Journal Pre-proof

A survey on machine learning techniques applied to source code

Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari,
Indira Vats, Hadi Moazen, Federica Sarro

PII: S0164-1212(23)00329-1
DOI: https://doi.org/10.1016/j.jss.2023.111934
Reference: JSS 111934

To appear in: *The Journal of Systems & Software*

Received date : 16 April 2023
Revised date : 10 November 2023
Accepted date : 14 December 2023

Please cite this article as: T. Sharma, M. Kechagia, S. Georgiou et al., A survey on machine learning techniques applied to source code. *The Journal of Systems & Software* (2023), doi: https://doi.org/10.1016/j.jss.2023.111934.

This is a PDF file of an article that has undergone enhancements after acceptance, such as the addition of a cover page and metadata, and formatting for readability, but it is not yet the definitive version of record. This version will undergo additional copyediting, typesetting and review before it is published in its final form, but we are providing this version to give early visibility of the article. Please note that, during the production process, errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

© 2023 Published by Elsevier Inc.

Journal Pre-proof

1 # A Survey on Machine Learning
2 # Techniques Applied to Source Code

3 **Tushar Sharma[1] ✉, Maria Kechagia[2], Stefanos Georgiou[3], Rohit Tiwari[4], Indira**
4 **Vats[5], Hadi Moazen[6], Federica Sarro[2]**

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

✉ **For correspondence:**
tushar@dal.ca

8

Data availability: Replication package can be found on GitHub - https://github.com/-tushartushar/ML4SCA

Funding: Maria Kechagia and Federica Sarro are supported by the ERC grant no. 741278 (EPIC).

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

24

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

27 ## 1. Introduction
28 In the last two decades, we have witnessed significant advancements in Machine Learning (ML),
29 including Deep Learning (DL) techniques, specifically in the domain of image [237, 476], text [255, 4],
30 and speech [418, 166, 165] processing. These advancements, coupled with a large amount of
31 open-source code and associated artifacts, as well as the availability of accelerated hardware, have
32 encouraged researchers and practitioners to use ML techniques to address software engineering
33 problems [513, 561, 27, 248, 34].
34 The software engineering community has employed ML and DL techniques for a variety of appli-
35 cations such as software testing [275, 361, 564], source code representation [27, 191], source code
36 quality analysis [34, 45], program synthesis [248, 540], code completion [288], refactoring [40],
37 code summarization [295, 252, 24], and vulnerability analysis [440, 429, 501] that involve source
38 code analysis. As the field of *Machine Learning for Software Engineering* (ML4SE) is expanding, the
39 number of available resources, methods, and techniques as well as tools and datasets, is also in-
40 creasing. This poses a challenge, to both researchers and practitioners, to fully comprehend the
41 landscape of the available resources and infer the potential directions that the field is taking. In

Journal Pre-proof

Journal Pre-proof

42 this context, literature surveys play an important role in understanding existing research, finding
43 gaps in research or practice, and exploring opportunities to improve the state of the art. By sys-
44 tematically examining existing literature, surveys may uncover hidden patterns, recurring themes,
45 and promising research directions. Surveys also identify untapped opportunities and formulation
46 of new hypotheses. A survey also serves as an educational tool, offering comprehensive coverage
47 of the field to a newcomer.

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

66 In this paper, we focus on the usage of ML, including DL, techniques for source code analysis.
67 Source code analysis involves tasks that take the source code as input, process it, and/or produce
68 source code as output. Source code representation, code quality analysis, testing, code summa-
69 rization, and program synthesis are applications that involve source code analysis. To the best of
70 our knowledge, the software engineering literature lacks a survey covering a wide range of source
71 code analysis applications using machine learning; this work is an attempt to fill this research gap.

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

79 This survey makes the following contributions to the field:

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

86 The rest of the paper is organized as follows. We present the followed methodology, including
87 the literature search protocol and research questions, in Section 2. Section 2.3, Section 3, Section 4,
88 and Section 5 provide the detailed results of our findings. We present threats to validity in Section 5,
89 and conclude the paper in Section 6.

Journal Pre-proof

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.

Category	Article	Scope	Data & Tools	Challenges	Type	#Studies
Program Comprehension	Nazar et al. [348]	TML	Tools	No	Lit. survey	59
	Zhang et al. [560]	DL	Data	No	Lit. survey	NA
	Song et al. [458]	TML & DL	No	Yes	Lit. survey	NA
Testing	Omri and Sinz [361]	DL	No	No	Lit. survey	NA
	Durelli et al. [132]	TML & DL	No	Yes	Mapping study	48
	Hall and Bowes [181]	TML	Yes	Yes	Meta-analysis	21
	Zhang et al. [564]	TML & DL	No	Yes	Lit. survey	46
	Pandey et al. [368]	TML	No	Yes	Lit. survey	154
	Singh et al. [452]	TML	No	No	Lit. survey	13
Vulnerability analysis	Li et al. [271]	DL	Yes	Yes	Meta-analysis	–
	Shen and Chen [440]	DL	No	Yes	Meta-analysis	–
	Ucci et al. [501]	TML	No	Yes	Lit. survey	64
	Jie et al. [215]	TML	No	No	Lit. survey	19
	Hanif et al. [187]	TML & DL	No	Yes	Lit. survey	90
Quality assessment	Alsolai and Roper [34]	TML	No	No	Lit. survey	56
	Tsintzira et al. [487]	TML	Yes	Yes	Lit. survey	90
	Azeem et al. [45]	TML	Yes	No	Lit. survey	15
	Caram et al. [77]	TML	No	No	Mapping study	25
	Lewowski and Madeyski [259]	TML	Yes	No	Lit. survey	45
Prog. synthesis	Goues et al. [162]	TML & DL	No	Yes	Lit. survey	NA
	Le et al. [248]	DL	Yes	Yes	Lit. survey	NA
Prog. synthesis & code representation	Allamanis et al. [27]	TML & DL	Yes	Yes	Lit. survey	39+48
Software engg. tasks	Yang et al. [544]	DL	Data	Yes	Lit. survey	250
Source-code analysis	Our study	TML & DL	Yes	Yes	Lit. survey	494

90 ## 2. Methodology

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

95 ### 2.1 Research objectives

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

97 RO1. *Identifying specific software engineering tasks involving source code that have been attempted*
98 *using machine learning.*
99 Our objective is to explore the extent to which machine learning has been applied to analyze
100 and process source code for SE tasks.We aim to summarize how ML can help engineers tackle
101 specific SE tasks.
102 RO2. *Summarizing the machine learning techniques used for these tasks.*
103 This objective explores the ML techniques commonly applied to source code for performing
104 the software engineering tasks identified above. We attempt to synthesize a mapping of tasks
105 (along with related sub-tasks) and corresponding ML techniques.
106 RO3. *Providing a list of available datasets and tools.*
107 With this goal, we aim to provide a consolidated summary of publicly available datasets and
108 tools along with their purpose.
109 RO4. *Identifying the challenges and perceived deficiencies in ML-enabled source code analysis and ma-*
110 *nipulation for software engineering.*
111 With this objective, we aim to identify challenges, and opportunities arising when applying
112 ML techniques to source code for SE tasks, as well as to understand the extent to which they
113 have been addressed in the articles surveyed.

114 ### 2.2 Literature search protocol

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

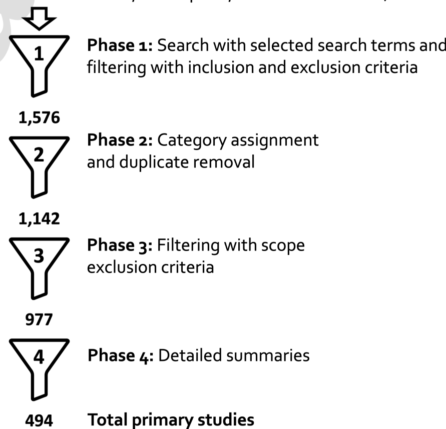

Figure 1. Overview of the search process

Journal Pre-proof

117 2.2.1 Literature search—Phase 1

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

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

Table 2. Search terms and corresponding relevant studies found in the second round of phase 1.

144

Category	Search terms	#Studies
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
Testing	word embedding in software testing	2
	automated Software Testing with machine learning	12
	optimal machine learning based random test generation	1
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
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
Program synthesis	automated program repair methods with machine learning	45

	program generation with machine learning	2
	object-oriented program repair with machine learning	15
	predicting patch correctness with machine learning	3
	multihunk program repair with machine learning	9
Program comprehension	autogenerated code with machine learning	6
	commits analysis with machine learning	34
	supplementary bug fixes with machine learning	9
Code summarization	automatic source code summarization with machine learning	43
	automatic commit message generation with machine learning	19
	comments generation with machine learning	11
Code review	security flaws detection in source code with machine learning	20
	intelligent source code security review with machine learning	2
Code representation	design pattern detection with machine learning	10
	human-machine-comprehensible software representation	1
	feature learning in source code	6
Code completion	missing software architectural tactics prediction with machine learning	1
	software system quality analysis with machine learning	6
	package-level tactic recommendation generation in source code	3
	identifier prediction in source code	13
	token prediction in source code	29

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

Journal Pre-proof

172 and sub-categories of common software engineering tasks can be found in Figure 3. Then, we
173 manually assigned a category and sub-category, if applicable, to each selected article based on the
174 (sub-)category to which an article contributes the most. The assignment was carried out by one of
175 the authors and verified by two other authors. We computed Cohen's Kappa [329] to measure the
176 initial disagreement; we found a strong agreement among the authors with $\kappa = 0.87$. In case of
177 disagreement, each author specified a key goal, operation, or experiment in the article, indicating
178 the rationale of the category assignment for the article. This exercise resolved the majority of the
179 disagreements. In the rest of the cases, we discussed the rationale identified by individual authors
180 and voted to decide a category or sub-category to which the article contributes the most. In this
181 phase, we also discarded duplicates or irrelevant studies not meeting our inclusion criteria after
182 reading their title and abstract. After this phase, we were left with $1,098$ studies.

183 2.2.3 Literature search—Phase 3

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

193 2.2.4 Literature search—Phase 4

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

200 2.3 Assigning articles to software engineering task categories

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

211 Some of the categories are quite generic and hence further categorization is possible based on
212 specific tasks. For each category, we identified sub-categories by grouping related studies together
213 and assigning an intuitive name representing the set of the studies. For example, the *testing* cate-
214 gory is further divided into *defect prediction*, and *test data/case generation*. We attempted to assign
215 a sub-category to each study; if none of the sub-categories was appropriate for a study, we did not
216 assign any sub-category to the study. One author of this paper assigned a sub-category to each
217 study based on the topic to which that study contributed the most. The initial assignment was
218 verified by two other authors of this paper, where disagreements were discussed and resolved to
219 reach a consensus. Figure 3 presents the distribution of studies per year *w.r.t.* each category and

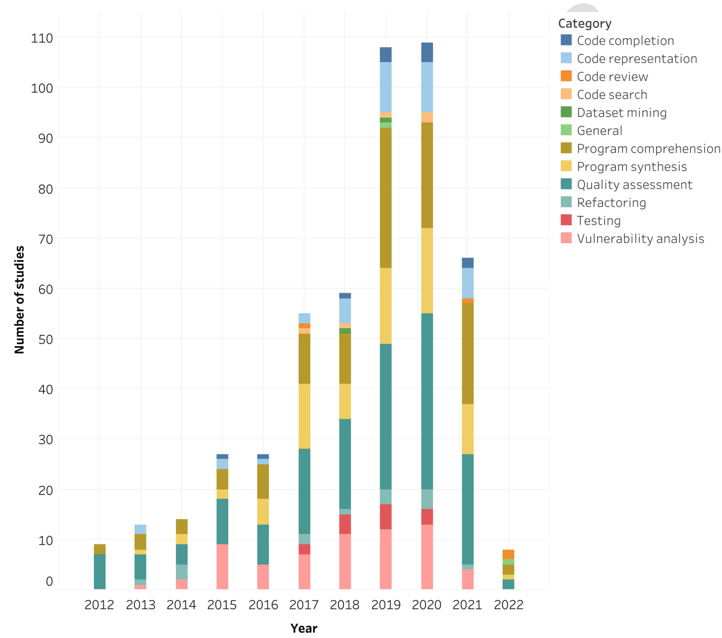

Figure 2. Category-wise distribution of studies

Category	Sub-category	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022
Code completion					1	1		1	3	4	2	
Code representation			2		2	1	2	5	10	10	6	
Code review							1				1	2
Code search							1	1	1	2		
Dataset mining								1	1			
General									1			1
Program comprehension		1			1	1	2		5	1	3	
	Change analysis		1				2		1	1	1	
	Code summarization	1	1			4	2	3	13	16	13	2
	Entity identification/recommendation		1	3	3	1	3	3	5	2		
	Program classification					1	1	4	4	1	3	
Program synthesis	Code generation			1		4	11	4	4	2	1	1
	Program Repair		1	1	1	1	2	3	11	11	9	
	Program translation				1					4		
Quality assessment									1	1		
	Clone detection				1	2		2	2	2	2	
	Code smell detection		1		1	1	2	4	10	14	12	1
	Defect prediction	7	4	4	7	5	12	11	13	15	7	1
	Quality prediction						3	1	3	2	1	
	Technical debt identification									1		
Refactoring			1	3			2	1	3	4	1	
Testing							1		1	2		
	Test data/case generation						1	4	4	1		
Vulnerability analysis			1	2	9	5	7	11	12	13	4	

Year

Figure 3. Category- and sub-categories-wise distribution of studies

corresponding sub-categories.

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.

				Code representation	Code completion	Code review	Code search	Dataset mining	Program comprehension	Program synthesis	Quality assessment	Refactoring	Testing	Vulnerability analysis	Total
Traditional Machine Learning	Model-based	Support Vector Regression	TML-SUP-MOD-SVR	0	0	0	0	0	0	0	1	0	1	0	2
		Support Vector Machine	TML-SUP-MOD-SVM	0	0	0	0	0	8	2	41	4	3	31	89
		Polynomial Regression	TML-SUP-MOD-POLY	0	0	0	0	0	0	0	1	0	0	0	1
		Logistic Regression	TML-SUP-MOD-LOG	0	1	0	0	1	2	2	22	4	1	8	41
		Locally Deep Support Vector Machines	TML-SUP-MOD-LDSVM	0	0	0	0	0	0	0	0	0	0	1	1
		Linear Regression	TML-SUP-MOD-LR	0	0	0	0	0	2	0	10	1	1	7	21
		Linear Discriminant Analysis	TML-SUP-MOD-LDA	1	1	0	0	0	0	0	0	0	0	2	4
		Least Median Square Regression	TML-SUP-MOD-LMSR	0	0	0	0	0	0	0	1	0	0	0	1
		LASSO	TML-SUP-MOD-LSS	0	0	0	0	0	0	0	0	0	0	1	1
	Tree-based	Boosted Decision Trees	TML-SUP-TR-BDT	0	0	0	0	0	0	0	0	0	0	1	1
		Classification And Regression Tree	TML-SUP-TR-CART	0	0	0	0	0	0	1	1	0	0	0	2
		Co-forest Random Forest	TML-SUP-TR-CRF	0	0	0	0	0	0	0	1	0	0	1	2
		Decision Forest	TML-SUP-TR-DF	0	0	0	0	0	0	0	0	0	0	1	1
		Decision Jungle	TML-SUP-TR-DJ	0	0	0	0	0	0	0	0	0	0	1	1
		Decision Stump	TML-SUP-TR-DS	0	0	0	0	0	0	0	0	0	0	2	2
		Decision Tree	TML-SUP-TR-DT	0	1	1	0	0	8	3	52	2	1	19	87
		Extra Trees	TML-SUP-TR-ET	0	0	0	0	0	0	0	3	0	0	0	3
		Gradient Boosted Trees	TML-SUP-TR-GBT	0	0	0	0	0	0	1	1	0	0	0	2
		Gradient Boosted Decision Tree	TML-SUP-TR-GBDT	0	0	0	0	0	0	0	0	0	0	2	2
		ID3	TML-SUP-TR-ID3	0	0	0	0	0	0	0	0	0	0	1	1
		Random Tree	TML-SUP-TR-RT	0	0	0	0	0	0	0	2	0	0	2	4
		Random Forest	TML-SUP-TR-RF	1	1	1	0	0	12	3	45	3	1	21	88
	Instance-based	COBWEB	TML-SUP-IN-CWEB	0	0	0	0	0	0	0	1	0	0	0	1
		KStar	TML-SUP-IN-KS	0	0	0	0	0	0	0	5	0	0	0	5
		K-Nearest Neighbours	TML-SUP-IN-KNN	0	0	0	0	0	3	0	13	0	1	9	26
	Probabilistic-based	Bayes Net	TML-SUP-PRO-BN	0	1	1	0	0	1	0	8	1	0	6	18
		Bayes Point Machine	TML-SUP-PRO-BPM	0	0	0	0	0	0	0	0	0	0	1	1
		Bernoulli Naives Bayes	TML-SUP-PRO-BNB	0	0	0	0	0	0	0	3	0	0	2	5
		Gaussian Naive Bayes	TML-SUP-PRO-GNB	0	0	0	0	0	0	0	5	0	0	1	6
		Graph random-walk with absorbing states	TML-SUP-PRO-GRASSHOPER	0	0	0	0	0	1	0	0	0	0	0	1
		Transfer Naive Bayes	TML-SUP-PRO-TNB	0	0	0	0	0	0	0	1	0	0	0	1
		Naive Bayes	TML-SUP-PRO-NB	0	0	0	0	0	7	1	40	2	2	16	68
		Multinomial Naive Bayes	TML-SUP-PRO-MNB	0	0	0	0	0	0	0	3	1	0	1	5
	Rule-based	Decision Table	TML-SUP-RUL-DTB	0	0	0	0	0	0	0	1	0	0	0	1
		Ripper	TML-SUP-RUL-Ripper	0	0	0	0	0	1	0	10	0	0	4	15
	Learn-to-Rank	Diverse Rank	TML-SUP-LR-DR	0	0	0	0	0	1	0	0	0	0	0	1
	Clustering	Hierarchical Clustering	TML-UNSUP-CLS-HC	0	0	0	0	0	0	1	0	0	0	0	1
		KMeans	TML-UNSUP-CLS-KM	0	0	0	0	0	0	0	1	0	0	1	2
	Other	Fuzzy Logic	TML-UNSUP-OTH-FL	0	0	0	0	0	0	0	1	0	0	0	1
		Maximal Marginal Relevance	TML-UNSUP-OTH-MMR	0	0	0	0	0	1	0	0	0	0	0	1
		Latent Dirichlet Allocation	TML-UNSUP-OTH-LDAA	0	0	0	1	0	9	0	3	1	0	0	14
	Evolutionary	Gene Expression Programming	TML-EVO-GEP	0	0	0	0	0	0	0	2	0	0	0	2
		Genetic Programming	TML-EVO-GP	0	0	0	0	0	0	0	3	0	0	0	3
	Meta-algorithms / General Approaches	AdaBoost	TML-GEN-AB	0	0	0	0	0	0	0	13	2	2	4	21
		Binary Relevance	TML-GEN-BR	0	0	0	0	0	0	0	1	0	0	0	1
		Classifier Chain	TML-GEN-CC	0	0	0	0	0	0	0	1	0	0	0	1
		Cost-Sensitive Classifer	TML-GEN-CSC	0	0	0	0	0	0	0	2	0	0	0	2
		Ensemble Learning	TML-GEN-EL	0	0	0	0	0	1	0	3	0	0	0	4
		Ensemble Learning Machine	TML-GEN-ELM	0	0	0	0	0	0	0	1	0	0	0	1
		Gradient Boosting	TML-GEN-GB	0	0	0	0	0	2	1	8	0	0	3	14
		Gradient Boosting Machine	TML-GEN-GBM	0	0	0	0	0	1	0	1	0	0	1	3
		Statiscal Machine Translation	TML-GEN-SMT	0	0	0	0	0	0	1	0	0	0	0	1
		Neural Machine Translation	TML-GEN-NMT	1	1	0	0	0	0	5	1	0	0	0	8
		Multiple Kernel Ensemble Learning	TML-GEN-MKEL	0	0	0	0	0	0	0	1	0	0	0	1
		Neural Machine Model	TML-GEN-NLM	0	0	0	0	0	1	0	0	0	0	0	1
		Majority Voting Ensemble	TML-GEN-MVE	0	0	0	0	0	0	0	1	0	0	0	1
		Bagging	TML-GEN-B	0	0	0	0	0	0	0	11	0	0	1	12
		LogitBoost	TML-GEN-LB	0	0	0	0	0	0	0	4	1	0	1	6
		Kernel Based Learning	TML-GEN-KBL	0	0	0	0	0	0	0	1	0	0	0	1

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

Journal Pre-proof

Journal Pre-proof

Category	Subcategory	Technique	Abbreviation	Code representation	Code completion	Code review	Code search	Dataset mining	Program comprehension	Program synthesis	Quality assessment	Refactoring	Testing	Vulnerability analysis	Total
Deep Learning	RNN	Bidirectional GRU	DL-RNN-Bi-GRU	1	0	0	0	0	0	0	0	0	0	1	2
		Bidirectional RNN	DL-RNN-Bi-RNN	0	0	0	0	0	1	0	0	0	0	0	1
		Bidirectional LSTM	DL-RNN-Bi-LSTM	0	0	0	0	0	5	2	2	0	0	3	12
		Gated Recurrent Unit	DL-RNN-GRU	1	1	0	0	0	9	0	1	0	0	3	15
		Hierarchical Attention Network	DL-RNN-HAN	1	0	0	0	0	1	0	0	0	0	0	2
		Recurrent Neural Network	DL-RNN-RNN	3	3	0	1	0	9	5	0	0	0	2	23
		Pointer Network	DL-RNN-PN	0	1	0	0	0	0	0	0	0	0	0	1
		Modular Tree Structured RNN	DL-RNN-MTN	1	1	0	0	0	0	0	0	0	0	0	2
		Long Short Term Memory	DL-RNN-LSTM	3	4	0	1	0	21	10	6	1	1	5	52
	Graph	Gated Graph Neural Network	DL-GRA-GGNN	0	0	0	1	0	0	2	0	0	0	0	3
		Graph Convolutional Networks	DL-GRA-GCN	0	0	0	0	0	0	0	0	0	0	1	1
		Graph Interval Neural Network	DL-GRA-GINN	1	0	0	0	0	0	0	0	0	0	0	1
		Graph Neural Network	DL-GRA-GNN	2	0	0	0	0	3	0	1	0	0	0	6
	CNN	Convolutional Neural Network	DL-CNN-CNN	3	0	0	1	0	4	2	8	0	0	5	23
		Faster R-CNN	DL-CNN-FR-CNN	0	0	0	0	0	0	0	0	0	1	0	1
		Text-CNN	DL-CNN-TCNN	0	0	0	0	0	0	0	0	0	0	1	1
	Vanilla	Artificial Neural Network	DL-ANN	0	1	0	0	0	2	1	21	3	1	3	32
		Autoencoder	DL-AE	1	0	0	0	0	0	0	2	0	0	1	4
		Deep Neural Network	DL-DNN	2	0	0	1	0	6	2	5	1	0	4	21
		Regression Neural Network	DL-RGNN	0	0	0	0	0	0	0	1	0	0	0	1
		Multi Level Perceptron	DL-MLP	0	0	0	0	0	2	3	14	1	1	5	26
	Transformers	Bidirectional Encoder Representation from T	DL-XR-BERT	0	0	0	0	0	1	1	0	0	0	0	2
		CodeBERT	DL-XR-CodeBERT	1	0	0	0	0	0	1	0	0	0	0	2
		Generative Pretraining Transformer for Code	DL-XR-GPT-C	0	0	0	0	0	0	1	0	0	0	0	1
		Transformer	DL-XR-TF	2	1	2	0	0	4	3	1	0	0	0	13
	Other	Bilateral Neural Network	DL-OTH-BiNN	0	0	0	0	0	0	0	1	0	0	0	1
		Cascade Correlation Network	DL-OTH-CCN	0	0	0	0	0	0	0	1	0	0	0	1
		Code2Vec	DL-OTH-Code2Vec	5	0	0	0	0	1	0	0	0	0	0	6
		Deep Belief Network	DL-OTH-DBN	0	0	0	0	0	0	0	2	0	0	2	4
		Doc2Vec	DL-OTH-Doc2Vec	0	0	0	0	0	0	0	0	0	0	2	2
		Encoder-Decoder	DL-OTH-EN-DE	3	1	0	0	0	17	10	0	0	0	0	31
		FastText	DL-OTH-FT	0	0	0	0	0	0	0	0	0	0	1	1
		Functional Link ANN	DL-OTH-FLANN	0	0	0	0	0	0	0	1	0	0	0	1
		Guassian Encoder-Decoder	DL-OTH-GED	0	0	0	0	0	0	1	0	0	0	0	1
		Global Vectors for Word Representation	DL-OTH-Glove	1	0	0	0	0	0	0	0	0	0	0	1
		Word2Vec	DL-OTH-Word2Vec	0	0	0	0	0	0	0	1	0	0	0	1
		Sequence-to-Sequence	DL-OTH-Seq2Seq	1	0	0	0	0	2	2	0	0	1	0	6
		Reverse NN	DL-OTH-ReNN	0	0	0	0	0	0	0	1	0	0	0	1
		Residual Neural Network	DL-OTH-ResNet	0	0	0	0	0	0	1	1	0	0	0	2
		Radial Basis Function Network	DL-OTH-RBFN	0	0	0	0	0	0	0	1	0	0	0	1
		Probabilistic Neural Network	DL-OTH-PNN	0	0	0	0	0	0	0	1	1	0	0	2
		Node2Vec	DL-OTH-Node2Vec	0	0	0	0	0	0	0	1	0	0	0	1
		Neural Network for Discrete Goal	DL-OTH-NND	0	0	0	0	0	0	0	2	0	0	0	2
Reinforcement Learning		Double Deep Q-Networks	RL-DDQN	0	0	0	0	0	0	0	0	0	1	0	1
		Reinforcement Learning	RL-RL	0	0	0	0	0	3	0	0	0	0	0	3
Others	Hybrid	Adaptive neuro fuzzy inference system	OTH-HYB-ANFIS	0	0	0	0	0	0	0	1	0	0	0	1
	Optimization Techniques	Expectation Minimization	OTH-OPT-EM	0	0	0	0	0	0	0	1	0	0	0	1
		Gradient Descent	OTH-OPT-GD	0	0	0	0	0	0	1	0	0	0	0	1
		Stochastic Gradient Descent	OTH-OPT-SGD	0	0	0	0	0	0	0	2	0	0	0	2
		Sequential Minimal Optimization	OTH-OPT-SMO	0	0	0	0	0	0	0	5	0	0	1	6
		Particle Swarm Optimization	OTH-OPT-PSO	0	0	0	0	0	0	0	1	0	0	0	1

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 (TML), 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

Journal Pre-proof

234 each study while summarizing the study. Given that a study may use multiple ML techniques, we
235 developed a script to split the techniques and create a CSV file containing one ML technique and
236 the corresponding paper category. We then compute a number of times for each ML technique
237 for each software engineering task category to generate the tables. In these tables we refer to ML
238 techniques with their commonly used acronym along with their category and sub-category. It is ev-
239 ident from these tables that SVM, RF, and DT are the most frequently used traditional ML techniques,
240 whereas, the RNN family (including LSTM and GRU) is the most commonly used DL technique.

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

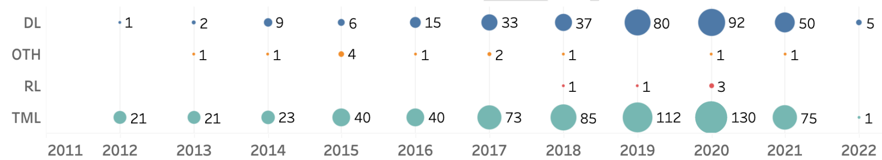

Figure 4. Usage of ML techniques by categories per year

246 **Venue and article categories:** We identified and manually curated the software engineering venue
247 for each study discussed in our literature review. Figure 5 shows the venues for the considered
248 categories. We show the most prominent venues per category. Each label includes a number
249 indicating the number of articles published at the same venue in that category.
250 We observe that ICSE is the top venue, appearing in three categories. IEEE *Access* is the top jour-
251 nal for the considered categories. Machine learning conferences such as ICLR also appear as the
252 top venues for the *program synthesis* category. The category *program comprehension* exhibits the
253 highest concentration of articles to a relatively small list of top venues where approximately 50%
254 of articles come from the top venues (with at least four studies). On the other hand, researchers
255 publish articles related to *testing*, *code completion*, and *vulnerability analysis* in a rather diverse set
256 of venues.

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

265 **Popular models:** As part of collecting metadata and summarizing studies, we identified the pro-
266 posed model, if any, for each selected study. We considered novel proposed models only and not
267 the name of the approach or method in this analysis. We also obtained the number of citations
268 for the study. In Table 5, we present the most popular model, in no particular order, by using the
269 number of citations as the metric to decide the popularity. We collected the number of citations
270 at the end of August 2023 and included all the models with corresponding citations over 100.
271 In the rest of this section, we delve into each category and sub-category at a time, break down
272 the entire workflow of a code analysis task into fine-grained steps, and summarize the method
273 and ML techniques used. It is worth emphasizing that we structure the discussion around the cru-

Journal Pre-proof

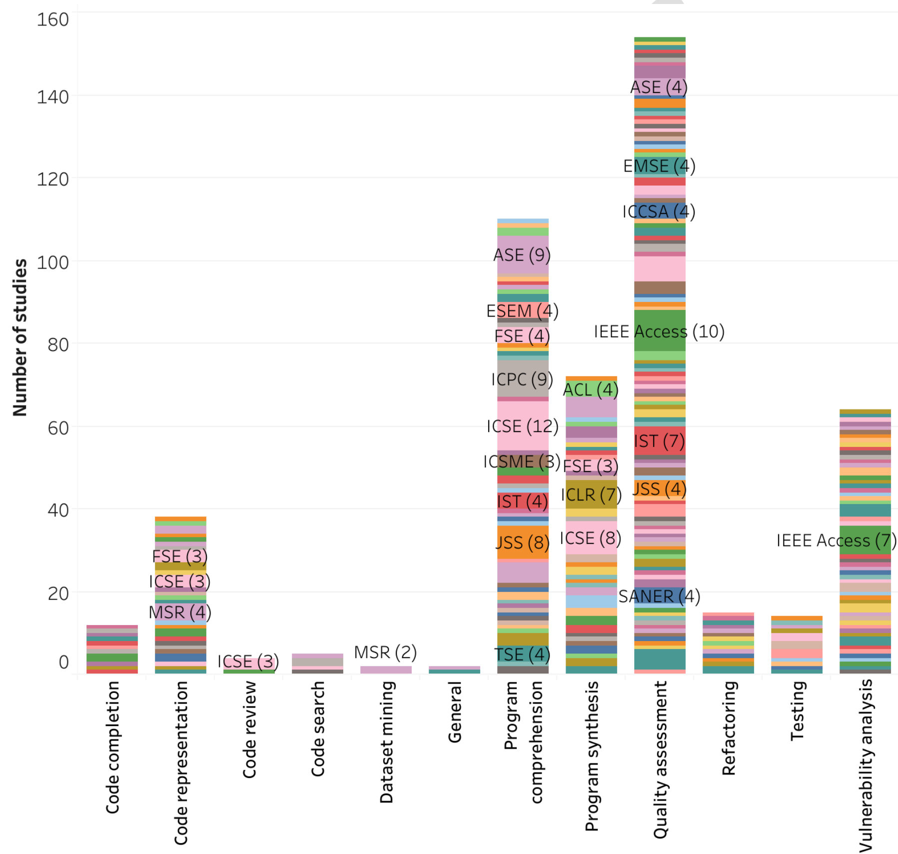

Figure 5. Top venues for each considered category

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

Journal Pre-proof

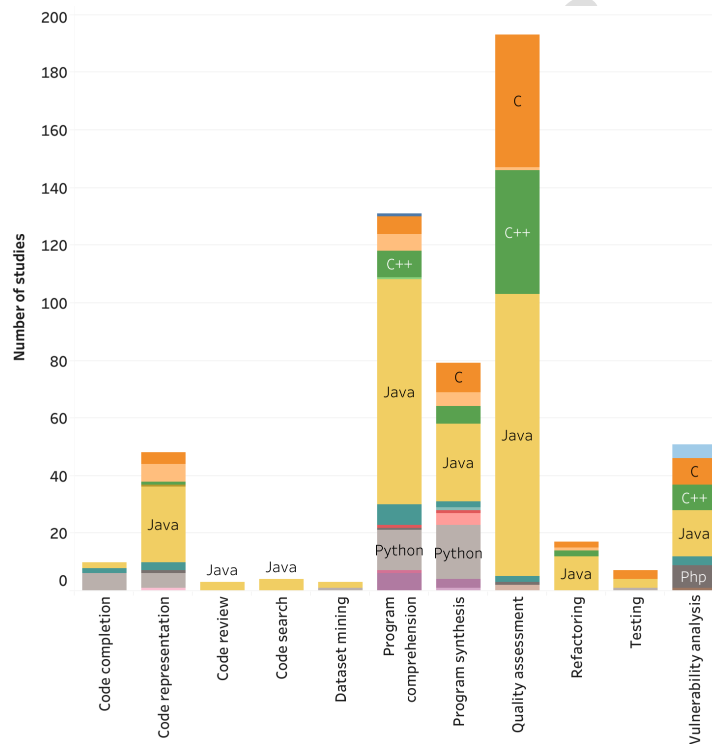

Figure 6. Target programming languages for each considered category

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 *syntaxeme*), 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

Journal Pre-proof

Table 5. Popular models proposed in the selected studies.

