A Survey on Machine Learning Techniques for Source Code Analysis

TUSHAR SHARMA, Dalhousie University, Canada
MARIA KECHAGIA, University College London, UK
STEFANOS GEORGIOU, Queen’s University, Canada
ROHIT TIWARI, DevOn, India
INDIRA VATS, J.S.S. Academy of Technical Education, India
HADI MOAZEN, Dalhousie University, Canada
FEDERICA SARRO, University College London, UK

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 479 primary studies published between 2011 and 2021. 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.

CCS Concepts: • Software and its engineering → Software libraries and repositories; Software maintenance tools; Software post-development issues; Maintaining software; • Computing methodologies → Machine learning.

Additional Key Words and Phrases: Machine learning for software engineering, source code analysis, deep learning, datasets, tools

ACM Reference Format:
Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, Indira Vats, Hadi Moazen, and Federica Sarro. 2022. A Survey on Machine Learning Techniques for Source Code Analysis. 0, 0, Article 0 (2022), 73 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In the last two decades, we have witnessed significant advancements in Machine Learning (ml) and Deep Learning (dl) techniques, specifically in the domain of image [213, 432], text [4, 233], and speech [146, 147, 379] 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 and dl techniques to address software engineering problems [22, 33, 225, 463, 503].

The software engineering community has employed ml and dl techniques for a variety of applications such as software testing [249, 328, 509], source code representation [22, 171], source code quality analysis [33, 44].

Authors’ addresses: Tushar Sharma, tushar@dal.ca, Dalhousie University, Halifax, Canada; Maria Kechagia, m.kechagia@ucl.ac.uk, University College London, London, UK; Stefanos Georgiou, stefanos.georgiou@queensu.ca, Queen’s University, Kingston, Canada; Rohit Tiwari, rohit.beawar@gmail.com, DevOn, Bangalore, India; Indira Vats, indiravats@gmail.com, J.S.S. Academy of Technical Education, Noida, India; Hadi Moazen, hadimoazen@ymail.com, Dalhousie University, Halifax, Canada; Federica Sarro, f.sarro@ucl.ac.uk, University College London, London, UK.

2022. XXXX-XXXX/2022/0-ART0 $15.00
https://doi.org/10.1145/nmmnnn.nmmnnn

, Vol. 0, No. 0, Article 0. Publication date: 2022.
program synthesis [225, 488], code completion [262], refactoring [38], code summarization [21, 230, 267], and vulnerability analysis [390, 401, 451] 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 fact, there have been numerous recent attempts to summarize the application-specific knowledge in the form of surveys. For example, Allamanis et al. [22] present key methods to model source code using ML techniques. Shen and Chen [401] provide a summary of research methods associated with software vulnerability detection, software program repair, and software defect prediction. Durelli et al. [115] collect 48 primary studies focusing on software testing using machine learning. Alsolai and Roper [33] present a systematic review of 56 studies related to maintainability prediction using ML techniques. Recent surveys [13, 44, 440] summarize application of ML techniques on software code smells and technical debt identification. Similarly, literature reviews on program synthesis [225] and code summarization [315] 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. We can observe that our survey makes the following additions to the state-of-the-art surveys: it covers a wide range of software engineering activities; it summarizes a significantly large number of primary 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 and DL techniques for source code analysis. Source code analysis involves tasks that take the source code as input, process it, and/or produces 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 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 challenges in this domain and present them in a synthesized form. 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 source code analysis domain.
- It consolidates resources (such as datasets and tools) relevant for future studies in this domain.
- It provides a synthesized summary of the open challenges that requires the attention of the researchers.

In this paper, for the sake of simplicity, we use ML techniques to refer to both ML and DL techniques and models, unless explicitly specified.

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 3, Section 4, Section 5, and Section 6 provide the detailed results of our documented findings corresponding to each research question. We present discussion in Section 7, threats to validity in Section ??, and conclude the paper in Section 8.
Table 1. Comparison Among Surveys. The “Category” column refers to the software engineering task-based category of the survey where ML is used, column “Data&Tools” means that a survey reviews available datasets and tools for ML-based applications, column “Challenges” shows whether the study identifies challenges related to ML applications, column “Type” refers to the type of literature survey, and column “#Studies” refers to the number of primary studies included in a considered study. We tag a study with “-” to indicate that the field is not applicable for the study and NA for the number of studies column where the study does not explicitly mention the selection criteria and the number of selected studies.

| Category                        | Article                  | Data & Tools | Challenges | Type         | #Studies |
|---------------------------------|--------------------------|--------------|------------|--------------|----------|
| **Program Comprehension**       | Abbas et al. [2]         | Yes          | No         | Meta-analysis | –        |
|                                 | Uchôa et al. [453]       | Yes          | No         | Meta-analysis | –        |
| **Testing**                     | Omri and Sinz [328]      | No           | No         | Lit. survey   | NA       |
|                                 | Durelli et al. [115]     | No           | Yes        | Mapping study | 48       |
|                                 | Hall and Bowes [160]     | Yes          | Yes        | Meta-analysis | 21       |
|                                 | Zhang et al. [509]       | No           | Yes        | Lit. survey   | 46       |
|                                 | Pandey et al. [334]      | No           | Yes        | Lit. survey   | 154      |
|                                 | Singh et al. [410]       | No           | No         | Lit. survey   | 13       |
| **Vulnerability analysis**      | Li et al. [246]          | Yes          | Yes        | Meta-analysis | –        |
|                                 | Shen and Chen [401]      | No           | Yes        | Meta-analysis | –        |
|                                 | Ucci et al. [451]        | No           | Yes        | Lit. survey   | 64       |
|                                 | Jie et al. [192]         | No           | No         | Lit. survey   | 19       |
|                                 | Hanif et al. [166]       | No           | Yes        | Lit. survey   | 90       |
| **Quality assessment**          | Alsolai and Roper [33]   | No           | No         | Lit. survey   | 56       |
|                                 | Tsintzira et al. [440]   | Yes          | Yes        | Lit. survey   | 90       |
|                                 | Azeem et al. [44]        | Yes          | No         | Lit. survey   | 15       |
|                                 | Caram et al. [70]        | No           | No         | Mapping study | 25       |
|                                 | Lewowski and Madeyski [235] | Yes         | No         | Lit. survey   | 45       |
| **Program synthesis**           | Goues et al. [143]       | No           | Yes        | Lit. survey   | NA       |
|                                 | Le et al. [225]          | Yes          | Yes        | Lit. survey   | NA       |
| **Program synthesis & code representation** | Allamanis et al. [22] | Yes | Yes | Lit. survey | 39+48 |
| **Source-code analysis**        | Our study                | Yes          | Yes        | Lit. survey   | 479      |

2 Methodology

First, we present the objectives of this study and the research questions derived from such objectives. Second, we describe the search protocol that 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 objectives.

- **Identifying specific tasks involving source code analysis that has been attempted using machine learning.**
  
  We would like to investigate different types of code analysis tasks that have been attempted using ML
techniques. We aim to summarize how ml methods and techniques are helping specific software engineering tasks.

- **Summarizing the machine learning techniques used for source code analysis tasks.**
  This objective explores different ml techniques commonly used for source code analysis. We attempt to synthesize a mapping of code analysis tasks along with sub-tasks and steps, and corresponding ml techniques.

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

- **Discussing the challenges and perceived deficiencies in ml-enabled source code analysis.**
  With this concrete objective, we aim to present the perceived deficiencies, challenges, and opportunities in the software engineering field specifically in the context of applying ml techniques observed from the collected articles.

2.2 Literature search protocol

We identified 479 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 the section.

![Fig. 1. Overview of the search process](image)

### 2.2.1 Literature search — Phase 1

We split the phase 1 literature search into two rounds. In the first round, we carried out an extensive initial search on six well-known digital libraries—Google Scholar, SpringerLink, ACM Digital Library, ScienceDirect, IEEE Xplore, and Web of Science during Feb-Mar 2021. We formulated a set of search terms based on common tasks and software engineering activities related to source code analysis.
Specifically, we used the following terms for the search: 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 combined the keyword with other keywords or with other relevant words. 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. We carried out this additional round of literature search to augment our initial search terms and reduce the risk of missing relevant articles in our search. The search terms used in the second round of phase 1 can be found in our replication package [396]. Next, we defined inclusion and exclusion criteria to filter out irrelevant studies.

Inclusion criteria:

- Studies that discuss source code analysis using a ML technique (including DL).
- Surveys discussing source code analysis using ML techniques.
- 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 analysis 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 small size).
- Studies whose full text is not available, or be written in any other language than English.

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 replication package online [396]. 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 In 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 [22, 44, 129, 401]. 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 the article contributes the most. The assignment is carried out by one of the authors and verified by two other authors; disagreements were discussed and resolved to reach a consensus. 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 To keep our focus on the most recent studies, we marked the studies published before 2011 as out of scope. 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 studies that do not satisfy our inclusion criteria (such as when the article is too small or do not employ any ML technique for source code analysis and processing tasks) after reading the whole article. The remaining 480 articles are the primary studies that we examine in detail. For each study, we extracted the core idea and contribution, the ML techniques and tools used as well as challenges and findings unveiled. Next, we present our observations corresponding to each research goal we pose.

3 Categorizing ML-enabled Source code analysis tasks

![Fig. 2. Category-wise distribution of studies](image)

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 it 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 attracting 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.

| Category Sub-category | Count of Category |
|-----------------------|------------------|
| ...                  | 17               |

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

Some of the categories are quite generic and hence further categorization is possible based on specific tasks. For example, category testing is further divided into defect prediction, and test data/case generation. We assigned sub-categories to the studies wherever applicable; if none of the sub-categories is appropriate for a study, we assigned it to the parent category. Figure 3 presents the distribution of studies per year w.r.t. each category and corresponding sub-categories.

To quantify the growth of each category, we compute the average increase in the number of articles from the last year for each category between the years 2012 and 2021. We did not include the year 2022 because the year has not completed at the time of writing this survey and the obviously our collection includes the partial set of articles published in 2022. We observed that the program comprehension and vulnerability analysis categories grew most with approximately 63.9% and 64.2% average growth each year, respectively.

4 Machine learning techniques used for source code analysis

We document our observations per category and subcategory by providing a summary of the existing efforts. Figure 4 and Figure 5 show the frequency of various ML techniques per category used in the primary studies. The figures use commonly used acronyms for ML techniques along with their corresponding expanded form; we utilize these acronyms throughout the paper. It is evident from the figures that SVM and RF are the most frequently employed ML techniques. From the DL side, the RNN family (including LSTM and GRU) is the most commonly used in this context.

