A Survey on Machine Learning Techniques Applied to Source Code

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

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

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

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).

Abstract

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

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

1. Introduction

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

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

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

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

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

This survey makes the following contributions to the field:

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

The rest of the paper is organized as follows. We present the followed methodology, including the literature search protocol and research questions, in Section 2. Section 2.3, Section 3, Section 4, and Section 5 provide the detailed results of our findings. We present threats to validity in Section 6, and conclude the paper in Section 7.

Table 1. Comparison Among Surveys. The "Category" column refers to the software engineering task the survey covers. The "Scope" column indicates the focus of the study; TML refers to traditional machine learning and DL refers to deep learning techniques. The "Data&Tools" column indicates if a survey reviews available datasets and tools for ml-based applications, the "Challenges" column shows whether the study identifies challenges in the field studied, the "Type" column refers to the type of literature survey, and the "#Studies" column refers to the number of studies included in a given survey. We use "–" to indicate that a field is not applicable to a certain study and *NA* for the number of studies column, where the study does not explicitly mention selection criteria and the number of selected studies.

Category	Article	Scope	Data & Tools	Chall-enges	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

2. Methodology

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

2.1 Research objectives

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

RO1. *Identifying specific software engineering tasks involving source code that have been attempted using machine learning.*
Our objective is to explore the extent to which machine learning has been applied to analyze and process source code for SE tasks.We aim to summarize how ML can help engineers tackle specific SE tasks.

RO2. *Summarizing the machine learning techniques used for these tasks.*
This objective explores the ML techniques commonly applied to source code for performing the software engineering tasks identified above. We attempt to synthesize a mapping of tasks (along with related sub-tasks) and corresponding ML techniques.

RO3. *Providing a list of available datasets and tools.*
With this goal, we aim to provide a consolidated summary of publicly available datasets and tools along with their purpose.

RO4. *Identifying the challenges and perceived deficiencies in ML-enabled source code analysis and manipulation for software engineering.*
With this objective, we aim to identify challenges, and opportunities arising when applying ML techniques to source code for SE tasks, as well as to understand the extent to which they have been addressed in the articles surveyed.

2.2 Literature search protocol

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

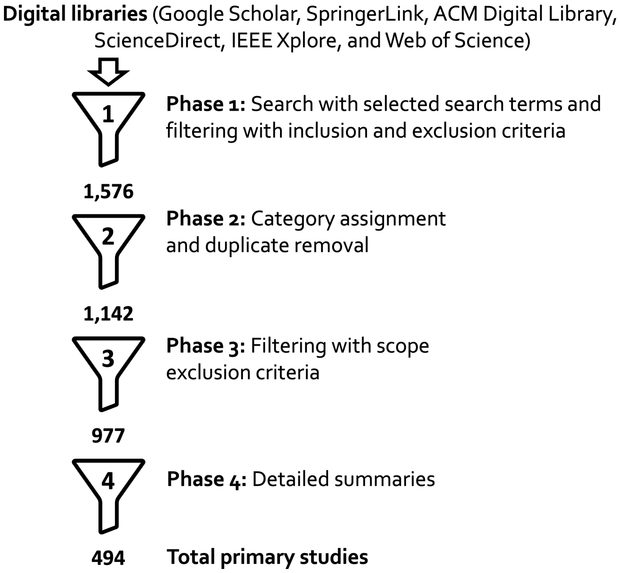

Digital libraries (Google Scholar, SpringerLink, ACM Digital Library, ScienceDirect, IEEE Xplore, and Web of Science)
1
Phase 1: Search with selected search terms and filtering with inclusion and exclusion criteria
1,576
2
Phase 2: Category assignment and duplicate removal
1,142
3
Phase 3: Filtering with scope exclusion criteria
977
4
Phase 4: Detailed summaries
494
Total primary studies