Model	#Citations	Model	#Citations
Transfer Naive Bayes [307]	513	Code Generation Model [551]	651
Path-based code representation [30]	230	Multi-headed pointer network [507]	128
Inst2Vec [57]	234	Code-NN [204]	681
DeepCoder [47]	612	ASTNN [563]	498
Code2Seq [31]	643	Code2Vec [32]	1,093
TBCNN [342]	695	Program as graph model [67]	159
SLAMC [352]	130	Coding criterion [377]	128
TransCoder [408]	115	TreeGen [468]	124
Codex [93]	897	AlphaCode [270]	317

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-interpretation, 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[1] AST. 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 GRU-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

[1] https://clang.llvm.org/

Journal Pre-proof

346 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* subcategories in this section.

Journal Pre-proof

391 ### 3.3.1 Program repair

392 Automated Program Repair (APR) refers to techniques that attempt to automatically identify patches
393 for a given bug (*i.e.,* programming mistakes that can cause an unintended run-time behavior), which
394 can be applied to software with a little or without human intervention [162]. Program repair typ-
395 ically consists of two phases. Initially, the repair tool uses fault localization to detect a bug in the
396 software under examination, then, it generates patches using techniques such as search-based
397 software engineering and logic rules that can possibly fix a given bug. To validate the generated
398 patch, the (usually manual) evaluation of the semantic correctness[2] of that patch follows.

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

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

413 **Dataset preparation:** The majority of the studies extract buggy project revisions and manual
414 fixes from buggy software projects. Most studies leverage source-code naturalness. For instance,
415 Tufano et al. [492] extracted millions of bug-fixing pairs from GitHub, Amorim et al. [39] lever-
416 aged the naturalness obtained from a corpus of known fixes, and Chen et al. [97] used natural
417 language structures from source code. Furthermore, many studies develop their own large-scale
418 bug benchmarks. Ahmed et al. [10] leveraged 4,500 erroneous C programs, Gopinath et al. [161]
419 used a suite of programs and datasets stemmed from real-world applications, Long and Rinard
420 [297] used a set of successful manual patches from open-source software repositories, and Mash-
421 hadi and Hemmati [326] used the *ManySStuBs4J* dataset containing natural language description
422 and code snippets to automatically generate code fixes. Le et al. [249] created an oracle for predict-
423 ing which bugs should be delegated to developers for fixing and which should be fixed by repair
424 tools. Jiang et al. [211] used a dataset containing more than 4 million methods extracted. White
425 et al. [533] used Spoon, an open-source library for analyzing and transforming Java source code,
426 to build a model for each buggy program revision. Pinconschi et al. [382] constructed a dataset
427 containing vulnerability-fix pairs by aggregating five existing dataset (Mozilla Foundation Security
428 Advisories, SecretPatch, NVD, Secbench, and Big-Vul). The dataset *i.e., PatchBundle* is publicly avail-
429 able on GitHub. Cambronero and Rinard [76] proposed a method to generate new supervised
430 machine learning pipelines. To achieve the goal, the study trained using a collection of 500 super-
431 vised learning programs and their associated target datasets from Kaggle. Liu et al. [287] prepared
432 their dataset by selecting 636 closed bug reports from the Linux kernel and Mozilla databases.
433 Svyatkovskiy et al. [475] constructed their experimental dataset from the 2700 top-starred Python
434 source code repositories on GitHub. CODIT [82] collects a new dataset—*Code-ChangeData*, consist-
435 ing of 32,473 patches from 48 open-source GitHub projects collected from Travis Torrent.

436 Other studies use existing bug benchmarks, such as Defects4J [218] and IntroClass [250], which
437 already include buggy revisions and human fixes, to evaluate their approaches. For instance, Saha
438 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].

Journal Pre-proof

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, asts, 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 apis 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

Journal Pre-proof

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, GGF, 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] utilized 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,[3] 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. Pinconschi et al. [382] used SequenceR, a sequence-to-sequence model, to patch security faults in C programs. Zhu et al. [582] proposed a tool Recoder, 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,[4] 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 amendable 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.

[3]https://github.com/GumTreeDiff/gumtree
[4]https://opennmt.net/

Journal Pre-proof

537 Tian et al. [483] applied a ML predictor with BERT transformer-based embeddings associated with lo-
538 gistic regression to learn code representations in order to learn deep features that can encode the
539 properties of patch correctness. Saha et al. [416] used similarity analysis for repairing bugs that
540 may require applying a substantially similar patch at a number of locations. Additionally, Wang
541 et al. [518] used also similarity metrics to compare the differences among machine-generated and
542 human patches. Santos et al. [421] used n-grams and NNs to detect and correct syntax errors.

543 *Logic-based rules:* Alrajeh et al. [33] combined model checking and logic-based learning to sup-
544 port automated program repair. Cai et al. [74] also combined model-checking and ML for program
545 repair. Shim et al. [444] used inductive program synthesis (DeeperCoder), by creating a simple Do-
546 main Specific Language (DSL), and ML to generate computer programs that satisfies user require-
547 ments and specification. Sakkas et al. [419] combined type rules and ML (*i.e.,* multi-class classifica-
548 tion, DNNs, and MLP) for repairing compile errors.

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

569 *Recurrent neural networks:* DL approaches such as RNNs (*e.g.,* LSTM and Transformer) have been used
570 for synthesizing new code statements by learning patterns from a previous list of code statement,
571 *i.e.,* this techniques can be used to mainly predict the next statement. Such approaches often
572 leverage word embeddings. Dantas et al. [118] combined Doc2Vec and LSTM, to capture dependen-
573 cies between source code statements, and improve the fault-localization step of program repair.
574 Ahmed et al. [10] developed a repair approach (Tracer) for fixing compilation errors using RNNs.
575 Recently, Li et al. [267] introduced DLFix, which is a context-based code transformation learning
576 for automated program repair. DLFix uses RNNs and treats automated program repair as code
577 transformation learning, by learning patterns from prior bug fixes and the surrounding code con-
578 texts of those fixes. Svyatkovskiy et al. [474] presented IntelliCode that uses a Transformer model
579 that predicts sequences of code tokens of arbitrary types, and generates entire lines of syntacti-
580 cally correct code. Chen et al. [97] used the LSTM for synthesizing `if-then` constructs. Similarly,
581 Vasic et al. [507] applied the LSTM in multi-headed pointer networks for jointly learning to localize
582 and repair variable misuse bugs. Bhatia et al. [59] combined neural networks, and in particular
583 RNNs, with constraint-based reasoning to repair syntax errors in buggy programs. Chen et al. [101]
584 applied LSTM for sequence-to-sequence learning achieving end-to-end program repair through the
585 SequenceR repair tool they developed. Majd et al. [313] developed SLDeep, statement-level soft-
586 ware defect prediction, which uses LSTM on static code features.

Journal Pre-proof

587 Apart from above-mentioned techniques, White et al. [533] developed DeepRepair, a recur-
588 sive unsupervised deep learning-based approach, that automatically creates a representation of
589 source code that accounts for the structure and semantics of lexical elements. The neural network
590 language model is trained from the file-level corpus using embeddings.

591 ### 3.3.2 Code generation

592

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

596 **Dataset preparation:** Yin and Neubig [552] proposed a transition-based neural semantic parser,
597 namely TRANX, which generates formal meaning representation from natural language text. They
598 used multiple datasets for their study—dataset proposed by Dong and Lapata [128] containing 880
599 geography-related questions, *Django* dataset [358], as well as *WikiSQL* dataset [576]. Similarly, Sun
600 et al. [468] and Shin et al. [446] used the *HearthStone* dataset [283] for Python code generation;
601 in addition, Shin et al. [446] used the Spider [557] dataset for training. Liang et al. [272] used the
602 semantic parsing dataset *WebQuestionsSP*[550] consisting 3,098 question-answer pairs for training
603 and 1,639 for testing. Bielik et al. [60] used the *Linux Kernel* dataset [222], and the *Hutter Prize*
604 *Wikipedia* dataset.[5] Devlin et al. [122] evaluated their architecture on 205 real-world *Flash-Fill* in-
605 stances [170]. Xiong et al. [537] used training data stemming from two *Defects4J* projects and their
606 related JDK packages. Wei et al. [530] conducted experiments on Java and Python projects collected
607 from GITHUB used by previous work (such as by Hu et al. [198], Hu et al. [199], Wan et al. [511]).
608 Some studies curated datasets for their experiments. For example, Chen et al. [93] created
609 *HumanEval*, a dataset containing 164 programming problems crafted manually for evaluation. Sim-
610 ilarly, Li et al. [270] first used a curated set of public GITHUB repositories implemented in several
611 popular languages such as C++, C#, Java, Go, and Python for pre-training. They created a dataset,
612 *CodeContests*, for fine-tuning. The dataset includes problems, solutions, and test cases scraped
613 from the Codeforces platform. Furthermore, IntelliCode [474] is trained on 1.2 billion lines of
614 source code written in the Python, C#, JavaScript and TypeScript programming languages. Alla-
615 manis et al. [28] evaluated their models on a large dataset of 2.9 million lines of code. Cai et al. [75]
616 used a training set that contains 200 traces for addition, 100 traces for bubble sort, 6 traces for topo-
617 logical sort, and 4 traces for quicksort. Devlin et al. [121] used programming examples that involve
618 induction, such as I/O examples. Shu and Zhang [449] used training data to generate programs at
619 various levels of complexity according to 45 predefined tasks (*e.g.,* Split, Join, Select). Murali et al.
620 [344] used a corpus of about 150,000 API-manipulating Android methods. Shin et al. [447] propose
621 a new approach to generate desirable distribution for the target datasets for program induction
622 and synthesis tasks.

623 **Feature extraction:** Studies in this category extensively used AST during the feature extraction
624 step. TRANX [552] maps natural language text into an AST using a series of tree-construction ac-
625 tions. Similarly, Sun et al. [468] parsed a program as an AST and decomposed the program into
626 several context-free grammar rules. Also, the study by Yin and Neubig [551] transformed state-
627 ments to ASTs. These ASTs are generated for all well-formed programs using parsers provided by
628 the programming language under examination. Furthermore, Rabinovich et al. [395] developed a
629 model that used a modular decoder, whose sub-models are composed using natively generated
630 ASTs. Each sub-model is associated with a specific construct in the AST grammar, and, then, it is
631 invoked when that construct is required in the output tree.
632 Some studies in the category used examples of input and output to learn code generation.
633 *Euphony* [257] learns good representation using easily obtainable solutions for given programs.
634 *DeepCoder* [47] observes inputs and outputs, by leveraging information from interpreters. Then,

[5] http://prize.hutter1.net/

Journal Pre-proof

635 *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.

640 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.

656 **ML model training:** A majority of the studies in this category relies on the RNN-based ecoder-decoder architecture. TRANX [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.

673 Some of the studies employed transformer-based models. Sun et al. [468] proposed TreeGen for code generation. They implemented an AST readerer 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.

681 *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 684

Journal Pre-proof

685 source code generation of machine learning models as well as the generation algorithms for com-
686 monly used ML methods. Chen et al. [99] introduced a technique that is based on execution-guided
687 synthesis and uses a synthesizer ensemble. This approach leverages semantic information to en-
688 semble multiple neural program synthesizers. Chen et al. [97] used latent attention to compute
689 token weights. They found that latent attention performs better in capturing the sentence struc-
690 ture. Allamanis et al. [28] used DL models to learn semantics from programs. They used the code's
691 graph structure and learned program representations over the generated graphs. Xiong et al. [537]
692 applied the gradient boosting tree algorithm to train their models. Devlin et al. [121] used the trans-
693 fer learning and k-shot learning approach for cross-task knowledge transfer to improve program
694 induction in limited-data scenarios. Shu and Zhang [449] proposed NPBE (Neural Programming by
695 Example) that teaches a DNN to compose a set of predefined atomic operations for string manipula-
696 tions. Murali et al. [344] trained a neural generator on program sketches to generate source code
697 in a strongly typed, Java-like programming language. Liang et al. [272] introduced the Neural Sym-
698 bolic Machine (NSM), based on a sequence-to-sequence neural network induction, and apply it to
699 semantic parsing. Shin et al. [446] employed non-parametric Bayesian inference to mine the code
700 idioms that frequently occur in a given corpus and trained a neural generative model to option-
701 ally emit named idioms instead of the original code fragments. Maddison and Tarlow [310] used
702 models that are based on probabilistic context free grammars (PCFGs) and a neuro-probabilistic
703 language, which are extended to incorporate additional source code-specific structures.

704 3.3.3 Program translation

705

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

720 3.4 Quality assessment

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

724 3.4.1 Code smell detection

725 Code smells impair the code quality and make the software difficult to extend and maintain [435].
726 Extensive literature is available on detecting smells automatically [435]; ML techniques have been
727 used to classify smelly snippets from non-smelly code. First, source code is pre-processed to ex-
728 tract individual samples (such as a class, file, or method). These samples are classified into positive
729 and negative samples. Afterwards, relevant features are identified from the source code and those
730 features are then fed into an ML model for training. The trained model classifies a source code sam-
731 ple into a smelly or non-smelly code.

Journal Pre-proof

Journal Pre-proof

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

Journal Pre-proof

781 models. These models include *Decision Tree*, *Support Vector Machine*, *Random Forest*, *Naive Bayes*,
782 *Logistic Regression*, *Linear Regression*, *Polynomial Regression*, *Bagging*, and *Multilayer Perceptron*. The
783 majority of studies [303, 240, 174, 8, 360, 390, 149, 148, 374, 481, 111, 127, 114, 495, 110, 498, 499,
784 226, 124, 14, 175, 206, 180, 173] in this category compared the performance of various ML models.
785 Some of the authors experimented with individual ML models; for example, Kaur et al. [223] and
786 Amorim et al. [38] used *Support Vector Machine* and *Decision Tree*, respectively, for smell detection.

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

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

800 *Discussion:* A typical ML model trained to classify samples into either smelly or non-smelly samples.
801 The majority of the studies focused on a relatively small set of known code smells— *god class* [52,
802 303, 223, 174, 8, 360, 149, 167, 42, 111, 78, 179], *feature envy* [52, 223, 8, 149, 42, 148, 111, 437, 179],
803 *long method* [223, 174, 149, 167, 42, 148, 111, 45, 179], *data class* [223, 360, 149, 167, 42, 148], and
804 *complex class* [303, 174, 360]. Results of these efforts vary significantly; F1 score of the ML models
805 vary between 0.3 to 0.99. Among the investigated ML models, authors widely report that *Decision*
806 *Tree* [45, 148, 13, 174] and *Random Forest* [45, 148, 240, 42, 336] perform the best. Other methods
807 that have been reported better than other ML models in their respective studies are *Support Vector*
808 *Machine* [496], *Boosting* [302], and *autoencoders* [437].

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

815 ### 3.4.2 Code clone detection

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

823 **Dataset preparation:** Manual annotation is a common way to prepare a dataset for applying ML
824 to identify code clones [340, 341, 532]. Mostaeen et al. [340] used a set of tools (NiCad, Deckard,
825 iClones, CCFinderX and SourcererCC) to first identify a list of code clones; they then manually vali-
826 dated each of the identified clone set. Yang et al. [542] used existing code clone detection tools to
827 generate their training set. Some authors (such as Bandara and Wijayarathna [49] and Hammad
828 et al. [183]) relied on existing code-clone datasets. Zhang and Khoo [562] used NiCad to detect all
829 clone groups from each version of the software. The study mapped the clones from a consecu-

Journal Pre-proof

830 tive version and used the mapping to predict clone consistency at both the clone-creating and the
831 clone-changing time. Bui et al. [72] deployed an interesting mechanism to prepare their code-clone
832 dataset. They crawled through GitHub repositories to find different implementations of sorting al-
833 gorithms; they collected 3,500 samples from this process.

834 **Feature extraction:** The majority of the studies relied on the textual properties of the source code
835 as features. Bandara and Wijayarathna [49] identified features such as the number of characters
836 and words, identifier count, identifier character count, and underscore count using the ANTLR tool.
837 Some studies [340, 341, 339] utilized line similarity and token similarity. Yang et al. [542] and Ham-
838 mad et al. [183] computed TF-IDF along with other metrics such as position of clones in the file.
839 Cesare et al. [79] extracted 30 package-level features including the number of files, hashes of the
840 files, and common filenames as they detected code clones at the package level. Zhang and Khoo
841 [562] obtained a set of code attributes (*e.g.,* lines of code and the number of parameters), context
842 attribute set (*e.g.,* method name similarity, and sum of parameter similarity). Similarly, Sheneamer
843 and Kalita [441] obtained metrics such as the number of constructors, number of field access, and
844 super-constructor invocation from the program AST. They also employed program dependence
845 graph features such as *decl_assign* and *control_decl*. Along the similar lines, Zhao and Huang [571]
846 used CFG and DFG (Data Flow Graph) for clone detection. Some of the studies [72, 532, 142] relied
847 on DL methods to encode the required features automatically without specifying an explicit set of
848 features.

849 **ML model training:**

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

856 DL*-based models:* DL models such as ANN [340, 339], DNN [142, 571], and RNN with *Reverse neural*
857 *network* [532] are also employed extensively. Bui et al. [71] and Bui et al. [72] combined neural
858 networks for ML models' training. Specifically, Bui et al. [71] built a *Bilateral neural network* on
859 top of two underlying sub-networks, each of which encodes syntax and semantics of code in one
860 language. Bui et al. [72] constructed BiTBCNNs—a combination layer of sub-networks to encode
861 similarities and differences among code structures in different languages. Hammad et al. [183]
862 proposed a Clone-Advisor, a DNN model trained by fine-tuning GPT-2 over the BigCloneBench code
863 clone dataset, for predicting code tokens and clone methods.

864 3.4.3 Defect prediction

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

872 **Dataset preparation:** To train an ML model for predicting defects in source code a labeled dataset
873 is required. For this purpose, researchers have used some well-known and publicly available
874 datasets. For instance, a large number of studies [80, 157, 316, 454, 85, 58, 320, 453, 81, 517, 106,
875 265, 125, 386, 307, 229, 90, 116, 520, 442, 129, 455, 568, 73, 126, 423, 521, 281, 404, 263, 224, 359,
876 246, 457, 366, 318, 393, 323, 470, 137, 365, 554, 469, 120, 12, 15] used the PROMISE dataset [424].
877 Some studies used other datasets in addition to PROMISE dataset. For example, Liang et al. [273]

Journal Pre-proof

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 GitHub 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 Children, Coupling Between Objects, and Cyclomatic Complexity. 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. Dos Santos et al. [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 DL 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-

Journal Pre-proof

927 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 for 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, Alikhashashneh 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[1] 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: Alikhashashneh 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

Journal Pre-proof

974 and Python code completion. Svyatkovskiy et al. [473] used 2,700 Python open-source software
975 GitHub repositories for the evaluation of their novel approach, Pythia.

976 The rest of the approaches employed existing benchmarks and datasets. Rahman et al. [398]
977 trained their proposed model using the data extracted from Aizu Online Judge (aoj) system. Liu et al.
978 [289], Liu et al. [288] performed experiments on three real-world datasets to evaluate the effective-
979 ness of their model when compared with the state-of-the-art approaches. Li et al. [264] conducted
980 experiments on two datasets to demonstrate the effectiveness of their approach consisting of an
981 attention mechanism and a pointer mixture network on code completion tasks. Schuster et al.
982 [426] used a public archive of GitHub from 2020 [1].

983 **Feature extraction:** Studies in this category extract source code information in variety of forms.
984 Gopalakrishnan et al. [158] extracted relationships between topical concepts in the source code
985 and the use of specific architectural developer tactics in that code. Liu et al. [289], Liu et al. [288]
986 introduced a self-attentional neural architecture for code completion with multi-task learning. To
987 achieve this, they extracted the hierarchical source code structural information from the programs
988 considered. Also, they captured the long-term dependency in the input programs, and derived
989 knowledge sharing between related tasks. Li et al. [264] used locally repeated terms in program
990 source code to predict out-of-vocabulary (OoV) words that restrict the code completion. Chen and
991 Wan [92] proposed a tree-to-sequence (Tree2Seq) model that captures the structure information
992 of source code to generate comments for source code. Raychev et al. [401] used asts and per-
993 formed prediction of a program element on a dynamically computed context. Svyatkovskiy et al.
994 [473] introduced a novel approach for code completion called Pythia, which exploits state-of-the-
995 art large-scale dl models trained on code contexts extracted from asts.

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

998 *Recurrent Neural Networks:* For code completion, researchers mainly try to predict the next token.
999 Therefore, most approaches use rnns. In particular, Terada and Watanobe [479] used lstm for
1000 code completion to facilitate programming education. Rahman et al. [398] also used lstm. Wang
1001 et al. [519] used an lstm-based neural network combined with several techniques such as *Word*
1002 *Embedding* models and *Multi-head Attention Mechanism* to complete programming code. Zhong
1003 et al. [575] applied several dl techniques, including lstm, *Attention Mechanism* (AM), and *Sparse*
1004 *Point Network* (spn) for JavaScript code suggestions.

1005 Apart from lstm, researchers have used rnn with different approaches to perform code sugges-
1006 tions. Li et al. [264] applied neural language models, which involve attention mechanism for rnn,
1007 by learning from large codebases to facilitate effective code completion for dynamically-typed pro-
1008 gramming languages. Hussain et al. [202] presented CodeGRU that uses gru for capturing source
1009 codes contextual, syntactical, and structural dependencies. Yang et al. [545] presented rep to im-
1010 prove language modeling for code completion. Their approach uses learning of general token rep-
1011 etition of source code with optimized memory, and it outperforms lstm. Schumacher et al. [425]
1012 combined neural and classical ml including rnns, to improve code recommendations.

1013 *Probabilistic Models:* Earlier approaches for code completion used statistical learning for recom-
1014 mending code elements. In particular, Gopalakrishnan et al. [158] developed a recommender sys-
1015 tem using prediction models including neural networks for latent topics. Han et al. [185], Han et al.
1016 [186] applied *Hidden Markov Models* to improve the efficiency of code-writing by supporting code
1017 completion of multiple keywords based on non-predefined abbreviated input. Proksch et al. [391]
1018 used *Bayesian Networks* for intelligent code completion. Raychev et al. [401] utilized a probabilistic
1019 model for code in any programming language with *Decision Tree*. Svyatkovskiy et al. [473] proposed
1020 Pythia that employs a *Markov Chain* language model. Their approach can generate ranked lists of
1021 methods and api recommendations, which can be used by developers while writing programs.

1022 *Other techniques:* Recently, new approaches have been developed for code completion based on

Journal Pre-proof

1023 multi-task learning, code representations, and NMT. For instance, Liu et al. [289], Liu et al. [288] ap-
1024 plied Multi-Task Learning (MTL) for suggesting code elements. Lee et al. [256] developed MergeLog-
1025 ging, a DLbased merged network that uses code representations for automated logging decisions.
1026 Chen and Wan [92] applied Tree2Seq model with NMT techniques for code comment generation.

1027 3.6 Program Comprehension

1028 Program comprehension techniques attempt to understand the theory of comprehension process
1029 of developers as well as the tools, techniques, and processes that influence the comprehension
1030 activity [463]. We summarized, in the rest of the section, program comprehension studies into
1031 four sub-categories *i.e.,* code summarization, program classification, change analysis, and entity
1032 identification/recommendation.

1033 3.6.1 Code summarization

1034 Code summarization techniques attempt to provide a consolidated summary of the source code
1035 entity (typically a method). A variety of attempts has been made in this direction. The majority of
1036 the studies [94, 252, 285, 9, 443, 548, 198, 260, 516, 253, 549, 523, 565, 204, 268, 580, 188, 581]
1037 produces a summary for a small block (such as a method). This category also includes studies that
1038 summarize small code fragments [347], code folding within IDEs [510], commit message genera-
1039 tion [212, 295, 214, 213, 96, 526], and title generation for online posts from code [151].

1040 **Dataset preparation:** The majority of the studies [26, 94, 252, 285, 9, 198, 95, 260, 516, 511, 523,
1041 96, 581] in this category prepares pairs of code snippets and their corresponding natural language
1042 description. Specifically, Chen and Zhou [94] used more than 66 thousand pairs of C# code and
1043 natural language description where source code is tokenized using a modified version of the ANTLR
1044 parser. Ahmad et al. [9] conducted their experiments on a dataset containing Java and Python
1045 snippets; sequences of both the code and summary tokens are represented by a sequence of
1046 vectors. Hu et al. [198] and Li et al. [260] prepared a large dataset from 9,714 GitHub projects.
1047 Similarly, Wang et al. [516] mined code snippets and corresponding `javadoc` comments for their
1048 experiment. Chen et al. [95] created their dataset from 12 popular open-source Java libraries with
1049 more than 10 thousand stars. They considered method bodies as their inputs and method names
1050 along with method comments as prediction targets. Psarras et al. [392] prepared their dataset by
1051 using Weka, SystemML, DL4J, Mahout, Neuroph, and Spark as their subject systems. The authors
1052 retained names and types of methods, and local and class variables. Choi et al. [104] collected
1053 and refined more than 114 thousand pairs of methods and corresponding code annotations from
1054 100 open-source Java projects. Iyer et al. [204] mined StackOverflow and extracted title and code
1055 snippet from posts that contain exactly one code snippet. Similarly, Gao et al. [151] used a dump
1056 of StackOverflow dataset. They tokenized code snippets with respect to each programming lan-
1057 guage for pre-processing. The common steps in preprocessing identifiers include making them
1058 lower case, splitting the camel-cased and underline identifiers into sub-tokens, and normalizing
1059 the code with special tokens such as "VAR" and "NUMBER". Nazar et al. [347] used human anno-
1060 tators to summarize 127 code fragments retrieved from Eclipse and NetBeans official frequently
1061 asked questions. Yang et al. [546] built a dataset with over 300K pairs of method and comment
1062 to evaluate their approach. Chen et al. [96] used dataset provided by Hu et al. [198] and man-
1063 ually categorized comments into six intention categories for 20,000 code-comment pairs. Wang
1064 et al. [526] created a Python dataset that contains 128 thousand code-comment pairs. Zhou et al.
1065 [579] crawled over 6700 Java projects from Github to extract their methods and the corresponding
1066 Javadoc comments to create their dataset.

1067 Jiang [213] used 18 popular Java projects from GitHub to prepare a dataset with approximately
1068 50 thousand commits to generate commit messages automatically. Liu et al. [292] processed 56
1069 popular open-source projects and selected approximately 160K commits after filtering out the ir-
1070 relevant commits. Liu et al. [296] used RepoRepears to identify Java repositories to process. They

Journal Pre-proof

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. Simiarly, 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 (SBT). 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.,* SBT 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

1121 traversal algorithm to generate the code token sequence preserving the semantic, syntactic and
1122 structural information in the code snippet.
1123 The most common way of representing features in this category is to encode the features in the
1124 form of embeddings or feature vectors. Specifically, LeClair et al. [252] used embeddings layer for
1125 code, text, as well as for AST. Similarly, Choi et al. [104] transformed each of the tokenized source
1126 code into a vector of fixed length through an embedding layer. Wang et al. [516] extracted the
1127 functional keyword from the code and perform positional encoding. Yao et al. [548] used a code
1128 retrieval pre-trained model with natural language query and code snippet and annotated each
1129 code snippet with the help of a trained model. Ye et al. [549] utilized two separate embedding
1130 layers to convert input sequences, belonging to both text and code, into high-dimensional vectors.
1131 Furthermore, some authors encode source code models using various techniques. For instance,
1132 Chen et al. [95] represented every input code snippet as a series of AST paths where each path is
1133 seen as a sequence of embedding vectors associated with all the path nodes. LeClair et al. [253]
1134 used a single embedding layer for both the source code and AST node inputs to exploit a large over-
1135 lap in vocabulary. Wang et al. [523] prepared a large-scale corpus of training data where each code
1136 sample is represented by three sequences—code (in text form), AST, and CFG. These sequences are
1137 encoded into vector forms using work2vec. Studies also explored other mechanisms to encode
1138 features. For example, Liu et al. [295] extracted commit *diffs* and represented them as bag of
1139 words. The corresponding model ignores grammar and word order, but keeps term frequencies.
1140 The vector obtained from the model is referred to as *diff vector*. Zhang et al. [565] parsed code
1141 snippets into ASTs and calculated their similarity using ASTs. Allamanis et al. [26] and Ahmad et al.
1142 [9] employed attention-based mechanism to encode tokens. Li et al. [268] used GloVe, a word em-
1143 bedding technique, to obtain the vector representation of the context; the study included method
1144 callers and callee as well as other methods in the enclosing class as the context for a method. Sim-
1145 ilarly, Li et al. [262] calculated edit vectors based on the lexical and semantic differences between
1146 input code and the similar code.

1147 **ML model training:** The ML techniques used by the studies in this category can be divided into the
1148 following four categories.

1149 *Encoder-decoder models:* The majority of the studies used attention-based *Encoder-Decoder* models
1150 to generate code summaries for code snippets. We further classify the studies in three categories
1151 based on their ML implementation.
1152 A large portion of the studies use *sequence-to-sequence based approaches*. For instance, Gao et al.
1153 [151] proposed an end-to-end sequence-to-sequence system enhanced with an attention mecha-
1154 nism to perform better content selection. A code snippet is transformed by a source-code encoder
1155 into a vector representation; the decoder reads the code embeddings to generate the target ques-
1156 tion titles. Jiang et al. [212] trained an NTM algorithm to ``translate'' from diffs to commit messages.
1157 Iyer et al. [204] used an attention-based neural network to model the conditional distribution of a
1158 natural language summary. Their approach uses an LSTM model guided by attention on the source
1159 code snippet to generate a summary of one word at a time. Choi et al. [104] transformed input
1160 source code into a context vector by detecting local structural features with CNNs. Also, attention
1161 mechanism is used with encoder CNNs to identify interesting locations within the source code. Sim-
1162 ilarly, Jiang [213], Haque et al. [188], Liu et al. [296], Lu et al. [301], Takahashi et al. [478] employed
1163 LSTM-based *Encoder-Decoder* model to generate summaries. Their last module decoder generates
1164 source code summary. Ahmad et al. [9] proposed to use Transformer to generate a natural lan-
1165 guage summary given a piece of source code. For both encoder and decoder, the Transformer
1166 consists of stacked multi-head attention and parameterized linear transformation layers. LeClair
1167 et al. [252] used attention mechanism to not only attend words in the output summary to words
1168 in the code word representation but also to attend the summary words to parts of the AST. The
1169 concatenated context vector is used to predict the summary of one word at a time. Xie et al. [536]
1170 designed a novel multi-task learning (MLT) approach for code summarization through mining the

Journal Pre-proof

1171 relationship between method-code summaries and method names. Li et al. [268] used RNN-based
1172 encoder-decoder model to generate a code representation of a method and check whether the cur-
1173 rent method name is inconsistent with the predicted name based on the semantic representation.
1174 Haque et al. [189] compared five seq2seq-like approaches (*attendgru*, *ast-attendgru*, *ast-attendgru-*
1175 *fc*, *graph2seq*, and *code2seq*) to explore the role of action word identification in code summarization.
1176 Wang et al. [515] proposed a new approach, named CoRec, to translate git diffs, using attentional
1177 Encoder-Decoder model, that include both code changes and non-code changes into commit mes-
1178 sages. Zhou et al. [578] presented ContextCC that uses a Seq2Seq Neural Network model with an
1179 attention mechanism to generate comments for Java methods.