In the rest of the 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 crucial steps for each category (e.g., model generation, data sampling, feature extraction, and model training).
Fig. 4. Usage of ML techniques in the primary studies
### 4.1 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. 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 \[22, 318\]; the proposed techniques extensively

---

#### Table: Usage of ML techniques in the primary studies

| ML techniques                  | Acronyms | Code presentation | Code review | Code search | Data mining | Program comprehension | Quality assessment | Refactoring | Testing | Vulnerability analysis | Total |
|-------------------------------|----------|-------------------|-------------|-------------|-------------|-----------------------|-------------------|-------------|---------|------------------------|-------|
| ID3                           | ID3      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| JIR                           | JR       |                   |             |             |             |                       |                   |             |         |                       | 1     |
| K Nearest Neighbours          | KNN      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Kernel based learning         | KBL      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| KMeans                        | KM       |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Kstar                         | KST      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Lasso                         | LASSO    |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Least Median Square Regression | LMSR     |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Light GBM                     | LGBM     |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Linear Discriminant Analysis  | LDA      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Linear Regression             | LR       |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Locally deep support vector machine | LDSVM |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Logistic Linear Regression    | LLR      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Logistic regression           | LOG      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| LogitBoost                    | LOG      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Long Short Term Memory        | LSTM     |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Majority Voting Ensemble      | MVE      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Maximal Marginal Relevance    | MRR      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Modular Tree-structured RNN    | MTN      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Multi Level Perceptron        | MLP      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Multinomial Naive Bayes       | MNB      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Multiple kernel ensemble learning | MKEL |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Naive Bayes                   | NB       |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Neural Language Model          | NLM      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Neural Machine Translation    | NMT      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Neural Network                | NN       |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Neural Network for Discrete goal | NND  |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Node2Vec                      | Node2Vec |                   |             |             |             |                       |                   |             |         |                       | 1     |
| One Class Classifier          | OCC      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Particle Swarm Optimization   | PSO      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Pointer Network               | PN       |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Polynomial regression         | POLY     |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Probabilistic Neural Network  | PNN      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Radial Basis Function Network | RBFN     |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Random Forrest                | RF       |                   |             |             |             |                       |                   |             |         |                       | 1     |
| RandomTree                    | RT       |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Recurrent Neural Network      | RNN      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Regression Neural Network     | RNN      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Reinforcement Learning        | RL       |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Residual Neural Network       | ResNet   |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Reverse NN                    | ReNN     |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Ripper                        | Ripper   |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Sequence-to-Sequence          | Seq2Seq  |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Sequential Minimal Optimization | SMO    |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Simple Logistic               | SL       |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Statistical Machine Translation | SMT    |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Stochastic Gradient Descent   | SGD      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Support Vector Machine        | SVM      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Support Vector Regression     | SVR      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Text CNN                      | Text CNN |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Transfer Naive Bayes          | TNB      |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Transformer                   | TF       |                   |             |             |             |                       |                   |             |         |                       | 1     |
| Word2Vec                      | Word2Vec |                   |             |             |             |                       |                   |             |         |                       | 1     |
| XGBoost                       | XG       |                   |             |             |             |                       |                   |             |         |                       | 1     |

Fig. 5. Usage of ML techniques in the primary studies
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 [311, 435], method name prediction [31], and comment classification [464].

Figure 6 provides an overview of a typical pipeline associated with code representation. 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 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.

**Model generation:** Code representation efforts start with preparing a source code model. The majority of the studies generate ast [27, 29–31, 62, 76, 83, 318, 474, 488, 506]. Some studies [25, 43, 77, 117, 196, 312, 320, 400, 515] parsed source code as tokens and prepared a sequence of tokens in this step. Hoang et al. [173] generated tokens representing only the code changes. Furthermore, Sui et al. [421] compiled a program into llvm-ir. The inter-procedural value-flow graph (ivfg) is built on top of the intermediate representation. Thaller et al. [435] used abstract semantic graph as their code model. Nie et al. [321] used dataset offered by Jiang et al. [186] that offers a large number code snippets and comment pairs. Finally, Brauckmann et al. [61] and Tufano et al. [442] 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. [318] prepared vectors of syntactic context (referred to as syntaxeme), type context (sememes), and lexical tokens. Shedko et al. [400] generated a stream of tokens corresponding to function calls and control flow expressions. Karampatsis et al. [198] 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. [211] extracted path context representing two tokens in code and a structural connection along with paths between ast nodes. Alon et al. [29] 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. Finally, Alon et al. [31] 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. [83] created AST node identified by an API name and attached each node to the corresponding AST node belonging to the identifier. Thaller et al. [435] proposed feature maps; feature maps are human-interpretation, stacked, named subtrees extracted from abstract semantic graph. Brauckmann et al. [61] created a dataflow-enriched AST graph, where nodes are labeled as declarations, statements, and types as found in the Clang\footnote{https://clang.llvm.org/} AST. Cvetkovic et al. [103] augmented AST with semantic information by adding a graph-structured vocabulary cache. Finally, Zhang et al. [506] 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 [173, 442] 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 [29, 61, 171, 464, 474] employed LSTM-based models; while others [62, 173, 198, 488, 506] used GRU-based models. Among the other kinds of ML models, studies employed GNN-based [103, 478], DNN [318], conditional random fields [30], SVM [248, 357], and CNN-based models [83, 311, 435]. Some of the studies rely on the combination of different DL models. For example, Tufano et al. [442] 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. [197] carried out an empirical study to observe whether the embeddings generated by Code2Vec can be used in other contexts. Similarly, Pour et al. [348] 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 depends 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.

### 4.2 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. Following the definition, we include defect prediction studies in this category where authors extract features to train ML models to find bugs in software applications. Then, we offer a discussion on effort prediction models used to identify the time needed to test an application. Finally, we present studies associated with test cases generation by employing ML techniques.

#### 4.2.1 Defect prediction

To pinpoint bugs in software, researchers used various ML approaches. Figure 7 depicts a common pipeline used to train a defect prediction model. 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.

**Data labeling:** 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 \[12, 15, 54, 67, 73, 74, 78, 82, 95, 104, 107, 109, 110, 112, 120, 139, 200, 204, 223, 238, 241, 255, 279, 283, 287, 289, 292, 326, 331, 349, 356, 364, 384, 403, 411–413, 415, 425, 426, 468, 469, 471, 498, 510\] used the PROMISE dataset \[385\]. Some studies used other datasets in addition to the PROMISE dataset. For example, Liang et al. \[247\] used Apache projects and Qiao et al. \[356\] used mis dataset \[277\]. Xiao et al. \[483\] utilized a Continuous Integration \(\text{(ci)}\) dataset and Pradel and Sen \[350\] generated a synthetic dataset. Apart from using the existing datasets, some other studies prepared their own datasets by utilizing various GitHub projects \[7, 169, 284, 286, 338, 413, 443\] including Apache \[59, 94, 105, 124, 245, 285, 330, 361, 417\], Eclipse \[105, 521\] and Mozilla \[209, 281\] projects, or industrial data \[59\].

**Feature extraction:** The most common features to train a defect prediction model are the source code metrics introduced by Halstead \[161\], Chidamber and Kemerer \[92\], and McCabe \[297\]. Most of the examined studies \[12, 15, 20, 39, 67, 73, 78, 79, 94, 95, 107, 120, 124, 139, 194, 200, 204, 209, 223, 279, 283–287, 289, 292, 326, 330, 338, 351, 356, 361, 388, 412, 415, 417, 425, 426, 471, 472, 498\] used a large number of metrics such as Lines of Code, Number of Children, Coupling Between Objects, and Cyclomatic Complexity. Some authors \[331, 414\] combined detected code smells with code quality metrics. Furthermore, Felix and Lee \[127\] used defect metrics such as defect density and defect velocity along with traditional code smells.

In addition to the above, some authors \[54, 74, 109, 349\] suggested the use of dimensional space reduction techniques—such as Principal Component Analysis \(\text{(PCA)}\)—to limit the number of features. Pandey and Gupta \[333\] used Sequential Forward Search \(\text{(sfs)}\) to extract relevant source code metrics. Dos Santos et al. \[112\] suggested a sampling-based approach to extract source code metrics to train defect prediction models. Kaur et al. \[201\] suggested an approach to fetch entropy of change metrics. Bowes et al. \[59\] introduced a novel set of metrics constructed in terms of mutants and the test cases that cover and detect them.

Other authors \[350, 510\] used embeddings as features to train models. Such studies, first generate \text{ast} \[124, 238, 245, 247, 332\], a variation of \text{ast} such as Simplified \text{ast} \[81, 255\], or \text{ast-diff} \[443, 468\] for a selected method or

\[\text{Vol. 0, No. 0, Article 0. Publication date: 2022.}\]
A Survey on Machine Learning Techniques for Source Code Analysis

file. Then, embeddings are generated either using the token vector corresponding to each node in the generated tree or extracting a set of paths from ast. Singh et al. [413] 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. [104]. 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 the file is clear from bugs or not.

Wang et al. [469] employed the Deep Belief Network algorithm (dbn) to learn semantic features from token vectors, which are fetched from applications’ asts. Shi et al. [403] used a dnn model to automate the features extraction from the source code. Xiao et al. [483] 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 [281] 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. [82] 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 [458]. 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 [15, 20, 54, 67, 73, 74, 78, 94, 95, 107, 109, 112, 127, 139, 163, 200, 201, 204, 223, 283–287, 289, 292, 326, 330, 333, 338, 349, 351, 356, 361, 364, 411–415, 417, 425, 426, 469, 498] used traditional ml algorithms such as Decision Tree, Random Forest, Support Vector Machine, and AdaBoost. Similarly, Jing et al. [194], Wang et al. [472] used Cost Sensitive Discriminative Learning. In addition, authors [241, 279, 471] proposed changes to traditional ml algorithms to train their models. Specifically, Wang and Yao [471] suggested a dynamic version of AdaBoost.NC that adjusts its parameters automatically during training. Similarly, Li et al. [241] proposed ACoForest, an active semi-supervised learning method to sample the most useful modules to train defect prediction models. Ma et al. [279] 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 [82, 104, 245, 350, 388] used dl models such as cnn and rnn-based models for defect prediction. Specifically, Al Qasem et al. [12], Chen et al. [82], Li et al. [238], Pan et al. [332] used cnn-based models to predict bugs. rnn-based methods [81, 104, 124, 247, 255, 443] are also frequently used where variations of lstm are used to do defect prediction. Moreover, by using dl approaches, authors achieved improved accuracy for defect prediction and they pointed out bugs in real-world applications [245, 350].

**4.2.2 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.

**Data generation and pre-processing:** 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. [60] recorded traces for applications to capture usage data. They sanitized any irrelevant information collected from the programs recording components. AppFlow [176] captures human-event sequences from a smart-phone screen in order to identify tests. Similarly, Nguyen et al. [319] suggested Shinobi, a framework that uses a fast r-cnn model to identify input data fields from multiple web-sites. Utting et al. [455] captured user and system execution traces to
help generating missing API tests. To automatically identify metamorphic relations, Nair et al. [313] 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 [60, 455] used execution traces as features. Kim et al. [206] suggested an approach that replaces Snort’s meta-heuristic algorithms with deep reinforcement learning to generate test cases based on branch coverage information. [145] used code quality metrics such as coupling, DIT, and NOR to generate test data; they use the generate test data to predict the code coverage in a continuous integration pipeline.

**Train ML model:** 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 [60, 206, 313, 455] used traditional ML algorithms, such as Support Vector Machine, Naive Bayes, Decision Tree, Multilayer Perceptron, Random Forest, AdaBoost, Linear Regression. Nguyen et al. [319] used the DL algorithm Fast R-CNN. Similarly, [138] used LSTM to automate generating the input grammar data for fuzzing.

### 4.3 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 refactoring and program translation sub-categories in this section.

#### 4.3.1 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 unintended run-time behavior), which can be applied to software with a little or without human intervention [143]. Program repair typically consists of two phases. Initially, the repair tool uses fault localization to detect a bug in the software under examination, then, it generates patches using techniques such as search-based software engineering and logic rules that can possibly fix a given bug. To validate the generated patch, the (usually manual) evaluation of the semantic correctness of that patch follows.

According to Goues et al. [143], 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.

Figure 8 depicts the typical process performed by learning-aided-based repair tools. Typically, at the preprocessing 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.