Figure 1. Overview of the search process

2.2.1 Literature search—Phase 1

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

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

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

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

2.2.3 Literature search—Phase 3

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

2.2.4 Literature search—Phase 4

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

2.3 Assigning articles to software engineering task categories

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

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

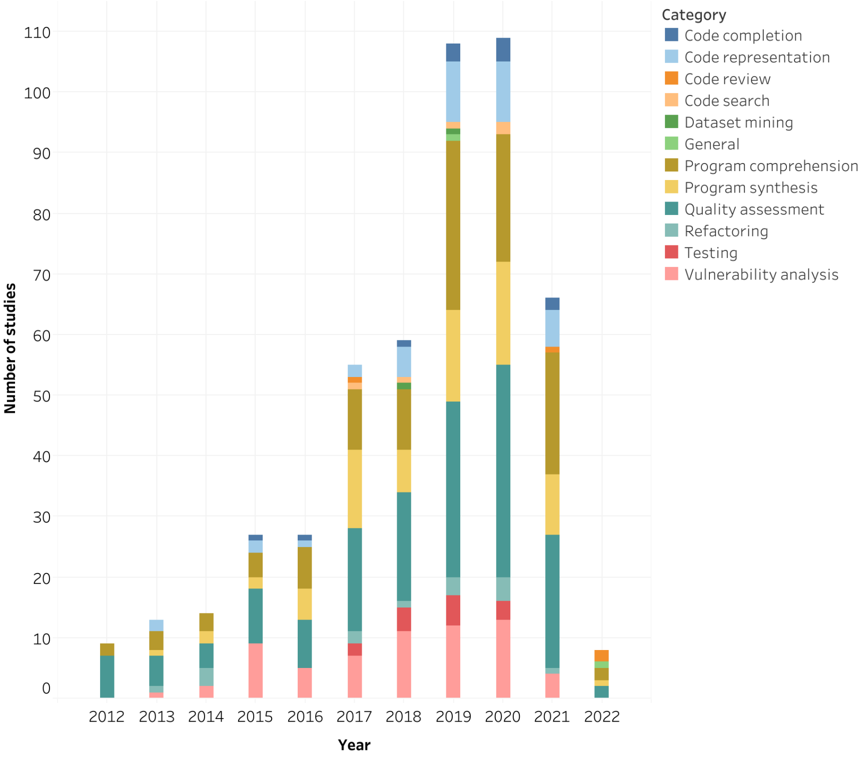
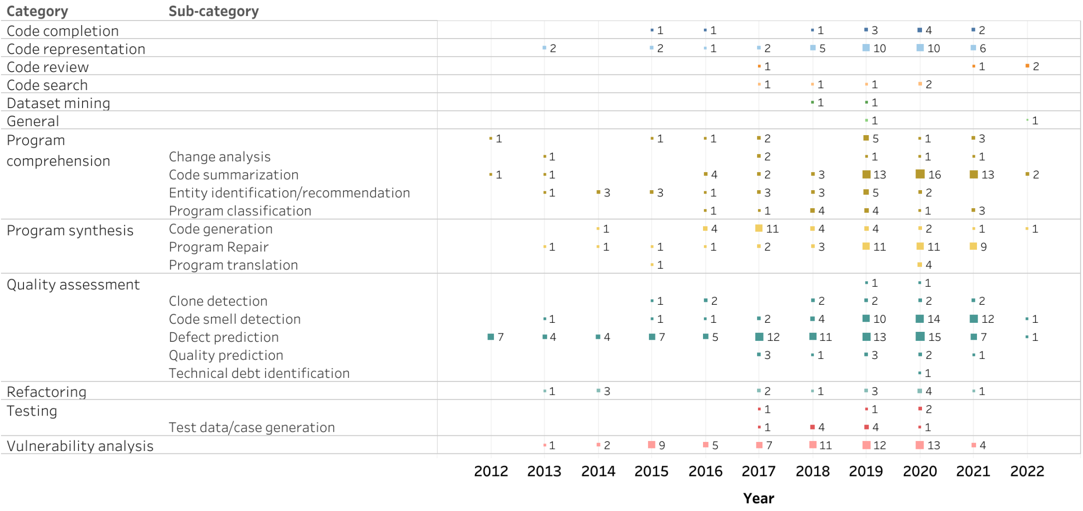


Figure 2. Category-wise distribution of studies



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	

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

220 corresponding sub-categories.