1180 Other studies relied on *tree-based approaches*. For example, Yang et al. [546] developed a multi-
1181 modal transformer-based code summarization approach for smart contracts. Bansal et al. [51]
1182 introduced a project-level encoder DL model for code summarization. Chen et al. [95], Hu et al.
1183 [198] employed LSTM-based *Encoder-Decoder* model to generate summaries.

1184 Rest of the studies employed *retrieval-based techniques*. Zhang et al. [565] proposed *Rencos* in
1185 which they first trained an attentional *Encoder-Decoder* model to obtain an encoder for all code
1186 samples and a decoder for generating natural language summaries. Second, the approach re-
1187 trieves the most similar code snippets from the training set for each input code snippet. Rencos
1188 uses the trained model to encode the input and retrieves two code snippets as context vectors. It
1189 then decodes them simultaneously to adjust the conditional probability of the next word using the
1190 similarity values from the retrieved two code snippets. Li et al. [262] implemented their retrieve-
1191 and-edit approach by using LSTM-based models.

1192 *Extended encoder-decoder models:* Many studies extended the traditional *Encoder-Decoder* mech-
1193 anism in a variety of ways. Among them, *sequence-to-sequence based approaches* include an ap-
1194 proach proposed by Liu et al. [285]; they introduced *CallNN* that utilizes call dependency informa-
1195 tion. They employed two encoders, one for the source code and another for the call dependency
1196 sequence. The generated output from the two encoders are integrated and used in a decoder
1197 for the target natural language summarization. Wang et al. [516] implemented a three step ap-
1198 proach. In the first step, functional reinforcer extracts the most critical function-indicated tokens
1199 from source code which are fed into the second module code encoder along with source code. The
1200 output of the code encoder is given to a decoder that generates the target sequence by sequen-
1201 tially predicting the probability of words one by one. LeClair et al. [253] proposed to use GNN-based
1202 encoder to encode AST of each method and RNN-based encoder to model the method as a sequence.
1203 They used an attention mechanism to learn important tokens in the code and corresponding AST.
1204 Finally, the decoder generates a sequence of tokens based on the encoder output. Zhou et al.
1205 [580] used two encoders, semantic and structural, to generate summaries for Java methods. Their
1206 method combined text features with structure information of code snippets to train encoders with
1207 multiple graph attention layers.

1208 Li et al. [260] presented a *tree-based approach* Hybrid-DeepCon model containing two encoders
1209 for code and AST along with a decoder to generate sequences of natural language annotations.
1210 Shido et al. [443] extended Tree-LSTM and proposed Multi-way Tree-LSTM as their encoder. The ra-
1211 tional behind the extension is that the proposed approach not only can handle an arbitrary number
1212 of ordered children, but also factor-in interactions among children. Zhou et al. [581] trained two
1213 separate *Encoder-Decoder* models, one for source code sequence and another for AST via adversar-
1214 ial training, where each model is guided by a well-designed discriminator that learns to evaluate its
1215 outputs. Lin et al. [277] used a transformer to generate high-quality code summaries. The learned
1216 syntax encoding is combined with code encoding, and fed into the transformer.

1217 Rest of the approaches adopted *retrieval-based approaches*. Ye et al. [549] employed dual learn-
1218 ing mechanism by using Bi-LSTM. In one direction, the model is trained for code summarization task
1219 that takes code sequence as input and summarized into a sequence of text. On the other hand,
1220 the code generation task takes the text sequence and generate code sequence. They reused the

Journal Pre-proof

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 *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 *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 *Actor* network that generates comments for input code snippets. The *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 *Naive Bayes* to classify the diff files into the verb groups. For automated code folding, Viuginov and Filchenkov [510] used *Random Forest* and *Decision Tree* to classify whether a code block needs to be folded. Similarly, Nazar et al. [347] used *Support Vector Machine* and *Naive Bayes* classifiers to generate summaries from the extracted features. Chen et al. [96] compared six 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 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 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 `[ci skip]' or `[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

Journal Pre-proof

1270 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 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 Bezzi [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. Ghadhab 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 Bezzi [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-

Journal Pre-proof

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 *Naive Bayes* ML 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 the 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.

Journal Pre-proof

1367 Other studies used source code or code metadata as features. Abuhamad et al. [5] extracted
1368 code authorship attributes from samples of code. Malik et al. [322] used function names, formal
1369 parameters, and corresponding comments as features. Ali et al. [18] extracted source code en-
1370 tity names, such as class, method, and variable names. Bavota et al. [56] retrieved 618 features
1371 from six open-source Java systems to apply *Latent Dirichlet Allocation*-based feature location tech-
1372 nique. Similarly, De Lucia et al. [119] extracted class name, signature of methods, and attribute
1373 names from Java source code. They applied *Latent Dirichlet Allocation* to label source code arti-
1374 facts. Gopalakrishnan et al. [159] processed tactics in the form of a set of textual descriptions and
1375 produced a set of weighted indicator terms. Mahmoud and Bradshaw [312] extracted code term
1376 co-occurrence, pair-wise term similarity, and clusters of terms features and applied their apporach
1377 Semantic Topic Models (STM) on them.

1378 In addition, Uchiyama et al. [502], Chaturvedi et al. [87], Dwivedi et al. [133], Alhusain et al. [17]
1379 used several source-code metrics as features to detect design patterns in software programs.

1380 **ML model training:** The majority of studies in this category use RNN-based DL models. In particular,
1381 Huang et al. [200] and Hellendoorn et al. [192] used bidirectional RNN models. Similarly, Abuhamad
1382 et al. [5] and Malik et al. [322] also employed RNN models to identify code authorship and function
1383 signatures respectively. Zhang et al. [567] created a bug-localization tool, KGBugLocator utilizing
1384 knowledge graph embeddings and bi-directional attention models. Xu et al. [539] employed the
1385 GRU-based *Encoder-Decoder* model for method name prediction. Uchiyama et al. [502] used a hier-
1386 archical neural network as their classifier. Allamanis et al. [24] utilized neural language models for
1387 predicting method and class names.

1388 Other studies used traditional ML techniques. Specifically, Chaturvedi et al. [87] compared four
1389 ML techniques (*Linear Regression*, *Polynomial Regression*, *support vector regression*, and *neural net-*
1390 *work*). Dwivedi et al. [133] used *Decision Tree* and Zanoni et al. [559] trained *Naive Bayes*, *Decision*
1391 *Tree*, *Random Forest*, and *Support Vector Machine* to detect design patterns using ML. Ali et al. [18]
1392 employed *Latent Dirichlet Allocation* to distinguish domain-level terms from implementation-level
1393 terms. Gopalakrishnan et al. [159] discovered latent topics using *Latent Dirichlet Allocation* in the
1394 large-scale corpus. The study used *Decision Tree*, *Random Forest*, and *Linear Regression* as classifiers
1395 to compute the likelihood that a given source file is associated with a given tactic.

1396 ### 3.7 Code review

1397 Code Review is the process of systematically check the code written by a developer performed by
1398 one or more different developers. A very small set of studies explore the role of ML in the process
1399 of code review that we present in this section.

1400 **Dataset preparation:** Lal and Pahwa [245] labeled check-in code samples as *clean* and *buggy*. On
1401 code samples, they carried out extensive pre-processing such as normalization and label encoding.
1402 Aiming to automate code review process, Tufano et al. [493] trained two DL architectures one for
1403 both contributor and for reviewer. They mined Gerrit and GitHub to prepare their dataset from
1404 $8,904$ projects. Furthermore, Thongtanunam et al. [482] proposed AutoTransform to better handle
1405 new tokens using Byte-Pair Encoding (BPE) approach. They leveraged the dataset proposed by
1406 Tufano et al. [493] consisting 630,858 changed methods to train a Transformer-based NMT model.

1407 **Feature extraction:** Lal and Pahwa [245] used TF-IDF to convert the code samples into vectors after
1408 applying extensive pre-processing. Tufano et al. [493] used n-grams extracted from each commit
1409 to train their classifiers.

1410 **ML model training:** Lal and Pahwa [245] used a *Naive Bayes* model to classify samples into buggy
1411 or clean. Tufano et al. [493] trained two DL architectures one for both contributor and for reviewer.
1412 The authors use n-grams extracted from each commit and implement their classifiers using *Deci-*
1413 *sion Tree*, *Naive Bayes*, and *Random Forest*. In their revised work [494], the authors used Text-To-Text
1414 Transfer Transformer (T5) model and shown significant improvements in DL code review models.

Journal Pre-proof

1415 ### 3.8 Code search

1416 Code search is an activity of searching a code snippet based on individual's need typically in Q&A
1417 sites such as StackOverflow [413, 450, 512]. The studies in this category define the following coarse-
1418 grained steps. In the first step, the techniques prepare a training set by collecting source code and
1419 often corresponding description or query. A feature extraction step then identifies and extracts
1420 relevant features from the input code and text. Next, these features are fed into ML models for
1421 training which is later used to execute test queries.

1422 **Dataset preparation:** Shuai et al. [450] utilized commented code as input. Wan et al. [512] used
1423 source code in the the form of tokens, AST, and CFG. Sachdev et al. [413] employed a simple tok-
1424 enizer to extract all tokens from source code by removing non–alphanumeric tokens. Ling et al.
1425 [282] mined software projects from GitHub for the training of their approach. Jiang et al. [208]
1426 used existing McGill corpus and Android corpus.

1427 **Feature extraction:** Code search studies typically use embeddings representing the input code.
1428 Shuai et al. [450] performed embeddings on code, where source code elements (method name,
1429 API sequence, and tokens) are processed separately. They generated embeddings for code com-
1430 ments independently. Wan et al. [512] employed a multi-modal code representation, where they
1431 learnt the representation of each modality via LSTM, Tree-LSTM and GGNN, respectively. Sachdev et al.
1432 [413] identified words from source code and transformed the extracted tokens into a natural lan-
1433 guage documents. Similarly, Ling et al. [282] used an unsupervised word embedding technique
1434 to construct a matching matrix to represent lexical similarities in software projects and used an
1435 RNN model to capture latent syntactic patterns for adaptive code search. Jiang et al. [208] used a
1436 fragment parser to parse a tutorial fragment in four steps (API discovery, pronoun and variable
1437 resolution, sentence identification, and sentence type identification).

1438 **ML model training:** Shuai et al. [450] used a CNN-based ML model named CARLCS-CNN. The cor-
1439 responding model learns interdependent representations for embedded code and query by a
1440 co-attention mechanism. Based on the embedded code and query, the co-attention mechanism
1441 learns a correlation matrix and leverages row/column-wise max-pooling on the matrix. Wan et al.
1442 [512] employed a multi-modal attention fusion. The model learns representations of different
1443 modality and assigns weights using an attention layer. Next, the attention vectors are fused into
1444 a single vector. Sachdev et al. [413] utilized word and documentation embeddings and performed
1445 code search using the learned embeddings. Similarly, Ling et al. [282] used an *autoencoder* network
1446 and a metric (believability) to measure the degree to which a sentence is approved or disapproved
1447 within a discussion in a issue-tracking system. Jiang et al. [208] used *Latent Dirichlet Allocation* to
1448 segregate all tutorial fragments into relevant clusters and identify relevant tutorial for an API.

1449 Once an ML model is trained, code search can be initiated using a query and a code snippet.
1450 Shuai et al. [450] used the given query and code sample to measure the semantic similarity using
1451 cosine similarity. Wan et al. [512] ranked all the code snippets by their similarities with the input
1452 query. Similarly, Sachdev et al. [413] were able to answer almost 43% of the collected StackOver-
1453 flow questions directly from code.

1454 ### 3.9 Refactoring

1455 Refactoring transformations are intended to improve code quality (specifically maintainability),
1456 while preserving the program behavior (functional requirements) from users' perspective [471].
1457 This section summarizes the studies that identify refactoring candidates or predict refactoring com-
1458 mits by analyzing source code and by applying ML techniques on code. A process pipeline typically
1459 adopted by the studies in this category can be viewed as a three step process. In the first step, the
1460 source code of the projects is used to prepare a dataset for training. Then, individual samples (*i.e.,*
1461 either a method, class, or a file) is processed to extract relevant features. The extracted features
1462 are then fed to an ML model for training. Once trained, the model is used to predict whether an

Journal Pre-proof

input sample is a candidate for refactoring or not.

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, LOC, 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 fragements 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.

Journal Pre-proof

Journal Pre-proof

1512 **ML model training:** Majority of the studies in this category utilized traditional ML techniques. Rodriguez et al. [407] proposed a method to identify web-service groups for refactoring using *K-means*, COBWEB, and expectation maximization. Kosker et al. [234] trained a *Naive Bayes*-based classifier to identify classes that need refactoring. Kumar and Sureka [239] used *Least Square-Support Vector Machine* (LS-SVM) along with SMOTE as classifier. They found that LS-SVM with *Radial Basis Function* (RBF) kernel gives the best results. Nyamawe et al. [354] recommended refactorings based on the history of requested features and applied refactorings. Their approach involves two classification tasks; first, a binary classification that suggests whether refactoring is needed or not and second, a multi-label classification that suggests the type of refactoring. The authors used *Linear Regression*, *Multinomial Naive Bayes* (MNB), *Support Vector Machine*, and *Random Forest* classifiers. Yue et al. [558] presented CREC—a learning-based approach that automatically extracts refactored and non-refactored clones groups from software repositories, and trains an *AdaBoost* model to recommend clones for refactoring. Kumar et al. [242] employed a set of ML models such as *Linear Regression*, *Naive Bayes*, *Bayes Network*, *Random Forest*, *AdaBoost*, and *Logit Boost* to develop a recommendation system to suggest the need of refactoring for a method. Amal et al. [37] proposed the use of ANN to generate a sequence of refactoring. Aribandi et al. [43] predicted the classes that are likely to be refactored in the future iterations. To achieve their aim, the authors used various variants of ANN, *Support Vector Machine*, as well as *Best-in-training based Ensemble* (BTE) and *Majority Voting Ensemble* (MVE) as ensemble techniques. Kurbatova et al. [244] proposed an approach to recommend move method refactoring based on a path-based presentation of code using *Support Vector Machine*. Similarly, Aniche et al. [40] used *Linear Regression*, *Naive Bayes*, *Support Vector Machine*, *Decision Tree*, *Random Forest*, and *Neural Network* to predict applicable refactoring operations. Sidhu et al. [451], Xu et al. [538], Wang and Godfrey [524] used DNN, *gradient boosting*, and *Decision Tree* respectively to identify refactoring candidate. Sagar et al. [414], Nyamawe et al. [355] employed various classifiers such as *Support Vector Machine*, *Linear Regression*, and *Random Forest* to predict commits with refactoring operations. 1537

1538 Bavota et al. [55] and Bavota et al. [56] applied *Latent Dirichlet Allocation* to identify move class and move method refactoring candidates respectively. They model the documents in a given corpus as a probabilistic mixture of latent topics and model the links between document pairs as a binary variable. 1541

1542 3.10 Vulnerability analysis

1543 The studies in this domain analyze source code to identify potential security vulnerabilities. In this section, we point out the state-of-the-art in software vulnerability detection using ML techniques. First, the studies prepare a dataset or identify an existing dataset for ML training. Next, the studies extract relevant features from the identified subject systems. Then, the features are fed into a ML model for training. The trained model is then used to predict vulnerabilities in the source code. 1547

1548 **Dataset preparation:** Authors used existing labeled datasets as well as created their own datasets to train ML models. Specifically, a set of studies [378, 337, 397, 412, 231, 61, 461, 280, 555, 467, 247, 370, 6, 556, 509, 228, 232, 570, 327, 130, 448, 131, 541, 54, 346, 527, 100, 269, 403, 48] used available labeled datasets for PHP, Java, C, C++, and Android applications to train vulnerability detection models. In other cases, Russell et al. [409] extended an existing dataset with millions of C and C++ functions and then labeled it based on the output of three static analyzers (*i.e.,* Clang, CppCheck, and Flawfinder). 1554

1555 Many studies [309, 19, 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 CFGs of approximately 10 thousand, both benign and vulnerable, Android applications from *AndroZoo* and *Android Malware* datasets; Ali Alatwi et al. [19] collected 5,063 Android applications where 1,000 of them were marked as be- 1560

Journal Pre-proof

1561 nign and the remaining as malware; Cui et al. [112] selected an open-source dataset comprised of
1562 1,179 Android applications that have 4,416 different version (of the 1,179 applications) and labeled
1563 the selected dataset by using the Androrisk tool; and Gupta et al. [172] used two Android applica-
1564 tions (Android-universal-image-loader and JHotDraw) which they have manually labeled based on
1565 the projects PMD reports (true if a vulnerability was reported in a PMD file and false otherwise). To
1566 create datasets of PHP projects, Medeiros et al. [331] collected 35 open-source PHP projects and in-
1567 tentionally injected 76 vulnerabilities in their dataset. Shar et al. [430] used *phpminer* to extract 15
1568 datasets that include SQL injections, cross-site scripting, remote code execution, and file inclusion
1569 vulnerabilities, and labeled only 20% of their dataset to point out the precision of their approach.
1570 Ndichu et al. [349] collected 5,024 JavaScript code snippets from D3M, JSUNPACK, and 100 top web-
1571 sites where the half of the code snippets were benign and the other half malicious. In other cases,
1572 authors [543, 397, 379] collected large number of commit messages and mapped them to known
1573 vulnerabilities by using Google's Play Store, National Vulnerability Database (NVD), Synx, Node Secu-
1574 rity Project, and so on, while in limited cases authors [383] manually label their dataset. Hou et al.
1575 [196], Moskovitch et al. [338] and Santos et al. [422] created their datasets by collecting web-page
1576 samples from StopBadWare and VxHeavens. Lin et al. [280] constructed a dataset and manually
1577 labeled 1,471 vulnerable functions and 1,320 vulnerable files from nine open-source applications,
1578 named Asterisk, FFmpag, HTTPD, LibPNG, LibTIFF, OpenSSL, Pidgin, VLC Player, and Xen. Lin et al.
1579 [278] have used more then 30,000 non-vulnerable functions and manually labeled 475 vulnerable
1580 functions for their experiments.

1581 **Feature extraction:** Authors used static source code metrics, CFGs, ASTs, source code tokens, and
1582 word embeddings as features.

1583 *Source code metrics:* A set of studies [331, 146, 36, 172, 107, 397, 112, 383, 403, 130, 232, 332, 6, 247,
1584 467] used more than 20 static source code metrics (such as *cyclomatic complexity*, *maximum depth*
1585 *of class in inheritance tree*, *number of statements,* and *number of blank lines*).

1586 *Data/control flow and AST:* Ma et al. [307], Kim et al. [231], Bilgin et al. [61], Kronjee et al. [238],
1587 Wang et al. [527], Du et al. [131], Medeiros et al. [332] used CFGs, ASTs, or data flow analysis as
1588 features. More specifically, Ma et al. [309] extracted the API calls from the CFGs of their dataset and
1589 collected information such as the usage of APIs (which APIs the application uses), the API frequencies
1590 (how many times the application uses APIs) and API sequence (the order the application uses APIs).
1591 Kim et al. [231] extracted ASTs and GFCs which they tokenized and fed into ML models, while Bilgin
1592 et al. [61] extracted ASTs and translated their representation of source code into a one-dimensional
1593 numerical array to fed them to a model. Kronjee et al. [238] used data-flow analysis to extract
1594 features, while Spreitzenbarth et al. [461] used static, dynamic analysis, and information collected
1595 from `ltrace` to collect features and train a linear vulnerability detection model. Lin et al. [278]
1596 created ASTs and from there they extracted code semantics as features.

1597 *Repository and file metrics:* Perl et al. [379] collected GitHub repository meta-data (*i.e., programming*
1598 *language, star count, fork count,* and *number of commits*) in addition to source code metrics. Other
1599 authors [378, 135] used file meta-data such as *files' creation and modification time, machine type, file*
1600 *size,* and *linker version*.

1601 *Code and Text tokens:* Chernis and Verma [102] used simple token features (*character count, char-*
1602 *acter diversity, entropy, maximum nesting depth, arrow count, ``if'' count, ``if'' complexity, ``while''*
1603 *count,* and *``for'' count*) and complex features (*character n-grams, word n-grams,* and *suffix trees*).
1604 Hou et al. [196] collected 10 features such as *length of the document, average length of word, word*
1605 *count, word count in a line,* and *number of NULL characters*. The remaining studies [409, 369, 338,
1606 422, 543, 412, 573, 430, 100, 346, 409, 327, 143, 570, 370, 48, 555, 280] tokenized parts of the source
1607 code or text-based information with various techniques such as the most frequent occurrences of
1608 operational codes, capture the meaning of critical tokens, or applied techniques to reduce the vo-
1609 cabulary size in order to retrieve the most important tokens. In some other cases, authors [269]

Journal Pre-proof

1610 used statistical techniques to reduce the feature space to reduce the number of code tokens.

1611 *Other features:* Ali Alatwi et al. [19], Ndichu et al. [349] and Milosevic et al. [337] extracted permission-
1612 related features. In other cases, authors [541] combined software metrics and N-grams as features
1613 to train models and others [448] created text-based images to extract features. Likewise, Sultana
1614 [466] extracted traceable patterns such as CompoundBox, Immutable, Implementor, Overrider,
1615 Sink, Stateless, FunctionObject, and LimitSel and used Understand tool to extract various software
1616 metrics. Wei et al. [531] extracted system calls and function call-related information to use as
1617 features, while Vishnu and Jevitha [509] extracted URL-based features like number of chars, dupli-
1618 cated characters, special characters, script tags, cookies, and re-directions. Padmanabhuni and
1619 Tan [362] extracted buffer usage patterns and defensive mechanisms statements constructs by
1620 analyzing files.

1621 **Model training:** To train models, the selected studies used a variety of traditional ML and DL algo-
1622 rithms.

1623 *Traditional ML techniques:* One set of studies [19, 349, 378, 409, 369, 338, 379, 430, 555, 467, 362,
1624 247, 6, 556, 466, 509, 531, 130, 143, 332, 131, 346, 527, 100, 403] used traditional ML algorithms
1625 such as *Naive Bayes*, *Decision Tree*, *Support Vector Machine*, *Linear Regression*, *Decision Tree*, and *Ran-*
1626 *dom Forest* to train their models. Specifically, Ali Alatwi et al. [19], Russell et al. [409], Perl et al. [379]
1627 selected *Support Vector Machine* because it is not affected by over-fitting when having very high di-
1628 mensional variable spaces. Along the similar lines, Ndichu et al. [349] used *Support Vector Machine*
1629 to train their model with linear kernel. Pereira et al. [378] used *Decision Tree*, *Linear Regression*,
1630 and *Lasso* to train their models, while [6] found that *Random Forest* is the best model for predicting
1631 cross-project vulnerabilities. Compared to the above studies, Shar et al. [430] used both supervised
1632 (*i.e., Linear Regression* and *Random Forest*) and semi-supervised (*i.e., Co-trained Random Forest*) al-
1633 gorithms to train their models since most of that datasets were not labeled. Yosifova et al. [555]
1634 used text-based features to train *Naive Bayes*, *Support Vector Machine*, and *Random Forest* models.
1635 Du et al. [130] created the LEOPARD framework that does not require prior knowledge about known
1636 vulnerabilities and used *Random Forest*, *Naive Bayes*, *Support Vector Machine*, and *Decision Tree* to
1637 point them out.

1638 Other studies [331, 146, 383, 238, 36, 172, 107, 337, 102, 196, 422, 397, 112] used up to 32
1639 different ML algorithms to train models and compared their performance. Specifically, Medeiros
1640 et al. [331] experimented with multiple variants of *Decision Tree*, *Random Forest*, *Naive Bayes*, *K*
1641 *Nearest Neighbors*, *Linear Regression*, *Multilayer Perceptron*, and *Support Vector Machine* models and
1642 identified *Support Vector Machine* as the best performing classifier for their experiment. Likewise,
1643 Milosevic et al. [337] and Rahman et al. [397] employed multiple ML algorithms, respectively, and
1644 found that *Support Vector Machine* offers the highest accuracy rate for training vulnerability detec-
1645 tors. In contrast to the above studies, Ferenc et al. [146] showed that *K Nearest Neighbors* offers
1646 the best performance for their dataset after experimenting with DNN, *K Nearest Neighbors*, *Support*
1647 *Vector Machine*, *Linear Regression*, *Decision Tree*, *Random Forest*, and *Naive Bayes*. In order to find
1648 out which is the best model for the SWAN tool, Piskachev et al. [383] evaluated the *Support Vector*
1649 *Machine*, *Naive Bayes*, *Bayes Network*, *Decision Tree*, *Stump*, and *Ripper*. Their results pointed out the
1650 *Support Vector Machine* as the best performing model to detect vulnerabilities. Similarly, Kronjee
1651 et al. [238], Cui et al. [112], and Gupta et al. [172] compared different ML algorithms and found
1652 *Decision Tree* and *Random Forest* as the best performing algorithms.

1653 *DL techniques:* A large number of studies [543, 412, 231, 280, 48, 232, 327, 278, 448, 54] used DL
1654 methods such as CNN, RNN, and ANN to train models. In more details, Yang et al. [543] utilized the BP-
1655 ANN algorithm to train vulnerability detectors. For the project *Achilles*, Saccente et al. [412] used an
1656 array of LSTM models to train on data containing Java code snippets for a specific set of vulnerability
1657 types. In another study, Kim et al. [231] suggested a DL framework that makes use of RNN models
1658 to train vulnerability detectors. Specifically, the authors framework first feeds the code embed-

1659 dings into a BI-LSTM model to capture the feature semantics, then an attention layer is used to get
1660 the vector weights, and, finally, passed into a dense layer to output if a code is safe or vulnerable.
1661 Compared to the studies that examined traditional ML or DL algorithms, Zheng et al. [573] exam-
1662 ined both of them. They used *Random Forest*, *K Nearest Neighbors*, *Support Vector Machine*, *Linear*
1663 *Regression* among the traditional ML algorithms along with BI-LSTM, GRU, and CNN. There results indi-
1664 cate BI-LSTM as the best performing model. Lin et al. [280] developed a benchmarking framework
1665 that can use BI-LSTM, LSTM, BI-GRU, GRU, DNN and Text-CNN, but can be extended to use more deep
1666 learning models. Kim et al. [232] generating graphical semantics that reflect on code semantic fea-
1667 tures and use them for Graph Convolutional Network to automatically identify and learn semantic
1668 and extract features for vulnerability detection, while Shiqi et al. [448] created textual images and
1669 fed them to Deep Belief Networks to classify malware.

1670 ### 3.11 Summary

1671 In this section, we briefly summarize the usage of ML in a software engineering task involving source
1672 code analysis. Figure 7 presents an overview of the pipeline that is typically used in a software
1673 engineering task that uses ML.

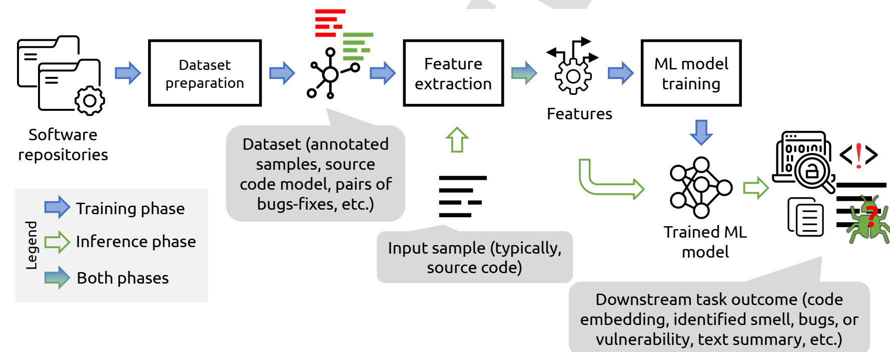

Figure 7. Overview of the software engineering task implementation pipeline using ML

1674 **Dataset preparation:** Preparing a dataset is the first major activity in the pipeline. The activity
1675 starts with identifying the source of required data, typically source code repositories. The activ-
1676 ity involves selecting and downloading the required repositories, collecting supplementary data
1677 (such as GitHub issues), create individual samples sometimes by combining information, and an-
1678 notate samples. Depending upon the specific software engineering task at hand, these steps are
1679 customized and extended.

1680 The outcome of this activity is a dataset. Depending upon the context, the dataset may contain
1681 information such as annotated code samples, source code model (*e.g.,* AST), and pairs of buggy
1682 code and fixed code.

1683 **Feature extraction:** Performance of a ML model depends significantly on the provided kind and
1684 quality of features. Various techniques are applied on the prepared dataset to extract the required
1685 features that help the ML model perform well for the given task. Features may take variety of form
1686 and format; for source code analysis applications, typical features include source code metrics,
1687 source code tokens, their properties, and representation, changes in the code (code *diff*), vector
1688 representation of code and text, dependency graph, and vector representation of AST, CFG, or AST
1689 diff. Obviously, selection of the specific features depends on the downstream task.

1690 **ML model training:** Selecting a ML model for a given task depends on many factors such as the
1691 nature of the problem, the properties of training and input samples, and the expected output.

Journal Pre-proof

1692 Below, we provide an analysis of employed ML models based on these factors.

- 1693 One of the factors that influence the choice of ML models is the chosen features and their 1694 properties. Studies in the *quality assessment* category majorly relied on token-based features 1695 and code quality metrics. Such features allowed studies in this categories to use traditional 1696 ML models. Some authors applied DL models such as DNN when higher-granularity constructs 1697 such as CFG and DFG are used as features.
- 1698 Similarly, the majority of the studies in *testing* category relied on code quality metrics. There- 1699 fore, they have fixed size, fixed meaning (for each column) vectors to feed to a ML model. 1700 With such inputs, traditional ML approaches, such as *Random Forest* and *Support Vector Ma-* 1701 *chine*, work well. Other studies used a variation of AST or AST of the changes to generate the 1702 embeddings. DL models including DNN and RNN-based models are used to first train a model 1703 for embeddings. A typical ML classifier use the embeddings to classify samples in buggy or 1704 benign.
- 1705 Typical output of a *code representation* study is embeddings representing code in the vec- 1706 tor form. The semantics of the produced embeddings significantly depend on the selected 1707 features. Studies in this domain identify this aspect and, hence, they are swiftly focused to 1708 extract features that capture the relevant semantics; for example, path-based features en- 1709 code the order among the tokens. The chosen ML model plays another important role to 1710 generate effective embeddings. Given the success of RNN with text processing tasks, due to 1711 its capability to identify a sequence or pattern, RNN-based models dominate this category.
- 1712 *Program repair* is typically a sequence to sequence transformation *i.e.,* a sequence of buggy 1713 code is the input and a sequence of fixed code is the output. Given the nature of the problem, 1714 it is not surprising to observe that the majority of the studies in this category used Encoder- 1715 Decoder-based models. RNN are considered a popular choice to realize Encoder-Decoder 1716 models due to its capability to remember long sequences.

1717 4. Datasets and Tools

1718 For RO3, this section provides a consolidated summary of available datasets and tools that are 1719 used by the studies considered in the survey. We carefully examined each selected study and 1720 noted the resources (*i.e.,* datasets and tools). We define the following criteria to include a resource 1721 in our catalog.

- 1722 The referenced resource must have been used by at least one primary study.
- 1723 The referenced resource must be publicly available at the time of writing this article (Dec 1724 2022).
- 1725 The resource provides bare-minimum usage instructions to build and execute (wherever ap- 1726 plicable) and to use the artifact.
- 1727 The resource is useful either by providing an implementation of a ML technique, helping the 1728 user to generate information/data which is further used by a ML technique, or by providing a 1729 processed dataset that can be directly employed in a ML study.

1730 Table 6 lists all the tools that we found in this exploration. Each resource is listed with it's 1731 category, name and link to access the resource, number of citations (as of Dec 2022), and the time 1732 when it was first introduced along with the time when the resource was last updated. We collected 1733 the metadata about the resources manually by searching the digital libraries, repositories, and 1734 authors' websites. The cases where we could not find the required information, are marked as 1735 ``–''. We also provide a short description of the resource.