**Data collection:** 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. [444] extracted millions of bug-fixing pairs from GitHub, Amorim et al. [37] leveraged the naturalness obtained from a corpus of known fixes, and Chen et al. [88] 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,

---

2The term semantic correctness is a criterion for evaluating whether a generated patch is similar to the human fix for a given bug [264].
Gopinath et al. [141] used a suite of programs and datasets stemmed from real-world applications, Long and Rinard [269] used a set of successful manual patches from open-source software repositories, and Mashhadi and Hemmati [295] used ManySSStuBs4J dataset containing natural language description and code snippets to automatically generate code fixes. Le et al. [226] 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. [188] used a dataset containing more than 4 million methods extracted. White et al. [480] 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. [345] 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 [69] 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. [260] prepared their dataset by selecting 636 closed bug reports from the Linux kernel and Mozilla databases. Svyatkovskiy et al. [430] constructed their experimental dataset from the 2700 top-starred Python source code repositories on GitHub.

Other studies use existing bug benchmarks, such as Defects4J [195] and IntroClass [227], which already include buggy revisions and human fixes, to evaluate their approaches. For instance, Saha et al. [377], Lou et al. [271], Zhu et al. [520], Renzullo et al. [367], Wang et al. [470], and Chen et al. [90] leveraged Defects4J for the evaluations of their approaches. Additionally, Dantas et al. [106] used the IntroClass benchmark and Majd et al. [282] conducted experiments using 119,989 C/C++ programs within Code4Bench. Wu et al. [482] used the DeepFix dataset that contains 46,500 correct C programs and 6,975 programs with errors for their graph-based dt. approach for syntax error correction.

Some studies examine bugs in different programming languages. For instance, Svyatkovskiy et al. [429] used 1.2 billion lines of source code in Python, C#, JavaScript, and TypeScript programming languages. Also, Lutellier et al. [276] 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. [156] used 6,975 erroneous C programs with typographic errors, Santos et al. [382] used source code files with syntax errors, and Sakkas
et al. [380] used a corpus of 4,500 ill-typed OCaml programs that lead to compile-time errors. Bhatia et al. [55] 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. [55], Gupta et al. [156], and Sakkas et al. [380] 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. [37], Jiang et al. [188], Santos et al. [382], Svyatkovskiy et al. [429], and Chen et al. [88], leveraged source-code naturalness and applied NLP-based metrics. Tian et al. [437] employed different representation learning approaches for code changes to derive embeddings for similarity computations. Similarly, White et al. [480] 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. [377] 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. [470] 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. [380] also applied similarity metrics. Svyatkovskiy et al. [430] 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. [456] 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 [269] more efficient and precise. Alrajeh et al. [32] identified counterexamples and witness traces using model checking for logic-based learning to perform repair process automatically. Cai et al. [68] 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 [69] 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. [444] 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 [123]. Additionally, Soto and Le Goues [416] 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. [303] 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. [243] used the prior bug fixes and the surrounding code contexts of the fixes for code transformation learning. Saha et al. [376] 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 [295] 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. [45] 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. [457] used results from localization.
of variable-misuse bugs. Wu et al. [482] developed an approach, ggr, for syntax-error correction that treats the code as a mixture of the token sequences and graphs. LIN et al. [250] and Zhu et al. [520] utilized ast paths to generate code embeddings to predict the correctness of a patch.

**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 languages. Lutellier et al. [276] 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. [444] used NMT techniques (Encoder-Decoder model) for learning bug-fixing patches for real defects, and generated repair patches. Mesbah et al. [303] introduced DeepDelta, which used NMT for learning to repair compilation errors. Jiang et al. [188] proposed CURE, a NMT-based approach to automatically fix bugs. Pinconschi et al. [345] used SequenceR, a sequence-to-sequence model, to patch security faults in C programs. Zhu et al. [520] 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.

**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. [437] introduced an empirical work that investigates 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. [37] 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. [456] 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 [269]. Gupta et al. [156] used word embeddings and reinforcement learning to fix erroneous C student programs with typographic errors. Tian et al. [437] 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. [377] used similarity analysis for repairing bugs that may require applying a substantially similar patch at a number of locations. Additionally, Wang et al. [470] used also similarity metrics to compare the differences among machine-generated and human patches. Santos et al. [382] used n-grams and NNs to detect and correct syntax errors.

**Logic-based rules:** Alrajeh et al. [32] combined model checking and logic-based learning to support automated program repair. Cai et al. [68] also combined model checking and ML for program repair. Shim et al. [405] 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. [380] 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 [269] 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 [416] 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. [141] 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. [376] developed Elixir repair approach that uses Logistic Regression models and
similarity-score metrics. Bader et al. [45] 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. [485] 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. [142] used Support Vector Machine and ID3 with path exploration to repair bugs in complex data structures. Le et al. [226] 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. [106] 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. [243] 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. [429] 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. [88] used the LSTM for synthesizing if-then constructs. Similarly, Vasic et al. [457] applied the LSTM in multi-headed pointer networks for jointly learning to localize and repair variable misuse bugs. Bhatia et al. [55] combined neural networks, and in particular RNNs, with constraint-based reasoning to repair syntax errors in buggy programs. Chen et al. [90] applied LSTM for sequence-to-sequence learning achieving end-to-end program repair through the SEQUENCE repair tool they developed. Majd et al. [282] developed SLDeep, statement-level software defect prediction, which uses LSTM on static code features.

Apart from above-mentioned techniques, White et al. [480] 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.

**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. [225] presented a survey on ML techniques including machine translation algorithms and applications. Chakraborty et al. [75] 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 to learn the probability distribution of changes in code. They evaluate CodIT on a dataset of 30k real-world changes and 6k patches. The evaluation reveals that CodIT can effectively learn and suggest patches, as well as learn specific bug fix patterns on Defects4J.

Oda et al. [325] 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.

**4.4 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.
4.4.1 Code smell detection

Code smells impair the code quality and make the software difficult to extend and maintain [399]. Extensive literature is available on detecting smells automatically [399]; ML techniques have been used to classify smelly snippets from non-smelly code. Figure 9 presents a common workflow for code smells detection using ML. 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.

**Sample generation and classification:** 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. [122] developed a manually validated oracle containing 1,700 instances of linguistic smells. Pecorelli et al. [340] created a dataset of 8.5 thousand samples of smells from 13 open-source projects. Some authors [11, 99, 159, 184, 304] employed existing datasets (Landfill and Qualitas) in their studies. Tummalapalli et al. [445, 449, 450] used 226 WSDL files from the tera-PROMISE dataset. Oliveira et al. [327] relied on historical data and mined smell instances from history where the smells are refactored. Some efforts such as one by Sharma et al. [395] used CodeSplit [393, 394] first to split source code files into individual classes and methods. Then, they used existing smell detection tools [392, 398] 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 [202] used smells identified by Dr Java, EMMA, and FindBugs as their gold-set. Alazba and Aljamaan [14] and Dewangan et al. [108] 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 [511] 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. [263] adopted an unusual 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 [8, 14, 36, 40, 50, 99, 100, 102, 108, 130, 131, 152, 154, 155, 158, 184, 199, 202, 218, 263, 304, 327, 353, 378, 436, 445, 448–450, 511] 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 [436] 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. [36] employed 62, Mhawish and Gupta [304] utilized 82, and Arcelli Fontana and Zanoni [40] 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. [274] used warnings generated from existing static analysis tools as features. Similarly, Ochodek et al. [324] 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. [447] and Gupta et al. [153] 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 [159] generated AST first and obtain the corresponding vector representation to train a model for smell detection. Furthermore, Sharma et al. [395] 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 [8, 14, 99, 100, 102, 108, 111, 130, 131, 152, 153, 155, 159, 184, 202, 218, 274, 327, 341, 353, 436, 447–449] in this category compared the performance of various ML models. Some of the authors experimented with individual ML models; for example, Kaur et al. [199] and Amorim et al. [36] 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. [50] and TummalaPalli et al. [446] experimented with ensemble techniques such as majority training ensemble and best training ensemble. Saidani et al. [378] 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. [395] used CNN, RNN (LSTM), and Autoencoders-based DL models. Hadj-Kacem and Bouassida [158] 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. [263] 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. [154] used eight DL models and Zhang and Dong [511] 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 \([8, 40, 50, 71, 100, 131, 148, 155, 158, 199, 274, 327]\), feature envy \([8, 40, 50, 100, 130, 131, 158, 199, 395]\), long method \([40, 44, 100, 130, 131, 148, 155, 158, 199]\), data class \([40, 130, 131, 148, 199, 327]\), and complex class \([155, 274, 327]\). 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 \([13, 44, 130, 155]\) and Random Forest \([40, 44, 130, 218, 304]\) perform the best. Other methods that have been reported better than other ML models in their respective studies are Support Vector Machine \([446]\), Boosting \([273]\), and Autoencoders \([395]\).

Traditional ML techniques are the prominent choice in this category because these techniques work well with fixed size, fixed column meaning vectors. Code quality metrics capture the features relevant to identify smells and they are 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 from AST to feed to DL models such as RNN.

4.4.2 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. Figure 10 provides an overview of techniques that detect code clones 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 \([307, 310, 481]\). Mostaen et al. \([310]\) 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. \([490]\) used existing code clone detection tools to generate their training set. Some authors (such as Bandara and Wijayarathna \([47]\) and Hammad et al. \([162]\)) relied on existing code-clone datasets. Zhang and Khoo
Sharma et al. [504] used NiCad to detect all clone groups from each version of the software. The study mapped the clones from consecutive version and used the mapping to predict clone consistency at both the clone-creating and clone-changing time. Bui et al. [65] 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 [47] identified features such as the number of characters and words, identifier count, identifier character count, and underscore count using ANTLR tool. Some studies [307, 309, 310] utilized line similarity and token similarity. Yang et al. [490] and Hammad et al. [162] computed TF-IDF along with other metrics such as position of clones in the file. Cesare et al. [72] 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 [504] obtained code attribute set (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 [402] 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 [513] used CFG and DFG (Data Flow Graph) for clone detection. Some of the studies [65, 125, 481] 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 [47, 307, 309, 402, 504] experimented with a number of ML approaches. For example, Mostaen et al. [307] used Bayes Network, Logistic Regression, and Decision Tree; Bandara and Wijayarathna [47] employed Naive Bayes, K Nearest Neighbors, AdaBoost. Similarly, Sheneamer and Kalita [402] 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 [309, 310], DNN [125, 513], and RNN with Reverse neural network [481] are also employed extensively. Bui et al. [66] and Bui et al. [65] combined neural networks for ML models training. Specifically, Bui et al. [66] 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. [65] constructed BiTBCNNs—a combination layer of sub-networks to encode similarities and differences among code structures in different languages. Hammad et al. [162] 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.

### 4.4.3 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 preprocessing and labeling:** Heo et al. [172] 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. Ribeiro et al. [368] used ensemble learning to learn from multiple static analyzers and show that ensemble learning improves the accuracy. Specifically, they took three static analyzers (Clang-analyzer, CppCheck, and Frama-C) and detected issues in Juliet dataset. Once the report is generated from all three tools, the authors combined the reports by converting them to a uniform format. Then, they tagged the samples. Similarly,
Alikhashashneh et al. [19] used the Understand tool to detect various metrics, and employed them on the Juliet test suite for C++. Reddivari and Raman [363] extracted a subset of data belonging to open source projects such as Ant, Tomcat, and Jedit to predict reliability and maintainability using machine learning techniques. Malhotra and Chug [290] also prepared a custom dataset using two proprietary software systems as their subjects to predict maintainability of a class.

**Feature extraction:** Heo et al. [172] 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). Ribeiro et al. [368] generated features only from the warnings (such as redundancy level and number of warnings in the same file). Some studies [19, 208, 290, 363] used source code metrics as features.