221 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)

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

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

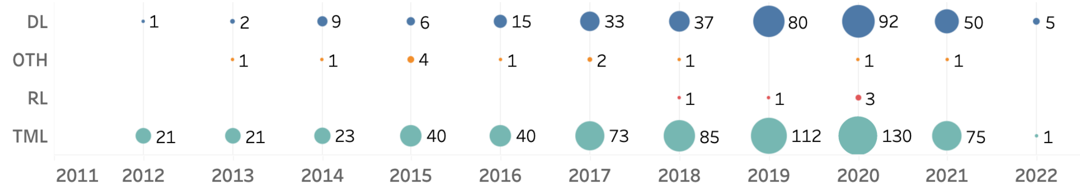


Figure 4. Usage of ML techniques by categories per year

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

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

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

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

In the rest of this section, we delve into each category and sub-category at a time, break down the entire workflow of a code analysis task into fine-grained steps, and summarize the method and ML techniques used. It is worth emphasizing that we structure the discussion around the cru-

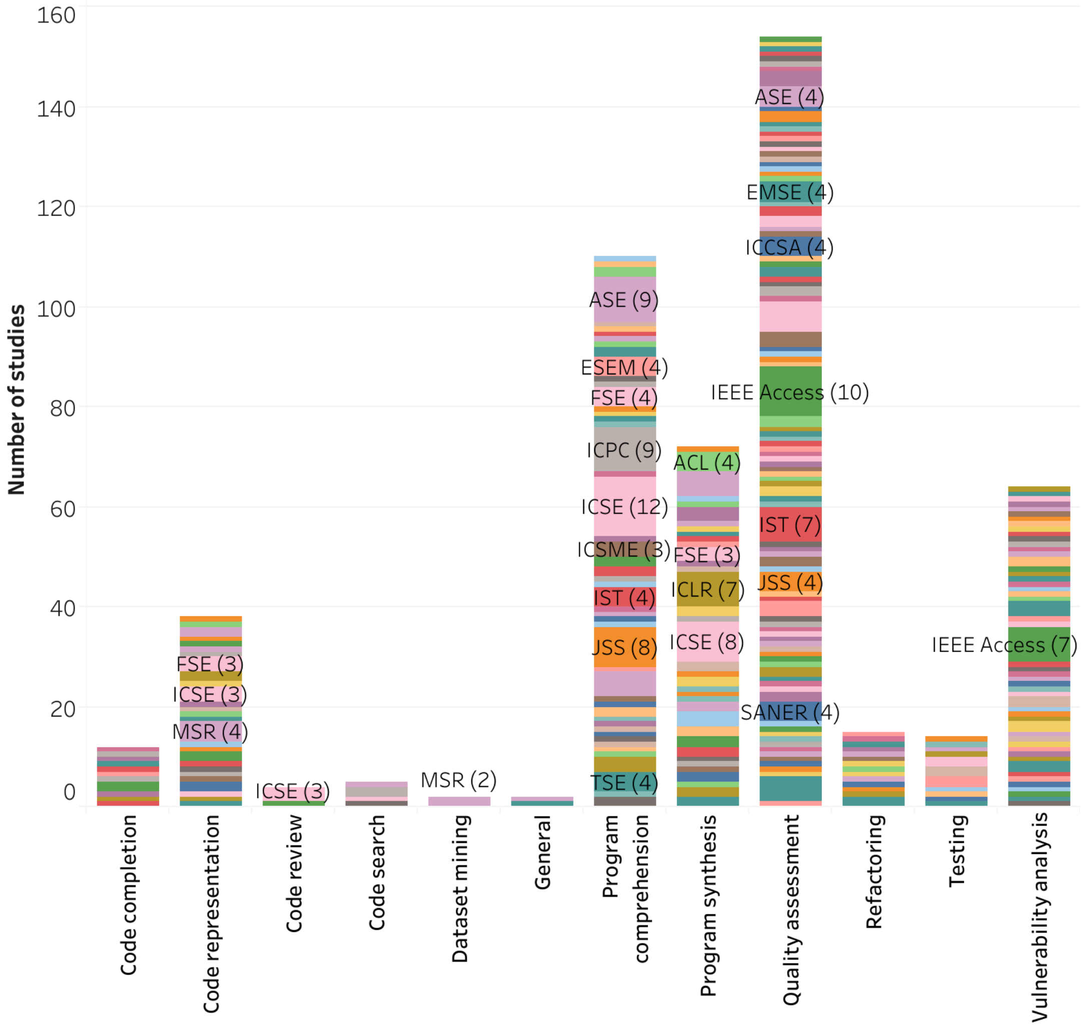


Figure 5. Top venues for each considered category

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

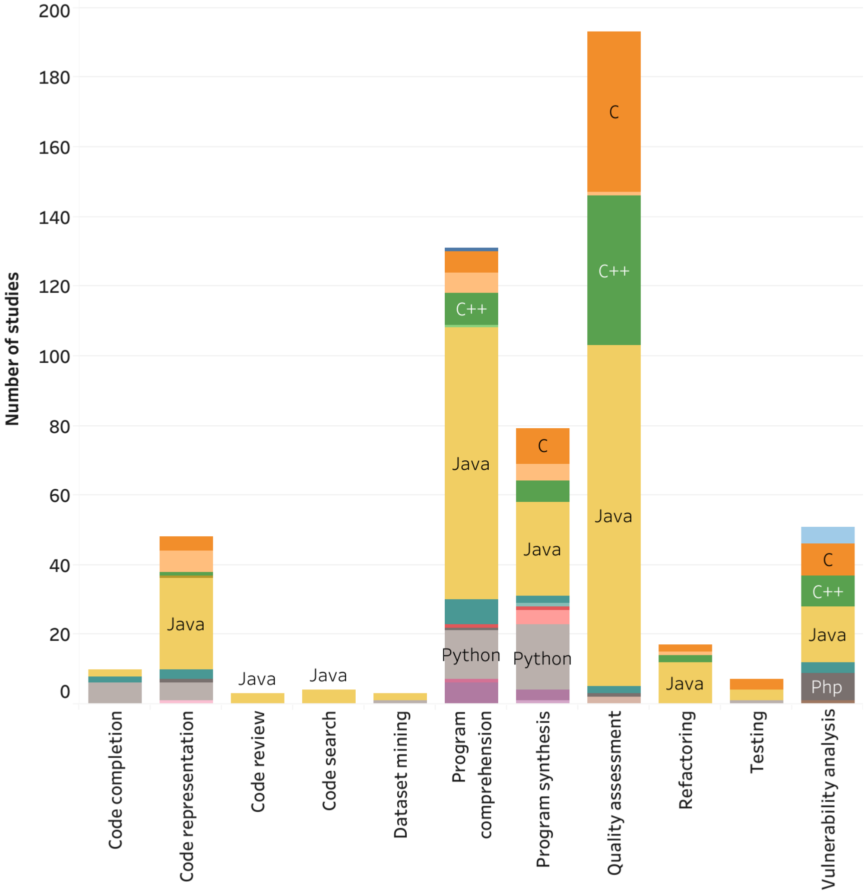


Figure 6. Target programming languages for each considered category

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

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/

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.

3.3.1 Program repair

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

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

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

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

Other studies use existing bug benchmarks, such as Defects4J [218] and IntroClass [250], which already include buggy revisions and human fixes, to evaluate their approaches. For instance, Saha et al. [416], Lou et al. [299], Zhu et al. [582], Renzullo et al. [406], Wang et al. [518], and Chen

[2] The term semantic correctness is a criterion for evaluating whether a generated patch is similar to the human fix for a given bug [291].