Table 6. A list of tools useful for applying machine learning to source code

Journal Pre-proof

Category	Name	#Citation	Introd.	Updated	Description
Code Representation	ncc [57]	234	Dec 2018	Aug 2021	Learns representations of code semantics
	Code2vec [32]	487	Jan 2019	Feb 2022	Generates distributed representation of code
	Code2seq [31]	536	May 2019	Jul 2022	Generates sequences from structured representation of code
	Vector representation for coding style [235]	3	Sep 2020	Jul 2022	Implements vector representation of individual coding style
	CC2Vec [194]	69	Oct 2020	–	Implements distributed representation of code changes
	Autoen-CODE [490]	75	–	–	Encodes source code fragments into vector representations
	Graph-based code modeling [28]	544	May 2018	May 2021	Generates code modeling with graphs
	Vocabulary learning on code [115]	34	Jan 2019	–	Generates an augmented AST from Java source code
	User2code2vec [44]	29	Mar 2019	May 2019	Generates embeddings for developers based on distributed representation of code
Code Search	Deep Code Search [168]	472	May 2018	May 2022	Searches code by using code embeddings
	FRAPT [208]	43	Jul 2017	–	Searches relevant tutorial fragments for APIs
	Obfuscated-code2vec [108]	23	Oct 2022	–	Embeds Java Classes with Code2vec
	DeepTyper [192]	87	Oct 2018	Feb 2020	Annotates types for JavaScript and TypeScript
	CallNN [285]	9	Oct 2019	–	Implements a code summarization approach by using call dependencies
	Neural-CodeSum [9]	277	May 2020	Oct 2021	Implements a code summarization method by using transformers
	Summarization_tf [443]	30	Jul 2019	–	Summarizes code with Extended Tree-LSTM
	CoaCor [548]	36	Jul 2019	May 2020	Explores the role of rich annotation for code retrieval

1736

Journal Pre-proof

Program Comprehension	DeepCom [260]	102	Nov 2020	May 2021	Generates code comments
	Rencos [565]	79	Oct 2020	–	Generates code summary by using both neural and retrieval-based techniques
	CODES [371]	121	Jul 2012	Jul 2016	Extracts method description from StackOverflow discussions
	CFS	–	–	–	Summarizes code fragments using SVM and NB
	TASSAL	–	–	–	Summarizes code using autofolding
	ChangeScribe [109]	180	Dec 2014	Dec 2015	Generates commit messages
	CodeInsight [399]	59	Nov 2015	May 2019	Recommends insightful comments for source code
	CodeNN [204]	681	Aug 2016	May 2017	Summarizes code using neural attention model
	Code2Que [151]	25	Jul 2020	Aug 2021	Suggests improvements in question titles from mined code in Stack-Overflow
	BI-TBCNN [72]	34	Mar 2019	May 2019	Implements a bi-TBCNN model to classify algorithms
	DeepSim [571]	139	Oct 2018	–	Implements a DL approach to measure code functional similarity
	FCDetector [142]	48	Jul 2020	–	Proposes a fine-grained granularity of source code for functionality identification
	LASCAD [35]	12	Aug 2018	–	Categorizes software into relevant categories
	FunCom[252]	46	May 2019	–	Summarizes code
Quality Assessment	SonarQube	–	–	–	Analyzes code quality
	SVF [464]	317	Mar 2016	Jul 2022	Enables inter-procedural dependency analysis for LLVM-based languages
	Designite [436]	101	Mar 2016	Jul 2023	Detects code smells and computes quality metrics in Java and C# code

1737

Journal Pre-proof

	CloneCognition [339]	10	Nov 2018	May 2019	Proposes a ML framework to validate code clones
	SMAD [52]	25	Mar 2020	Feb 2021	Implements smell detection (God class and Feature envy) using ML
	Checkstyle	–	–	–	Checks for coding convention in Java code
	FindBugs	–	–	–	Implements a static analysis tool for Java
	PMD	–	–	–	Finds common programming flaws in Java and six other languages
	py-ccflex [356]	12	Mar 2017	Oct 2020	Mimics code metrics by using ML
	Deep learning smells [437]	27	Jul 2021	Nov 2020	Implements DL (CNN, RNN, and autoencoder-based models) to identify four smells
	CREC [558]	26	Nov 2018	–	Recommends clones for refactoring
	ML for software refactoring [40]	31	Sep 2020	–	Recommends refactoring by using ML
	DTLDP [90]	28	Aug 2019	–	Implements a deep transfer learning framework
	BugDetection [266]	66	Oct 2019	May 2021	Trains models for defect prediction
	DeepBugs [387]	210	Nov 2018	May 2021	Implements a framework for learning name-based bug detectors
Program Synthesis	CoCoNuT [305]	97	Jul 2020	Sep 2021	Repairs Java programs
	DeepFix [177]	498	Feb 2017	Dec 2017	Fixes common C errors
	TRANX [552]	187	Oct 2018	–	Translates natural language text to formal meaning representations
	TreeGen	83	Nov 2019	–	Generates code
Testing	AppFlow [197]	47	Oct 2018	–	Automates UI tests generation
	DeepFuzz [293]	72	Jul 2019	Mar 2020	Grammar fuzzer that generates C programs
	Agilika [505]	7	Aug 2020	Mar 2022	Generates tests from execution traces

1738

Journal Pre-proof

	TestDescriber	–	–	–	Implements test case summary generator and evaluator
	Randoop	–	–	Jul 2022	Generates tests automatic for Java code
Vulnerability Analysis	WAP [330]	9	Oct 2013	Nov 2015	Detects and corrects input validation vulnerabilities
	SWAN[383]	8	Oct 2019	May 2022	Identifies vulnerabilities
	VCCFinder [379]	174	Oct 2015	May 2017	Finds potentially dangerous code in repositories
General	BERT [123]	76,767	Oct 2018	Mar 2020	NLP pre-trained models
	BC3 Annotation Framework	–	–	–	Annotates emails/conversations easily
	JGibLDA	–	–	–	Implements Latent Dirichlet Allocation
	Stanford NLP Parser	–	–	–	A statistical NLP parser
	srcML	–	–	May 2022	Generates XML representation of sourcecode
	CallGraph	–	Oct 2017	Oct 2018	Generates static and dynamic call graphs for Java code
	ML for programming	–	–	–	Offers various tools such as JSNice, Nice2Predict, and DEBIN

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

Category	Name	#Citation	Introd.	Updated	Description
Code Representation	Code2seq [32]	418	Jan 2019	Feb 2022	Sequences generated from structured representation of code
	GHTorrent [163]	728	Oct 2013	Sep 2020	Meta-data from GitHub repositories
Code Completion	Neural Code Completion	148	Nov 2017	Sep 2019	Dataset and code for code completion with neural attention and pointer networks

Journal Pre-proof

Program Synthesis	CoNaLa corpus [553]	201	Dec 2018	Oct 2021	Python snippets and corresponding natural language description
	IntroClass [250]	299	Jul 2015	Feb 2016	Program repair dataset of C programs
	Code contest[270]	84	Dec 2022	–	Code generation dataset for AlphaCode
Program Comprehension	Program comprehension dataset [462]	61	May 2018	Aug 2021	Contains code for a program comprehension user survey
	CommitGen [212]	116	–	–	Commit messages and the diffs from 1,006 Java projects
	StaQC [547]	80	Nov 2019	Aug 2021	148K Python and 120K SQL question-code pairs from StackOverflow
	TL-CodeSum [199]	241	Feb 2019	Sep 2020	Dataset for code summarization
	DeepCom [198]	–	May 2018	–	Dataset for code completion
Quality Assessment	src-d datasets	–	–	–	Various labeled datasets (commit messages, duplicates, DockerHub, and Nuget)
	BigCloneBench [472]	272	Dec 2014	Mar 2021	Known clones in the IJaDataset source repository
	Multi-label smells [169]	28	May 2020	–	A dataset of 445 instances of two code smells and 82 metrics
	Deep learning smells [437]	27	Jul 2021	Nov 2020	A dataset of four smells in tokenized form from 1,072 C# and 100 Java repositories
	ML for software refactoring [40]	31	Nov 2019	–	Dataset for applying ML to recommend refactoring
	QScored [431]	11	Aug 2021	–	Code smell and metrics dataset for more than 86 thousand open-source repositories
	Landfill [363]	34	May 2015	–	Code smell dataset with public evaluation
	KeepItSimple [139]	16	Jul 2018	–	A dataset of linguistic antipatterns of 1,753 instances of source code elements

1745

Journal Pre-proof

	Code smell dataset [110]	8	Sept 2018	–	A dataset of four code smells
	Defects4J [218]	858	Jul 2014	Jul 2022	Java reproducible bugs
	PROMISE [424]	434	–	Jan 2021	Various datasets including defect prediction and cost estimation
	BugDetection [266]	59	Oct 2019	May 2021	A bug prediction dataset containing 4.973M methods belonging to 92 different Java project versions
	DeepBugs [387]	155	Oct 2018	Apr 2021	A JavaScript code corpus with 150K code snippets
	DTLDP [90]	28	Oct 2020	–	Dataset for deep transfer learning for defect prediction
Testing	DAMT [345]	15	Aug 2019	Dec 2019	Metamorphic testing dataset
Vulnerability Analysis	WPScan	–	–	–	a PHP dataset for WordPress plugin vulnerabilities
	Genome [577]	2,898	Jul 2012	Dec 2015	1,200 malware samples covering the majority of existing malware families
	Juliet [63]	147	–	–	81K synthetic C/C++ and Java programs with known flaws
	AndroZoo [29]	–	–	–	15.7M APKS from Google's Play Store
	TRL [279]	108	Apr 2018	Jan 2019	Vulnerabilities in six C programs
	Draper VDISC [410]	479	Jul 2018	Nov 2018	1.27 million functions mined from C and C++ applications
	SAMATE [62]	–	–	–	A set of known security flaws from NIST for C, C++, and Java programs
	JSVulner [146]	3	–	–	JavaScript Vulnerability Analysis dataset
	SWAN [383]	8	Jul 2019	Jul 2022	A Vulnerability Analysis collection of 12 Java applications
	Project-KB [384]	49	Aug 2019	–	A Manually-Curated dataset of fixes to vulnerabilities of open-source software

1746

Journal Pre-proof

General	GitHub Java Corpus [22]	411	–	–	A large collection of Java repositories
	150k Python dataset [401]	89	–	–	Contains parsed AST for 150K Python files
	UCI source code dataset [298]	38	Apr 2010	Nov 2013	Various large scale source code analysis datasets

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-

[6] https://datasetsearch.research.google.com/
[7] https://zenodo.org/
[8] https://www.kaggle.com/datasets

1786 ported that the replicated results were lower by up to 90% compared to what was re-
1787 ported in the original study.
1788 – *Handling of data imbalance:* It is very common to have imbalanced datasets in software
1789 engineering applications. Authors use techniques such as under-sampling and over-
1790 sampling to overcome the challenge for training. However, test datasets must retain
1791 the original sample ratio as found in the real world [127]; carrying out a performance
1792 evaluation based on a balanced dataset is flawed. Obviously, the model will perform
1793 significantly inferior when it is put at work in a real-world context. We noted many stud-
1794 ies [8, 360, 169, 149, 148, 481, 114] that used balanced samples and often did not provide
1795 the size and ratio of the training and testing dataset. Such improper handling of data
1796 imbalance contributes to poor reproducibility.

1797 *Mitigation:* The importance of reproducibility and replicability has been emphasized and un-
1798 derstood by the software engineering community [286]. It has lead to a concrete artifact
1799 evaluation mechanism adopted by leading software engineering conferences. For example,
1800 FSE artifact evaluation divides artifacts into five categories—*functional, reusable, available, re-*
1801 *sults reproduced,* and *results replicated*.[9] Such thorough evaluation encouraging software en-
1802 gineering authors to produce high-quality documentation along with easily replicate experi-
1803 ment results using their developed artifacts. In addition, efforts (such as model engineering
1804 process [50]) are being made to support ML research reproducible and replicable. Finally,
1805 identifying practices (such as assumptions related to hardware or dependencies) that may
1806 hinder reproducibility improve reproducibility.

1807 • ***Maturity in ML development:*** Development of ML systems are inherently different from tra-
1808 ditional software development [513]. Phases of ML development are very exploratory in na-
1809 ture and highly domain and problem dependent [513]. Identifying the most appropriate ML
1810 model, their appropriate parameters, and configuration is largely driven by *trial and error*
1811 manner [513, 45, 440]. Such an *ad hoc* and immature software development environment
1812 poses a huge challenge to the community.
1813 A related challenge is lack of tools and techniques for various phases and tasks involved in ML
1814 software development. It includes effective tools for testing ML programs, ensuring that the
1815 dataset are pre-processed adequately, debugging, and effective data management [513, 373,
1816 155]. In addition, quality aspects such as explainability and trust-worthiness are new desired
1817 quality aspects especially applicable for ML code where current practices and knowledge is
1818 inadequate [155].
1819 *Mitigation:* The ad-hoc trial and error ML development can be addressed by improved tools
1820 and techniques. Even though the variety of ML development environments including man-
1821 aged services such as AWS Sagemaker and Google Notebooks attempt to make ML develop-
1822 ment easier, they essentially do not offer much help in reducing the ad-hoc nature of the
1823 development. A significant research push from the community would make ML development
1824 relatively systematic and organized.
1825 Recent advancements in the form of available tools not only help a developer to comprehend
1826 the process but also let them effectively manage code, data, and experimental results. Exam-
1827 ples of such tools and methods include DARVIZ [420] for DL model visualization, MLFlow[10] for
1828 managing the ML lifecycle, and DeepFault [136] for identifying faults in DL programs. Such
1829 efforts are expected to address the challenge.
1830 Software Engineering for Machine Learning (SE4ML) brings another perspective to this issue
1831 by bringing best practices from software engineering to ML development. Efforts in this di-
1832 rection not only can make ML specific code maintainable and reliable but also can contribute
1833 back to reproducibility and replicability.

[9] https://2021.esec-fse.org/track/fse-2021-artifacts
[10] https://mlflow.org/

Journal Pre-proof

- ***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 user-survey 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 DL 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.

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

Journal Pre-proof

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[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[12] open-source to allow the community to extend the living literature survey.

6. 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 manuscript. Maria Kechagia and Federica Sarro are supported by the ERC grant no. 741278 (EPIC).

[11] http://www.tusharma.in/ML4SCA
[12] https://github.com/tushartushar/ML4SCA

Journal Pre-proof

1929 ## References

1930 [1] Github archive, 2020. URL https://www.gharchive.org/.

1931 [2] Raja Abbas, Fawzi Abdulaziz Albalooshi, and Mustafa Hammad. Software change proneness
1932 prediction using machine learning. In *2020 International Conference on Innovation and Intelli-*
1933 *gence for Informatics, Computing and Technologies (3ICT)*, pages 1--7. IEEE, 2020.

1934 [3] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. A machine learning approach to
1935 improve the detection of ci skip commits. *IEEE Transactions on Software Engineering*, 2020.

1936 [4] Osama Abdeljaber, Onur Avci, Serkan Kiranyaz, Moncef Gabbouj, and Daniel J Inman. Real-
1937 time vibration-based structural damage detection using one-dimensional convolutional neu-
1938 ral networks. *Journal of Sound and Vibration*, 388:154--170, 2017.

1939 [5] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. Large-scale
1940 and language-oblivious code authorship identification. In *Proceedings of the 2018 ACM SIGSAC*
1941 *Conference on Computer and Communications Security*, CCS '18, page 101–114, 2018. ISBN
1942 9781450356930. doi: 10.1145/3243734.3243738.

1943 [6] Ibrahim Abunadi and Mamdouh Alenezi. Towards cross project vulnerability prediction in
1944 open source web applications. In *Proceedings of the The International Conference on Engineer-*
1945 *ing & MIS 2015*, ICEMIS '15, New York, NY, USA, 2015. Association for Computing Machinery.
1946 ISBN 9781450334181. doi: 10.1145/2832987.2833051. URL https://doi.org/10.1145/2832987.
1947 2833051.

1948 [7] Simran Aggarwal. Software code analysis using ensemble learning techniques. In *Proceedings*
1949 *of the International Conference on Advanced Information Science and System*, AISS '19, 2019.
1950 ISBN 9781450372916. doi: 10.1145/3373477.3373486.

1951 [8] Mansi Agnihotri and Anuradha Chug. Application of machine learning algorithms for code
1952 smell prediction using object-oriented software metrics. *Journal of Statistics and Management*
1953 *Systems*, 23(7):1159--1171, 2020. doi: 10.1080/09720510.2020.1799576.

1954 [9] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based
1955 approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the*
1956 *Association for Computational Linguistics*, pages 4998--5007, July 2020. doi: 10.18653/v1/2020.
1957 acl-main.449.

1958 [10] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. Com-
1959 pilation error repair: For the student programs, from the student programs. In *Proceedings*
1960 *of the 40th International Conference on Software Engineering: Software Engineering Education*
1961 *and Training*, ICSE-SEET '18, page 78–87, 2018. ISBN 9781450356602. doi: 10.1145/3183377.
1962 3183383.

1963 [11] H. A. Al-Jamimi and M. Ahmed. Machine learning-based software quality prediction models:
1964 State of the art. In *2013 International Conference on Information Science and Applications (ICISA)*,
1965 pages 1--4, 2013. doi: 10.1109/ICISA.2013.6579473.

1966 [12] Osama Al Qasem, Mohammed Akour, and Mamdouh Alenezi. The influence of deep learning
1967 algorithms factors in software fault prediction. *IEEE Access*, 8:63945--63960, 2020.

1968 [13] A. AL-Shaaby, Hamoud I. Aljamaan, and M. Alshayeb. Bad smell detection using machine
1969 learning techniques: A systematic literature review. *Arabian Journal for Science and Engineer-*
1970 *ing*, 45:2341--2369, 2020.

Journal Pre-proof

[14] Amal Alazba and Hamoud Aljamaan. Code smell detection using feature selection and stacking ensemble: An empirical investigation. *Information and Software Technology*, 138:106648, 2021.

[15] Saiqa Aleem, Luiz Fernando Capretz, Faheem Ahmed, et al. Comparative performance analysis of machine learning techniques for software bug detection. In *Proceedings of the 4th International Conference on Software Engineering and Applications*, number 1, pages 71--79. AIRCC Press Chennai, Tamil Nadu, India, 2015.

[16] Aldeida Aleti and Matias Martinez. E-apr: mapping the effectiveness of automated program repair techniques. *Empirical Software Engineering*, 26(5):1--30, 2021.

[17] Sultan Alhusain, Simon Coupland, Robert John, and Maria Kavanagh. Towards machine learning based design pattern recognition. In *2013 13th UK Workshop on Computational Intelligence (UKCI)*, pages 244--251. IEEE, 2013.

[18] Nasir Ali, Zohreh Sharafi, Yann-Ga"el Guéhéneuc, and Giuliano Antoniol. An empirical study on the importance of source code entities for requirements traceability. *Empirical software engineering*, 20(2):442--478, 2015.

[19] Huda Ali Alatwi, Tae Oh, Ernest Fokoue, and Bill Stackpole. Android malware detection using category-based machine learning classifiers. In *Proceedings of the 17th Annual Conference on Information Technology Education*, SIGITE '16, page 54–59, 2016. ISBN 9781450344524. doi: 10.1145/2978192.2978218.

[20] E. A. Alikhashashneh, R. R. Raje, and J. H. Hill. Using machine learning techniques to classify and predict static code analysis tool warnings. In *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*, pages 1--8, 2018. doi: 10.1109/AICCSA.2018.8612819.

[21] Hamoud Aljamaan and Amal Alazba. Software defect prediction using tree-based ensembles. In *Proceedings of the 16th ACM international conference on predictive models and data analytics in software engineering*, pages 1--10, 2020.

[22] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207--216, 2013. doi: 10.1109/MSR.2013.6624029.

[23] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *10th Working Conference on Mining Software Repositories (MSR)*, pages 207--216, 2013. doi: 10.1109/MSR.2013.6624029.

[24] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 38–49, 2015. ISBN 9781450336758. doi: 10.1145/2786805.2786849.

[25] Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 2123–2132, 2015.

[26] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code, 2016.

[27] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), July 2018. ISSN 0360-0300. doi: 10.1145/3212695.

Journal Pre-proof

[28] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[29] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468--471, 2016. ISBN 978-1-4503-4186-8. doi: 10.1145/2901739.2903508.

[30] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. *SIGPLAN Not.*, 53(4):404–419, June 2018. ISSN 0362-1340. doi: 10.1145/3296979.3192412.

[31] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code, 2019.

[32] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi: 10.1145/3290353.

[33] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastian Uchitel. Automated support for diagnosis and repair. *Commun. ACM*, 58(2):65–72, January 2015. ISSN 0001-0782. doi: 10.1145/2658986.

[34] Hadeel Alsolai and Marc Roper. A systematic literature review of machine learning techniques for software maintainability prediction. *Information and Software Technology*, 119: 106214, 2020. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2019.106214.

[35] Doaa Altarawy, Hossameldin Shahin, Ayat Mohammed, and Na Meng. Lascad: Language-agnostic software categorization and similar application detection. *Journal of Systems and Software*, 142:21--34, 2018.

[36] H. Alves, B. Fonseca, and N. Antunes. Experimenting machine learning techniques to predict vulnerabilities. In *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, pages 151--156, 2016. doi: 10.1109/LADC.2016.32.

[37] Boukhdhir Amal, Marouane Kessentini, Slim Bechikh, Josselin Dea, and Lamjed Ben Said. On the use of machine learning and search-based software engineering for ill-defined fitness function: A case study on software refactoring. In Claire Le Goues and Shin Yoo, editors, *Search-Based Software Engineering*, pages 31--45, 2014. ISBN 978-3-319-09940-8.

[38] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 261--269, 2015. doi: 10.1109/ISSRE.2015.7381819.

[39] L. A. Amorim, M. F. Freitas, A. Dantas, E. F. de Souza, C. G. Camilo-Junior, and W. S. Martins. A new word embedding approach to evaluate potential fixes for automated program repair. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1--8, 2018. doi: 10.1109/IJCNN.2018.8489079.

[40] M. Aniche, E. Maziero, R. Durelli, and V. Durelli. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, pages 1--1, 2020. doi: 10.1109/TSE.2020.3021736.

[41] "Omer Faruk Arar and K"urşat Ayan. Software defect prediction using cost-sensitive neural network. *Applied Soft Computing*, 33:263--277, 2015.

Journal Pre-proof

[42] Francesca Arcelli Fontana and Marco Zanoni. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems*, 128:43 -- 58, 2017. ISSN 0950-7051. doi: https://doi.org/10.1016/j.knosys.2017.04.014.

[43] Vamsi Krishna Aribandi, Lov Kumar, Lalita Bhanu Murthy Neti, and Aneesh Krishna. Prediction of refactoring-prone classes using ensemble learning. In Tom Gedeon, Kok Wai Wong, and Minho Lee, editors, *Neural Information Processing*, pages 242--250, 2019. ISBN 978-3-030-36802-9.

[44] David Azcona, Piyush Arora, I-Han Hsiao, and Alan Smeaton. User2code2vec: Embeddings for profiling students based on distributional representations of source code. In *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*, LAK19, page 86–95, 2019. ISBN 9781450362566. doi: 10.1145/3303772.3303813.

[45] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115 -- 138, 2019. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2018.12.009.

[46] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi: 10.1145/3360585.

[47] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016.

[48] Xinbo Ban, Shigang Liu, Chao Chen, and Caslon Chua. A performance evaluation of deep-learnt features for software vulnerability detection. *Concurrency and Computation: Practice and Experience*, 31(19):e5103, 2019. ISSN 1532-0634. doi: 10.1002/cpe.5103. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5103. _eprint: https://onlinelibrary.wiley.com/-doi/pdf/10.1002/cpe.5103.

[49] U. Bandara and G. Wijayarathna. A machine learning based tool for source code plagiarism detection. *International Journal of Machine Learning and Computing*, pages 337--343, 2011.

[50] Vishnu Banna, Akhil Chinnakotla, Zhengxin Yan, Anirudh Vegesana, Naveen Vivek, Kruthi Krishnappa, Wenxin Jiang, Yung-Hsiang Lu, George K. Thiruvathukal, and James C. Davis. An experience report on machine learning reproducibility: Guidance for practitioners and tensorflow model garden contributors. *CoRR*, abs/2107.00821, 2021. URL https://arxiv.org/abs/2107.00821.

[51] A. Bansal, S. Haque, and C. McMillan. Project-level encoding for neural source code summarization of subroutines. In *2021 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) (ICPC)*, pages 253--264. IEEE Computer Society, may 2021. doi: 10.1109/ICPC52881.2021.00032.

[52] Antoine Barbez, Foutse Khomh, and Yann-Gaël Guéhéneuc. A machine-learning based ensemble method for anti-patterns detection. *Journal of Systems and Software*, 161:110486, 2020. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.110486.

[53] Antonio Valerio Miceli Barone and Rico Sennrich. A parallel corpus of python functions and documentation strings for automated code documentation and code generation, 2017.

[54] Canan Batur Şahin and Laith Abualigah. A novel deep learning-based feature selection model for improving the static analysis of vulnerability detection. *Neural Comput. Appl.*, 33(20):14049–14067, oct 2021. ISSN 0941-0643. doi: 10.1007/s00521-021-06047-x. URL https://doi.org/10.1007/s00521-021-06047-x.

Journal Pre-proof

[55] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 40(7):671--694, 2013.

[56] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea de Lucia. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(1):1--33, 2014.

[57] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3589–3601, 2018.

[58] G. P. Bhandari and R. Gupta. Machine learning based software fault prediction utilizing source code metrics. In *2018 IEEE 3rd International Conference on Computing, Communication and Security (ICCCS)*, pages 40--45, 2018. doi: 10.1109/CCCS.2018.8586805.

[59] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. Neuro-symbolic program corrector for introductory programming assignments. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 60–70, 2018. ISBN 9781450356381. doi: 10.1145/3180155.3180219.

[60] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. Program synthesis for character level language modeling. In *ICLR*, 2017.

[61] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay. Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672--150684, 2020. doi: 10.1109/ACCESS.2020.3016774.

[62] Paul E. Black. Software Assurance with SAMATE Reference Dataset, Tool Standards, and Studies. October 2007.

[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.

[64] David Bowes, Tracy Hall, Mark Harman, Yue Jia, Federica Sarro, and Fan Wu. Mutation-aware fault prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 330–341, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2931039. URL https://doi.org/10.1145/2931037.2931039.

[65] Ronyérison Braga, Pedro Santos Neto, Ricardo Rabêlo, José Santiago, and Matheus Souza. A machine learning approach to generate test oracles. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, SBES '18, page 142–151, 2018. ISBN 9781450365031. doi: 10.1145/3266237.3266273.

[66] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*, CC 2020, page 201–211, 2020. ISBN 9781450371209.

[67] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. In *International Conference on Learning Representations*, 2019.

Journal Pre-proof

2144 [68] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal,
2145 Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel
2146 Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M.
2147 Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz
2148 Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec
2149 Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.
2150 URL https://arxiv.org/abs/2005.14165.

2151 [69] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code
2152 completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineer-*
2153 *ing Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*,
2154 ESEC/FSE '09, page 213–222, 2009. ISBN 9781605580012. doi: 10.1145/1595696.1595728.

2155 [70] Yuriy Brun and Alexandra Meliou. Software fairness. In *Proceedings of the 2018 26th ACM*
2156 *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations*
2157 *of Software Engineering*, ESEC/FSE 2018, page 754–759, New York, NY, USA, 2018. Association
2158 for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3264838. URL https:
2159 //doi.org/10.1145/3236024.3264838.

2160 [71] N. D. Q. Bui, Y. Yu, and L. Jiang. Bilateral dependency neural networks for cross-language
2161 algorithm classification. In *2019 IEEE 26th International Conference on Software Analysis, Evolu-*
2162 *tion and Reengineering (SANER)*, pages 422--433, 2019. doi: 10.1109/SANER.2019.8667995.

2163 [72] Nghi D. Q. Bui, Lingixao Jiang, and Y. Yu. Cross-language learning for program classification
2164 using bilateral tree-based convolutional neural networks. In *AAAI Workshops*, 2018.

2165 [73] L. Butgereit. Using machine learning to prioritize automated testing in an agile environment.
2166 In *2019 Conference on Information Communications Technology and Society (ICTAS)*, pages 1--6,
2167 2019. doi: 10.1109/ICTAS.2019.8703639.

2168 [74] Cheng-Hao Cai, Jing Sun, and Gillian Dobbie. Automatic b-model repair using model checking
2169 and machine learning. *Automated Software Engineering*, 26(3), January 2019. ISSN 1573-7535.
2170 doi: 10.1007/s10515-019-00264-4.

2171 [75] Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures gen-
2172 eralize via recursion. *CoRR*, abs/1704.06611, 2017.

2173 [76] José P Cambronero and Martin C Rinard. Al: autogenerating supervised learning programs.
2174 *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1--28, 2019.

2175 [77] Frederico Luiz Caram, Bruno Rafael De Oliveira Rodrigues, Amadeu Silveira Campanelli, and
2176 Fernando Silva Parreiras. Machine learning techniques for code smells detection: a system-
2177 atic mapping study. *International Journal of Software Engineering and Knowledge Engineering*,
2178 29(02):285--316, 2019.

2179 [78] Frederico Luiz Caram, Bruno Rafael De Oliveira Rodrigues, Amadeu Silveira Campanelli, and
2180 Fernando Silva Parreiras. Machine learning techniques for code smells detection: A system-
2181 atic mapping study. *International Journal of Software Engineering and Knowledge Engineering*,
2182 29(02):285--316, 2019. doi: 10.1142/S021819401950013X.

2183 [79] Silvio Cesare, Yang Xiang, and Jun Zhang. Clonewise -- detecting package-level clones us-
2184 ing machine learning. In Tanveer Zia, Albert Zomaya, Vijay Varadharajan, and Morley Mao,
2185 editors, *Security and Privacy in Communication Networks*, pages 197--215, 2013. ISBN 978-3-
2186 319-04283-1.

Journal Pre-proof

Journal Pre-proof

2187 [80] M. Cetiner and O. K. Sahingoz. A comparative analysis for machine learning based software
2188 defect prediction systems. In *2020 11th International Conference on Computing, Communica-*
2189 *tion and Networking Technologies (ICCCNT)*, pages 1--7, 2020. doi: 10.1109/ICCCNT49239.2020.
2190 9225352.

2191 [81] E. Ceylan, F. O. Kutlubay, and A. B. Bener. Software defect identification using machine learn-
2192 ing techniques. In *32nd EUROMICRO Conference on Software Engineering and Advanced Appli-*
2193 *cations (EUROMICRO'06)*, pages 240--247, 2006. doi: 10.1109/EUROMICRO.2006.56.

2194 [82] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray. Codit: Code editing with tree-based neural
2195 models. *IEEE Transactions on Software Engineering*, pages 1--1, 2020. doi: 10.1109/TSE.2020.
2196 3020502.

2197 [83] Saikat Chakraborty and Baishakhi Ray. On multi-modal learning of editing source code. In
2198 *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages
2199 443--455, 2021. doi: 10.1109/ASE51524.2021.1003_Chakraborty2021.

2200 [84] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. Codit: Code
2201 editing with tree-based neural models. *IEEE Transactions on Software Engineering*, 48(4):
2202 1385--1399, 2022. doi: 10.1109/TSE.2020.3020502.

2203 [85] VENKATA UDAYA B. CHALLAGULLA, FAROKH B. BASTANI, I-LING YEN, and RAYMOND A.
2204 PAUL. Empirical assessment of machine learning based software defect prediction tech-
2205 niques. *International Journal on Artificial Intelligence Tools*, 17(02):389--400, 2008. doi:
2206 10.1142/S0218213008003947.

2207 [86] T. Chappelly, C. Cifuentes, P. Krishnan, and S. Gevay. Machine learning for finding bugs:
2208 An initial report. In *2017 IEEE Workshop on Machine Learning Techniques for Software Quality*
2209 *Evaluation (MaLTeSQuE)*, pages 21--26, 2017. doi: 10.1109/MALTESQUE.2017.7882012.

2210 [87] Shivam Chaturvedi, Amrita Chaturvedi, Anurag Tiwari, and Shalini Agarwal. Design pattern
2211 detection using machine learning techniques. In *2018 7th International Conference on Relia-*
2212 *bility, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*, pages 1--6.
2213 IEEE, 2018.

2214 [88] Deyu Chen, Xiang Chen, Hao Li, Junfeng Xie, and Yanzhou Mu. Deepcpdp: Deep learning
2215 based cross-project defect prediction. *IEEE Access*, 7:184832--184848, 2019.