**ML model training:** Kim et al. [208] used Support Vector Machine to identify risky modules from a software system. Alikhashashneh et al. [19] 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 [363] predicted reliability and maintainability using the similar set of machine learning techniques. The study by Ribeiro et al. [368] claimed that ensemble methods such as AdaBoost works superior than standalone machine learning methods. Anomaly-detection techniques such as One-class Support Vector Machine have been used by Heo et al. [172]. They applied their method on taint analysis and buffer overflow detection to improve the recall of static analysis. Whereas, some other studies [19, 368] aimed to rank and classify static analysis warnings. Kim et al. [208] estimated risky modules in the subject system.

### 4.5 Code completion

Code auto-completion is a state-of-the-art integral feature of modern source-code editors and IDEs [63]. The latest generation of auto-completion methods uses NLP and advanced machine learning models, trained on publicly available software repositories, to suggest source-code completions, given the current context of the software-project under examination.

**Data collection:** The majority of the studies mined a large number of repositories to construct their own dataset. Specifically, Gopalakrishnan et al. [140] 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. [164], Han et al. [165] trained and tested their system by sampling 4,919 source code lines from open-source projects. Raychev et al. [362] used large codebases from GitHub to make predictions for JavaScript and Python code completion. Svyatkovskiy et al. [431] used 2,700 Python open-source software repositories for the evaluation of their novel approach, Pythia.

The rest of the approaches employed existing benchmarks and datasets. Rahman et al. [359] trained their proposed model using the data extracted from Aizu Online Judge (Aoj) system. Liu et al. [261]. Liu et al. [262] 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. [240] 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. Phan and Jannesari [344] used three corpus for their experiments—a large-scale corpus of English-German translation in NLP [275], the Conala corpus [497], which contains Python software documentation as 116,000 English sentences, and the MSR 2013 corpus [26]. Schuster et al. [387] used a public archive of GrtHub repositories for the evaluation of their novel approach, Pythia.

**Feature extraction:** Studies in this category extract source code information in variety of forms. Gopalakrishnan et al. [140] extracted relationships between topical concepts in the source code and the use of specific architectural developer tactics in that code. Phan and Jannesari [344] used machine translation to learn the mapping from prefixes to code tokens for code suggestion. They extracted the tokens from the documentation of the source
Sharma et al. [261], Liu et al. [262] 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. [240] used locally repeated terms in program source code to predict out-of-vocabulary (OoV) words that restrict the code completion. Chen and Wan [84] proposed a tree-to-sequence (Tree2Seq) model that captures the structure information of source code to generate comments for source code. Raychev et al. [362] used ASTs and performed prediction of a program element on a dynamically computed context. Svyatkovskiy et al. [431] 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 [434] used LSTM for code completion to facilitate programming education. Rahman et al. [359] also used LSTM. Wang et al. [467] used 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. [516] 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. [240] 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. [181] presented CodeGRU that uses GRU for capturing source codes contextual, syntactical, and structural dependencies. Yang et al. [492] 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. [386] 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. [140] developed a recommender system using prediction models including neural networks for latent topics. Han et al. [164], Han et al. [165] 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. [354] used **Bayesian Networks** for intelligent code completion. Raychev et al. [362] utilized a probabilistic model for code in any programming language with Decision Tree. Svyatkovskiy et al. [431] 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. [261], Liu et al. [262] applied Multi-Task Learning (MTL) for suggesting code elements. Lee et al. [232] developed MERGELOGGING, a DL-based merged network that uses code representations for automated logging decisions. Chen and Wan [84] applied Tree2Seq model with NMT techniques for code comment generation. Phan and Jannesari [344] 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.

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 [420]. 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.

**4.5.1 Code summarization**

Code summarization techniques attempt to provide a consolidated summary of the source code entity (typically
A variety of attempts has been made in this direction. The majority of the studies [9, 87, 168, 177, 183, 229, 230, 236, 244, 258, 404, 466, 475, 494, 496, 505, 518, 519] produces a summary for a small block (such as a method). This category also includes studies that summarize small code fragments [316], commit message generation [86, 189–191, 267, 476], and title generation for online posts from code [133]. Figure 11 provides an overview of the mechanism used by code summarization techniques.

Data collection and processing: The majority of the studies [9, 24, 85–87, 177, 230, 236, 258, 462, 466, 475, 519] in this category prepares pairs of code snippets and their corresponding natural language description. Specifically, Chen and Zhou [87] 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. [177] and Li et al. [236] prepared a large dataset from 9,714 GitHub projects. Similarly, Wang et al. [466] mined code snippets and corresponding javadoc comments for their experiment. Chen et al. [85] 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. [355] 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. [93] collected and refined more than 114 thousand pairs of methods and corresponding code annotations from 100 open-source Java projects. Iyer et al. [183] mined StackOverflow and extracted title and code snippet from posts that contain exactly one code snippet. Similarly, Gao et al. [133] 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 “NUMER”. Nazar et al. [316] used human annotators to summarize 127 code fragments retrieved from Eclipse and NetBeans official frequently asked questions. Yang et al. [493] built a dataset with over 300K pairs of method and comment to evaluate their approach. Chen et al. [86] used dataset provided by Hu et al. [177] and manually categorize comments into six intention categories for 20,000 code-comment pairs. Wang et al. [476] created a Python dataset that contains 128
thousand code-comment pairs. Lal and Pahwa [222] crawled over 6700 Java projects from Github to extract their methods and the corresponding Javadoc comments to create their dataset.

Jiang [189] used 18 popular Java projects from GitHub to prepare a dataset with approximately 50 thousand commits to generate commit messages automatically. Liu et al. [265] processed 56 popular open-source projects and selected approximately 160K commits after filtering out the irrelevant commits. Liu et al. [268] 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. [465] 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. [230] used ast along with code and their corresponding summaries belonging to more than 2 million Java methods. Likewise, Shido et al. [404] and Zhang et al. [505] also generated asts of the collected code samples. Liu et al. [258] utilized call dependencies along with source code and corresponding comments from more than a thousand GitHub repositories. LeClair et al. [229] employed AST along with adjacency matrix of AST edges.

Some of the studies used existing datasets such as StaQC [495] and the dataset created by Jiang et al. [190]. Specifically, Jiang and McMillan [191], Liu et al. [267] utilized a dataset of commits provided by Jiang et al. [190] that contains two million commits from one thousand popular Java projects. Yao et al. [494] and Ye et al. [496] used StaQC dataset [495]; it contains more than 119 thousand pairs of question title and code snippet related to SQL mined from StackOverflow. Xie et al. [484] utilized two existing datasets—one each for Java [231] and Python [51]. Bansal et al. [49] evaluated their code summarization technique using a Java dataset of 2.1M Java methods from 28K projects created by LeClair and McMillan [231]. Li et al. [244] also used the Java dataset of 2.1M methods LeClair and McMillan [231] to predict the inconsistent names from the implementation of the methods. Simiarly, Haque et al. [167, 168], LeClair et al. [228] relied on the Java dataset by LeClair and McMillan [231] for summarizing methods. The first dataset [177] contains over 87 thousand Java methods. The other datasets contained 2.1M Java methods [231] and 500 thousand Java methods respectively.

Efforts in the direction of automatic code folding also utilize techniques similar to code summarization. Viuginov and Filchenkov [460] collected projects developed using IntelliJ platform. They identified 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. [190] tokenized their code and text. Jiang and McMillan [191] extracted commit messages starting from “verb + object” and computed TFIDF for each word. Haque et al. [167] extracted top-40 most-common action words from the dataset of 2.1m Java methods provided by LeClair and McMillan [231]. Psarras et al. [355] 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. [258] 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 [460] 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. [177] and Li et al. [236] transformed AST into Structure-Based Traversal (SBT). Yang et al. [493] 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. [518] extracted AST and prepared tokenized code sequences and tokenized AST to feed to semantic and structural encoders respectively. Lal and Pahwa [222], Zhou et al. [519] tokenized source code and parse them into AST. Lin et al. [251] 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. [265] worked with AST diff between commits as input to generate a commit summary. Lu et al. [272] 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. [180] 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. [230] used embeddings layer for code, text, as well as for AST. Similarly, Choi et al. [93] transformed each of the tokenized source code into a vector of fixed length through an embedding layer. Wang et al. [466] extracted the functional keyword from the code and perform positional encoding. Yao et al. [494] 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. [496] 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. [85] 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. [229] used a single embedding layer for both the source code and AST node inputs to exploit a large overlap in vocabulary. Wang et al. [475] 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 word2vec. Studies also explored other mechanisms to encode features. For example, Liu et al. [267] 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. [505] parsed code snippets into ASTs and calculated their similarity using ASTs. Allamanis et al. [24] and Ahmad et al. [9] employed attention-based mechanism to encode tokens. Li et al. [244] 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. [239] 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. For instance, Gao et al. [133] 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. [190] trained an NTM algorithm to “translate” from diffs to commit messages. Similarly, Chen et al. [85], Haque et al. [168], Hu et al. [177], Jiang [189], Li et al. [239], Liu et al. [268], Lu et al. [272], Takahashi et al. [433] employed LSTM-based Encoder-Decoder model to generate summaries. Zhang et al. [505] 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. Iyer et al. [183] 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. [93] 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. 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. [230] 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. Yang et al. [493] developed a multi-modal transformer-based code summarization approach for smart contracts. Xie et al. [484] designed a novel multi-task learning (MTL) approach for code summarization through mining the relationship between method-code summaries and method names. Bansal et al. [49] introduced a project-level encoder DL model for code summarization. Li et al. [244] 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. [167] 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. [465] 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. Lal and Pahwa [222] presented ContextCC that uses a Seq2Seq Neural Network model with an attention mechanism to generate comments for Java methods.

Extended encoder-decoder models: Many studies extended the traditional Encoder-Decoder mechanism in a variety of ways. Liu et al. [258] proposed 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. Similarly, Li et al. [236] presented Hybrid-DeepCon model containing two encoders for code and ast along with a decoder to generate sequences of natural language annotations. Shido et al. [404] 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. Wang et al. [466] 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. [229] 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. Ye et al. [496] 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. Zhou et al. [518] 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. Zhou et al. [519] 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. [251] used a transformer to generate high-quality code summaries. The learned syntax encoding is combined with code encoding, and fed into the transformer. Liu et al. [265] 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. [494] 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. [462] and Wang et al. [475] 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. [476] 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. [180] 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 [191] used Naive Bayes to classify the diff files into the verb groups. For automated code folding, Viuginov and Filchenkov [460] used Random Forest and Decision Tree to classify whether a code block needs to be folded. Similarly, Nazar et al. [316] used Support Vector Machine and Naive Bayes classifiers to generate summaries from the extracted features. Chen et al. [86] compared six ml techniques to demonstrate that comment category prediction can boost code summarization to reach better results. Etemadi and Monperrus [121] compared NNGen, SimpleNNGen, and EXC-NNGen to explore the origin of nearest diffs selected by the neural network.

4.5.2 Program classification

Studies targeting this category classify software artifacts based on programming language [454], application domain [454], and type of commits (such as buggy and adaptive) [185, 302]. We summarize these efforts below from dataset preparation, feature extraction, and ml model training perspective.

Dataset and benchmarks: Ma et al. [278] 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. [406] 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. [185] and Meqdadi et al. [302] analyzed the GitHub commit history. Ugurel et al. [454] relied on C and C++ projects from libblio and the Sourceforge archives. Levin and Yehudai [234] 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. [293] and Mariano et al. [294] classify commits by maintenance activities; they identify a large number of open-source GitHub repositories. Along the similar lines, Meng et al. [301] classified commits messages into categories such as bug fix and feature addition and Li et al. [237] 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 [372] 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. [136] 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.