et al. [101] leveraged Defects4J for the evaluations of their approaches. Additionally, Dantas et al. [118] used the IntroClass benchmark and Majd et al. [313] conducted experiments using 119,989 C/C++ programs within Code4Bench. Wu et al. [534] used the DeepFix dataset that contains 46,500 correct C programs and 6,975 programs with errors for their graph-based dl approach for syntax error correction.

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

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

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

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

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

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

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

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

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

3.3.2 Code generation

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

Dataset preparation: Yin and Neubig [552] proposed a transition-based neural semantic parser, namely TRANX, which generates formal meaning representation from natural language text. They used multiple datasets for their study—dataset proposed by Dong and Lapata [128] containing 880 geography-related questions, *Django* dataset [358], as well as *WikiSQL* dataset [576]. Similarly, Sun et al. [468] and Shin et al. [446] used the *HearthStone* dataset [283] for Python code generation; in addition, Shin et al. [446] used the Spider [557] dataset for training. Liang et al. [272] used the semantic parsing dataset *WebQuestionsSP*[550] consisting 3,098 question-answer pairs for training and 1,639 for testing. Bielik et al. [60] used the *Linux Kernel* dataset [222], and the *Hutter Prize Wikipedia* dataset.[5] Devlin et al. [122] evaluated their architecture on 205 real-world *Flash-Fill* instances [170]. Xiong et al. [537] used training data stemming from two *Defects4J* projects and their related JDK packages. Wei et al. [530] conducted experiments on Java and Python projects collected from GitHub used by previous work (such as by Hu et al. [198], Hu et al. [199], Wan et al. [511]).