2216 [89] Fuxiang Chen, Mijung Kim, and Jaegul Choo. Novel natural language summarization of
2217 program code via leveraging multiple input representations. In *Findings of the Association*
2218 *for Computational Linguistics: EMNLP 2021*, pages 2510--2520, Punta Cana, Dominican Re-
2219 public, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.
2220 findings-emnlp.214. URL https://aclanthology.org/2021.findings-emnlp.214.

2221 [90] Jinyin Chen, Keke Hu, Yue Yu, Zhuangzhi Chen, Qi Xuan, Yi Liu, and Vladimir Filkov. Soft-
2222 ware visualization and deep transfer learning for effective software defect prediction. In
2223 *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20,
2224 page 578–589, 2020. ISBN 9781450371216. doi: 10.1145/3377811.3380389.

2225 [91] Long Chen, Wei Ye, and Shikun Zhang. Capturing source code semantics via tree-based
2226 convolution over api-enhanced ast. In *Proceedings of the 16th ACM International Conference*
2227 *on Computing Frontiers*, CF '19, page 174–182, 2019. ISBN 9781450366854. doi: 10.1145/
2228 3310273.3321560.

2229 [92] M. Chen and X. Wan. Neural comment generation for source code with auxiliary code classifi-
2230 cation task. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 522--529,
2231 2019. doi: 10.1109/APSEC48747.2019.00076.

2232 [93] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared
2233 Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large
2234 language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

2235 [94] Q. Chen and M. Zhou. A neural framework for retrieval and summarization of source code.
2236 In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages
2237 826--831, 2018. doi: 10.1145/3238147.3240471.

2238 [95] Qiuyuan Chen, Han Hu, and Zhaoyi Liu. Code summarization with abstract syntax tree. In
2239 Tom Gedeon, Kok Wai Wong, and Minho Lee, editors, *Neural Information Processing*, pages
2240 652--660, 2019. ISBN 978-3-030-36802-9.

2241 [96] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. Why my code summarization
2242 model does not work: Code comment improvement with category prediction. *ACM Transac-*
2243 *tions on Software Engineering and Methodology (TOSEM)*, 30(2):1--29, 2021.

2244 [97] Xinyun Chen, Chang Liu, Richard Shin, Dawn Song, and Mingcheng Chen. Latent attention
2245 for if-then program synthesis. In *Proceedings of the 30th International Conference on Neural*
2246 *Information Processing Systems*, NIPS'16, page 4581–4589, 2016. ISBN 9781510838819.

2247 [98] Xinyun Chen, Chang Liu, and Dawn Song. Towards synthesizing complex programs from
2248 input-output examples, 2018.

2249 [99] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In
2250 *International Conference on Learning Representations*, 2019.

2251 [100] Yang Chen, Andrew E. Santosa, Ang Ming Yi, Abhishek Sharma, Asankhaya Sharma, and
2252 David Lo. *A Machine Learning Approach for Vulnerability Curation*, page 32–42. Associa-
2253 tion for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450375177. URL
2254 https://doi.org/10.1145/3379597.3387461.

2255 [101] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus. Se-
2256 quencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions*
2257 *on Software Engineering*, pages 1--1, 2019. doi: 10.1109/TSE.2019.2940179.

2258 [102] Boris Chernis and Rakesh Verma. Machine learning methods for software vulnerability detec-
2259 tion. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*,
2260 IWSPA '18, page 31–39, 2018. ISBN 9781450356343. doi: 10.1145/3180445.3180453.

2261 [103] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transaction*
2262 *of Software Engineering*, 20(6):476--493, June 1994. ISSN 0098-5589. doi: 10.1109/32.295895.

2263 [104] Y. Choi, S. Kim, and J. Lee. Source code summarization using attention-based keyword mem-
2264 ory networks. In *2020 IEEE International Conference on Big Data and Smart Computing (Big-*
2265 *Comp)*, pages 564--570, 2020. doi: 10.1109/BigComp48618.2020.00011.

2266 [105] Garvit Rajesh Choudhary, Sandeep Kumar, Kuldeep Kumar, Alok Mishra, and Cagatay Catal.
2267 Empirical analysis of change metrics for software fault prediction. *Computers & Electrical*
2268 *Engineering*, 67:15--24, 2018.

2269 [106] A. Chug and S. Dhall. Software defect prediction using supervised learning algorithm and un-
2270 supervised learning algorithm. In *Confluence 2013: The Next Generation Information Technol-*
2271 *ogy Summit (4th International Conference)*, pages 173--179, 2013. doi: 10.1049/cp.2013.2313.

2272 [107] C. J. Clemente, F. Jaafar, and Y. Malik. Is predicting software security bugs using deep learning
2273 better than the traditional machine learning algorithms? In *2018 IEEE International Conference*
2274 *on Software Quality, Reliability and Security (QRS)*, pages 95--102, 2018. doi: 10.1109/QRS.2018.
2275 00023.

Journal Pre-proof

Journal Pre-proof

2276 [108] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. Embedding java classes with
2277 code2vec: Improvements from variable obfuscation. In *Proceedings of the 17th Interna-*
2278 *tional Conference on Mining Software Repositories*, MSR '20, page 243–253, 2020. ISBN
2279 9781450375177. doi: 10.1145/3379597.3387445.

2280 [109] Luis Fernando Cortes-Coy, M. Vásquez, Jairo Aponte, and D. Poshyvanyk. On automatically
2281 generating commit messages via summarization of source code changes. *2014 IEEE 14th*
2282 *International Working Conference on Source Code Analysis and Manipulation*, pages 275--284,
2283 2014.

2284 [110] Daniel Cruz, Amanda Santana, and Eduardo Figueiredo. Detecting bad smells with machine
2285 learning algorithms: an empirical study. In *Proceedings of the 3rd International Conference on*
2286 *Technical Debt*, pages 31--40, 2020.

2287 [111] Daniel Cruz, Amanda Santana, and Eduardo Figueiredo. Detecting bad smells with machine
2288 learning algorithms: An empirical study. In *Proceedings of the 3rd International Conference on*
2289 *Technical Debt*, TechDebt '20, page 31–40, 2020. ISBN 9781450379601. doi: 10.1145/3387906.
2290 3388618.

2291 [112] Jianfeng Cui, Lixin Wang, Xin Zhao, and Hongyi Zhang. Towards predictive analysis of android
2292 vulnerability using statistical codes and machine learning for iot applications. *Computer Com-*
2293 *munications*, 155:125 -- 131, 2020. ISSN 0140-3664. doi: https://doi.org/10.1016/j.comcom.
2294 2020.02.078.

2295 [113] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. Synthesizing benchmarks for predic-
2296 tive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization*
2297 *(CGO)*, pages 86--99, 2017. doi: 10.1109/CGO.2017.7863731.

2298 [114] Warteruzannan Soyer Cunha, Guisella Angulo Armijo, and Valter Vieira de Camargo. *Inves-*
2299 *tigating Non-Usually Employed Features in the Identification of Architectural Smells: A Machine*
2300 *Learning-Based Approach*, page 21–30. 2020. ISBN 9781450387545.

2301 [115] Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. Open vocabulary learning on
2302 source code with a graph-structured cache. volume 97 of *Proceedings of Machine Learning*
2303 *Research*, pages 1475--1485, 09--15 Jun 2019.

2304 [116] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose,
2305 Taeksu Kim, and Chul-Joo Kim. Lessons learned from using a deep tree-based model for
2306 software defect prediction in practice. In *Proceedings of the 16th International Conference on*
2307 *Mining Software Repositories*, MSR '19, page 46–57, 2019. doi: 10.1109/MSR.2019.00017.

2308 [117] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction ap-
2309 proaches: A benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4–5):
2310 531–577, August 2012. ISSN 1382-3256. doi: 10.1007/s10664-011-9173-9. URL https:
2311 //doi.org/10.1007/s10664-011-9173-9.

2312 [118] Altino Dantas, Eduardo F. de Souza, Jerffeson Souza, and Celso G. Camilo-Junior. Code nat-
2313 uralness to assist search space exploration in search-based program repair methods. In
2314 Shiva Nejati and Gregory Gay, editors, *Search-Based Software Engineering*, pages 164--170,
2315 2019. ISBN 978-3-030-27455-9.

2316 [119] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano
2317 Panichella. Labeling source code with information retrieval methods: an empirical study.
2318 *Empirical Software Engineering*, 19(5):1383--1420, 2014.

Journal Pre-proof

2319 [120] Karel Dejaeger, Thomas Verbraken, and Bart Baesens. Toward comprehensible software
2320 fault prediction models using bayesian network classifiers. *IEEE Transactions on Software*
2321 *Engineering*, 39(2):237--257, 2012.

2322 [121] Jacob Devlin, Rudy Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural
2323 program meta-induction. In *Proceedings of the 31st International Conference on Neural Infor-*
2324 *mation Processing Systems*, NIPS'17, page 2077–2085, 2017. ISBN 9781510860964.

2325 [122] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed,
2326 and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings*
2327 *of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 990–998,
2328 2017.

2329 [123] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of
2330 deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*,
2331 2018.

2332 [124] Seema Dewangan, Rajwant Singh Rao, Alok Mishra, and Manjari Gupta. A novel approach
2333 for code smell detection: An empirical study. *IEEE Access*, 9:162869--162883, 2021.

2334 [125] N. Dhamayanthi and B. Lavanya. Improvement in software defect prediction outcome using
2335 principal component analysis and ensemble machine learning algorithms. In Jude Hemanth,
2336 Xavier Fernando, Pavel Lafata, and Zubair Baig, editors, *International Conference on Intelligent*
2337 *Data Communication Technologies and Internet of Things (ICICI) 2018*, pages 397--406, 2019.
2338 ISBN 978-3-030-03146-6.

2339 [126] Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, and Federica Sarro. A genetic algo-
2340 rithm to configure support vector machines for predicting fault-prone components. In Danilo
2341 Caivano, Markku Oivo, Maria Teresa Baldassarre, and Giuseppe Visaggio, editors, *Product-*
2342 *Focused Software Process Improvement*, pages 247--261, Berlin, Heidelberg, 2011. Springer
2343 Berlin Heidelberg. ISBN 978-3-642-21843-9.

2344 [127] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. Detecting code
2345 smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th International*
2346 *Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612--621, 2018.
2347 doi: 10.1109/SANER.2018.8330266.

2348 [128] Li Dong and Mirella Lapata. Language to logical form with neural attention. In *Proceedings of*
2349 *the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*,
2350 pages 33--43, Berlin, Germany, August 2016. Association for Computational Linguistics. doi:
2351 10.18653/v1/P16-1004. URL https://aclanthology.org/P16-1004.

2352 [129] Geanderson Esteves Dos Santos, E. Figueiredo, Adriano Veloso, Markos Viggiato, and N. Zi-
2353 viani. Understanding machine learning software defect predictions. *Autom. Softw. Eng.*, 27:
2354 369--392, 2020.

2355 [130] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang.
2356 Leopard: Identifying vulnerable code for vulnerability assessment through program metrics.
2357 In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 60--71,
2358 2019. doi: 10.1109/ICSE.2019.00024.

2359 [131] Yao Du, Xiaoqing Wang, and Junfeng Wang. A static android malicious code detection method
2360 based on multi-source fusion. *Sec. and Commun. Netw.*, 8(17):3238–3246, nov 2015. ISSN
2361 1939-0114. doi: 10.1002/sec.1248. URL https://doi.org/10.1002/sec.1248.

Journal Pre-proof

2362 [132] V. H. S. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. C. Dias, and M. P.
2363 Guimarães. Machine learning applied to software testing: A systematic mapping study. *IEEE*
2364 *Transactions on Reliability*, 68(3):1189--1212, 2019. doi: 10.1109/TR.2019.2892517.

2365 [133] Ashish Kumar Dwivedi, Anand Tirkey, Ransingh Biswajit Ray, and Santanu Kumar Rath. Soft-
2366 ware design pattern recognition using machine learning techniques. In *2016 ieee region 10*
2367 *conference (tencon)*, pages 222--227. IEEE, 2016.

2368 [134] Vasiliki Efstathiou and Diomidis Spinellis. Semantic source code models using identifier em-
2369 beddings. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories*
2370 *(MSR)*, pages 29--33, 2019. doi: 10.1109/MSR.2019.00015.

2371 [135] Yuval Elovici, Asaf Shabtai, Robert Moskovitch, Gil Tahan, and Chanan Glezer. Applying ma-
2372 chine learning techniques for detection of malicious code in network traffic. In Joachim
2373 Hertzberg, Michael Beetz, and Roman Englert, editors, *KI 2007: Advances in Artificial Intelli-*
2374 *gence*, pages 44--50, 2007. ISBN 978-3-540-74565-5.

2375 [136] Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. Deepfault: Fault localization for deep
2376 neural networks. In Reiner Hähnle and Wil van der Aalst, editors, *Fundamental Approaches to*
2377 *Software Engineering*, pages 171--191, Cham, 2019. Springer International Publishing. ISBN
2378 978-3-030-16722-6.

2379 [137] Ezgi Erturk and Ebru Akcapinar Sezer. A comparison of some soft computing methods for
2380 software fault prediction. *Expert systems with applications*, 42(4):1872--1879, 2015.

2381 [138] Khashayar Etemadi and Martin Monperrus. On the relevance of cross-project learning with
2382 nearest neighbours for commit message generation. In *Proceedings of the IEEE/ACM 42nd*
2383 *International Conference on Software Engineering Workshops*, pages 470--475, 2020.

2384 [139] S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh, and G. Antoniol. Keep it simple: Is deep
2385 learning good for linguistic smell detection? In *2018 IEEE 25th International Conference on*
2386 *Software Analysis, Evolution and Reengineering (SANER)*, pages 602--611, 2018. doi: 10.1109/
2387 SANER.2018.8330265.

2388 [140] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus.
2389 Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE In-*
2390 *ternational Conference on Automated Software Engineering*, ASE '14, page 313–324, 2014. ISBN
2391 9781450330138. doi: 10.1145/2642937.2642982.

2392 [141] Guisheng Fan, Xuyang Diao, Huiqun Yu, Kang Yang, and Liqiong Chen. Deep semantic feature
2393 learning with embedded static metrics for software defect prediction. In *2019 26th Asia-*
2394 *Pacific Software Engineering Conference (APSEC)*, pages 244--251. IEEE, 2019.

2395 [142] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. Functional code clone
2396 detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT*
2397 *International Symposium on Software Testing and Analysis*, ISSTA 2020, page 516–527, 2020.
2398 ISBN 9781450380089. doi: 10.1145/3395363.3397362.

2399 [143] Yong Fang, Yongcheng Liu, Cheng Huang, and Liang Liu. FastEmbed: Predicting vulnerabil-
2400 ity exploitation possibility based on ensemble machine learning algorithm. *PLoS ONE*, 15:
2401 e0228439, February 2020. doi: 10.1371/journal.pone.0228439. URL https://ui.adsabs.harvard.
2402 edu/abs/2020PLoSO..1528439F. ADS Bibcode: 2020PLoSO..1528439F.

2403 [144] Ebubeogu Amarachukwu Felix and Sai Peck Lee. Integrated approach to software defect
2404 prediction. *IEEE Access*, 5:21524--21547, 2017.

Journal Pre-proof

2405 [145] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou,
2406 Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for program-
2407 ming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP*
2408 *2020*, pages 1536--1547, Online, November 2020. Association for Computational Linguistics.
2409 doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.
2410 139.

2411 [146] Rudolf Ferenc, Péter Hegedundefineds, Péter Gyimesi, Gábor Antal, Dénes Bán, and Tibor
2412 Gyimóthy. Challenging machine learning algorithms in predicting vulnerable javascript func-
2413 tions. In *Proceedings of the 7th International Workshop on Realizing Artificial Intelligence Syner-*
2414 *gies in Software Engineering*, RAISE '19, page 8–14, 2019. doi: 10.1109/RAISE.2019.00010.

2415 [147] Fabio Ferreira, Luciana Lourdes Silva, and Marco Tulio Valente. Software engineering meets
2416 deep learning: A mapping study. In *Proceedings of the 36th Annual ACM Symposium on Applied*
2417 *Computing*, SAC '21, page 1542–1549, New York, NY, USA, 2021. Association for Computing
2418 Machinery. ISBN 9781450381048. doi: 10.1145/3412841.3442029. URL https://doi.org/10.
2419 1145/3412841.3442029.

2420 [148] F. Fontana, M. Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experiment-
2421 ing machine learning techniques for code smell detection. *Empirical Software Engineering*, 21:
2422 1143--1191, 2015.

2423 [149] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä. Code smell detection: Towards a
2424 machine learning-based approach. In *2013 IEEE International Conference on Software Mainte-*
2425 *nance*, pages 396--399, 2013.

2426 [150] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Pat-*
2427 *terns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Profes-
2428 sional. Part of the Addison-Wesley Professional Computing Series series., 1st edi-
2429 tion, October 1994. ISBN 978-0-201-63361-0. URL https://www.informit.com/store/
2430 design-patterns-elements-of-reusable-object-oriented-9780201633610?w_ptgrevartcl=Grady+
2431 Booch+on+Design+Patterns%2c+OOP%2c+and+Coffee_1405569.

2432 [151] Zhipeng Gao, Xin Xia, John Grundy, David Lo, and Yuan-Fang Li. Generating question titles for
2433 stack overflow from mined code snippets. *ACM Trans. Softw. Eng. Methodol.*, 29(4), September
2434 2020. ISSN 1049-331X. doi: 10.1145/3401026.

2435 [152] Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud.
2436 Augmenting commit classification by using fine-grained source code changes and a pre-
2437 trained deep neural language model. *Information and Software Technology*, 135:106566, 2021.

2438 [153] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and
2439 discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv.*,
2440 50(4), August 2017. ISSN 0360-0300. doi: 10.1145/3092566.

2441 [154] Sirine Gharbi, Mohamed Wiem Mkaouer, Ilyes Jenhani, and Montassar Ben Messaoud. On
2442 the classification of software change messages using multi-label active learning. In *Proceed-*
2443 *ings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1760--1767, 2019.

2444 [155] Görkem Giray. A software engineering perspective on engineering machine learning sys-
2445 tems: State of the art and challenges. *Journal of Systems and Software*, 180:111031, 2021.
2446 ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2021.111031. URL https://www.sciencedirect.
2447 com/science/article/pii/S016412122100128X.

Journal Pre-proof

2448 [156] P. Godefroid, H. Peleg, and R. Singh. Learn fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50--59, 2017. doi: 10.1109/ASE.2017.8115618.

2451 [157] Iker Gondra. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software*, 81(2):186 -- 195, 2008. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2007.05.035. Model-Based Software Testing.

2454 [158] R. Gopalakrishnan, P. Sharma, M. Mirakhorli, and M. Galster. Can latent topics in source code predict missing architectural tactics? In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 15--26, 2017. doi: 10.1109/ICSE.2017.10.

2457 [159] Raghuram Gopalakrishnan, Palak Sharma, Mehdi Mirakhorli, and Matthias Galster. Can latent topics in source code predict missing architectural tactics? In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 15--26. IEEE, 2017.

2460 [160] D. Gopinath, K. Wang, J. Hua, and S. Khurshid. Repairing intricate faults in code using machine learning and path exploration. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 453--457, 2016. doi: 10.1109/ICSME.2016.75.

2463 [161] Divya Gopinath, Sarfraz Khurshid, Diptikalyan Saha, and Satish Chandra. Data-guided repair of selection statements. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 243–253, 2014. ISBN 9781450327565. doi: 10.1145/2568225.2568303.

2467 [162] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Commun. ACM*, 62(12):56–65, November 2019. ISSN 0001-0782. doi: 10.1145/3318162.

2469 [163] Georgios Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233--236, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. URL http://dl.acm.org/citation.cfm?id=2487085.2487132.

2473 [164] G. Grano, T. V. Titov, S. Panichella, and H. C. Gall. How high will it be? using machine learning models to predict branch coverage in automated testing. In *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pages 19--24, 2018. doi: 10.1109/MALTESQUE.2018.8368454.

2477 [165] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. Hybrid speech recognition with deep bidirectional lstm. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pages 273--278. IEEE, 2013.

2480 [166] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28 (10):2222--2232, 2017.

2483 [167] Hanna Grodzicka, Arkadiusz Ziobrowski, Zofia Łakomiak, Michał Kawa, and Lech Madeyski. *Code Smell Prediction Employing Machine Learning Meets Emerging Java Language Constructs*, pages 137--167. 2020. ISBN 978-3-030-34706-2. doi: 10.1007/978-3-030-34706-2_8.

2486 [168] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933--944, 2018. doi: 10.1145/3180155.3180167.

2489 [169] Thirupathi Guggulothu and S. A. Moiz. Code smell detection using multi-label classification approach. *Software Quality Journal*, pages 1--24, 2020.

Journal Pre-proof

2491 [170] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using
2492 examples. *Commun. ACM*, 55(8):97–105, aug 2012. ISSN 0001-0782. doi: 10.1145/2240236.
2493 2240260.

2494 [171] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan,
2495 Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations
2496 with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

2497 [172] Aakanshi Gupta, Bharti Suri, Vijay Kumar, and Pragyashree Jain. Extracting rules for vulner-
2498 abilities detection with static metrics using machine learning. *International Journal of System*
2499 *Assurance Engineering and Management*, 12:65--76, 2021.

2500 [173] Aakanshi Gupta, Bharti Suri, and Lakshay Lamba. Tracing bad code smells behavior using
2501 machine learning with software metrics. *Smart and Sustainable Intelligent Systems*, pages
2502 245--257, 2021.

2503 [174] H. Gupta, L. Kumar, and L. B. M. Neti. An empirical framework for code smell prediction using
2504 extreme learning machine*. In *2019 9th Annual Information Technology, Electromechanical*
2505 *Engineering and Microelectronics Conference (IEMECON)*, pages 189--195, 2019. doi: 10.1109/
2506 IEMECONX.2019.8877082.

2507 [175] Himanshu Gupta, Abhiram Anand Gulanikar, Lov Kumar, and Lalita Bhanu Murthy Neti. Em-
2508 pirical analysis on effectiveness of nlp methods for predicting code smell. In *International*
2509 *Conference on Computational Science and Its Applications*, pages 43--53. Springer, 2021.

2510 [176] Himanshu Gupta, Tanmay Girish Kulkarni, Lov Kumar, Lalita Bhanu Murthy Neti, and Aneesh
2511 Krishna. An empirical study on predictability of software code smell using deep learning mod-
2512 els. In *International Conference on Advanced Information Networking and Applications*, pages
2513 120--132. Springer, 2021.

2514 [177] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. DeepFix: Fixing common c
2515 language errors by deep learning. In *AAAI*, pages 1345--1351, 2017.

2516 [178] Rahul Gupta, Aditya Kanade, and Shirish Shevade. Deep reinforcement learning for syntactic
2517 error repair in student programs. *Proceedings of the AAAI Conference on Artificial Intelligence*,
2518 33:930--937, 07 2019. doi: 10.1609/aaai.v33i01.3301930.

2519 [179] Mouna Hadj-Kacem and Nadia Bouassida. A hybrid approach to detect code smells using
2520 deep learning. In *ENASE*, pages 137--146, 2018.

2521 [180] Mouna Hadj-Kacem and Nadia Bouassida. Deep representation learning for code smells
2522 detection using variational auto-encoder. In *2019 International Joint Conference on Neural*
2523 *Networks (IJCNN)*, pages 1--8. IEEE, 2019.

2524 [181] T. Hall and D. Bowes. The state of machine learning methodology in software fault prediction.
2525 In *2012 11th International Conference on Machine Learning and Applications*, volume 2, pages
2526 308--313, 2012. doi: 10.1109/ICMLA.2012.226.

2527 [182] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*.
2528 USA, 1977. ISBN 0444002057.

2529 [183] Muhammad Hammad, "Onder Babur, Hamid Abdul Basit, and Mark van den Brand. Clone-
2530 advisor: recommending code tokens and clone methods with deep learning and information
2531 retrieval. *PeerJ Computer Science*, 7:e737, 2021.

Journal Pre-proof

2532 [184] Awni Hammouri, Mustafa Hammad, Mohammad Alnabhan, and Fatima Alsarayrah. Software
2533 bug prediction using machine learning approach. *International Journal of Advanced Computer*
2534 *Science and Applications*, 9, 01 2018. doi: 10.14569/IJACSA.2018.090212.

2535 [185] S. Han, D. R. Wallace, and R. C. Miller. Code completion from abbreviated input. In *2009*
2536 *IEEE/ACM International Conference on Automated Software Engineering*, pages 332--343, 2009.
2537 doi: 10.1109/ASE.2009.64.

2538 [186] Sangmok Han, David R. Wallace, and Robert C. Miller. Code completion of multiple keywords
2539 from abbreviated input. *Automated Software Engg.*, 18(3–4):363–398, December 2011. ISSN
2540 0928-8910. doi: 10.1007/s10515-011-0083-2.

2541 [187] Hazim Hanif, Mohd Hairul Nizam Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and
2542 Nor Badrul Anuar. The rise of software vulnerability: Taxonomy of software vulnerabili-
2543 ties detection and machine learning approaches. *Journal of Network and Computer Appli-*
2544 *cations*, 179:103009, April 2021. ISSN 1084-8045. doi: 10.1016/j.jnca.2021.103009. URL
2545 https://www.sciencedirect.com/science/article/pii/S1084804521000369.

2546 [188] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. Improved automatic sum-
2547 marization of subroutines via attention to file context. In *Proceedings of the 17th International*
2548 *Conference on Mining Software Repositories*, pages 300--310, 2020.

2549 [189] Sakib Haque, Aakash Bansal, Lingfei Wu, and Collin McMillan. Action word prediction for
2550 neural source code summarization. In *2021 IEEE International Conference on Software Analysis,*
2551 *Evolution and Reengineering (SANER)*, pages 330--341. IEEE, 2021.

2552 [190] Mark Harman, Syed Islam, Yue Jia, Leandro L. Minku, Federica Sarro, and Komsan Srivisut.
2553 Less is more: Temporal fault predictive performance over multiple hadoop releases. In Claire
2554 Le Goues and Shin Yoo, editors, *Search-Based Software Engineering*, pages 240--246, Cham,
2555 2014. Springer International Publishing. ISBN 978-3-319-09940-8.

2556 [191] Vincent J. Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice
2557 for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of*
2558 *Software Engineering*, ESEC/FSE 2017, page 763–773, 2017. ISBN 9781450351058. doi: 10.
2559 1145/3106237.3106290.

2560 [192] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning
2561 type inference. ESEC/FSE 2018, page 152–162, 2018. ISBN 9781450355735. doi: 10.1145/
2562 3236024.3236051.

2563 [193] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-learning-guided selectively unsound
2564 static analysis. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE
2565 '17, page 519–529, 2017. ISBN 9781538638682. doi: 10.1109/ICSE.2017.54.

2566 [194] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: Distributed representations
2567 of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software*
2568 *Engineering*, ICSE '20, page 518–529, 2020. ISBN 9781450371216. doi: 10.1145/3377811.
2569 3380361.

2570 [195] Max Hort, Maria Kechagia, Federica Sarro, and Mark Harman. A survey of performance
2571 optimization for mobile applications. *IEEE Transactions on Software Engineering (TSE)*, 2021.

2572 [196] Yung-Tsung Hou, Yimeng Chang, Tsuhan Chen, Chi-Sung Laih, and Chia-Mei Chen. Malicious
2573 web content detection by machine learning. *Expert Systems with Applications*, 37(1):55 -- 60,
2574 2010. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2009.05.023.

Journal Pre-proof

Journal Pre-proof

2575 [197] Gang Hu, Linjie Zhu, and Junfeng Yang. Appflow: Using machine learning to synthesize ro-
2576 bust, reusable ui tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software*
2577 *Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE
2578 2018, page 269–282, 2018. ISBN 9781450355735. doi: 10.1145/3236024.3236055.

2579 [198] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation. In *2018 IEEE/ACM 26th*
2580 *International Conference on Program Comprehension (ICPC)*, pages 200--20010, 2018.

2581 [199] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with trans-
2582 ferred api knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on*
2583 *Artificial Intelligence, IJCAI-18*, pages 2269--2275. International Joint Conferences on Artificial
2584 Intelligence Organization, 7 2018. doi: 10.24963/ijcai.2018/314.

2585 [200] Yuan Huang, Xinyu Hu, Nan Jia, Xiangping Chen, Zibin Zheng, and Xiapu Luo. Commtpst:
2586 Deep learning source code for commenting positions prediction. *Journal of Systems and Soft-*
2587 *ware*, 170:110754, 2020. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2020.110754.

2588 [201] Yuan Huang, Shaohao Huang, Huanchao Chen, Xiangping Chen, Zibin Zheng, Xiapu Luo, Nan
2589 Jia, Xinyu Hu, and Xiaocong Zhou. Towards automatically generating block comments for
2590 code snippets. *Information and Software Technology*, 127:106373, 2020.

2591 [202] Yasir Hussain, Zhiqiu Huang, Yu Zhou, and Senzhang Wang. Codegru: Context-aware deep
2592 learning with gated recurrent unit for source code modeling. *Information and Software Tech-*
2593 *nology*, 125:106309, 2020. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2020.106309.

2594 [203] J. Ivers, I. Ozkaya, and R. L. Nord. Can ai close the design-code abstraction gap? In *2019 34th*
2595 *IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages
2596 122--125, 2019. doi: 10.1109/ASEW.2019.00041.

2597 [204] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source
2598 code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Asso-*
2599 *ciation for Computational Linguistics (Volume 1: Long Papers)*, pages 2073--2083, August 2016.
2600 doi: 10.18653/v1/P16-1195.

2601 [205] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. Con-
2602 trastive code representation learning. In *Proceedings of the 2021 Conference on Empirical*
2603 *Methods in Natural Language Processing*. Association for Computational Linguistics, 2021. doi:
2604 10.18653/v1/2021.emnlp-main.482.

2605 [206] Shivani Jain and Anju Saha. Improving performance with hybrid feature selection and ensem-
2606 ble machine learning techniques for code smell detection. *Science of Computer Programming*,
2607 212:102713, 2021.

2608 [207] T. Ji, J. Pan, L. Chen, and X. Mao. Identifying supplementary bug-fix commits. In *2018 IEEE 42nd*
2609 *Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 184--193,
2610 2018. doi: 10.1109/COMPSAC.2018.00031.

2611 [208] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. An unsupervised approach for discov-
2612 ering relevant tutorial fragments for apis. In *2017 IEEE/ACM 39th International Conference on*
2613 *Software Engineering (ICSE)*, pages 38--48. IEEE, 2017.

2614 [209] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program
2615 repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT*
2616 *International Symposium on Software Testing and Analysis*, ISSTA 2018, page 298–309, 2018.
2617 ISBN 9781450356992. doi: 10.1145/3213846.3213871.

Journal Pre-proof

Journal Pre-proof

2618 [210] Lin Jiang, Hui Liu, and He Jiang. Machine learning based recommendation of method names:
2619 How far are we. In *Proceedings of the 34th IEEE/ACM International Conference on Automated*
2620 *Software Engineering*, ASE '19, page 602–614, 2019. ISBN 9781728125084. doi: 10.1109/ASE.
2621 2019.00062.

2622 [211] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation
2623 for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software*
2624 *Engineering (ICSE)*, pages 1161--1173. IEEE, 2021.

2625 [212] S. Jiang, A. Armaly, and C. McMillan. Automatically generating commit messages from diffs
2626 using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Auto-*
2627 *mated Software Engineering (ASE)*, pages 135--146, 2017. doi: 10.1109/ASE.2017.8115626.

2628 [213] Shuyao Jiang. Boosting neural commit message generation with code semantic analysis. In
2629 *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages
2630 1280--1282. IEEE, 2019.

2631 [214] Siyuan Jiang and Collin McMillan. Towards automatic generation of short summaries of com-
2632 mits. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages
2633 320--323. IEEE, 2017.

2634 [215] Gong Jie, Kuang Xiao-Hui, and Liu Qiang. Survey on software vulnerability analysis method
2635 based on machine learning. In *2016 IEEE First International Conference on Data Science in*
2636 *Cyberspace (DSC)*, pages 642--647, 2016. doi: 10.1109/DSC.2016.33.