Feature extraction: Features in this category of studies belong to either source code features category or repository features. A subset of studies [278, 406, 454] relies on features extracted from source code token including language specific keywords and other syntactic information. Other studies [185, 302] collect repository metrics (such as number of changed statements, methods, hunks, and files) to classify commits. Ben-Nun et al. [53] leveraged both the underlying data- and control-flow of a program to learn code semantics performance prediction. Gharbi et al. [136] used TF-IDF to weight the tokens extracted from change messages. Ghadhab et al. [134] 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. [293] and Mariano et al. [294] extracted 71 features majorly belonging to source code changes and keyword occurrences categories. Meng et al. [301] and Li et al. [237] 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 [372] 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).

**ML model training:** A variety of ml approaches have been applied. Specifically, Ma et al. [278] used Support Vector Machine, Decision Tree, and Bayes Network for artifact classification. Meqdadi et al. [302] employed Naive Bayes, Ripper, as well as Decision Tree and Ugurel et al. [454] used Support Vector Machine to classify specific commits. Ben-Nun et al. [53] proposed an approach based on an RNN architecture and fixed inst2vec embeddings for code analysis tasks. Levin and Yehudai [234], Mariano et al. [293, 294] used Decision Tree and Random Forest for commits classification into maintenance activities. Gharbi et al. [136] applied Logistic Regression model to determine the commit classes for each new commit message. Ghadhab et al. [134] trained a DNN classifier to fine-tune the BERT model on the task of commit classification. Meng et al. [301] used a CNN-based model to classify code commits. Sabetta and Bezzi [372] trained Random Forest, Naive Bayes, and Support Vector Machine to identify security-relevant commits.

4.5.3 Change analysis
Researchers have explored applications of ml techniques to identify or predict relevant code changes [438, 441].

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. [438] performed their study on two industrial projects. Tufano et al. [441] extracted 236K pairs of code snippets identified before and after the implementation of the changes provided in the pull requests. Kumar et al. [215] used eBay web-services as their subject systems. Uchôa et al. [453] 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 [288] considered three open-source projects to investigate the relationship between code quality metrics and change proneness.

**Feature extraction:** Tollin et al. [438] extracted features related to the code quality from the issues of two industrial projects. Tufano et al. [441] used features from pull requests to investigate the ability of a nmt modes. Abbas et al. [2] and Malhotra and Khanna [288] computed well-known C&K metrics to investigate the relationship between change proneness and object-oriented metrics. Similarly, Kumar et al. [215] computed 21 code quality metrics to predict change-prone web-services. Uchôa et al. [453] 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[392].

**ML model training:** Tollin et al. [438] employed Decision Tree, Random Forest, and Naive Bayes ml algorithms for their prediction task. Tufano et al. [441] used Encoder-Decoder architecture of a typical nmt model to learn the changes introduced in pull requests. Malhotra and Khanna [288] 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. [215] used Support Vector Machine to the predict change proneness in web-services. Uchôa et al. [453] 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.

4.5.4 Entity identification/recommendation
This category represents studies that recommend source code entities (such as method and class names)
A Survey on Machine Learning Techniques for Source Code Analysis

• 0:31

or identify entities such as design patterns [132] in code using ml [17, 80, 116, 452, 502]. Specifically, Linstead et al. [257] 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. [179] found commenting position in code using ml techniques. Uchiyama et al. [452] identified design patterns and Abuhamad et al. [5] recommended code authorship. Similar approaches include recommending method name [21, 187, 487], method signature [291], class name [21], and type inference [170]. 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. [257] used two large, open source Java projects, Eclipse and ArgoUML in their experiments to apply unsupervised statistical topic models. Similarly, Hellendoorn et al. [170] 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. [21] mined 20 software projects from GitHub to predict method and class names. Jiang et al. [187] used the Code2Seq dataset containing 3.8 million methods as their experimental data.

A subset of studies focused on identifying design patterns using ml techniques. Uchiyama et al. [452] 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. [502] manually identified 2,794 design patterns instances from ten open-source repositories. Dwivedi et al. [116] analyzed JHotDraw and identified 59 instances of abstract factory and 160 instances of adapter pattern for their experiment.

Feature extraction: Several studies generated embeddings from their feature set. Specifically, Huang et al. [179] used embeddings generated from Word2vec capturing code semantics. Similarly, Jiang et al. [187] employed Code2vec embeddings and Allamanis et al. [21] used embeddings that contain semantic information about sub-tokens of a method name to identify similar embeddings utilized in similar contexts. Zhang et al. [507] utilized knowledge graph embeddings to extract interrelations of code for bug localization. Abuhamad et al. [5] extracted code authorship attributes from samples of code. Malik et al. [291] used function names, formal parameters, and corresponding comments as features.

In addition, Alhusain et al. [17], Chaturvedi et al. [80], Dwivedi et al. [116], Uchiyama et al. [452] 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. [179] and Hellendoorn et al. [170] used bidirectional RNN models. Similarly, Abuhamad et al. [5] and Malik et al. [291] also employed RNN models to identify code authorship and function signatures respectively. Zhang et al. [507] created a bug-localization tool, KGBugLocator utilizing knowledge graph embeddings and bi-directional attention models. Xu et al. [487] employed the gru-based Encoder-Decoder model for method name prediction. Uchiyama et al. [452] used a hierarchical neural network as their classifier. Allamanis et al. [21] utilized neural language models for detecting prediction and class names.

Other studies used traditional ml techniques. Specifically, Chaturvedi et al. [80] compared four ml techniques (Linear Regression, Polynomial Regression, support vector regression, and neural network). Dwivedi et al. [116] used Decision Tree and Zanoni et al. [502] trained Naive Bayes, Decision Tree, Random Forest, and Support Vector Machine to detect design patterns using ml.
4.6 Code review

Code Review is the process of systematically checking 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. Specifically, Lal and Pahwa [221] labeled check-in code samples as clean and buggy. On code samples, they carried out extensive pre-processing such as normalization and label encoding before using TF-IDF to convert the samples into vectors. They used a Naive Bayes model to classify samples into buggy or clean. Similarly, Axelsson et al. [42] developed a tool referred to as ‘Code Distance Visualiser’ to help reviewers find problematic sections of code. The authors experimented on two subject systems—one open-source and another proprietary. The authors provide an interactive, supervised self-learning static analysis tool based on Normalised Compression Distance (NCD) metric that relies on K Nearest Neighbors.

4.7 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 [374, 408, 461]. 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. [408] utilized commented code as input. Wan et al. [461] used source code in the the form of tokens, ast, and crg. Sachdev et al. [374] employed a simple tokenizer to extract all tokens from source code by removing non–alphanumeric tokens. Ling et al. [256] mined software projects from GitHub for the training of their approach.

Feature extraction: Code search studies typically use embeddings representing the input code. Shuai et al. [408] 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. [461] employed a multi-modal code representation, where they learnt the representation of each modality via LSTM, Tree-LSTM and GGN, respectively. Sachdev et al. [374] identified words from source code and transformed the extracted tokens into a natural language documents. Similarly, Ling et al. [256] 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.

ML model training: Shuai et al. [408] 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. [461] 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. [374] utilized word and documentation embeddings and performed code search using the learned embeddings. Similarly, Ling et al. [256] 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.

Once an ML model is trained, code search can be initiated using a query and a code snippet. Shuai et al. [408] used the given query and code sample to measure the semantic similarity using cosine similarity. Wan et al. [461] ranked all the code snippets by their similarities with the input query. Similarly, Sachdev et al. [374] were able to answer almost 43% of the collected StackOverflow questions directly from code.
4.8 Refactoring

Refactoring transformations are intended to improve code quality (specifically maintainability), while preserving the program behavior (functional requirements) from users’ perspective [427]. 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. [369] and Amal et al. [35] 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. [210] 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. [322] analyzed 43 open-source repositories with 13.5 thousand commits to prepare their dataset. Similarly, Aniche et al. [38] created a dataset comprising over two million refactorings from more than 11 thousand open-source repositories. Sagar et al. [375] identified 5004 commits randomly selected from all the commits obtained from 800 open-source repositories where RefactoringMiner [439] identified at least one refactoring. Along the similar lines, Li et al. [244] used RefactoringMiner and RefDiff tools to identify refactoring operations in the selected commits. Krasniqi and Cleland-Huang [212], Xu et al. [486] used manual analysis and tagging for identifying refactoring operations. Finally, Kurbatova et al. [220] 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 [41, 216, 217], 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. [210] computed 25 different code quality metrics along with 25 other code quality metrics. Similarly, Kumar et al. [216] 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 [41, 217, 409, 473] calculated a large number of metrics. Specifically, Kumar and Sureka [217] computed 102 metrics and then applied PCA to reduce the number of features to 31, while Aribandi et al. [41] used 125 metrics. Sidhu et al. [409] used metrics capturing design characteristics of a model including inheritance, coupling and modularity, and size. Wang and Godfrey [473] computed a wide range of metrics related to clones such as number of clone fragments in a class, clone type (type1, type2, or type3), and lines of code in the cloned method.

Some other studies did not limit themselves to only code quality metrics. Particularly, Yue et al. [501] collected 34 features belonging to code, evolution history, diff between commits, and co-change. Similarly, Aniche et al. [38] extracted code quality metrics, process metrics, and code ownership metrics.

In addition, Nyamawe et al. [322]. Nyamawe et al. [323] carried out standard NLP preprocessing and generated TF-IDF embeddings for each sample. Along the similar lines, Kurbatova et al. [220] used code2vec to generate embeddings for each method. Sagar et al. [375] extracted keywords from commit messages and used GloVe to obtain the corresponding embedding. Krasniqi and Cleland-Huang [212] tagged each commit message with their parts-of-speech and prepared a language model dependency tree to detect refactoring operations from commit messages.

ML model training: Majority of the studies in this category utilized traditional ML techniques. Rodriguez et al. [369] proposed a method to identify web-service groups for refactoring using K-means, COBWEB, and expectation
maximization. Kosker et al. [210] trained a Naive Bayes-based classifier to identify classes that need refactoring. Kumar and Sureka [217] 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. [322] 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. [501] 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. [216] 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. [35] proposed the use of ANN to generate a sequence of refactoring. Aribandi et al. [41] 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. [220] proposed an approach to recommend move method refactoring based on a path-based presentation of code using Support Vector Machine. Similarly, Aniche et al. [38] used Linear Regression, Naive Bayes, Support Vector Machine, Decision Tree, Random Forest, and Neural Network to predict applicable refactoring operations. Sidhu et al. [409], Wang and Godfrey [473], Xu et al. [486] used mnn, gradient boosting, and Decision Tree respectively to identify refactoring candidate. Nyamawe et al. [323], Sagar et al. [375] employed various classifiers such as Support Vector Machine, Linear Regression, and Random Forest to predict commits with refactoring operations.