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

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

Some studies in the category used examples of input and output to learn code generation. *Euphony* [257] learns good representation using easily obtainable solutions for given programs. *DeepCoder* [47] observes inputs and outputs, by leveraging information from interpreters. Then,

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

DeepCoder searches for a program that matches the input-output examples. Similarly, Chen et al. [99] developed a neural program synthesis from input-output examples. Shu and Zhang [449] extracted features from string transformations, *i.e.,* input-output strings, and use the learned features to induce correct programs. Devlin et al. [122] used I/O programming examples and developed a DSL for synthesizing related programs.

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

ML model training: A majority of the studies in this category relies on the RNN-based 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.

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.

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

3.3.3 Program translation

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

3.4 Quality assessment

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

3.4.1 Code smell detection

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

Dataset preparation: The process of identifying code smells requires a dataset as a ground truth for training an ML model. Each sample of the training dataset must be tagged appropriately as smelly sample (along with target smell types) or non-smelly sample. Many authors built their datasets tagged manually with annotations. For example, Fakhoury et al. [139] developed a manually validated oracle containing $1,700$ instances of linguistic smells. Pecorelli et al. [375] created a dataset of 8.5 thousand samples of smells from 13 open-source projects. Some authors [11, 336, 110, 206, 180] employed existing datasets (Landfill and Qualitas) in their studies. Tummalapalli et al. [500, 497, 499] used 226 WSDL files from the tera-PROMISE dataset. Oliveira et al. [360] relied on historical data and mined smell instances from history where the smells were refactored.

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

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

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

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

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

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

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

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

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

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

3.4.2 Code clone detection

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

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

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

ML model training:

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

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

3.4.3 Defect prediction

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

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

used Apache projects and Qiao et al. [393] used MIS dataset [306]. Xiao et al. [535] utilized a Continuous Integration (CI) dataset and Pradel and Sen [387] generated a synthetic dataset. Apart from using the existing datasets, some other studies prepared their own datasets by utilizing various 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 models. 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 and Python code completion. Svyatkovskiy et al. [473] used 2,700 Python open-source software GitHub repositories for the evaluation of their novel approach, Pythia.

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

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

ML model training: The studies can be classified based on the used ml technique for code completion.

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

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

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

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

3.6 Program Comprehension

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

3.6.1 Code summarization

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

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

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

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

Some of the studies used existing datasets such as StaQC [547] and the dataset created by Jiang et al. [212]. Specifically, Liu et al. [295], Jiang and McMillan [214] utilized a dataset of commits provided by Jiang et al. [212] that contains two million commits from one thousand popular Java projects. Yao et al. [548] and Ye et al. [549] used StaQC dataset [547]; it contains more than 119 thousand pairs of question title and code snippet related to SQL mined from StackOverflow. Xie et al. [536] utilized two existing datasets—one each for Java [251] and Python [53]. Bansal et al. [51] evaluated their code summarization technique using a Java dataset of 2.1M Java methods from 28K projects created by LeClair and McMillan [251]. Li et al. [268] also used the Java dataset of 2.1M methods LeClair and McMillan [251] to predict the inconsistent names from the implementation of the methods. 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 traversal algorithm to generate the code token sequence preserving the semantic, syntactic and structural information in the code snippet.

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

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

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

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

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

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

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

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

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

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

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

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

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

3.7 Code review

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

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

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

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

3.8 Code search

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

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

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

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

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

3.9 Refactoring

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

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.

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.

3.10 Vulnerability analysis

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.

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).