2637 [216] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and
2638 Mark Harman. The importance of accounting for real-world labelling when predicting soft-
2639 ware vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Soft-*
2640 *ware Engineering Conference and Symposium on the Foundations of Software Engineering*, ES-
2641 EC/FSE 2019, page 695–705, New York, NY, USA, 2019. Association for Computing Machinery.
2642 ISBN 9781450355728. doi: 10.1145/3338906.3338941. URL https://doi.org/10.1145/3338906.
2643 3338941.

2644 [217] Xiao-Yuan Jing, Shi Ying, Zhi-Wu Zhang, Shan-Shan Wu, and Jin Liu. Dictionary learning based
2645 software defect prediction. In *Proceedings of the 36th international conference on software*
2646 *engineering*, pages 414--423, 2014.

2647 [218] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to
2648 enable controlled testing studies for java programs. In *Proceedings of the 2014 International*
2649 *Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA,
2650 2014. Association for Computing Machinery. ISBN 9781450326452. doi: 10.1145/2610384.
2651 2628055. URL https://doi.org/10.1145/2610384.2628055.

2652 [219] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and eval-
2653 uating contextual embedding of source code. In Hal Daumé III and Aarti Singh, editors,
2654 *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Pro-*
2655 *ceedings of Machine Learning Research*, pages 5110--5121. PMLR, 13--18 Jul 2020. URL https:
2656 //proceedings.mlr.press/v119/kanade20a.html.

2657 [220] Hong Jin Kang, Tegawendé F. Bissyandé, and David Lo. Assessing the generalizability of
2658 code2vec token embeddings. In *2019 34th IEEE/ACM International Conference on Automated*
2659 *Software Engineering (ASE)*, pages 1--12, 2019. doi: 10.1109/ASE.2019.00011.

2660 [221] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes.
2661 Big code != big vocabulary: Open-vocabulary models for source code. In *Proceedings of the*

Journal Pre-proof

2662 *ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1073–1085,
2663 2020. ISBN 9781450371216. doi: 10.1145/3377811.3380342.

2664 [222] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent net-
2665 works. *arXiv preprint arXiv:1506.02078*, 2015.

2666 [223] A. Kaur, S. Jain, and S. Goel. A support vector machine based approach for code smell de-
2667 tection. In *2017 International Conference on Machine Learning and Data Science (MLDS)*, pages
2668 9--14, 2017. doi: 10.1109/MLDS.2017.8.

2669 [224] Arvinder Kaur and Kamaldeep Kaur. An empirical study of robustness and stability of ma-
2670 chine learning classifiers in software defect prediction. In *Advances in intelligent informatics*,
2671 pages 383--397. Springer, 2015.

2672 [225] Arvinder Kaur, Kamaldeep Kaur, and Deepti Chopra. An empirical study of software en-
2673 tropy based bug prediction using machine learning. *International Journal of System Assur-*
2674 *ance Engineering and Management*, 8(2):599--616, November 2017. ISSN 0976-4348. doi:
2675 10.1007/s13198-016-0479-2.

2676 [226] Inderpreet Kaur and Arvinder Kaur. A novel four-way approach designed with ensemble
2677 feature selection for code smell detection. *IEEE Access*, 9:8695--8707, 2021.

2678 [227] Patrick Keller, Abdoul Kader Kaboré, Laura Plein, Jacques Klein, Yves Le Traon, and
2679 Tegawendé F. Bissyandé. What you see is what it means! semantic representation learn-
2680 ing of code based on visualization and transfer learning. *ACM Trans. Softw. Eng. Methodol.*, 31
2681 (2), dec 2021. ISSN 1049-331X. doi: 10.1145/3485135. URL https://doi.org/10.1145/3485135.

2682 [228] Muhammad Noman Khalid, Humera Farooq, Muhammad Iqbal, Muhammad Talha Alam,
2683 and Kamran Rasheed. Predicting Web Vulnerabilities in Web Applications Based on Machine
2684 Learning. In Imran Sarwar Bajwa, Fairouz Kamareddine, and Anna Costa, editors, *Intelligent*
2685 *Technologies and Applications*, Communications in Computer and Information Science, pages
2686 473--484, Singapore, 2019. Springer. ISBN 9789811360527. doi: 10.1007/978-981-13-6052-7_
2687 41.

2688 [229] Bilal Khan, Danish Iqbal, and Sher Badshah. Cross-project software fault prediction using
2689 data leveraging technique to improve software quality. In *Proceedings of the Evaluation and*
2690 *Assessment in Software Engineering*, EASE '20, page 434–438, 2020. ISBN 9781450377317. doi:
2691 10.1145/3383219.3383281.

2692 [230] J. Kim, M. Kwon, and S. Yoo. Generating test input with deep reinforcement learning. In *2018*
2693 *IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)*, pages 51--58,
2694 2018.

2695 [231] Junae Kim, David Hubczenko, and Paul Montague. Towards attention based vulnerability
2696 discovery using source code representation. In Igor V. Tetko, Věra Kůrková, Pavel Karpov,
2697 and Fabian Theis, editors, *Artificial Neural Networks and Machine Learning -- ICANN 2019: Text*
2698 *and Time Series*, pages 731--746, 2019. ISBN 978-3-030-30490-4.

2699 [232] Sangwoo Kim, Seokmyung Hong, Jaesang Oh, and Heejo Lee. Obfuscated vba macro detec-
2700 tion using machine learning. In *2018 48th Annual IEEE/IFIP International Conference on Depend-*
2701 *able Systems and Networks (DSN)*, pages 490--501, 2018. doi: 10.1109/DSN.2018.00057.

2702 [233] Patrick Knab, Martin Pinzger, and Abraham Bernstein. Predicting defect densities in source
2703 code files with decision tree learners. In *Proceedings of the 2006 International Workshop on*
2704 *Mining Software Repositories*, MSR '06, page 119–125, 2006. ISBN 1595933972. doi: 10.1145/
2705 1137983.1138012.

Journal Pre-proof

2706 [234] Yasemin Kosker, Burak Turhan, and Ayse Bener. An expert system for determining candidate
2707 software classes for refactoring. *Expert Systems with Applications*, 36(6):10000 -- 10003, 2009.
2708 ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2008.12.066.

2709 [235] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. Building im-
2710 plicit vector representations of individual coding style. In *Proceedings of the IEEE/ACM 42nd
2711 International Conference on Software Engineering Workshops*, ICSEW'20, page 117–124, 2020.
2712 ISBN 9781450379632. doi: 10.1145/3387940.3391494.

2713 [236] Rrezarta Krasniqi and Jane Cleland-Huang. Enhancing source code refactoring detection
2714 with explanations from commit messages. In *2020 IEEE 27th International Conference on
2715 Software Analysis, Evolution and Reengineering (SANER)*, pages 512--516, 2020. doi: 10.1109/
2716 SANER48275.2020.770_Krasniqi2020.

2717 [237] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep
2718 convolutional neural networks. In *Advances in neural information processing systems*, pages
2719 1097--1105, 2012.

2720 [238] Jorrit Kronjee, Arjen Hommersom, and Harald Vranken. Discovering software vulnerabilities
2721 using data-flow analysis and machine learning. In *Proceedings of the 13th International Con-
2722 ference on Availability, Reliability and Security*, ARES 2018, 2018. ISBN 9781450364485. doi:
2723 10.1145/3230833.3230856.

2724 [239] L. Kumar and A. Sureka. Application of lssvm and smote on seven open source projects for
2725 predicting refactoring at class level. In *2017 24th Asia-Pacific Software Engineering Conference
2726 (APSEC)*, pages 90--99, 2017. doi: 10.1109/APSEC.2017.15.

2727 [240] L. Kumar and A. Sureka. An empirical analysis on web service anti-pattern detection using a
2728 machine learning framework. In *2018 IEEE 42nd Annual Computer Software and Applications
2729 Conference (COMPSAC)*, volume 01, pages 2--11, 2018. doi: 10.1109/COMPSAC.2018.00010.

2730 [241] Lov Kumar, Santanu Kumar Rath, and Ashish Sureka. Using source code metrics to predict
2731 change-prone web services: A case-study on ebay services. In *2017 IEEE workshop on machine
2732 learning techniques for software quality evaluation (MaLTeSQuE)*, pages 1--7. IEEE, 2017.

2733 [242] Lov Kumar, Shashank Mouli Satapathy, and Lalita Bhanu Murthy. Method level refactoring
2734 prediction on five open source java projects using machine learning techniques. In *Proceed-
2735 ings of the 12th Innovations on Software Engineering Conference (Formerly Known as India Soft-
2736 ware Engineering Conference)*, ISEC'19, 2019. ISBN 9781450362153. doi: 10.1145/3299771.
2737 3299777.

2738 [243] Pradeep Kumar and Yogesh Singh. Assessment of software testing time using soft computing
2739 techniques. *SIGSOFT Softw. Eng. Notes*, 37(1):1–6, January 2012. ISSN 0163-5948. doi: 10.1145/
2740 2088883.2088895.

2741 [244] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. Recommendation
2742 of move method refactoring using path-based representation of code. In *Proceedings of the
2743 IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page
2744 315–322, 2020. ISBN 9781450379632. doi: 10.1145/3387940.3392191.

2745 [245] H. Lal and G. Pahwa. Code review analysis of software system using machine learning tech-
2746 niques. In *2017 11th International Conference on Intelligent Systems and Control (ISCO)*, pages
2747 8--13, 2017. doi: 10.1109/ISCO.2017.7855962.

2748 [246] Issam H Laradji, Mohammad Alshayeb, and Lahouari Ghouti. Software defect prediction
2749 using ensemble learning on selected features. *Information and Software Technology*, 58:
2750 388--402, 2015.

2751 [247] Michael R. Law and Karen A. Grépin. Is newer always better? Re-evaluating the benefits of
2752 newer pharmaceuticals. *Journal of Health Economics*, 29(5):743--750, September 2010. ISSN
2753 1879-1646. doi: 10.1016/j.jhealeco.2010.06.007.

2754 [248] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. Deep learning for source code modeling
2755 and generation: Models, applications, and challenges. *ACM Comput. Surv.*, 53(3), June 2020.
2756 ISSN 0360-0300. doi: 10.1145/3383458.

2757 [249] X. D. Le, T. B. Le, and D. Lo. Should fixing these failures be delegated to automated program
2758 repair? In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*,
2759 pages 427--437, 2015. doi: 10.1109/ISSRE.2015.7381836.

2760 [250] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu,
2761 Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for au-
2762 tomated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236--1256,
2763 2015. doi: 10.1109/TSE.2015.2454513.

2764 [251] Alexander LeClair and Collin McMillan. Recommendations for datasets for source code sum-
2765 marization, 2019.

2766 [252] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural
2767 language summaries of program subroutines. In *Proceedings of the 41st International Confer-
2768 ence on Software Engineering*, ICSE '19, page 795–806, 2019. doi: 10.1109/ICSE.2019.00087.

2769 [253] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. Improved code summa-
2770 rization via a graph neural network. In *Proceedings of the 28th International Conference
2771 on Program Comprehension*, ICPC '20, page 184–195, 2020. ISBN 9781450379588. doi:
2772 10.1145/3387904.3389268.

2773 [254] Alexander LeClair, Aakash Bansal, and Collin McMillan. Ensemble models for neural source
2774 code summarization of subroutines. In *2021 IEEE International Conference on Software Main-
2775 tenance and Evolution (ICSME)*, pages 286--297. IEEE, 2021.

2776 [255] Song-Mi Lee, Sang Min Yoon, and Heeryon Cho. Human activity recognition from accelerom-
2777 eter data using convolutional neural network. In *Big Data and Smart Computing (BigComp),
2778 2017 IEEE International Conference on*, pages 131--134. IEEE, 2017.

2779 [256] Suin Lee, Youngseok Lee, Chan-Gun Lee, and Honguk Woo. Deep learning-based logging
2780 recommendation using merged code representation. In Hyuncheol Kim and Kuinam J. Kim,
2781 editors, *IT Convergence and Security*, pages 49--53, 2021. ISBN 978-981-15-9354-3.

2782 [257] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program
2783 synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Confer-
2784 ence on Programming Language Design and Implementation*, PLDI 2018, page 436–449, 2018.
2785 ISBN 9781450356985. doi: 10.1145/3192366.3192410.

2786 [258] Stanislav Levin and Amiram Yehudai. Boosting automatic commit classification into mainte-
2787 nance activities by utilizing source code changes. In *Proceedings of the 13th International Con-
2788 ference on Predictive Models and Data Analytics in Software Engineering*, pages 97--106, 2017.

2789 [259] Tomasz Lewowski and Lech Madeyski. Code smells detection using artificial intelligence tech-
2790 niques: A business-driven systematic review. *Developments in Information I& Knowledge Man-
2791 agement for Business Applications*, pages 285--319, 2022.

Journal Pre-proof

Journal Pre-proof

2792 [260] Boao Li, Meng Yan, Xin Xia, Xing Hu, Ge Li, and David Lo. Deepcommenter: A deep code
2793 comment generation tool with hybrid lexical and syntactical information. In *Proceedings*
2794 *of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium*
2795 *on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1571–1575, 2020. ISBN
2796 9781450370431. doi: 10.1145/3368089.3417926.

2797 [261] Daoyuan Li, Li Li, Dongsun Kim, Tegawendé F Bissyandé, David Lo, and Yves Le Traon. Watch
2798 out for this commit! a study of influential software changes. *Journal of Software: Evolution*
2799 *and Process*, 31(12):e2181, 2019.

2800 [262] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. Editsum: A retrieve-and-edit framework
2801 for source code summarization. In *2021 36th IEEE/ACM International Conference on Automated*
2802 *Software Engineering (ASE)*, pages 155--166. IEEE, 2021.

2803 [263] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolu-
2804 tional neural network. In *2017 IEEE International Conference on Software Quality, Reliability*
2805 *and Security (QRS)*, pages 318--328. IEEE, 2017.

2806 [264] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention
2807 and pointer networks. In *Proceedings of the 27th International Joint Conference on Artificial*
2808 *Intelligence*, IJCAI'18, page 4159–25, 2018. ISBN 9780999241127.

2809 [265] M. Li, H. Zhang, Rongxin Wu, and Z. Zhou. Sample-based software defect prediction with
2810 active and semi-supervised learning. *Automated Software Engineering*, 19:201--230, 2011.

2811 [266] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. Improving bug detection via
2812 context-based code representation learning and attention-based neural networks. *Proc. ACM*
2813 *Program. Lang.*, 3(OOPSLA), October 2019. doi: 10.1145/3360588.

2814 [267] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dlfix: Context-based code transformation learning
2815 for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference*
2816 *on Software Engineering*, ICSE '20, page 602–614, 2020. ISBN 9781450371216. doi: 10.1145/
2817 3377811.3380345.

2818 [268] Yi Li, Shaohua Wang, and Tien N Nguyen. A context-based automated approach for method
2819 name consistency checking and suggestion. In *2021 IEEE/ACM 43rd International Conference*
2820 *on Software Engineering (ICSE)*, pages 574--586. IEEE, 2021.

2821 [269] Yuancheng Li, Rong Ma, and Runhai Jiao. A hybrid malicious code detection method based
2822 on deep learning. *International journal of security and its applications*, 9:205--216, 2015.

2823 [270] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond,
2824 Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code
2825 generation with alphacode. *Science*, 378(6624):1092--1097, 2022.

2826 [271] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin. A comparative study of deep learning-based
2827 vulnerability detection system. *IEEE Access*, 7:103184--103197, 2019. doi: 10.1109/ACCESS.
2828 2019.2930578.

2829 [272] Chen Liang, Jonathan Berant, Quoc V. Le, Kenneth D. Forbus, and N. Lao. Neural symbolic
2830 machines: Learning semantic parsers on freebase with weak supervision. In *ACL*, 2017.

2831 [273] Hongliang Liang, Yue Yu, Lin Jiang, and Zhuosi Xie. Seml: A semantic lstm model for software
2832 defect prediction. *IEEE Access*, 7:83812--83824, 2019.

2833 [274] H. Lim. Applying code vectors for presenting software features in machine learning. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 803--804, 2018. doi: 10.1109/COMPSAC.2018.00128.

2836 [275] R. Lima, A. M. R. da Cruz, and J. Ribeiro. Artificial intelligence applied to software testing: A literature review. In *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1--6, 2020. doi: 10.23919/CISTI49556.2020.9141124.

2839 [276] BO LIN, SHANGWEN WANG, MING WEN, and XIAOGUANG MAO. Context-aware code change embedding for better patch correctness assessment. *J. ACM*, 1(1), 2021.

2841 [277] Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. Improving code summarization with block-wise abstract syntax tree splitting. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 184--195. IEEE, 2021.

2844 [278] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Yang Xiang, Olivier De Vel, and Paul Montague. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Transactions on Industrial Informatics*, 14(7):3289--3297, 2018. doi: 10.1109/TII.2018.2821768.

2847 [279] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Yang Xiang, Olivier De Vel, and Paul Montague. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Transactions on Industrial Informatics*, 14(7):3289--3297, 2018. doi: 10.1109/TII.2018.2821768.

2850 [280] Guanjun Lin, Wei Xiao, Jun Zhang, and Yang Xiang. Deep Learning-Based Vulnerable Function Detection: A Benchmark. In Jianying Zhou, Xiapu Luo, Qingni Shen, and Zhen Xu, editors, *Information and Communications Security*, Lecture Notes in Computer Science, pages 219--232, Cham, 2020. Springer International Publishing. ISBN 978-3-030-41579-2. doi: 10.1007/978-3-030-41579-2_13.

2855 [281] Junhao Lin and Lu Lu. Semantic feature learning via dual sequences for defect prediction. *IEEE Access*, 9:13112--13124, 2021.

2857 [282] Chunyang Ling, Zeqi Lin, Yanzhen Zou, and Bing Xie. Adaptive deep code search. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, pages 48–--59. Association for Computing Machinery, 2020. ISBN 9781450379588. doi: 10.1145/3387904.3389278.

2861 [283] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation, 2016. URL https://arxiv.org/abs/1603.06744.

2864 [284] E. Linstead, C. Lopes, and P. Baldi. An application of latent dirichlet allocation to analyzing software evolution. In *2008 Seventh International Conference on Machine Learning and Applications*, pages 813--818, 2008. doi: 10.1109/ICMLA.2008.47.

2867 [285] Bohong Liu, Tao Wang, Xunhui Zhang, Qiang Fan, Gang Yin, and Jinsheng Deng. A neural-network based code summarization approach by using source code and its call dependencies. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, Internetware '19, 2019. ISBN 9781450377010. doi: 10.1145/3361242.3362774.

2871 [286] Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. On the replicability and reproducibility of deep learning in software engineering, 2020.

2873 [287] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. R2fix: Automatically generating bug fixes from bug reports. In *2013 IEEE Sixth international conference on software testing, verification and validation*, pages 282--291. IEEE, 2013.

Journal Pre-proof

2876 [288] F. Liu, G. Li, Y. Zhao, and Z. Jin. Multi-task learning based pre-trained language model for code
2877 completion. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering*
2878 *(ASE)*, pages 473--485, 2020.

2879 [289] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. A self-attentional neural architecture
2880 for code completion with multi-task learning. In *Proceedings of the 28th International Con-*
2881 *ference on Program Comprehension*, ICPC '20, page 37–47, 2020. ISBN 9781450379588. doi:
2882 10.1145/3387904.3389261.

2883 [290] Hui Liu, Jiahao Jin, Zhifeng Xu, Yifan Bu, Yanzhen Zou, and Lu Zhang. Deep learning based
2884 code smell detection. *IEEE Transactions on Software Engineering*, 2019.

2885 [291] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim,
2886 Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite
2887 based program repair: A systematic assessment of 16 automated repair systems for Java
2888 programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineer-*
2889 *ing*, ICSE '20, pages 615?--627, 2020. ISBN 9781450371216. doi: 10.1145/3377811.3380338.

2890 [292] Shangqing Liu, Cuiyun Gao, Sen Chen, Nie Lun Yiu, and Yang Liu. Atom: Commit message
2891 generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software*
2892 *Engineering*, 2020.

2893 [293] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation
2894 of syntax valid c programs for fuzz testing. *Proceedings of the AAAI Conference on Artificial*
2895 *Intelligence*, 33(01):1044--1051, Jul. 2019. doi: 10.1609/aaai.v33i01.33011044.

2896 [294] Yang Liu. Fine-tune bert for extractive summarization, 2019. URL https://arxiv.org/abs/1903.
2897 10318.

2898 [295] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-
2899 machine-translation-based commit message generation: How far are we? In *Proceedings of*
2900 *the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page
2901 373–384, 2018. ISBN 9781450359375. doi: 10.1145/3238147.3238190.

2902 [296] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. Automatic generation
2903 of pull request descriptions. In *2019 34th IEEE/ACM International Conference on Automated*
2904 *Software Engineering (ASE)*, pages 176--188. IEEE, 2019.

2905 [297] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Pro-*
2906 *ceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Lan-*
2907 *guages*, POPL '16, page 298–312, 2016. ISBN 9781450335492. doi: 10.1145/2837614.2837617.

2908 [298] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi. UCI source code data sets, 2010. URL
2909 http://www.ics.uci.edu/$\sim$lopes/datasets/.

2910 [299] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang.
2911 Can automated program repair refine fault localization? a unified debugging approach. In
2912 *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*,
2913 pages 75--87, 2020.

2914 [300] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin
2915 Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou,
2916 Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng,
2917 Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code
2918 understanding and generation, 2021. URL https://arxiv.org/abs/2102.04664.

Journal Pre-proof

2919 [301] Yangyang Lu, Zelong Zhao, Ge Li, and Zhi Jin. Learning to generate comments for api-based
2920 code snippets. In *Software Engineering and Methodology for Emerging Domains*, pages 3--14.
2921 Springer, 2017.

2922 [302] Frederico Caram Luiz, Bruno Rafael de Oliveira Rodrigues, and Fernando Silva Parreiras. Ma-
2923 chine learning techniques for code smells detection: An empirical experiment on a highly im-
2924 balanced setup. In *Proceedings of the XV Brazilian Symposium on Information Systems*, SBSI'19,
2925 2019. ISBN 9781450372374. doi: 10.1145/3330204.3330275.

2926 [303] Savanna Lujan, Fabiano Pecorelli, Fabio Palomba, Andrea De Lucia, and Valentina Lenar-
2927 duzzi. A preliminary study on the adequacy of static analysis warnings with respect to code
2928 smell prediction. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-*
2929 *Learning Techniques for Software-Quality Evaluation*, MaLTeSQuE 2020, page 1–6, 2020. ISBN
2930 9781450381246. doi: 10.1145/3416505.3423559.

2931 [304] Minh-Thang Luong, Eugene Brevdo, and Rui Zhao. Neural machine translation (seq2seq)
2932 tutorial. *https://github.com/tensorflow/nmt*, 2017.

2933 [305] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Co-
2934 CoNuT: Combining context-aware neural translation models using ensemble for program
2935 repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing*
2936 *and Analysis*, ISSTA 2020, page 101–114, 2020. ISBN 9781450380089. doi: 10.1145/3395363.
2937 3397369.

2938 [306] Michael R. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill, Inc., USA,
2939 1996. ISBN 0070394008.

2940 [307] Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. Transfer learning for cross-company
2941 software defect prediction. *Information and Software Technology*, 54(3):248 -- 256, 2012. ISSN
2942 0950-5849. doi: https://doi.org/10.1016/j.infsof.2011.09.007.

2943 [308] Yuzhan Ma, Sarah Fakhoury, Michael Christensen, Venera Arnaoudova, Waleed Zogaan, and
2944 Mehdi Mirakhorli. Automatic classification of software artifacts in open-source applications.
2945 In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18,
2946 page 414–425, 2018. ISBN 9781450357166. doi: 10.1145/3196398.3196446.

2947 [309] Z. Ma, H. Ge, Y. Liu, M. Zhao, and J. Ma. A combination method for android malware detection
2948 based on control flow graphs and machine learning algorithms. *IEEE Access*, 7:21235--21245,
2949 2019. doi: 10.1109/ACCESS.2019.2896003.

2950 [310] Chris J. Maddison and Daniel Tarlow. Structured generative models of natural source code. In
2951 *Proceedings of the 31st International Conference on International Conference on Machine Learn-*
2952 *ing - Volume 32*, ICML'14, page II–649–II–657, 2014.

2953 [311] Janaki T. Madhavan and E. James Whitehead. Predicting buggy changes inside an integrated
2954 development environment. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technol-*
2955 *ogy EXchange*, eclipse '07, page 36–40, 2007. ISBN 9781605580159. doi: 10.1145/1328279.
2956 1328287.

2957 [312] Anas Mahmoud and Gary Bradshaw. Semantic topic models for source code analysis. *Em-*
2958 *pirical Software Engineering*, 22(4):1965--2000, 2017.

2959 [313] Amirabbas Majd, Mojtaba Vahidi-Asl, Alireza Khalilian, Pooria Poorsarvi-Tehrani, and Hassan
2960 Haghighi. SLDeep: Statement-level software defect prediction using deep-learning model on
2961 static code features. *Expert Systems with Applications*, 147:113156, 2020. ISSN 0957-4174. doi:
2962 https://doi.org/10.1016/j.eswa.2019.113156.

Journal Pre-proof

2963 [314] R. Malhotra and Rupender Jangra. Prediction & assessment of change prone classes us-
2964 ing statistical & machine learning techniques. *Journal of Information Processing Systems*, 13:
2965 778--804, 01 2017. doi: 10.3745/JIPS.04.0013.

2966 [315] R. Malhotra, L. Bahl, S. Sehgal, and P. Priya. Empirical comparison of machine learn-
2967 ing algorithms for bug prediction in open source software. In *2017 International Confer-*
2968 *ence on Big Data Analytics and Computational Intelligence (ICBDAC)*, pages 40--45, 2017. doi:
2969 10.1109/ICBDACI.2017.8070806.

2970 [316] Ruchika Malhotra. Comparative analysis of statistical and machine learning methods for
2971 predicting faulty modules. *Applied Soft Computing*, 21:286 -- 297, 2014. ISSN 1568-4946. doi:
2972 https://doi.org/10.1016/j.asoc.2014.03.032.

2973 [317] Ruchika Malhotra and Ankita Jain. Fault prediction using statistical and machine learning
2974 methods for improving software quality. *Journal of Information Processing Systems*, 8(2):
2975 241--262, 2012.

2976 [318] Ruchika Malhotra and Shine Kamal. An empirical study to investigate oversampling meth-
2977 ods for improving software defect prediction using imbalanced data. *Neurocomputing*, 343:
2978 120--140, 2019.

2979 [319] Ruchika Malhotra and Megha Khanna. Investigation of relationship between object-oriented
2980 metrics and change proneness. *International Journal of Machine Learning and Cybernetics*, 4
2981 (4):273--286, 2013.

2982 [320] Ruchika Malhotra and Yogesh Singh. On the applicability of machine learning techniques for
2983 object-oriented software fault prediction. *Software Engineering: An International Journal*, 1, 01
2984 2011.

2985 [321] Ruchika Malhotra[1] and Anuradha Chug. Software maintainability prediction using machine
2986 learning algorithms. *Software engineering: an international Journal (SeiJ)*, 2(2), 2012.

2987 [322] R. S. Malik, J. Patra, and M. Pradel. Nl2type: Inferring javascript function types from natural
2988 language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering*
2989 *(ICSE)*, pages 304--315, 2019. doi: 10.1109/ICSE.2019.00045.

2990 [323] C Manjula and Lilly Florence. Deep neural network based hybrid approach for software
2991 defect prediction using software metrics. *Cluster Computing*, 22(4):9847--9863, 2019.

2992 [324] Richard VR Mariano, Geanderson E dos Santos, Markos V de Almeida, and Wladmir C Brand
2993 textasciitilde ao. Feature changes in source code for commit classification into maintenance
2994 activities. In *2019 18th IEEE International Conference On Machine Learning And Applications*
2995 *(ICMLA)*, pages 515--518. IEEE, 2019.

2996 [325] Richard VR Mariano, Geanderson E dos Santos, and Wladmir Cardoso Brandao. Improve
2997 classification of commits maintenance activities with quantitative changes in source code.
2998 2021.

2999 [326] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of
3000 java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Reposito-*
3001 *ries (MSR)*, pages 505--509. IEEE, 2021.

3002 [327] Roni Mateless, Daniel Rejabek, Oded Margalit, and Robert Moskovitch. Decompiled APK
3003 based malicious code classification. *Future Generation Computer Systems*, 110:135--147,
3004 2020. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2020.03.052. URL https://www.
3005 sciencedirect.com/science/article/pii/S0167739X19325129.

Journal Pre-proof

3006 [328] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):
3007 308--320, 1976.

3008 [329] Mary L. McHugh. Interrater reliability: the kappa statistic. *Biochemia Medica*, 22:276 -- 282,
3009 2012.

3010 [330] Iberia Medeiros, Nuno F. Neves, and Miguel Correia. Securing energy metering software with
3011 automatic source code correction. In *2013 11th IEEE International Conference on Industrial*
3012 *Informatics (INDIN)*, jul 2013. doi: 10.1109/indin.2013.6622969.

3013 [331] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. Automatic detection and correction of
3014 web application vulnerabilities using data mining to predict false positives. In *Proceedings*
3015 *of the 23rd International Conference on World Wide Web*, WWW '14, page 63–74, 2014. ISBN
3016 9781450327442. doi: 10.1145/2566486.2568024.

3017 [332] Ibéria Medeiros, Nuno Neves, and Miguel Correia. Detecting and removing web applica-
3018 tion vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, 65(1):
3019 54--69, 2016. doi: 10.1109/TR.2015.2457411.

3020 [333] Na Meng, Zijian Jiang, and Hao Zhong. Classifying code commits with convolutional neural
3021 networks. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1--8. IEEE,
3022 2021.

3023 [334] Omar Meqdadi, Nouh Alhindawi, Jamal Alsakran, Ahmad Saifan, and Hatim Migdadi. Min-
3024 ing software repositories for adaptive change commits using machine learning techniques.
3025 *Information and Software Technology*, 109:80 -- 91, 2019. ISSN 0950-5849. doi: https:
3026 //doi.org/10.1016/j.infsof.2019.01.008.

3027 [335] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. Deep-
3028 Delta: Learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint*
3029 *Meeting on European Software Engineering Conference and Symposium on the Foundations*
3030 *of Software Engineering*, ESEC/FSE 2019, page 925–936, 2019. ISBN 9781450355728. doi:
3031 10.1145/3338906.3340455.

3032 [336] Mohammad Y. Mhawish and Manjari Gupta. Predicting code smells and analysis of predic-
3033 tions: Using machine learning techniques and software metrics. *J. Comput. Sci. Technol.*, 35:
3034 1428--1445, 2020.

3035 [337] Nikola Milosevic, Ali Dehghantanha, and Kim-Kwang Raymond Choo. Machine learning aided
3036 android malware classification. *Computers & Electrical Engineering*, 61:266 -- 274, 2017. ISSN
3037 0045-7906. doi: https://doi.org/10.1016/j.compeleceng.2017.02.013.

3038 [338] Robert Moskovitch, Nir Nissim, and Yuval Elovici. Malicious code detection using active learn-
3039 ing. In Francesco Bonchi, Elena Ferrari, Wei Jiang, and Bradley Malin, editors, *Privacy, Security,*
3040 *and Trust in KDD*, pages 74--91, 2009. ISBN 978-3-642-01718-6.

3041 [339] G. Mostaeen, J. Svajlenko, B. Roy, C. K. Roy, and K. A. Schneider. [research paper] on the use
3042 of machine learning techniques towards the design of cloud based automatic code clone
3043 validation tools. In *2018 IEEE 18th International Working Conference on Source Code Analysis*
3044 *and Manipulation (SCAM)*, pages 155--164, 2018. doi: 10.1109/SCAM.2018.00025.

3045 [340] Golam Mostaeen, Jeffrey Svajlenko, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider.
3046 Clonecognition: Machine learning based code clone validation tool. In *Proceedings of the*
3047 *2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium*
3048 *on the Foundations of Software Engineering*, ESEC/FSE 2019, page 1105–1109, 2019. ISBN
3049 9781450355728. doi: 10.1145/3338906.3341182.

Journal Pre-proof

3050 [341] Golam Mostaeen, Banani Roy, Chanchal K. Roy, Kevin Schneider, and Jeffrey Svajlenko. A
3051 machine learning based framework for code clone validation. *Journal of Systems and Software*,
3052 169:110686, 2020. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2020.110686.