4.9 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. Figure 12 presents an overview of a typical process to detect vulnerabilities with the help of 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 [6, 46, 52, 56, 89, 113, 114, 203, 205, 207, 224, 242, 252, 296, 305, 314, 336, 342, 358, 359, 373, 407, 418, 424, 458, 477, 489, 499, 500, 512] used available labeled datasets for PHP, Java, C, C++, and Android applications to train vulnerability detection models. In other cases, Russell et al. [370] 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 [18, 34, 91, 96, 101, 118, 128, 151, 175, 193, 214, 252, 253, 280, 300, 306, 317, 335, 343, 346, 383, 391, 491, 514] created their own datasets. Ali Alatwi et al. [18], Cui et al. [101], Ma et al. [280], and Gupta et al. [151] created datasets to train vulnerability detectors for Android applications. In particular, Ma et al. [280] decompiled and generated CFGs of approximately 10 thousand, both benign and vulnerable, Android applications from AndroZoo and Android Malware datasets; Ali Alatwi et al. [18] collected 5,063 Android applications where 1,000 of them were marked as benign and the remaining as malware; Cui et al. [101] 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. [151] 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. [300] collected 35 open-source PHP projects and intentionally injected 76 vulnerabilities in their dataset. Shar et al. [391] 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. [317] 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 [343, 358, 491] 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 [346] manually label their dataset. Hou et al. [175], Moskovitch et al. [306] and Santos et al. [383] created their datasets by collecting web-page samples from StopBadWare and VxHeavens. Lin et al. [252] constructed a dataset and manually labeled 1,471 vulnerable functions and 1,320 vulnerable files from nine open-source applications, named Asterisk, FFmpeg, HTTPD, LibPNG, LibTIFF, OpenSSL, Pidgin, VLC Player, and Xen. Lin et al. [253] 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 [6, 34, 96, 101, 113, 128, 151, 207, 224, 298, 300, 346, 358, 365, 424] 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:* Bilgin et al. [56], Du et al. [114], Kim et al. [205], Kronjee et al. [214], Ma et al. [279], Medeiros et al. [298], Wang et al. [477] used CFGs, ASTs, or data flow analysis as features. More specifically, Ma et al. [280] 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. [205] extracted ASTs and GFCs which they tokenized and fed into ML models, while Bilgin et al. [56] extracted ASTs and translated their representation of source code into a one-dimensional numerical array to fed them to a model. Kronjee et al. [214] used data-flow analysis to extract features, while Spreitzenbarth et al. [418] used static, dynamic analysis, and information collected from ltrace.
to collect features and train a linear vulnerability detection model. Lin et al. [253] created ASTs and from there they extracted code semantics as features.

Repository and file metrics: Perl et al. [343] 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 [118, 342] 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 [91] 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. [175] 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 [46, 89, 296, 306, 314, 335, 336, 370, 371, 383, 391, 491, 499, 512, 514] 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 [242] used statistical techniques to reduce the feature space to reduce the number of code tokens.

Other features: Ali Alatwi et al. [18], Ndichu et al. [317] and Milosevic et al. [305] extracted permission-related features. In other cases, authors [489] combined software metrics and N-grams as features to train models and others [467] created text-based images to extract features. Likewise, Sultana [423] 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. [479] extracted system calls and function call-related information to use as features, while Vishnu and Jevitha [459] extracted URL-based features like number of chars, duplicated characters, special characters, script tags, cookies, and re-directions. Padmanabhu and Tan [329] 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 [6, 18, 89, 113, 114, 126, 224, 298, 306, 314, 317, 329, 335, 342, 343, 365, 370, 391, 423, 424, 459, 477, 479, 499, 500] 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. [18], Perl et al. [343], Russell et al. [370] 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. [317] used Support Vector Machine to train their model with linear kernel. Pereira et al. [342] 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. [391] 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. [499] used text-based features to train Naive Bayes, Support Vector Machine, and Random Forest models. Du et al. [113] 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 [34, 91, 96, 101, 128, 151, 175, 214, 300, 305, 346, 358, 383] used up to 32 different ML algorithms to train models and compared their performance. Specifically, Medeiros et al. [300] 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. [305] and Rahman et al. [358] 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. [128] showed that K Nearest Neighbors offers the best performance for their dataset after experimenting with DNN, K Nearest Neighbors, Support Vector Machine, Linear Regression, and Decision Tree.
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. [346] 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. [214], Cui et al. [101], and Gupta et al. [151] compared different ML algorithms and found Decision Tree and Random Forest as the best performing algorithms.

DL techniques: A large number of studies [46, 52, 205, 207, 252, 256, 296, 373, 407, 491] used DL methods such as CNN, RNN, and ANN to train models. In more details, Yang et al. [491] utilized the BP-ANN algorithm to train vulnerability detectors. For the project Achilles, Saccente et al. [373] 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. [205] 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. [514] 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. [252] 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. [207] 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. [407] created textual images and fed them to Deep Belief Networks to classify malware.

5 Datasets and tools
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 primary study and noted the used 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 paper (Jul 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 2 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 Jul 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, we mark the entry with "-". We also provide a short description of the resource.
Table 2. A list of tools useful for analyzing source code and applying machine learning techniques

| Category                      | Name                                    | #Citation | Introd.  | Updated | Description                                                                 |
|-------------------------------|-----------------------------------------|------------|----------|---------|-----------------------------------------------------------------------------|
| Code Representation           | ncc [53]                                | 3          | Dec 2018 | Aug 2021| Learns representations of code semantics                                     |
|                              | Code2vec [31]                           | 271        | Jan 2019 | Feb 2022| Generates distributed representation of code                                |
|                              | Code2seq [29]                           | 418        | May 2019 | Jul 2022| Generates sequences from structured representation of code                 |
|                              | Vector representation for coding style [211] | 1        | Sep 2020 | Jul 2022| Implements vector representation of individual coding style               |
|                              | CC2Vec [173]                            | 23         | Oct 2020 | –       | Implements distributed representation of code                              |
|                              | AutoenCODE [442]                        | 1          | –        | –       | Encodes source code fragments into vector representations                 |
|                              | Graph-based code modeling [23]          | 544        | May 2018 | May 2021| Generates code modeling with graphs                                        |
|                              | Vocabulary learning on code [103]        | 34         | Jan 2019 | –       | Generates an augmented AST from Java source code                           |
|                              | User2code2vec [43]                      | 14         | Mar 2019 | May 2019| Generates embeddings for developers based on distributed representation of code |
| Code Search                  | Deep Code Search [149]                  | 160        | May 2018 | May 2022| Searches code by using code embeddings                                    |
|                              | Obfuscated-code2vec [97]                | 14         | Oct 2022 | –       | Embeds Java Classes with Code2vec                                          |
|                              | DEEPtyper [170]                         | 54         | Oct 2018 | Feb 2020| Annotates types for JavaScript and TypeScript                               |
|                              | CallNN [258]                            | 6          | Oct 2019 | –       | Implements a code summarization approach by using call dependencies        |
| Program Comprehension        | NeuralCodeSum [9]                       | 147        | May 2020 | Oct 2021| Implements a code summarization method by using transformers               |
|                              | Summarization_tf [404]                  | 9          | Jul 2019 | –       | Summarizes code with Extended Tree-LSTM                                    |
|                              | CoaCor [494]                           | 16         | Jul 2019 | May 2020| Explores the role of rich annotation for code retrieval                    |
|                              | DeepCom [236]                           | 12         | Nov 2020 | May 2021| Generates code comments                                                    |
|                              | Rencos [505]                            | 79         | Oct 2020 | –       | Generates code summary by using both neural and retrieval-based techniques |
| Tool                  | Code Quality Assessment                                                                 | Function                                                                                             | Date       |
|----------------------|------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|------------|
| CODES [337]          | 107 Jul 2012 Jul 2016                                                                  | Extracts method description from StackOverflow discussions                                           |            |
| CFS                  | – – –                                                                                     | Summarizes code fragments using SVM and NB                                                             |            |
| TASSAL               | – – –                                                                                     | Summarizes code using autofolding                                                                      |            |
| ChangeScribe [98]    | 158 Dec 2014 Dec 2015                                                                  | Generates commit messages                                                                             |            |
| CodelInsight [360]   | 59 Nov 2015 May 2019                                                                   | Recommends insightful comments for source code                                                        |            |
| CodeNN [183]         | 479 Aug 2016 May 2017                                                                  | Summarizes code using neural attention model                                                           |            |
| Code2Que [133]       | 3 Jul 2020 Aug 2021                                                                    | Suggests improvements in question titles from mined code in StackOverflow                            |            |
| BI-TBCNN [65]        | 30 Mar 2019 May 2019                                                                  | Implements a bi-TBCNN model to classify algorithms                                                    |            |
| DeepSim [513]        | 150 Oct 2018 –                                                                         | Implements a DL approach to measure code functional similarity                                        |            |
| FCDetector [125]     | 22 Jul 2020 –                                                                           | Proposes a fine-grained granularity of source code for functionality identification                |            |
| SONARQUBE             | – – –                                                                                     | Analyzes code quality                                                                                 |            |
| svf [422]            | 216 Mar 2016 Jul 2022                                                                  | Enables inter-procedural dependency analysis for LLVM-based languages                              |            |
| Designite [398]      | 18 Mar 2016 Jul 2022                                                                   | Detects code smells and computes quality metrics in Java and C# code                                 |            |
| CloneCognition [308] | 4 Nov 2018 May 2019                                                                     | Proposes a ML framework to validate code clones                                                      |            |
| SMAD [50]            | 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                                      |            |
| Program Synthesis | ML Clone Validation Framework [309] | py-ccflex [324] | Deep learning smells [395] | CREC [501] | ML for software refactoring [38] |
|-------------------|-------------------------------------|----------------|----------------------------|-------------|--------------------------------|
|                   | ML Clone Validation Framework [309] | py-ccflex [324] | Deep learning smells [395] | CREC [501] | ML for software refactoring [38] |
|                   | 11 | Aug 2019 | Aug 2019 | Implements a ml framework for automatic code clone validation | Mimics code metrics by using ml |
|                   | 6 | Mar 2017 | Oct 2020 | Implements dl (cnn, rnn, and Autoencoder-based models) to identify four smells | Recommends clones for refactoring |
|                   | 18 | Jul 2021 | Nov 2020 | Recommends refactoring by using ml | Recommends clones for refactoring |
|                   | 26 | Nov 2018 | – | – | – |
|                   | 31 | Sep 2020 | – | – | – |
| CoCoNuT [276]     | 86 | Jul 2020 | Sep 2021 | Repairs Java programs | Repairs Java programs |
| DeepFix [157]     | 359 | Feb 2017 | Dec 2017 | Fixes common C errors | Fixes common C errors |
| AppFlow [176]     | 47 | Oct 2018 | – | – | – |
| DeepFuzz [266]    | 42 | Jul 2019 | Mar 2020 | Grammar fuzzer that generates C programs | Grammar fuzzer that generates C programs |
| Agilika [455]     | 7 | Aug 2020 | Mar 2022 | Generates tests from execution traces | Generates tests from execution traces |
| BugDetection [245]| 66 | Oct 2019 | May 2021 | Trains models for defect prediction | Trains models for defect prediction |
| DTLDP [82]        | 21 | Aug 2019 | – | – | – |
| DeepBugs [350]    | 210 | Nov 2018 | May 2021 | Implements a framework for learning name-based bug detectors | Implements a framework for learning name-based bug detectors |
| Randoop           | – | – | Jul 2022 | Generates tests automatic for Java code | Generates tests automatic for Java code |
| TestDescriber     | – | – | – | Implements test case summary generator and evaluator | Implements test case summary generator and evaluator |
| WAP [299]         | 3 | Oct 2013 | Nov 2015 | Detects and corrects input validation vulnerabilities | Detects and corrects input validation vulnerabilities |
| swan[346]         | 6 | Oct 2019 | May 2022 | Identifies vulnerabilities | Identifies vulnerabilities |
| vccfinder [343]   | 174 | Oct 2015 | May 2017 | Finds potentially dangerous code in repositories | Finds potentially dangerous code in repositories |
| BERT              | 43462 | Oct 2018 | Mar 2020 | NLP pre-trained models | NLP pre-trained models |
| bc3 Annotation Framework | – | – | – | Annotates emails/conversations easily | Annotates emails/conversations easily |
| JGibLDA           | – | – | – | Implements Latent Dirichlet Allocation | Implements Latent Dirichlet Allocation |
| Stanford NLP Parser | – | – | – | A statistical NLP parser | A statistical NLP parser |
| srcML             | – | – | May 2022 | Generates xml representation of sourcecode | Generates xml representation of sourcecode |
CallGraph 8 Oct 2017 Oct 2018 Generates static and dynamic call graphs for Java code
ML for programming – – – Offers various tools such as JS-Nice, Nice2Predict, and DEBIN