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

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

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

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

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

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

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

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

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

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

DL techniques: A large number of studies [543, 412, 231, 280, 48, 232, 327, 278, 448, 54] used DL methods such as CNN, RNN, and ANN to train models. In more details, Yang et al. [543] utilized the BP-ANN algorithm to train vulnerability detectors. For the project *Achilles*, Saccente et al. [412] used an array of LSTM models to train on data containing Java code snippets for a specific set of vulnerability types. In another study, Kim et al. [231] suggested a DL framework that makes use of RNN models to train vulnerability detectors. Specifically, the authors framework first feeds the code embeddings into a bi-lstm model to capture the feature semantics, then an attention layer is used to get the vector weights, and, finally, passed into a dense layer to output if a code is safe or vulnerable. Compared to the studies that examined traditional ml or dl algorithms, Zheng et al. [573] examined both of them. They used *Random Forest*, *K Nearest Neighbors*, *Support Vector Machine*, *Linear Regression* among the traditional ml algorithms along with bi-lstm, gru, and cnn. There results indicate bi-lstm as the best performing model. Lin et al. [280] developed a benchmarking framework that can use bi-lstm, lstm, bi-gru, gru, dnn and Text-cnn, but can be extended to use more deep learning models. Kim et al. [232] generating graphical semantics that reflect on code semantic features and use them for Graph Convolutional Network to automatically identify and learn semantic and extract features for vulnerability detection, while Shiqi et al. [448] created textual images and fed them to Deep Belief Networks to classify malware.

3.11 Summary

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

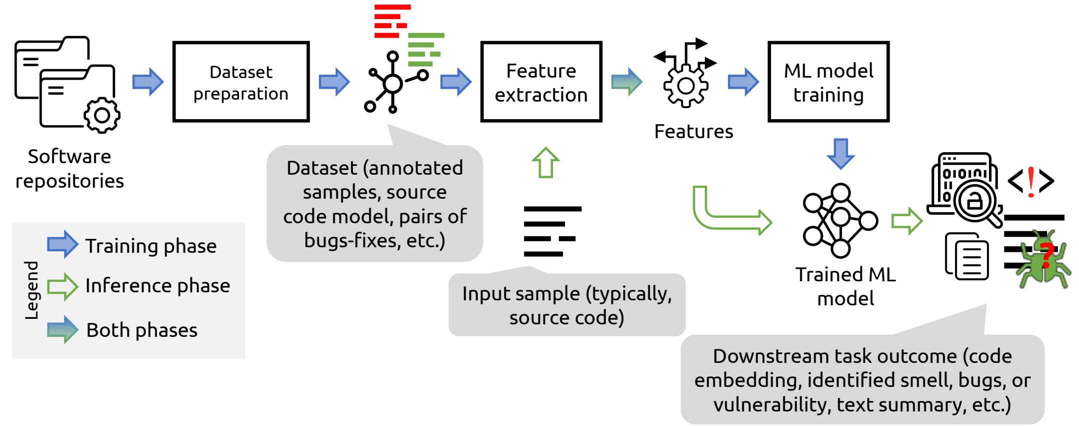

Software repositories
Dataset preparation
Feature extraction
Features
ML model training
Dataset (annotated samples, source code model, pairs of bugs-fixes, etc.)
Input sample (typically, source code)
Trained ML model
Downstream task outcome (code embedding, identified smell, bugs, or vulnerability, text summary, etc.)
Legend
Training phase
Inference phase
Both phases

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

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

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

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

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

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

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

4. Datasets and Tools

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

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

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

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

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

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
	Change-Scribe [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

	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

	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

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

	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

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

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

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

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

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

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

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

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

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

[10] https://mlflow.org/

- ***Data privacy and bias:*** Data hungry ML models are considered as good as the data they are consuming. Data collection and preparation without data diversity leads to bias and unfairness. Although we are witnessing more efforts to understand these sensitive aspects [566, 70], the present set of methods and practices lack the support to deal with data privacy issues at large as well as data diversity and fairness [70, 155].
  Mitigation: Data standards and best practices focusing on data privacy could be considered as an evaluation criterion to mitigate issues concerning data privacy and bias. In addition, mitigation of the issue is also linked with appropriate data pre-processing. Adoption of effective anonymization techniques and data quality assurance practices will further help us deal with the concern.
- ***Effective feature engineering:*** Features represent the problem-specific knowledge in pieces extracted from the data; the effectiveness of any ML model depends on the features fed into it. Many studies identified the importance of effective feature engineering and the challenges in gathering the same [487, 440, 373, 513, 203]. Specifically, software engineering researchers have notified that identifying and extracting relevant features beyond code quality metrics is non-trivial. For example, Ivers et al. [203] discusses that identifying features that establishes a relationship among different code elements is a significant challenge for ML implementations applied on source code analysis. Sharma et al. [437] have shown in their study that smell detection using ML techniques perform poorly especially for design smells where multiple code elements and their properties has to be observed.
  Mitigation: Recent advancements in the field of large language models (LLMs) trained on huge corpus of code and text have significantly eased the task for researchers. For example, tasks such as generating code embeddings and fine-tuning are supported natively by the LLMs. However, encoding code features specific to downstream tasks is required often and making the task easier requires a significant push from the research community.
- ***Skill gap:*** Wan et al. [513] identified that ML software development requires an extended set of skills beyond software development including ML techniques, statistics, and mathematics apart from the application domain. Similarly, Hall and Bowes [181] also reports a serious lack of ML expertise in academic software engineering efforts. Other authors [373] have emphasized the importance of domain knowledge to design effective ML models.
  Mitigation: Raising awareness and training sessions customized for the audience is considered the mitigation strategy for skill gap. Software engineering conferences organize tutorials that typically helps new researchers in the field. Availability of various hands-on courses and lecture series from known universities also help bringing the gap.
- ***Hardware resources:*** Given the need of large training datasets and many hidden layers, often ML training requires high-end processing units (such as GPUs and memory) [513, 155]. A 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.