3053 [342] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree
3054 structures for programming language processing. In *Proceedings of the Thirtieth AAAI Confer-*
3055 *ence on Artificial Intelligence*, AAAI'16, page 1287–1293, 2016.

3056 [343] Dana Movshovitz-Attias and William Cohen. Natural language models for predicting pro-
3057 gramming comments. *ACL 2013 - 51st Annual Meeting of the Association for Computational*
3058 *Linguistics, Proceedings of the Conference*, 2:35--40, 08 2013.

3059 [344] Vijayaraghavan Murali, Letao Qi, S. Chaudhuri, and C. Jermaine. Neural sketch learning for
3060 conditional program generation. In *ICLR*, 2018.

3061 [345] Aravind Nair, Karl Meinke, and Sigrid Eldh. Leveraging mutants for automatic prediction of
3062 metamorphic relations using machine learning. In *Proceedings of the 3rd ACM SIGSOFT Interna-*
3063 *tional Workshop on Machine Learning Techniques for Software Quality Evaluation*, MaLTeSQuE
3064 2019, page 1–6, 2019. ISBN 9781450368551. doi: 10.1145/3340482.3342741.

3065 [346] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, and Yang Liu. A multi-
3066 view context-aware approach to android malware detection and malicious code localiza-
3067 tion. *Empirical Softw. Engg.*, 23(3):1222–1274, jun 2018. ISSN 1382-3256. doi: 10.1007/
3068 s10664-017-9539-8. URL https://doi.org/10.1007/s10664-017-9539-8.

3069 [347] N. Nazar, He Jiang, Guojun Gao, Tao Zhang, Xiaochen Li, and Zhilei Ren. Source code frag-
3070 ment summarization with small-scale crowdsourcing based features. *Frontiers of Computer*
3071 *Science*, 10:504--517, 2015.

3072 [348] N. Nazar, Y. Hu, and He Jiang. Summarizing software artifacts: A literature review. *Journal of*
3073 *Computer Science and Technology*, 31:883--909, 2016.

3074 [349] Samuel Ndichu, Sangwook Kim, Seiichi Ozawa, Takeshi Misu, and Kazuo Makishima. A
3075 machine learning approach to detection of javascript-based attacks using ast features and
3076 paragraph vectors. *Applied Soft Computing*, 84:105721, 2019. ISSN 1568-4946. doi: https:
3077 //doi.org/10.1016/j.asoc.2019.105721.

3078 [350] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen. A deep neural network language
3079 model with contexts for source code. In *2018 IEEE 25th International Conference on Software*
3080 *Analysis, Evolution and Reengineering (SANER)*, pages 323--334, 2018. doi: 10.1109/SANER.2018.
3081 8330220.

3082 [351] Duc-Man Nguyen, Hoang-Nhat Do, Quyet-Thang Huynh, Dinh-Thien Vo, and Nhu-Hang Ha.
3083 Shinobi: A novel approach for context-driven testing (cdt) using heuristics and machine learn-
3084 ing for web applications. In Trung Q Duong and Nguyen-Son Vo, editors, *Industrial Networks*
3085 *and Intelligent Systems*, pages 86--102, 2019. ISBN 978-3-030-05873-9.

3086 [352] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. A statistical
3087 semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on*
3088 *Foundations of Software Engineering*, ESEC/FSE 2013, page 532–542, New York, NY, USA, 2013.
3089 Association for Computing Machinery. ISBN 9781450322379. doi: 1086_Nguyen2013. URL
3090 https://doi.org/1086_Nguyen2013.

3091 [353] Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, and Zenglin Xu. Coregen: Con-
3092 textualized code representation learning for commit message generation. *Neurocomputing*,
3093 459:97--107, 2021.

3094 [354] A. S. Nyamawe, H. Liu, N. Niu, Q. Umer, and Z. Niu. Automated recommendation of soft-
3095 ware refactorings based on feature requests. In *2019 IEEE 27th International Requirements*
3096 *Engineering Conference (RE)*, pages 187--198, 2019. doi: 10.1109/RE.2019.00029.

3097 [355] Ally S. Nyamawe, Hui Liu, Nan Niu, Qasim Umer, and Zhendong Niu. Feature requests-
3098 based recommendation of software refactorings. *Empirical Softw. Engg.*, 25(5):4315–4347,
3099 sep 2020. ISSN 1382-3256. doi: 10.1007/s10664-020-09871-2. URL https://doi.org/10.1007/
3100 s10664-020-09871-2.

3101 [356] Miroslaw Ochodek, Regina Hebig, Wilhelm Meding, Gert Frost, and Miroslaw Staron. Rec-
3102 ognizing lines of code violating company-specific coding guidelines using machine learning.
3103 *Empirical Software Engineering*, 25:220--265, 2019.

3104 [357] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. Learning to gen-
3105 erate pseudo-code from source code using statistical machine translation. In *2015 30th*
3106 *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574--584,
3107 2015. doi: 10.1109/ASE.2015.36.

3108 [358] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda,
3109 and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical
3110 machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software*
3111 *Engineering (ASE)*, pages 574--584, 2015. doi: 10.1109/ASE.2015.36.

3112 [359] Ahmet Okutan and Olcay Taner Yıldız. Software defect prediction using bayesian networks.
3113 *Empirical Software Engineering*, 19(1):154--181, 2014.

3114 [360] Daniel Oliveira, Wesley K. G. Assunção, Leonardo Souza, Willian Oizumi, Alessandro Garcia,
3115 and Baldoino Fonseca. Applying machine learning to customized smell detection: A multi-
3116 project study. SBES '20, page 233–242, 2020. ISBN 9781450387538. doi: 10.1145/3422392.
3117 3422427.

3118 [361] Safa Omri and Carsten Sinz. Deep learning for software defect prediction: A survey. In
3119 *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*,
3120 ICSEW'20, page 209–214, 2020. ISBN 9781450379632. doi: 10.1145/3387940.3391463.

3121 [362] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan. Buffer overflow vulnerability pre-
3122 diction from x86 executables using static analysis and machine learning. In *2015 IEEE 39th*
3123 *Annual Computer Software and Applications Conference*, volume 2, pages 450--459, 2015. doi:
3124 10.1109/COMPSAC.2015.78.

3125 [363] Fabio Palomba, Dario Di Nucci, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Denys Poshy-
3126 vanyk, and Andrea De Lucia. Landfill: An open dataset of code smells with public evaluation.
3127 In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 482--485,
3128 2015. doi: 10.1109/MSR.2015.69.

3129 [364] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto.
3130 Smells like teen spirit: Improving bug prediction performance using the intensity of code
3131 smells. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*,
3132 pages 244--255. IEEE, 2016.

3133 [365] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto.
3134 Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*, 45(2):
3135 194--218, 2017.

3136 [366] Cong Pan, Minyan Lu, Biao Xu, and Houleng Gao. An improved cnn model for within-project
3137 software defect prediction. *Applied Sciences*, 9(10):2138, 2019.

Journal Pre-proof

Journal Pre-proof

3138 [367] A. K. Pandey and Manjari Gupta. Software fault classification using extreme learning ma-
3139 chine: a cognitive approach. *Evolutionary Intelligence*, pages 1--8, 2018.

3140 [368] Sushant Kumar Pandey, Ravi Bhushan Mishra, and Anil Kumar Tripathi. Machine learning
3141 based methods for software fault prediction: A survey. *Expert Systems with Applications*, 172:
3142 114595, 2021.

3143 [369] Y. Pang, X. Xue, and A. S. Namin. Early identification of vulnerable software components
3144 via ensemble learning. In *2016 15th IEEE International Conference on Machine Learning and*
3145 *Applications (ICMLA)*, pages 476--481, 2016. doi: 10.1109/ICMLA.2016.0084.

3146 [370] Yulei Pang, Xiaozhen Xue, and Huaying Wang. Predicting vulnerable software compo-
3147 nents through deep neural network. In *Proceedings of the 2017 International Conference*
3148 *on Deep Learning Technologies*, ICDLT '17, page 6–10, New York, NY, USA, 2017. Association
3149 for Computing Machinery. ISBN 9781450352321. doi: 10.1145/3094243.3094245. URL
3150 https://doi.org/10.1145/3094243.3094245.

3151 [371] Sebastiano Panichella, Jairo Aponte, Massimiliano Di Penta, Andrian Marcus, and Gerardo
3152 Canfora. Mining source code descriptions from developer communications. In *2012 20th*
3153 *IEEE International Conference on Program Comprehension (ICPC)*, pages 63--72, 2012. doi: 10.
3154 1109/ICPC.2012.6240510.

3155 [372] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Re-evaluating method-level bug pre-
3156 diction. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengi-*
3157 *neering (SANER)*, pages 592--601. IEEE, 2018.

3158 [373] Kayur Patel, James Fogarty, James A. Landay, and Beverly Harrison. Investigating statistical
3159 machine learning as a tool for software development. In *Proceedings of the SIGCHI Conference*
3160 *on Human Factors in Computing Systems*, CHI '08, page 667–676, 2008. ISBN 9781605580111.
3161 doi: 10.1145/1357054.1357160.

3162 [374] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia. Comparing heuristic and machine learn-
3163 ing approaches for metric-based code smell detection. In *2019 IEEE/ACM 27th International*
3164 *Conference on Program Comprehension (ICPC)*, pages 93--104, 2019.

3165 [375] Fabiano Pecorelli, Dario Di Nucci, Coen De Roover, and Andrea De Lucia. On the role of data
3166 balancing for machine learning-based code smell detection. In *Proceedings of the 3rd ACM*
3167 *SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*,
3168 MaLTeSQuE 2019, page 19–24, 2019. ISBN 9781450368551. doi: 10.1145/3340482.3342744.

3169 [376] Han Peng, Ge Li, Wenhan Wang, YunFei Zhao, and Zhi Jin. Integrating tree path in trans-
3170 former for code representation. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and
3171 J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34,
3172 pages 9343--9354. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/
3173 2021/file/4e0223a87610176ef0d24ef6d2dcde3a-Paper.pdf.

3174 [377] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector rep-
3175 resentations for deep learning. In *International conference on knowledge science, engineering*
3176 *and management*, pages 547--553. Springer, 2015.

3177 [378] J. D. Pereira, J. R. Campos, and M. Vieira. An exploratory study on machine learning to com-
3178 bine security vulnerability alerts from static analysis tools. In *2019 9th Latin-American Sym-*
3179 *posium on Dependable Computing (LADC)*, pages 1--10, 2019. doi: 10.1109/LADC48089.2019.
3180 8995685.

Journal Pre-proof

3181 [379] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck,
3182 Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source
3183 projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer*
3184 *and Communications Security*, CCS '15, page 426–437, 2015. ISBN 9781450338325. doi: 10.
3185 1145/2810103.2813604.

3186 [380] Hung Phan and Ali Jannesari. Statistical machine translation outperforms neural machine
3187 translation in software engineering: Why and how. In *Proceedings of the 1st ACM SIGSOFT*
3188 *International Workshop on Representation Learning for Software Engineering and Program Lan-*
3189 *guages*, RL+SE&PL 2020, page 3–12, 2020. ISBN 9781450381253. doi: 10.1145/3416506.
3190 3423576.

3191 [381] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye.
3192 Cotext: Multi-task learning with code-text transformer, 2021. URL https://arxiv.org/abs/2105.
3193 08645.

3194 [382] Eduard Pinconschi, Rui Abreu, and Pedro Ad
3195 textasciitilde ao. A comparative study of automatic program repair techniques for security
3196 vulnerabilities. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering*
3197 *(ISSRE)*, pages 196--207. IEEE, 2021.

3198 [383] Goran Piskachev, Lisa Nguyen Quang Do, and Eric Bodden. Codebase-adaptive detection of
3199 security-relevant methods. In *Proceedings of the 28th ACM SIGSOFT International Symposium*
3200 *on Software Testing and Analysis*, ISSTA 2019, page 181–191, 2019. ISBN 9781450362245. doi:
3201 10.1145/3293882.3330556.

3202 [384] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A
3203 manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings*
3204 *of the 16th International Conference on Mining Software Repositories*, MSR '19, page 383–387,
3205 2019. doi: 10.1109/MSR.2019.00064.

3206 [385] Maryam Vahdat Pour, Zhuo Li, Lei Ma, and Hadi Hemmati. A search-based testing framework
3207 for deep neural networks of source code embedding. In *2021 14th IEEE Conference on Software*
3208 *Testing, Verification and Validation (ICST)*, pages 36--46. IEEE, 2021.

3209 [386] C. L. Prabha and N. Shivakumar. Software defect prediction using machine learning
3210 techniques. In *2020 4th International Conference on Trends in Electronics and Informatics*
3211 *(ICOEI)(48184)*, pages 728--733, 2020. doi: 10.1109/ICOEI48184.2020.9142909.

3212 [387] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detec-
3213 tion. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi: 10.1145/3276517.

3214 [388] Hosahalli Mahalingappa Premalatha and Chimanahalli Venkateshavittalachar Srikrishna.
3215 Software fault prediction and classification using cost based random forest in spiral life cycle
3216 model. *system*, 11, 2017.

3217 [389] Michael Prince. Does active learning work? a review of the research. *Journal of engineering*
3218 *education*, 93(3):223--231, 2004.

3219 [390] N. Pritam, M. Khari, L. Hoang Son, R. Kumar, S. Jha, I. Priyadarshini, M. Abdel-Basset, and
3220 H. Viet Long. Assessment of code smell for predicting class change proneness using machine
3221 learning. *IEEE Access*, 7:37414--37425, 2019. doi: 10.1109/ACCESS.2019.2905133.

3222 [391] Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent code completion with
3223 bayesian networks. *ACM Trans. Softw. Eng. Methodol.*, 25(1), December 2015. ISSN 1049-331X.
3224 doi: 10.1145/2744200.

Journal Pre-proof

[392] Christos Psarras, Themistoklis Diamantopoulos, and Andreas Symeonidis. A mechanism for automatically summarizing software functionality from source code. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 121--130. IEEE, 2019.

[393] Lei Qiao, Xuesong Li, Qasim Umer, and Ping Guo. Deep learning based software defect prediction. *Neurocomputing*, 385:100--110, 2020.

[394] Md Rafiqul Islam Rabin, Arjun Mukherjee, Omprakash Gnawali, and Mohammad Amin Alipour. Towards demystifying dimensions of source code embeddings. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages*, RL+SE&PL 2020, page 29–38, 2020. ISBN 9781450381253. doi: 10.1145/3416506.3423580.

[395] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139--1149, July 2017. doi: 10.18653/v1/P17-1105.

[396] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.

[397] Akond Rahman, Priysha Pradhan, Asif Partho, and Laurie Williams. Predicting android application security and privacy risk with static code metrics. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '17, page 149–153, 2017. ISBN 9781538626696. doi: 10.1109/MOBILESoft.2017.14.

[398] M. Rahman, Yutaka Watanobe, and K. Nakamura. A neural network based intelligent support model for program code completion. *Sci. Program.*, 2020:7426461:1--7426461:18, 2020. doi: 10.1155/2020/7426461.

[399] M. M. Rahman, C. K. Roy, and I. Keivanloo. Recommending Insightful Comments for Source Code using Crowdsourced Knowledge. In *Proc. SCAM*, pages 81--90, 2015.

[400] Santosh S Rathore and Sandeep Kumar. Software fault prediction based on the dynamic selection of learning technique: findings from the eclipse project study. *Applied Intelligence*, 51(12):8945--8960, 2021.

[401] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. *SIGPLAN Not.*, 51(10):731–747, October 2016. ISSN 0362-1340. doi: 10.1145/3022671.2984041.

[402] Sandeep Reddivari and Jayalakshmi Raman. Software quality prediction: an investigation based on machine learning. In *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*, pages 115--122. IEEE, 2019.

[403] Jiadong Ren, Zhangqi Zheng, Qian Liu, Zhiyao Wei, and Huaizhi Yan. A Buffer Overflow Prediction Approach Based on Software Metrics and Machine Learning. *Security and Communication Networks*, 2019:e8391425, March 2019. ISSN 1939-0114. doi: 10.1155/2019/8391425. URL https://www.hindawi.com/journals/scn/2019/8391425/. Publisher: Hindawi.

[404] Jinsheng Ren, Ke Qin, Ying Ma, and Guangchun Luo. On software defect prediction using machine learning. *Journal of Applied Mathematics*, 2014, 2014.

[405] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. A survey of deep active learning. *arXiv preprint arXiv:2009.00236*, 2020.

Journal Pre-proof

3268 [406] Joseph Renzullo, Westley Weimer, and Stephanie Forrest. Multiplicative weights algorithms for parallel automated software repair. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 984--993. IEEE, 2021.

3271 [407] Guillermo Rodriguez, Cristian Mateos, Luciano Listorti, Brian Hammer, and Sanjay Misra. A novel unsupervised learning approach for assessing web services refactoring. In Robertas Damaševičius and Giedrė Vasiljevienė, editors, *Information and Software Technologies*, pages 273--284, 2019. ISBN 978-3-030-30275-7.

3275 [408] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33:20601--20611, 2020.

3278 [409] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757--762, 2018. doi: 10.1109/ICMLA.2018.00120.

3282 [410] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757--762, 2018. doi: 10.1109/ICMLA.2018.00120.

3286 [411] Antonino Sabetta and Michele Bezzi. A practical approach to the automatic classification of security-relevant commits. In *2018 IEEE International conference on software maintenance and evolution (ICSME)*, pages 579--582. IEEE, 2018.

3289 [412] N. Saccente, J. Dehlinger, L. Deng, S. Chakraborty, and Y. Xiong. Project achilles: A prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 114--121, 2019. doi: 10.1109/ASEW.2019.00040.

3293 [413] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. Retrieval on source code: A neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, page 31–41, 2018. ISBN 9781450358347. doi: 10.1145/3211346.3211353.

3297 [414] Priyadarshni Suresh Sagar, Eman Abdulah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Christian D. Newman. Comparing commit messages and source code metrics for the prediction refactoring activities. *Algorithms*, 14(10), 2021. ISSN 1999-4893. doi: 10.3390/733_Sagar2021. URL https://www.mdpi.com/1999-4893/14/10/289.

3301 [415] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 648--659, 2017. doi: 10.1109/ASE.2017.8115675.

3304 [416] S. Saha, R. k. Saha, and M. r. Prasad. Harnessing evolution for multi-hunk program repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 13--24, 2019. doi: 10.1109/ICSE.2019.00020.

3307 [417] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Web service api anti-patterns detection as a multi-label learning problem. In *International Conference on Web Services*, pages 114--132. Springer, 2020.

3310 [418] Tara N Sainath, Brian Kingsbury, George Saon, Hagen Soltau, Abdel-rahman Mohamed, George Dahl, and Bhuvana Ramabhadran. Deep convolutional neural networks for large-scale speech tasks. *Neural Networks*, 64:39--48, 2015.

Journal Pre-proof

3313 [419] Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala.
3314 Type error feedback via analytic program repair. In *Proceedings of the 41st ACM SIGPLAN Con-*
3315 *ference on Programming Language Design and Implementation*, PLDI 2020, page 16–30, 2020.
3316 ISBN 9781450376136. doi: 10.1145/3385412.3386005.

3317 [420] Anush Sankaran, Rahul Aralikatte, Senthil Mani, Shreya Khare, Naveen Panwar, and Neela-
3318 madhav Gantayat. DARVIZ: deep abstract representation, visualization, and verification of
3319 deep learning models. *CoRR*, abs/1708.04915, 2017. URL http://arxiv.org/abs/1708.04915.

3320 [421] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral. Syntax and sensibility:
3321 Using language models to detect and correct syntax errors. In *2018 IEEE 25th International*
3322 *Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 311--322, 2018.
3323 doi: 10.1109/SANER.2018.8330219.

3324 [422] Igor Santos, Jaime Devesa, Félix Brezo, Javier Nieves, and Pablo Garcia Bringas. Opem: A
3325 static-dynamic approach for machine-learning-based malware detection. In Álvaro Herrero,
3326 Václav Snášel, Ajith Abraham, Ivan Zelinka, Bruno Baruque, Héctor Quintián, José Luis Calvo,
3327 Javier Sedano, and Emilio Corchado, editors, *International Joint Conference CISIS'12-ICEUTE´12-*
3328 *SOCO´12 Special Sessions*, pages 271--280, 2013. ISBN 978-3-642-33018-6.

3329 [423] F. Sarro, S. Di Martino, F. Ferrucci, and C. Gravino. A further analysis on the use of genetic
3330 algorithm to configure support vector machines for inter-release fault prediction. In *Pro-*
3331 *ceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, page 1215–1220,
3332 New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450308571. doi:
3333 10.1145/2245276.2231967. URL https://doi.org/10.1145/2245276.2231967.

3334 [424] J. Sayyad Shirabad and T.J. Menzies. The PROMISE Repository of Software Engineering
3335 Databases. School of Information Technology and Engineering, University of Ottawa, Canada,
3336 2005. URL http://promise.site.uottawa.ca/SERepository.

3337 [425] Max Eric Henry Schumacher, Kim Tuyen Le, and Artur Andrzejak. Improving code recom-
3338 mendations by combining neural and classical machine learning approaches. In *Proceedings*
3339 *of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20,
3340 page 476–482, 2020. ISBN 9781450379632. doi: 10.1145/3387940.3391489.

3341 [426] R. Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You autocomplete me: Poi-
3342 soning vulnerabilities in neural code completion. In *30th USENIX Security Symposium (USENIX*
3343 *Security 21)*, August 2021.

3344 [427] T. Sethi and Gagandeep. Improved approach for software defect prediction using artifi-
3345 cial neural networks. In *2016 5th International Conference on Reliability, Infocom Technolo-*
3346 *gies and Optimization (Trends and Future Directions) (ICRITO)*, pages 480--485, 2016. doi:
3347 10.1109/ICRITO.2016.7785003.

3348 [428] Burr Settles. Active learning literature survey. 2009.

3349 [429] Asaf Shabtai, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. Detection of malicious
3350 code by applying machine learning classifiers on static features: A state-of-the-art survey.
3351 *Information Security Technical Report*, 14(1):16 -- 29, 2009. ISSN 1363-4127. doi: https://doi.
3352 org/10.1016/j.istr.2009.03.003. Malware.

3353 [430] L. K. Shar, L. C. Briand, and H. B. K. Tan. Web application vulnerability prediction using hy-
3354 brid program analysis and machine learning. *IEEE Transactions on Dependable and Secure*
3355 *Computing*, 12(6):688--707, 2015. doi: 10.1109/TDSC.2014.2373377.

Journal Pre-proof

3356 [431] T. Sharma and M. Kessentini. Qscored: A large dataset of code smells and quality metrics. In
3357 *2021 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (MSR)*,
3358 pages 590--594, Los Alamitos, CA, USA, may 2021. IEEE Computer Society. doi: 10.1109/
3359 MSR52588.2021.00080. URL https://doi.ieeecomputersociety.org/10.1109/MSR52588.2021.00080.

3360 [432] Tushar Sharma. DesigniteJava, December 2018. URL https://doi.org/10.5281/zenodo.2566861.
3361 https://github.com/tushartushar/DesigniteJava.

3362 [433] Tushar Sharma. CodeSplit for C#, February 2019. URL https://doi.org/10.5281/zenodo.2566905.

3363 [434] Tushar Sharma. CodeSplitJava, February 2019. URL https://doi.org/10.5281/zenodo.2566865.
3364 https://github.com/tushartushar/CodeSplitJava.

3365 [435] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and*
3366 *Software*, 138:158--173, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2017.12.034.
3367 URL https://www.sciencedirect.com/science/article/pii/S0164121217303114.

3368 [436] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. Designite --- A Software Design Quality
3369 Assessment Tool. In *Proceedings of the First International Workshop on Bringing Architecture De-*
3370 *sign Thinking into Developers' Daily Activities*, BRIDGE '16, 2016. doi: 10.1145/2896935.2896938.

3371 [437] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. Code smell de-
3372 tection by deep direct-learning and transfer-learning. *Journal of Systems and Software*, 176:
3373 110936, 2021. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2021.110936.

3374 [438] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, Indira Vats, Hadi Moazen,
3375 and Federica Sarro. Replication package for Machine Learning for Source Code Analysis
3376 survey paper, Sept 2022. URL https://github.com/tushartushar/ML4SCA.

3377 [439] Andrey Shedko, Ilya Palachev, Andrey Kvochko, Aleksandr Semenov, and Kwangwon Sun. Ap-
3378 plying probabilistic models to c++ code on an industrial scale. In *Proceedings of the IEEE/ACM*
3379 *42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 595–602,
3380 2020. ISBN 9781450379632. doi: 10.1145/3387940.3391477.

3381 [440] Zhidong Shen and S. Chen. A survey of automatic software vulnerability detection,
3382 program repair, and defect prediction techniques. *Secur. Commun. Networks*, 2020:
3383 8858010:1--8858010:16, 2020.

3384 [441] A. Sheneamer and J. Kalita. Semantic clone detection using machine learning. In *2016 15th*
3385 *IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1024--1028,
3386 2016. doi: 10.1109/ICMLA.2016.0185.

3387 [442] Ke Shi, Yang Lu, Jingfei Chang, and Zhen Wei. Pathpair2vec: An ast path pair-based code rep-
3388 resentation method for defect prediction. *Journal of Computer Languages*, 59:100979, 2020.
3389 ISSN 2590-1184. doi: https://doi.org/10.1016/j.cola.2020.100979.

3390 [443] Y. Shido, Y. Kobayashi, A. Yamamoto, A. Miyamoto, and T. Matsumura. Automatic source
3391 code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural*
3392 *Networks (IJCNN)*, pages 1--8, 2019. doi: 10.1109/IJCNN.2019.8851751.

3393 [444] S. Shim, P. Patil, R. R. Yadav, A. Shinde, and V. Devale. DeeperCoder: Code generation using
3394 machine learning. In *2020 10th Annual Computing and Communication Workshop and Confer-*
3395 *ence (CCWC)*, pages 0194--0199, 2020. doi: 10.1109/CCWC47524.2020.9031149.

3396 [445] K. Shimonaka, S. Sumi, Y. Higo, and S. Kusumoto. Identifying auto-generated code by us-
3397 ing machine learning techniques. In *2016 7th International Workshop on Empirical Software*
3398 *Engineering in Practice (IWESEP)*, pages 18--23, 2016. doi: 10.1109/IWESEP.2016.18.

Journal Pre-proof

3399 [446] Eui Chul Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. Program synthesis
3400 and semantic parsing with learned code idioms. In *Advances in Neural Information Processing*
3401 *Systems*, pages 10825--10835, 2019.

3402 [447] Richard Shin, Neel Kant, Kavi Gupta, Chris Bender, Brandon Trabucco, Rishabh Singh, and
3403 Dawn Song. Synthetic datasets for neural program synthesis. In *International Conference on*
3404 *Learning Representations*, 2019.

3405 [448] L. Shiqi, T. Shengwei, Y. Long, Y. Jiong, and S. Hua. Android malicious code Classification
3406 using Deep Belief Network. *KSII Transactions on Internet and Information Systems*, 12:454--475,
3407 January 2018. doi: 10.3837/tiis.2018.01.022.

3408 [449] Chengxun Shu and Hongyu Zhang. Neural programming by example. *CoRR*, abs/1703.04990,
3409 2017.

3410 [450] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. Improving code search
3411 with co-attentive representation learning. In *Proceedings of the 28th International Conference*
3412 *on Program Comprehension*, ICPC '20, page 196–207, 2020. ISBN 9781450379588. doi: 10.
3413 1145/3387904.3389269.

3414 [451] Brahmaleen Kaur Sidhu, Kawaljeet Singh, and Neeraj Sharma. A machine learning approach
3415 to software model refactoring. *International Journal of Computers and Applications*, 44(2):
3416 166--177, 2022. doi: 10.1080/1206212X.2020.1711616. URL https://doi.org/10.1080/1206212X.
3417 2020.1711616.

3418 [452] Ajmer Singh, Rajesh Bhatia, and Anita Singhrova. Taxonomy of machine learning algorithms
3419 in software fault prediction using object oriented metrics. *Procedia computer science*, 132:
3420 993--1001, 2018.

3421 [453] P. Singh and A. Chug. Software defect prediction analysis using machine learning algorithms.
3422 In *2017 7th International Conference on Cloud Computing, Data Science Engineering - Confluence*,
3423 pages 775--781, 2017. doi: 10.1109/CONFLUENCE.2017.7943255.

3424 [454] P. Singh and R. Malhotra. Assessment of machine learning algorithms for determining defec-
3425 tive classes in an object-oriented software. In *2017 6th International Conference on Reliability,*
3426 *Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, pages 204--209,
3427 2017. doi: 10.1109/ICRITO.2017.8342425.

3428 [455] R. Singh, J. Singh, M. S. Gill, R. Malhotra, and Garima. Transfer learning code vector-
3429 izer based machine learning models for software defect prediction. In *2020 International*
3430 *Conference on Computational Performance Evaluation (ComPE)*, pages 497--502, 2020. doi:
3431 10.1109/ComPE49325.2020.9200076.

3432 [456] Behjat Soltanifar, Shirin Akbarinasaji, Bora Caglayan, Ayse Basar Bener, Asli Filiz, and Bryan M
3433 Kramer. Software analytics in practice: a defect prediction model using code smells. In
3434 *Proceedings of the 20th International Database Engineering & Applications Symposium*, pages
3435 148--155, 2016.

3436 [457] Qinbao Song, Yuchen Guo, and Martin Shepperd. A comprehensive investigation of the role
3437 of imbalanced learning for software defect prediction. *IEEE Transactions on Software Engineer-*
3438 *ing*, 45(12):1253--1269, 2019. doi: 10.1109/TSE.2018.2836442.

3439 [458] Xiaotao Song, Hailong Sun, Xu Wang, and Jiafei Yan. A survey of automatic generation of
3440 source code comments: Algorithms and techniques. *IEEE Access*, 7:111411--111428, 2019.

Journal Pre-proof

[459] M. Soto and C. Le Goues. Common statement kind changes to inform automatic program repair. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 102--105, 2018.

[460] Bruno Sotto-Mayor and Meir Kalech. Cross-project smell-based defect prediction. *Soft Computing*, 25(22):14171--14181, 2021.

[461] Michael Spreitzenbarth, Thomas Schreck, F. Echtler, D. Arp, and Johannes Hoffmann. Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security*, 14:141--153, 2014.

[462] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 2–13, 2020. ISBN 9781450379588. doi: 10.1145/3387904.3389258.

[463] M.-A. Storey. Theories, methods and tools in program comprehension: past, present and future. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 181--191, 2005. doi: 10.1109/WPC.2005.38.

[464] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265--266. ACM, 2016.

[465] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. Flow2vec: Value-flow-based precise code embedding. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi: 10.1145/3428301.

[466] Kazi Zakia Sultana. Towards a software vulnerability prediction model using traceable code patterns and software metrics. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1022--1025, 2017. doi: 10.1109/ASE.2017.8115724.

[467] Kazi Zakia Sultana, Vaibhav Anu, and Tai-Yin Chong. Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach. *Journal of Software: Evolution and Process*, 33(3):e2303, 2021. ISSN 2047-7481. doi: 10.1002/smr.2303. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2303.

[468] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8984--8991, 2020.

[469] Zhongbin Sun, Qinbao Song, and Xiaoyan Zhu. Using coding-based ensemble learning to improve software defect prediction. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1806--1817, 2012.

[470] Yeresime Suresh, Lov Kumar, and Santanu Ku Rath. Statistical and machine learning methods for software fault prediction using ck metric suite: a comparative analysis. *International Scholarly Research Notices*, 2014, 2014.

[471] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann, 1 edition, 2014. ISBN 0128013974.

[472] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476--480, 2014. doi: 10.1109/ICSME.2014.77.

Journal Pre-proof

[473] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 2727–2735, 2019. ISBN 9781450362016. doi: 10.1145/3292500.3330699.

[474] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1433–1443, 2020. ISBN 9781450370431. doi: 10.1145/3368089.3417058.

[475] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 329--340. IEEE, 2021.

[476] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1--9, 2015.

[477] Tomasz Szydlo, Joanna Sendorek, and Robert Brzoza-Woch. Enabling machine learning on resource constrained devices by source code generation of the learned models. In Yong Shi, Haohuan Fu, Yingjie Tian, Valeria V. Krzhizhanovskaya, Michael Harold Lees, Jack Dongarra, and Peter M. A. Sloot, editors, *Computational Science -- ICCS 2018*, pages 682--694, 2018. ISBN 978-3-319-93701-4.