The list of datasets found in this exploration are presented in Table 3. Similar to Tools’ table, Table 3 lists each resource with its category, name and link to access the resource, number of citations (as of Jul 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 3. A list of datasets useful for analyzing source code and applying machine learning techniques

| Category               | Name                          | #Citation | Introd.   | Updated   | Description                                                                 |
|------------------------|-------------------------------|-----------|-----------|-----------|-----------------------------------------------------------------------------|
| Code Representation    | Code2seq [31]                 | 271       | Jan 2019  | Feb 2022  | Sequences generated from structured representation of code                 |
|                        | GHTorrent [144]               | 645       | 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 |
|                        | TL-CodeSum [178]              | 150       | Feb 2019  | Sep 2020  | Dataset for code summarization                                              |
| Program Synthesis      | CoNLaLa corpus [497]          | 130       | Dec 2018  | Oct 2021  | Python snippets and corresponding natural language description             |
|                        | IntroClass [227]              | 144       | Jul 2015  | Feb 2016  | Program repair dataset of C programs                                         |
| Program Comprehension  | Program comprehension dataset [419] | 61       | May 2018  | Aug 2021  | Contains code for a program comprehension user survey                       |
|                        | CommitGen [190]               | –         | –         | –         | Commit messages and the diffs from 1,006 Java projects                      |
|                        | StaQC [495]                   | –         | –         | Nov 2019  | 148K Python and 120K sql question-code pairs from Stack Overflow            |
|                        | src-d datasets                | –         | –         | –         | Various labeled datasets (commit messages, duplicates, DockerHub, and Nuget) |
| Quality Assessment     | BigCloneBench [428]           | 187       | Dec 2014  | Mar 2021  | Known clones in the IJaDataset source repository                           |
|                        | Multi-label smells [150]      | 28        | May 2020  | –         | A dataset of 445 instances of two code smells and 82 metrics               |
|                        | Deep learning smells [395]    | 15        | Jul 2021  | Nov 2020  | A dataset of four smells in tokenized form from 1,072 C# and 100 Java repositories |
| Dataset | ML for software refactoring [38] | QScored [397] | Nov 2019 | Aug 2021 | Dataset for applying ML to recommend refactoring Code smell and metrics dataset for more than 86 thousand open-source repositories |
|---------|---------------------------------|--------------|----------|----------|------------------------------------------------------------------|
| Testing | Defects4J [195] | PROMISE [385] | 858 | 41 | Java reproducible bugs Various datasets including defect prediction and cost estimation |
| | BugDetection [245] | 66 | Oct 2019 | May 2021 | A bug prediction dataset containing 4.973M methods belonging to 92 different Java project versions |
| | DAMT [313] | 15 | Aug 2019 | Dec 2019 | Metamorphic testing dataset |
| | DTLDP [82] | 21 | Oct 2020 | – | Dataset for deep transfer learning for defect prediction |
| | DEEPBUGS [350] | 210 | Oct 2018 | Apr 2021 | A JavaScript code corpus with 150K code snippets |
| Vulnerability Analysis | wpscan | – | – | – | a PHP dataset for WordPress plugin vulnerabilities |
| | Genome [517] | 1139 | Jul 2012 | Dec 2015 | 1,200 malware samples covering the majority of existing malware families |
| | Juliet [58] | – | – | – | 81K synthetic C/C++ and Java programs with known flaws |
| | AndroZoo [28] | – | – | – | 15.7M APKs from Google’s Play Store |
| | trl [254] | 108 | Apr 2018 | Jan 2019 | Vulnerabilities in six C programs |
| | Draper vdisc [371] | 247 | Jul 2018 | Nov 2018 | 1.27 million functions mined from c and c++ applications |
| | SAMATE [57] | – | – | – | A set of known security flaws from nist for c, c++, and Java programs |
| | jsVulner [128] | – | – | – | JavaScript Vulnerability Analysis dataset |
| | swan [346] | 6 | Jul 2019 | Jul 2022 | A Vulnerability Analysis collection of 12 Java applications |
| | Project-KB [347] | 49 | Aug 2019 | – | A Manually-Curated dataset of fixes to vulnerabilities of open-source software |
| General | GitHub Java Corpus [25] | 333 | – | – | A large collection of Java repositories |
| | 150k Python dataset [362] | – | – | – | Contains parsed AST for 150K Python files |
6 Challenges and perceived deficiencies

The aim of this section is to focus on the perceived deficiencies, challenges, and opportunities in applying ML techniques in the context of source code analysis observed from the primary studies. We document challenges or deficiencies mentioned in the considered primary studies while studying and summarizing them. After the summarization phase was over, we consolidated all the documented notes and synthesized 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, lack of high-quality datasets (i.e., clean and reliably annotated) are one of the biggest challenges in the domain [33, 50, 82, 135, 139, 142, 193, 219, 263, 380, 401, 416, 429, 437, 440, 451, 463]. 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.

- **Reproducibility and replicability**: Reproducibility and replicability of any ML implementation can be compromised by factors discussed below.
  - Insufficient information: Aspects such as ML model, their hyper-parameters, data size and ratio (of benign and faulty samples, for instance) are needed 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. [111] carried out a detailed replication study and reported that the replicated results were lower by up to 90% compared to what was reported in the original study.
  - Handling of data imbalance: It is very common to have imbalanced datasets in software engineering applications. Authors use techniques such as under-sampling and over-sampling to overcome the challenge for training. However, test datasets must retain the original sample ratio as found in the real world [111]; 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, 102, 130, 131, 150, 327, 436] 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.

- **Maturity in ML development**: Development of ML systems are inherently different from traditional software development [463]. Phases of ML development are very exploratory in nature and highly domain and problem dependent [463]. Identifying the most appropriate ML model, their appropriate parameters, and configuration is largely driven by trial and error manner [44, 401, 463]. 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 ML software development. It includes effective tools for testing ML programs, ensuring that the dataset are pre-processed adequately, debugging, and effective data management [137, 339, 463]. 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 [137].

- **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 [64, 508], 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 [64, 137].

- **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 [182, 339, 401, 440, 463]. Specifically, software engineering researchers have notified that identifying and extracting relevant features beyond code quality metrics is non-trivial. For example, Ivers et al. [182] 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. [395] 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.

- **Skill gap:** Wan et al. [463] 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 [160] also reports a serious lack of ML expertise in academic software engineering efforts. Other authors [339] have emphasized the importance of domain knowledge to design effective ML models.

- **Hardware resources:** Given the need of large training dataset and many hidden layers, often ML training requires high-end processing units (such as GPUs and memory) [137, 463]. A user-survey study [463] highlights the need to special hardware for ML training. Such requirements poses a challenge to researchers constrained with limited hardware resources.

7 Discussion

This section provides a discussion on the top venues for articles belonging to each selected category in our scope as well as on potential mitigations for the challenges we identified in the previous section.

7.1 Venue and article categories

The goal of the exploration is to understand the top venues for each considered category. We identified and manually curated the software engineering venue for each primary study discussed in our literature review. Figure 13 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 and ASE are among the top venues, appearing in four and three categories respectively. TSE, JSS, and IEEE Access are the top journals for the considered categories. Machine learning conferences such as ICLR also appear as the one of top venues for program synthesis category).

The categories of program comprehension and program synthesis exhibit the highest concentration of articles to a relatively small list of top venues where 45% and 35% of articles, respectively, come from the top venues. On the other hand, researchers publish articles related to testing and vulnerability in a rather large number of venues.

7.2 Model selection

Selecting a ML model for a given task depends on many factors such as nature of the problem, properties of training and input samples, and 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 embedding representing code in the vector form. The semantics of the produced embeddings depends 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.
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. \texttt{RNN} are considered a popular choice to realize Encoder-Decoder models due to its capability to remember long sequences.

7.3 Mitigating the challenges

7.3.1 Availability of standard datasets: 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 \textsc{msr} data showcase track. Dataset search engines such as Google dataset search\footnote{https://datasetsearch.research.google.com/} 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\footnote{https://2021.esec-fse.org/track/fse-2021-artifacts} 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.\cite{193} 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.

7.3.2 Reproducibility and replicability: The importance of reproducibility and replicability has been emphasized and understood by the software engineering community\cite{259}. It has lead to a concrete artifact evaluation mechanism adopted by leading software engineering conferences. For example, \textsc{fse} artifact evaluation divides artifacts into five categories—\textit{functional}, \textit{reusable}, \textit{available}, \textit{results reproduced}, and \textit{results replicated}.\footnote{https://2021.esec-fse.org/track/fse-2021-artifacts} 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\cite{48}) are being made to support ml research reproducible and replicable.

7.3.3 Maturity in ML development: 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 \texttt{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 \texttt{DARVIZ}\footnote{https://darviz.org/} for \texttt{DL} model visualization, \texttt{MLFlow}\footnote{https://mlflow.org/} for managing the ml lifecycle, and DeepFault\footnote{https://deepfault.com/} for identifying faults in \texttt{DL} programs. Such efforts are expected to address the challenge.

Software Engineering for Machine Learning (\textsc{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.

7.3.4 Hardware resources: ml development is resource hungry. Certain \texttt{DL} models (such as models based on \texttt{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.

The first internal threats to validity relates to the concern of covering all the relevant articles in the selected domain. 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 extensive manual search process on commonly used digital libraries with the help of a comprehensive set of search terms; we augment the search terms that are used in related articles to maximize the chances of identifying the relevant articles.

Another threat to validity is the validity of data extraction and their interpretation applicable to the generated summary and metadata for each primary 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 [396] containing all the metadata for all the articles selected in each of the phases of article selection. In addition, inspired by previous surveys [22, 174], we have developed a website6 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 repository7 open-source to allow the community to extend the living literature survey.

8 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 479 primary studies spanning from 2011 to 2021 (and to some extent 2022) covering 12 software engineering categories. We present a synthesized 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.

Acknowledgments

This work is supported by the ERC Advanced fellowship grant no. 741278 (EPIC).

References

[1] 2020. GitHub archive. https://www.github.org/
[2] Raja Abbas, Fawzi Abdulaniz Albaloooshi, and Mustafa Hammad. 2020. Software change proneness prediction using machine learning. In 2020 International Conference on Innovation and Intelligence for Informatics, Computing and Technologies (3ICT). IEEE, 1–7.
[3] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. 2020. A machine learning approach to improve the detection of ci skip commits. IEEE Transactions on Software Engineering (2020).
[4] Osama Abdeljaber, Onur Avcı, Serkan Kiranyaz, Moncef Gabbouj, and Daniel J Inman. 2017. Real-time vibration-based structural damage detection using one-dimensional convolutional neural networks. Journal of Sound and Vibration 388 (2017), 154–170.
[5] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. 2018. Large-Scale and Language-Oblivious Code Authorship Identification. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (Toronto, Canada) (CCS ’18). 101–114. https://doi.org/10.1145/3243734.3243738

6http://www.tusharma.in/ML4SCA
7https://github.com/tushartushar/ML4SCA
[6] Ibrahim Abunadi and Mamdouh Alenezi. 2015. Towards Cross Project Vulnerability Prediction in Open Source Web Applications. In Proceedings of the The International Conference on Engineering & MIS 2015 (Istanbul, Turkey) (ICEMIS ’15). Association for Computing Machinery, New York, NY, USA, Article 42, 5 pages. https://doi.org/10.1145/2832987.2833051

[7] Simran Aggarwal. 2019. Software Code Analysis Using Ensemble Learning Techniques. In Proceedings of the International Conference on Advanced Information Science and System (Singapore, Singapore) (AISS ’19). Article 9, 7 pages. https://doi.org/10.1145/3373477.3373486

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

[9] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kwei-Wei Chang. 2020. A Transformer-Based Approach for Source Code Summarization. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. 4998–5007. https://doi.org/10.18653/v1/2020.acl-main.449

[10] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. 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 (Gothenburg, Sweden) (ICSE-SEET ’18). 78–87. https://doi.org/10.1145/3183377.3183383

[11] H. A. Al-Jamimi and M. Ahmed. 2013. Machine Learning-Based Software Quality Prediction Models: State of the Art. In Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (Lille, France). 1–10. http://jmlr.org/proceedings/papers/v37/Allamanis15.pdf

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

[13] A. Al-Shaaby, Hamoud I. Aljamaan, and M. Alshayeb. 2020. Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review. Arabian Journal for Science and Engineering 45 (2020), 2341–2369.