6. Threats to validity

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

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

External validity concerns the generalizability and reproducibility of the produced results and observations. We provide a spreadsheet [438] containing all the metadata for all the articles selected in each of the phases of article selection. In addition, inspired by previous surveys [27, 195], we have developed a website[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.

7. Conclusions

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

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

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

Acknowledgements

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

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

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

References

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

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

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

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

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

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

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

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

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

[10] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. Compilation error repair: For the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '18, page 78–87, 2018. ISBN 9781450356602. doi: 10.1145/3183377.3183383.

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

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

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

[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.

[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.

[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 &amp; 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.

[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.

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

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

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

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

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

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

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

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

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

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

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

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

[80] M. Cetiner and O. K. Sahingoz. A comparative analysis for machine learning based software defect prediction systems. In *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1--7, 2020. doi: 10.1109/ICCCNT49239.2020.9225352.

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

[82] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, pages 1--1, 2020. doi: 10.1109/TSE.2020.3020502.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[107] C. J. Clemente, F. Jaafar, and Y. Malik. Is predicting software security bugs using deep learning better than the traditional machine learning algorithms? In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 95--102, 2018. doi: 10.1109/QRS.2018.00023.

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

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

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

[111] Daniel Cruz, Amanda Santana, and Eduardo Figueiredo. Detecting bad smells with machine learning algorithms: An empirical study. In *Proceedings of the 3rd International Conference on Technical Debt*, TechDebt '20, page 31–40, 2020. ISBN 9781450379601. doi: 10.1145/3387906.3388618.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

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

[170] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, aug 2012. ISSN 0001-0782. doi: 10.1145/2240236.2240260.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[191] Vincent J. Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 763–773, 2017. ISBN 9781450351058. doi: 10.1145/3106237.3106290.

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

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

[194] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 518–529, 2020. ISBN 9781450371216. doi: 10.1145/3377811.3380361.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[228] Muhammad Noman Khalid, Humera Farooq, Muhammad Iqbal, Muhammad Talha Alam, and Kamran Rasheed. Predicting Web Vulnerabilities in Web Applications Based on Machine Learning. In Imran Sarwar Bajwa, Fairouz Kamareddine, and Anna Costa, editors, *Intelligent Technologies and Applications*, Communications in Computer and Information Science, pages 473--484, Singapore, 2019. Springer. ISBN 9789811360527. doi: 10.1007/978-981-13-6052-7_41.

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

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

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

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

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

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

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

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

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

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

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

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

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

[242] Lov Kumar, Shashank Mouli Satapathy, and Lalita Bhanu Murthy. Method level refactoring prediction on five open source java projects using machine learning techniques. In *Proceedings of the 12th Innovations on Software Engineering Conference (Formerly Known as India Software Engineering Conference)*, ISEC'19, 2019. ISBN 9781450362153. doi: 10.1145/3299771.3299777.

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

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

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

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

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

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

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

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

[251] Alexander LeClair and Collin McMillan. Recommendations for datasets for source code summarization, 2019.

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

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

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

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

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

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

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

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

[260] Boao Li, Meng Yan, Xin Xia, Xing Hu, Ge Li, and David Lo. Deepcommenter: A deep code comment generation tool with hybrid lexical and syntactical information. 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 1571–1575, 2020. ISBN 9781450370431. doi: 10.1145/3368089.3417926.