[478] Akiyoshi Takahashi, Hiromitsu Shiina, and Nobuyuki Kobayashi. Automatic generation of program comments based on problem statements for computational thinking. In *2019 8th International Congress on Advanced Applied Informatics (IIAI-AAI)*, pages 629--634. IEEE, 2019.

[479] K. Terada and Y. Watanobe. Code completion for programming education based on recurrent neural network. In *2019 IEEE 11th International Workshop on Computational Intelligence and Applications (IWCIA)*, pages 109--114, 2019. doi: 10.1109/IWCIA47330.2019.8955090.

[480] H. Thaller, L. Linsbauer, and A. Egyed. Feature maps: A comprehensible software representation for design pattern detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 207--217, 2019. doi: 10.1109/SANER.2019.8667978.

[481] P. Thongkum and S. Mekruksavanich. Design flaws prediction for impact on software maintainability using extreme learning machine. In *2020 Joint International Conference on Digital Arts, Media and Technology with ECTI Northern Section Conference on Electrical, Electronics, Computer and Telecommunications Engineering (ECTI DAMT NCON)*, pages 79--82, 2020. doi: 10.1109/ECTIDAMTNCON48261.2020.9090717.

[482] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. Autotransform: Automated code transformation to support modern code review process. 2022.

[483] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 981--992, 2020.

Journal Pre-proof

3526 [484] Irene Tollin, Francesca Arcelli Fontana, Marco Zanoni, and Riccardo Roveda. Change pre-
3527 diction through coding rules violations. In *Proceedings of the 21st International Conference*
3528 *on Evaluation and Assessment in Software Engineering*, EASE'17, page 61–64, 2017. ISBN
3529 9781450348041. doi: 10.1145/3084226.3084282.

3530 [485] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei,
3531 Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas
3532 Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernan-
3533 des, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal,
3534 Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez,
3535 Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux,
3536 Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mi-
3537 haylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi
3538 Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian,
3539 Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng
3540 Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien
3541 Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation
3542 and fine-tuned chat models, 2023.

3543 [486] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions*
3544 *on Software Engineering*, 2020. doi: 10.1109/TSE.2020.3007722.

3545 [487] Angeliki-Agathi Tsintzira, Elvira-Maria Arvanitou, Apostolos Ampatzoglou, and Alexander
3546 Chatzigeorgiou. Applying machine learning in technical debt management: Future oppor-
3547 tunities and challenges. In Martin Shepperd, Fernando Brito e Abreu, Alberto Rodrigues da
3548 Silva, and Ricardo Pérez-Castillo, editors, *Quality of Information and Communications Technol-*
3549 *ogy*, pages 53--67, 2020. ISBN 978-3-030-58793-2.

3550 [488] Naohiko Tsuda, Hironori Washizaki, Yoshiaki Fukazawa, Yuichiro Yasuda, and Shunsuke Sug-
3551 imura. Machine learning to evaluate evolvability defects: Code metrics thresholds for a given
3552 context. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*,
3553 pages 83--94, 2018. doi: 10.1109/QRS.2018.00022.

3554 [489] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk. On learning meaningful
3555 code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference*
3556 *on Software Engineering (ICSE)*, pages 25--36, 2019. doi: 10.1109/ICSE.2019.00021.

3557 [490] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and
3558 Denys Poshyvanyk. Deep learning similarities from different representations of source code.
3559 MSR '18, page 542–553, 2018. ISBN 9781450357166. doi: 10.1145/3196398.3196431.

3560 [491] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and
3561 Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. In *2019 IEEE In-*
3562 *ternational Conference on Software Maintenance and Evolution (ICSME)*, pages 301--312. IEEE,
3563 2019.

3564 [492] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and
3565 Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural
3566 machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4), September 2019. ISSN 1049-
3567 331X. doi: 10.1145/3340544.

3568 [493] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota.
3569 Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference*
3570 *on Software Engineering (ICSE)*, pages 163--174. IEEE, 2021.

Journal Pre-proof

Journal Pre-proof

3571 [494] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk,
3572 and Gabriele Bavota. Using pre-trained models to boost code review automation. *arXiv*
3573 *preprint arXiv:2201.06850*, 2022.

3574 [495] Sahithi Tummalapalli, Lov Kumar, and Lalita Bhanu Murthy Neti. An empirical framework for
3575 web service anti-pattern prediction using machine learning techniques. In *2019 9th Annual*
3576 *Information Technology, Electromechanical Engineering and Microelectronics Conference (IEME-*
3577 *CON)*, pages 137--143. IEEE, 2019.

3578 [496] Sahithi Tummalapalli, Lov Kumar, and N. L. Bhanu Murthy. Prediction of web service anti-
3579 patterns using aggregate software metrics and machine learning techniques. In *Proceedings*
3580 *of the 13th Innovations in Software Engineering Conference on Formerly Known as India Soft-*
3581 *ware Engineering Conference*, ISEC 2020, 2020. ISBN 9781450375948. doi: 10.1145/3385032.
3582 3385042.

3583 [497] Sahithi Tummalapalli, NL Murthy, Aneesh Krishna, et al. Detection of web service anti-
3584 patterns using neural networks with multiple layers. In *International Conference on Neural*
3585 *Information Processing*, pages 571--579. Springer, 2020.

3586 [498] Sahithi Tummalapalli, Lov Kumar, Lalitha Bhanu Murthy Neti, Vipul Kocher, and Srinivas Pad-
3587 manabhuni. A novel approach for the detection of web service anti-patterns using word
3588 embedding techniques. In *International Conference on Computational Science and Its Applica-*
3589 *tions*, pages 217--230. Springer, 2021.

3590 [499] Sahithi Tummalapalli, Juhi Mittal, Lov Kumar, Lalitha Bhanu Murthy Neti, and Santanu Kumar
3591 Rath. An empirical analysis on the prediction of web service anti-patterns using source code
3592 metrics and ensemble techniques. In *International Conference on Computational Science and*
3593 *Its Applications*, pages 263--276. Springer, 2021.

3594 [500] Sahithi Tummalapalli, Lov Kumar, NL Bhanu Murthy, and Aneesh Krishna. Detection of web
3595 service anti-patterns using weighted extreme learning machine. *Computer Standards & Inter-*
3596 *faces*, page 103621, 2022.

3597 [501] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques
3598 for malware analysis. *Computers & Security*, 81:123 -- 147, 2019. ISSN 0167-4048. doi: https:
3599 //doi.org/10.1016/j.cose.2018.11.001.

3600 [502] S. Uchiyama, A. Kubo, H. Washizaki, and Y. Fukazawa. Detecting design patterns in object-
3601 oriented program source code by using metrics and machine learning. *Journal of Software*
3602 *Engineering and Applications*, 07:983--998, 2014.

3603 [503] Anderson Uchôa, Caio Barbosa, Daniel Coutinho, Willian Oizumi, Wesley KG Assunçao, Sil-
3604 via Regina Vergilio, Juliana Alves Pereira, Anderson Oliveira, and Alessandro Garcia. Predict-
3605 ing design impactful changes in modern code review: A large-scale empirical study. In *2021*
3606 *IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 471--482.
3607 IEEE, 2021.

3608 [504] Secil Ugurel, Robert Krovetz, and C. Lee Giles. What's the code? automatic classification of
3609 source code archives. In *Proceedings of the Eighth ACM SIGKDD International Conference on*
3610 *Knowledge Discovery and Data Mining*, KDD '02, page 632–638, 2002. ISBN 158113567X. doi:
3611 10.1145/775047.775141.

3612 [505] M. Utting, B. Legeard, F. Dadeau, F. Tamagnan, and F. Bouquet. Identifying and generating
3613 missing tests using machine learning on execution traces. In *2020 IEEE International Confer-*
3614 *ence On Artificial Intelligence Testing (AITest)*, pages 83--90, 2020. doi: 10.1109/AITEST49225.
3615 2020.00020.

Journal Pre-proof

3616 [506] Hoang Van Thuy, Phan Viet Anh, and Nguyen Xuan Hoai. Automated large program repair
3617 based on big code. In *Proceedings of the Ninth International Symposium on Information and*
3618 *Communication Technology*, SoICT 2018, pages 375?--381, 2018. ISBN 9781450365390. doi:
3619 10.1145/3287921.3287958.

3620 [507] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural pro-
3621 gram repair by jointly learning to localize and repair, 04 2019.

3622 [508] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez,
3623 Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg,
3624 S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neu-*
3625 *ral Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https:
3626 //proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

3627 [509] B. A. Vishnu and K. P. Jevitha. Prediction of cross-site scripting attack using machine learning
3628 algorithms. In *Proceedings of the 2014 International Conference on Interdisciplinary Advances*
3629 *in Applied Computing*, ICONIAAC '14, New York, NY, USA, 2014. Association for Computing
3630 Machinery. ISBN 9781450329088. doi: 10.1145/2660859.2660969. URL https://doi.org/10.
3631 1145/2660859.2660969.

3632 [510] Nickolay Viuginov and Andrey Filchenkov. A machine learning based automatic folding of
3633 dynamically typed languages. In *Proceedings of the 3rd ACM SIGSOFT International Workshop*
3634 *on Machine Learning Techniques for Software Quality Evaluation*, MaLTeSQuE 2019, page 31–36,
3635 2019. ISBN 9781450368551. doi: 10.1145/3340482.3342746.

3636 [511] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu.
3637 Improving automatic source code summarization via deep reinforcement learning. In *Pro-*
3638 *ceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE
3639 2018, page 397–407, 2018. ISBN 9781450359375. doi: 10.1145/3238147.3238206.

3640 [512] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. Multi-
3641 modal attention network learning for semantic source code retrieval. In *Proceedings of*
3642 *the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page
3643 13–25, 2019. ISBN 9781728125084. doi: 10.1109/ASE.2019.00012.

3644 [513] Z. Wan, X. Xia, D. Lo, and G. C. Murphy. How does machine learning change software
3645 development practices? *IEEE Transactions on Software Engineering*, pages 1--1, 2019. doi:
3646 10.1109/TSE.2019.2937083.

3647 [514] Deze Wang, Wei Dong, and Shanshan Li. A multi-task representation learning approach
3648 for source code. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Representa-*
3649 *tion Learning for Software Engineering and Program Languages*, RL+SE&PL 2020, page 1–2,
3650 2020. ISBN 9781450381253. doi: 10.1145/3416506.3423575.

3651 [515] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. Context-aware
3652 retrieval-based deep commit message generation. *ACM Transactions on Software Engineering*
3653 *and Methodology (TOSEM)*, 30(4):1--30, 2021.

3654 [516] R. Wang, H. Zhang, G. Lu, L. Lyu, and C. Lyu. Fret: Functional reinforced transformer with
3655 bert for code summarization. *IEEE Access*, 8:135591--135604, 2020. doi: 10.1109/ACCESS.
3656 2020.3011744.

3657 [517] S. Wang and X. Yao. Using class imbalance learning for software defect prediction. *IEEE*
3658 *Transactions on Reliability*, 62(2):434--443, 2013. doi: 10.1109/TR.2013.2259203.

Journal Pre-proof

3659 [518] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao. How different is it between machine-generated
3660 and developer-provided patches? : An empirical study on the correct patches generated by
3661 automated program repair techniques. In *2019 ACM/IEEE International Symposium on Empiri-*
3662 *cal Software Engineering and Measurement (ESEM)*, pages 1--12, 2019. doi: 10.1109/ESEM.2019.
3663 8870172.

3664 [519] Shuai Wang, Jinyang Liu, Ye Qiu, Zhiyi Ma, Junfei Liu, and Zhonghai Wu. Deep learning
3665 based code completion models for programming codes. In *Proceedings of the 2019 3rd In-*
3666 *ternational Symposium on Computer Science and Intelligent Control*, ISCSIC 2019, 2019. ISBN
3667 9781450376617. doi: 10.1145/3386164.3389083.

3668 [520] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect
3669 prediction. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE
3670 '16, page 297–308, 2016. ISBN 9781450339001. doi: 10.1145/2884781.2884804.

3671 [521] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. Deep semantic feature learning for soft-
3672 ware defect prediction. *IEEE Transactions on Software Engineering*, 46(12):1267--1293, 2018.

3673 [522] Tiejian Wang, Zhiwu Zhang, Xiaoyuan Jing, and Liqiang Zhang. Multiple kernel ensemble
3674 learning for software defect prediction. *Automated Software Engineering*, 23(4):569--590,
3675 2016.

3676 [523] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, P. Yu, and G. Xu. Reinforcement-learning-
3677 guided source code summarization via hierarchical attention. *IEEE Transactions on Software*
3678 *Engineering*, pages 1--1, 2020. doi: 10.1109/TSE.2020.2979701.

3679 [524] Wei Wang and Michael W. Godfrey. Recommending clones for refactoring using design, con-
3680 text, and history. In *2014 IEEE International Conference on Software Maintenance and Evolution*,
3681 pages 331--340, 2014. doi: 10.1109/ICSME.2014.55.

3682 [525] Wenhan Wang, Ge Li, Sijie Shen, Xin Xia, and Zhi Jin. Modular tree network for source code
3683 representation learning. *ACM Trans. Softw. Eng. Methodol.*, 29(4), September 2020. ISSN 1049-
3684 331X. doi: 10.1145/3409331.

3685 [526] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip Yu, and Guan-
3686 dong Xu. Reinforcement-learning-guided source code summarization via hierarchical atten-
3687 tion. *IEEE Transactions on software Engineering*, 2020.

3688 [527] Xinda Wang, Shu Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. A machine learning
3689 approach to classify security patches into vulnerability types. In *2020 IEEE Conference on*
3690 *Communications and Network Security (CNS)*, pages 1--9, 2020. doi: 10.1109/CNS48642.2020.
3691 9162237.

3692 [528] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. Learning semantic program embed-
3693 dings with graph interval neural network. *Proc. ACM Program. Lang.*, 4(OOPSLA), November
3694 2020. doi: 10.1145/3428205.

3695 [529] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware uni-
3696 fied pre-trained encoder-decoder models for code understanding and generation. In *Pro-*
3697 *ceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages
3698 8696--8708, Online and Punta Cana, Dominican Republic, November 2021. Association for
3699 Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.
3700 org/2021.emnlp-main.685.

3701 [530] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code sum-
3702 marization. *Advances in neural information processing systems*, 32, 2019.

Journal Pre-proof

3703 [531] Linfeng Wei, Weiqi Luo, Jian Weng, Yanjun Zhong, Xiaoqian Zhang, and Zheng Yan. Machine
3704 learning-based malicious application detection of android. *IEEE Access*, 5:25591--25601, 2017.
3705 doi: 10.1109/ACCESS.2017.2771470.

3706 [532] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learn-
3707 ing code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM Interna-
3708 tional Conference on Automated Software Engineering*, ASE 2016, page 87–98, 2016. ISBN
3709 9781450338455. doi: 10.1145/2970276.2970326.

3710 [533] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk.
3711 Sorting and transforming program repair ingredients via deep learning code similarities.
3712 In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering
3713 (SANER)*, pages 479--490. IEEE, 2019.

3714 [534] Liwei Wu, Fei Li, Youhua Wu, and Tao Zheng. GGF: A graph-based method for programming
3715 language syntax error correction. In *Proceedings of the 28th International Conference on Pro-
3716 gram Comprehension*, ICPC '20, pages 139–--148. Association for Computing Machinery, 2020.
3717 ISBN 9781450379588. doi: 10.1145/3387904.3389252.

3718 [535] L. Xiao, HuaiKou Miao, Tingting Shi, and Y. Hong. Lstm-based deep learning for spatial–tem-
3719 poral software testing. *Distributed and Parallel Databases*, pages 1--26, 2020.

3720 [536] R. Xie, W. Ye, J. Sun, and S. Zhang. Exploiting method names to improve code summa-
3721 rization: A deliberation multi-task learning approach. In *2021 2021 IEEE/ACM 29th Interna-
3722 tional Conference on Program Comprehension (ICPC) (ICPC)*, pages 138--148, may 2021. doi:
3723 10.1109/ICPC52881.2021.00022.

3724 [537] Yingfei Xiong, Bo Wang, Guirong Fu, and Linfei Zang. Learning to synthesize. In *Proceedings
3725 of the 4th International Workshop on Genetic Improvement Workshop*, GI '18, page 37–44, 2018.
3726 ISBN 9781450357531. doi: 10.1145/3194810.3194816.

3727 [538] Sihan Xu, Aishwarya Sivaraman, Siau-Cheng Khoo, and Jing Xu. Gems: An extract method
3728 refactoring recommender. In *2017 IEEE 28th International Symposium on Software Reliability
3729 Engineering (ISSRE)*, pages 24--34, 2017. doi: 10.1109/ISSRE.2017.35.

3730 [539] Sihan Xu, Sen Zhang, Weijing Wang, Xinya Cao, Chenkai Guo, and Jing Xu. Method name sug-
3731 gestion with hierarchical attention networks. In *Proceedings of the 2019 ACM SIGPLAN Work-
3732 shop on Partial Evaluation and Program Manipulation*, PEPM 2019, page 10–21, 2019. ISBN
3733 9781450362269. doi: 10.1145/3294032.3294079.

3734 [540] Eran Yahav. From programs to interpretable deep models and back. In Hana Chockler and
3735 Georg Weissenbacher, editors, *Computer Aided Verification*, pages 27--37, 2018. ISBN 978-3-
3736 319-96145-3.

3737 [541] Hangfeng Yang, Shudong Li, Xiaobo Wu, Hui Lu, and Weihong Han. A novel solutions for
3738 malicious code detection and family clustering based on machine learning. *IEEE Access*, 7:
3739 148853--148860, 2019. doi: 10.1109/ACCESS.2019.2946482.

3740 [542] Jiachen Yang, K. Hotta, Yoshiki Higo, H. Igaki, and S. Kusumoto. Classification model for code
3741 clones based on machine learning. *Empirical Software Engineering*, 20:1095--1125, 2014.

3742 [543] Mutian Yang, Jingzheng Wu, Shouling Ji, Tianyue Luo, and Yanjun Wu. Pre-patch: Find hidden
3743 threats in open software based on machine learning method. In Alvin Yang, Siva Kantamneni,
3744 Ying Li, Awel Dico, Xiangang Chen, Rajesh Subramanyan, and Liang-Jie Zhang, editors, *Services
3745 -- SERVICES 2018*, pages 48--65, 2018. ISBN 978-3-319-94472-2.

Journal Pre-proof

Journal Pre-proof

3746 [544] Yanming Yang, Xin Xia, David Lo, and John Grundy. A survey on deep learning for software
3747 engineering. *ACM Comput. Surv.*, 54(10s), sep 2022. ISSN 0360-0300. doi: 10.1145/3505243.
3748 URL https://doi.org/10.1145/3505243.

3749 [545] Yixiao Yang, Xiang Chen, and Jiaguang Sun. Improve language modeling for code completion
3750 through learning general token repetition of source code with optimized memory. *Interna-*
3751 *tional Journal of Software Engineering and Knowledge Engineering*, 29(11n12):1801--1818, 2019.
3752 doi: 10.1142/S0218194019400229.

3753 [546] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, and M. Zhang. A multi-modal transformer-based
3754 code summarization approach for smart contracts. In *2021 2021 IEEE/ACM 29th International*
3755 *Conference on Program Comprehension (ICPC) (ICPC)*, pages 1--12, may 2021. doi: 10.1109/
3756 ICPC52881.2021.00010.

3757 [547] Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. Staqc: A systematically mined
3758 question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web*
3759 *Conference*, WWW '18, page 1693–1703, Republic and Canton of Geneva, CHE, 2018. Inter-
3760 national World Wide Web Conferences Steering Committee. ISBN 9781450356398. doi:
3761 10.1145/3178876.3186081. URL https://doi.org/10.1145/3178876.3186081.

3762 [548] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. Coacor: Code annotation for code
3763 retrieval with reinforcement learning. In *The World Wide Web Conference*, WWW '19, page
3764 2203–2214, 2019. ISBN 9781450366748. doi: 10.1145/3308558.3313632.

3765 [549] Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. Leveraging
3766 code generation to improve code retrieval and summarization via dual learning. In *Proceed-*
3767 *ings of The Web Conference 2020*, WWW '20, page 2309–2319, 2020. ISBN 9781450370233.
3768 doi: 10.1145/3366423.3380295.

3769 [550] Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. The value
3770 of semantic parse labeling for knowledge base question answering. In *Proceedings of the*
3771 *54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*,
3772 pages 201--206, Berlin, Germany, August 2016. Association for Computational Linguistics.
3773 doi: 10.18653/v1/P16-2033.

3774 [551] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code
3775 generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational*
3776 *Linguistics (Volume 1: Long Papers)*, pages 440--450, July 2017. doi: 10.18653/v1/P17-1041.

3777 [552] Pengcheng Yin and Graham Neubig. Tranx: A transition-based neural abstract syntax parser
3778 for semantic parsing and code generation. *arXiv preprint arXiv:1810.02720*, 2018.

3779 [553] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning
3780 to mine aligned code and natural language pairs from Stack Overflow. In *Proceedings of*
3781 *the 15th International Conference on Mining Software Repositories*, MSR '18, pages 476--486,
3782 New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357166. doi:
3783 10.1145/3196398.3196408.

3784 [554] Chubato Wondaferaw Yohannese and Tianrui Li. A combined-learning based framework for
3785 improved software fault prediction. *International Journal of Computational Intelligence Systems*,
3786 10(1):647, 2017.

3787 [555] Veneta Yosifova, Antoniya Tasheva, and Roumen Trifonov. Predicting vulnerability type in
3788 common vulnerabilities and exposures (cve) database with machine learning classifiers. In
3789 *2021 12th National Conference with International Participation (ELECTRONICA)*, pages 1--6, 2021.
3790 doi: 10.1109/ELECTRONICA52725.2021.9513723.

3791 [556] Awad A. Younis and Yashwant K. Malaiya. Using software structure to predict vulnerability
3792 exploitation potential. In *2014 IEEE Eighth International Conference on Software Security and*
3793 *Reliability-Companion*, pages 13--18, 2014. doi: 10.1109/SERE-C.2014.17.

3794 [557] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene
3795 Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale
3796 human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL
3797 task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Process-*
3798 *ing*, pages 3911--3921, Brussels, Belgium, October-November 2018. Association for Compu-
3799 tational Linguistics. doi: 10.18653/v1/D18-1425. URL https://aclanthology.org/D18-1425.

3800 [558] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler. Automatic clone rec-
3801 ommendation for refactoring based on the present and the past. In *2018 IEEE Interna-*
3802 *tional Conference on Software Maintenance and Evolution (ICSME)*, pages 115--126, 2018. doi:
3803 10.1109/ICSME.2018.00021.

3804 [559] Marco Zanoni, Francesca Arcelli Fontana, and Fabio Stella. On applying machine learning
3805 techniques for design pattern detection. *Journal of Systems and Software*, 103:102--117, 2015.

3806 [560] Chunyan Zhang, Junchao Wang, Qinglei Zhou, Ting Xu, Ke Tang, Hairen Gui, and Fudong Liu.
3807 A survey of automatic source code summarization. *Symmetry*, 14(3):471, 2022.

3808 [561] Du Zhang and Jeffrey J. P. Tsai. Machine learning and software engineering. *Software Quality*
3809 *Journal*, 11(2):87–119, June 2003. ISSN 0963-9314. doi: 10.1023/A:1023760326768.

3810 [562] Fanlong Zhang and Siau-cheng Khoo. An empirical study on clone consistency prediction
3811 based on machine learning. *Information and Software Technology*, 136:106573, 2021.

3812 [563] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. A novel neural source code repre-
3813 sentation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on*
3814 *Software Engineering (ICSE)*, pages 783--794, 2019. doi: 10.1109/ICSE.2019.00086.

3815 [564] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. Machine learning testing: Survey, landscapes and
3816 horizons. *IEEE Transactions on Software Engineering*, pages 1--1, 2020. doi: 10.1109/TSE.2019.
3817 2962027.

3818 [565] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Retrieval-based neural
3819 source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on*
3820 *Software Engineering*, ICSE '20, page 1385–1397, 2020. ISBN 9781450371216. doi: 10.1145/
3821 3377811.3380383.

3822 [566] Jie M. Zhang and Mark Harman. "ignorance and prejudice" in software fairness. In *2021*
3823 *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1436--1447, 2021.
3824 doi: 10.1109/ICSE43902.2021.00129.

3825 [567] Jinglei Zhang, Rui Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. Exploiting code knowledge
3826 graph for bug localization via bi-directional attention. In *Proceedings of the 28th International*
3827 *Conference on Program Comprehension*, ICPC '20, pages 219–--229. Association for Computing
3828 Machinery, 2020. ISBN 9781450379588. doi: 10.1145/3387904.3389281.

3829 [568] Q. Zhang and B. Wu. Software defect prediction via transformer. In *2020 IEEE 4th Information*
3830 *Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, volume 1, pages
3831 874--879, 2020. doi: 10.1109/ITNEC48623.2020.9084745.

3832 [569] Yang Zhang and Chunhao Dong. Mars: Detecting brain class/method code smell based on
3833 metric--attention mechanism and residual network. *Journal of Software: Evolution and Process*,
3834 page e2403, 2021.

Journal Pre-proof

Journal Pre-proof

3835 [570] Yu Zhang and Binglong Li. Malicious code detection based on code semantic features. *IEEE*
3836 *Access*, 8:176728--176737, 2020. doi: 10.1109/ACCESS.2020.3026052.

3837 [571] Gang Zhao and Jeff Huang. Deepsim: Deep learning code functional similarity. In *Proceedings*
3838 *of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Sympo-*
3839 *sium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 141–151, 2018. ISBN
3840 9781450355735. doi: 10.1145/3236024.3236068.

3841 [572] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min,
3842 Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen,
3843 Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong
3844 Wen. A survey of large language models, 2023.

3845 [573] Wei Zheng, Jialiang Gao, Xiaoxue Wu, Fengyu Liu, Yuxing Xun, Guoliang Liu, and Xiang Chen.
3846 The impact factors on the performance of machine learning-based vulnerability detection: A
3847 comparative study. *Journal of Systems and Software*, 168:110659, 2020. ISSN 0164-1212. doi:
3848 https://doi.org/10.1016/j.jss.2020.110659.

3849 [574] Wenhao Zheng, Hongyu Zhou, Ming Li, and Jianxin Wu. Codeattention: translating source
3850 code to comments by exploiting the code constructs. *Frontiers of Computer Science*, 13(3):
3851 565--578, 2019.

3852 [575] Chaoliang Zhong, Ming Yang, and Jun Sun. Javascript code suggestion based on deep learn-
3853 ing. In *Proceedings of the 2019 3rd International Conference on Innovation in Artificial Intelligence*,
3854 ICIAI 2019, page 145–149, 2019. ISBN 9781450361286. doi: 10.1145/3319921.3319922.

3855 [576] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries
3856 from natural language using reinforcement learning, 2017. URL https://arxiv.org/abs/1709.
3857 00103.

3858 [577] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution.
3859 In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, page 95–109, 2012.
3860 ISBN 9780769546810. doi: 10.1109/SP.2012.16.

3861 [578] Yu Zhou, Xin Yan, Wenhua Yang, Taolue Chen, and Zhiqiu Huang. Augmenting java method
3862 comments generation with context information based on neural networks. *Journal of Systems*
3863 *and Software*, 156:328--340, 2019. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.07.
3864 087. URL https://www.sciencedirect.com/science/article/pii/S0164121219301529.

3865 [579] Yu Zhou, Xin Yan, Wenhua Yang, Taolue Chen, and Zhiqiu Huang. Augmenting java method
3866 comments generation with context information based on neural networks. *Journal of Systems*
3867 *and Software*, 156:328--340, 2019.

3868 [580] Yu Zhou, Juanjuan Shen, Xiaoqing Zhang, Wenhua Yang, Tingting Han, and Taolue Chen. Au-
3869 tomatic source code summarization with graph attention networks. *Journal of Systems and*
3870 *Software*, 188:111257, 2022.

3871 [581] Ziyi Zhou, Huiqun Yu, and Guisheng Fan. Adversarial training and ensemble learning for au-
3872 tomatic code summarization. *Neural Computing and Applications*, 33(19):12571--12589, 2021.

3873 [582] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang.
3874 A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint*
3875 *Meeting on European Software Engineering Conference and Symposium on the Foundations of*
3876 *Software Engineering*, pages 341--353, 2021.

3877 [583] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse.
3878 In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE*
3879 *Workshops 2007)*, pages 9--9, 2007. doi: 10.1109/PROMISE.2007.10.

Journal Pre-proof

Highlights

- The use of ML techniques is constantly increasing for source code analysis
- A wide range SE tasks involving source code analysis use ML
- The study identifies challenges in the field and potential mitigations
- We identify commonly used datasets and tools used in the field

Journal Pre-proof

Journal Pre-proof

Tushar Sharma is an assistant professor at Dalhousie University, Canada. He leads Software Maintenance and Analytics Research Team (SMART) lab focusing on software design and architecture, refactoring, code quality, technical debt, and machine learning for software engineering (ML4SE). He earned PhD from Athens University of Economics and Business, Athens, Greece, specializing in software engineering. He obtained an MS in Computer Science from the Indian Institute of Technology-Madras, Chennai, India. His professional experience includes working with Siemens Research, USA, for approximately two years and with Siemens Corporate Technology, India, for seven years. He co-authored Refactoring for Software Design Smells: Managing Technical Debt and two Oracle Java certification books. He founded and developed Designite, a software design quality assessment tool many practitioners and researchers use worldwide.

Maria Kechagia is a Research Fellow at University College London. Previously, she was a postdoctoral researcher at the Delft University of Technology. She obtained a PhD degree on Software Engineering from the Athens University of Economics and Business and a MSc on Software Engineering from Imperial College London. Dr. Kechagia has been an active and prolific researcher focusing her effort on program analysis, software testing, automated program repair, and software analytics. On these topics, she has published high-quality scholarly articles in top-tier software engineering venues including TSE, EMSE, ICSE, ISSTA. She has co-organised the third International Workshop on Automated Program Repair (APR) at ICSE 2022. She is currently the co-chair of the ISSTA 2024 Tool Demonstrations track. She has been also invited to join the programme committees of top-tier events (ICSE, FSE, ASE, ISSTA, MSR, and ICSME) and to serve as a reviewer for the major SE journals (TSE, TOSEM, EMSE, JSS).

Stefanos Georgiou is a Backend and DevOps developer at SimpleTechs. He holds a PhD from the Athens University of Business and Economics focused on Energy and Run-Time Performance Practices in Software Engineering. He did his PostDoc fellowship at Queen's University, Ontario, Canada. He holds a BSc in Networks and Systems Programming from the University of Cyprus and an MSc in PERvasive Computing and COMmunications for sustainable development (PERCCOM).

Rohit Tiwari is a technical architect working with DevOn Software Services, India. He has more than six years of software development experience. He earned his bachelor of engineering degree in computer science from Rajasthan University, India.

Indira Vats is an undergraduate student at JSS academy of technical education, India.

Hadi Moazen is a research assistant at the advanced computer network lab of University of Tabriz, Iran.

Federica Sarro is a Professor of Software Engineering in the Department of Computer Science at UCL, where she is the Head of the Software Systems Engineering group and where she has established the SOLAR team within the CREST centre. She has extensive academic and

industrial expertise in Search-Based Software Engineering, Empirical Software Engineering and Software Analytics, with a focus on automated software management, optimisation, testing and repair. On these topics she has published over 100 peer-reviewed scholarly articles, and she has given several invited talks at academic and industrial international events. She has also worked in collaboration with several companies including Meta, Google and Microsoft. Professor Sarro has obtained numerous awards and generous funding for her research. In 2021, she was awarded the Rising Star Award by the IEEE Technical Community on Software Engineering in recognition of her "excellence in Software Engineering research with scholarly and real-world impact.

Journal Pre-proof

Journal Pre-proof

CRediT author statement

Tushar Sharma: Conceptualization, methodology, Writing - Original Draft, Investigation; **Maria Kechagia**: Data Curation, Writing - Original Draft; **Stefanos Georgiou:** Data Curation; **Rohit Tiwari:** Software, Data Curation, **Indira Vats:** Data Curation; **Hadi Moazen:** Data Curation; and **Federica Sarro:** Writing - Review & Editing.

Journal Pre-proof

Journal Pre-proof

Declaration of interests

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

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Journal Pre-proof