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

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

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

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

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

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

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

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

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

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

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

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

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

[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.

[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.

[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.

[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.

[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.

[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.

[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.

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

[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.

[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.

[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.

[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.

[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.

[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.

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

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

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

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

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

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

[294] Yang Liu. Fine-tune bert for extractive summarization, 2019. URL https://arxiv.org/abs/1903.10318.

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

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

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

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

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

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

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

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

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

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

[305] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 101–114, 2020. ISBN 9781450380089. doi: 10.1145/3395363.3397369.

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

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

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

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

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

[311] Janaki T. Madhavan and E. James Whitehead. Predicting buggy changes inside an integrated development environment. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology EXchange*, eclipse '07, page 36–40, 2007. ISBN 9781605580159. doi: 10.1145/1328279.1328287.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[350] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen. A deep neural network language model with contexts for source code. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 323--334, 2018. doi: 10.1109/SANER.2018.8330220.

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

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

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

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

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

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

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

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

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

[360] Daniel Oliveira, Wesley K. G. Assunção, Leonardo Souza, Willian Oizumi, Alessandro Garcia, and Baldoino Fonseca. Applying machine learning to customized smell detection: A multi-project study. SBES '20, page 233–242, 2020. ISBN 9781450387538. doi: 10.1145/3422392.3422427.

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

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

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

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

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

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

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

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

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

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

[371] Sebastiano Panichella, Jairo Aponte, Massimiliano Di Penta, Andrian Marcus, and Gerardo Canfora. Mining source code descriptions from developer communications. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 63--72, 2012. doi: 10.1109/ICPC.2012.6240510.

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

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

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

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

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

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

[378] J. D. Pereira, J. R. Campos, and M. Vieira. An exploratory study on machine learning to combine security vulnerability alerts from static analysis tools. In *2019 9th Latin-American Symposium on Dependable Computing (LADC)*, pages 1--10, 2019. doi: 10.1109/LADC48089.2019.8995685.

[379] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 426–437, 2015. ISBN 9781450338325. doi: 10.1145/2810103.2813604.

[380] Hung Phan and Ali Jannesari. Statistical machine translation outperforms neural machine translation in software engineering: Why and how. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages*, RL+SE&amp;PL 2020, page 3–12, 2020. ISBN 9781450381253. doi: 10.1145/3416506.3423576.

[381] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. Cotext: Multi-task learning with code-text transformer, 2021. URL https://arxiv.org/abs/2105.08645.

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

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

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

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

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

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

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

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

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

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

[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&amp;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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[450] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. Improving code search with co-attentive representation learning. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 196–207, 2020. ISBN 9781450379588. doi: 10.1145/3387904.3389269.

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

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

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

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

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

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

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

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

[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.

[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 &amp; 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.

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

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

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

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

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

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

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

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

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

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

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

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

[496] Sahithi Tummalapalli, Lov Kumar, and N. L. Bhanu Murthy. Prediction of web service anti-patterns using aggregate software metrics and machine learning techniques. In *Proceedings of the 13th Innovations in Software Engineering Conference on Formerly Known as India Software Engineering Conference*, ISEC 2020, 2020. ISBN 9781450375948. doi: 10.1145/3385032.3385042.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[518] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao. How different is it between machine-generated and developer-provided patches? : An empirical study on the correct patches generated by automated program repair techniques. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1--12, 2019. doi: 10.1109/ESEM.2019.8870172.

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

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

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

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

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

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

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

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

[527] Xinda Wang, Shu Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. A machine learning approach to classify security patches into vulnerability types. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1--9, 2020. doi: 10.1109/CNS48642.2020.9162237.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[564] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, pages 1--1, 2020. doi: 10.1109/TSE.2019.2962027.

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

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

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

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

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

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

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

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

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

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

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

[576] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning, 2017. URL https://arxiv.org/abs/1709.00103.

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

[578] Yu Zhou, Xin Yan, Wenhua Yang, Taolue Chen, and Zhiqiu Huang. Augmenting java method comments generation with context information based on neural networks. *Journal of Systems and Software*, 156:328--340, 2019. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.07.087. URL https://www.sciencedirect.com/science/article/pii/S0164121219301529.

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

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

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

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

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