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

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

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

[17] Sultan Alhussain, Simon Coupland, Robert John, and Maria Kavanagh. 2013. Towards machine learning based design pattern recognition. In Proceedings of the 13th UK Workshop on Computational Intelligence (UWCI). IEEE, 244–251.

[18] Huda Ali Altaw, Tae Oh, Ernest Fokoue, and Bill Stackpole. 2016. Android Malware Detection Using Category-Based Machine Learning Classifiers. In Proceedings of the 17th Annual Conference on Information Technology Education (Boston, Massachusetts, USA) (SIGITE ’16). 54–59. https://doi.org/10.1145/2978192.2978218

[19] E. A. Alikhashashneh, R. R. Raje, and J. H. Hill. 2018. 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). 1–8. https://doi.org/10.1109/AICCSA.2018.8612819

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

[21] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (Bergamo, Italy) (ESEC/FSE 2015). 38–49. https://doi.org/10.1145/2786805.2786849

[22] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. ACM Comput. Surv. 51, 4, Article 81 (July 2018), 37 pages. https://doi.org/10.1145/3212695

[23] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In International Conference on Learning Representations.

[24] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. arXiv:1602.03001 [cs.LG].

[25] M. Allamanis and C. Sutton. 2013. Mining source code repositories at massive scale using language modeling. In 2013 13th Working Conference on Mining Software Repositories (MSR). 207–216. https://doi.org/10.1109/MSR.2013.6624029

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

[27] Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. 2013. Bimodal Modelling of Source Code and Natural Language. In Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (Lille, France) (ICML’15). 2123–2132.

[28] Kevin Allix, Tegawendé F. Bisseyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In Proceedings of the 13th International Conference on Mining Software Repositories (Austin, Texas) (MSR ’16).
468–471. https://doi.org/10.1145/2901739.2903508

[29] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. arXiv:1808.01400 [cs.LG].

[30] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-Based Representation for Predicting Program Properties. SIGPLAN Not. 53, 4 (June 2018), 404–419. https://doi.org/10.1145/3192412

[31] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning Distributed Representations of Code. Proc. ACM Program. Lang. 3, POPL, Article 40 (January 2019), 29 pages. https://doi.org/10.1145/3290353

[32] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastian Uchitel. 2015. Automated Support for Diagnosis and Repair. Commun. ACM 58, 2 (January 2015), 65–72. https://doi.org/10.1145/2658986

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

[34] H. Alves, B. Fonseca, and N. Antunes. 2016. Experimenting Machine Learning Techniques to Predict Vulnerabilities. In 2016 Seventh Latin-American Symposium on Dependable Computing (LADC). 151–156. https://doi.org/10.1109/LADC.2016.32

[35] Boukhdir Amal, Marouane Kessentini, Slim Bechikh, Josselin Dea, and Lamjed Ben Said. 2014. On the Use of Machine Learning and Distributional Representations of Source Code. In Knowledge- Based Systems 128 (2017), 43 – 58. https://doi.org/10.1016/j.knosys.2017.04.014

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

[37] L. A. Amorim, M. F. Freitas, A. Dantas, E. F. de Souza, C. G. Camilo-Junior, and W. S. Martins. 2018. A New Word Embedding Approach to Evaluate Potential Fixes for Automated Program Repair. In 2018 International Joint Conference on Neural Networks (IJCNN). 1–8. https://doi.org/10.1109/IJCNN.2018.8499079

[38] M. Aniche, E. Maziero, R. Durelli, and V. Durelli. 2020. The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring. IEEE Transactions on Software Engineering (2020), 1–1. https://doi.org/10.1109/TSE.2020.3021736

[39] "Omer Furak Arar and Kürşat Ayan. 2015. Software defect prediction using cost-sensitive neural network. Applied Soft Computing 33 (2015), 263–277.

[40] Francesca Arcelli Fontana and Marco Zanoni. 2017. Code smell severity classification using machine learning techniques. Knowledge-Based Systems 128 (2017), 43 – 58. https://doi.org/10.1016/j.knosys.2017.04.014

[41] Vamsi Krishna Aribandi, Lov Kumar, Lalita Bhanu Murthy Neti, and Aneesh Krishna. 2019. Prediction of Refactoring-Prone Classes Using Ensemble Learning. In Neural Information Processing, Tom Gedeon, Kok Wai Wong, and Minho Lee (Eds.). 242–250.

[42] S. Axelsson, D. Baca, Robert Feldt, Darius Sidlauskas, and Denis Kacan. 2009. Detecting Defects with an Interactive Code Review Tool Based on Visualisation and Machine Learning. In SEKE.

[43] David Azcona, Pyush Arora, I-Han Hsiao, and Alan Smeaton. 2019. User2code2vec: Embeddings for Profiling Students Based on Distributional Representations of Source Code. In Proceedings of the 9th International Conference on Learning Analytics & Knowledge (Tempe, AZ, USA) (LAK’19). 86–95. https://doi.org/10.1109/LAK.2019.3303813

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

[45] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. Proc. ACM Program. Lang. 3, OOPSLA, Article 159 (October 2019), 27 pages. https://doi.org/10.1145/3360585

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

[47] U. Bandara and G. Wijayarathna. 2011. A Machine Learning Based Tool for Source Code Plagiarism Detection. International Journal of Machine Learning and Computing (2011), 337–343.

[48] Vishnu Banna, Akhil Chinnakotla, Zhengxin Yan, Anirudh Vegesana, Naveen Vivek, Kruthi Krishnappa, Wexin Jiang, Yung-Hsiang Lu, George K. Thiruvathukal, and James C. Davis. 2021. An Experience Report on Machine Learning Reproducibility: Guidance for Practitioners and TensorFlow Model Garden Contributors. CoRR abs/2107.00821 (2021). arXiv:2107.00821 https://arxiv.org/abs/2107.00821

[49] A. Bansal, S. Haque, and C. McMillan. 2021. Project-Level Encoding for Neural Source Code Summarization of Subroutines. In 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) (ICPC). IEEE Computer Society, 253–264. https://doi.org/10.1109/ICPC52881.2021.00032

[50] Antoine Barbez, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2020. A machine-learning based ensemble method for anti-patterns detection. Journal of Systems and Software 161 (2020), 110486. https://doi.org/10.1016/j.jss.2019.110486
[51] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation.

[52] Canan Batu Şahin and Laith Abuailah. 2021. A Novel Deep Learning-Based Feature Selection Model for Improving the Static Analysis of Vulnerability Detection. *Neural Comput. Appl.* 33, 20 (Oct 2021), 14049–14067. https://doi.org/10.1007/s00521-021-06047-x

[53] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montreal, Canada) (NIPS’18). 3589–3601.

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

[55] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-Symbolic Program Corrector for Introductory Programming Assignments. In *Proceedings of the 46th International Conference on Software Engineering* (Gothenburg, Sweden) (ICSE ’18). 60–70. https://doi.org/10.1145/3180155.3180219

[56] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay. 2020. Vulnerability Prediction From Source Code Using Machine Learning. *IEEE Access* 8 (2020), 150672–150684. https://doi.org/10.1109/ACCESS.2020.3016774

[57] Paul E. Black. 2007. Software Assurance with SAMATE Reference Dataset, Tool Standards, and Studies. (Oct. 2007).

[58] Frederick Boland and Paul Black. 2012. The Juliet 1.1 C/C++ and Java Test Suite. 45 (2012-10-01 2012). https://doi.org/10.1109/MC.

[59] David Bowes, Tracy Hall, Mark Harman, Yue Jia, Federica Sarro, and Fan Wu. 2016. Mutation-Aware Fault Prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (SSTTA 2016). Association for Computing Machinery, New York, NY, USA, 330–341. https://doi.org/10.1145/2931037.2931039

[60] Ronyérison Braga, Pedro Santos Neto, Ricardo Rabêlo, José Santiago, and Matheus Souza. 2018. A Machine Learning Approach to Generate Test Oracles. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering* (Sao Carlos, Brazil) (SBES’18). 142–151. https://doi.org/10.1109/3266237.3266273

[61] Alexander Brauckmann, André Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-Based Graph Representations for Deep Learning Models of Code. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) (CC 2020). 201–211.

[62] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative Code Modeling with Graphs. In *International Conference on Learning Representations*.

[63] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. 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* (Amsterdam, The Netherlands) (FSE ’09). 213–222. https://doi.org/10.1145/1595696.1595728

[64] Yuriy Brun and Alexandra Meliou. 2018. Software Fairness. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) (FSE 2018). Association for Computing Machinery, New York, NY, USA, 754–759. https://doi.org/10.1145/3236024.3264838

[65] Nghi D. Q. Bui, Lingxiao Jiang, and Y. Yu. 2018. Cross-Language Learning for Program Classification using Bilateral Tree-Based Convolutional Neural Networks. In *AAAI Workshops*.

[66] N. D. Q. Bui, Y. Yu, and L. Jiang. 2019. Bilateral Dependency Neural Networks for Cross-Language Algorithm Classification. In *1999 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 422–433. https://doi.org/10.1109/SANER.2019.8667995

[67] L. Butgereit. 2019. Using Machine Learning to Prioritize Automated Testing in an Agile Environment. In *2019 Conference on Information Communications Technology and Society* (ICTAS). 1–6. https://doi.org/10.1109/ICTAS.2019.8703639

[68] Cheng-Hao Cai, Jing Sun, and Gillian Dobbie. 2019. Automatic R-model repair using model checking and machine learning. *Automated Software Engineering* 26, 3 (Jan. 2019). https://doi.org/10.1007/s10515-019-00264-4

[69] José P. Cambronero and Martin C Rinard. 2019. AL: autogenerating supervised learning programs. *Proceedings of the ACM on Programming Languages* 3, OOPS (2019), 1–28.

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

[71] Frederico Luiz Caram, Bruno Rafael De Oliveira Rodrigues, Amadeu Silveira Campanelli, and Fernando Silva Parreiras. 2019. Machine Learning Techniques for Code Smells Detection: A Systematic Mapping Study. *International Journal of Software Engineering and Knowledge Engineering* 29, 02 (2019), 285–316. https://doi.org/10.1142/S021819401950013X arXiv:https://doi.org/10.1142/S021819401950013X

[72] Silvio Cesare, Yang Xiang, and Jun Zhang. 2013. Clonewise – Detecting Package-Level Clones Using Machine Learning. In *Security and Privacy in Communication Networks*, Tanveer Zia, Albert Zomaya, Vijay Varadharajan, and Morley Mao (Eds.), 197–215.

[73] M. Cetiner and O. K. Sahingoz. 2020. A Comparative Analysis for Machine Learning based Software Defect Prediction Systems. In *2020 11th International Conference on Computing, Communication and Networking Technologies* (ICCCNT). 1–7. https://doi.org/10.1109/
Vasiliki Efstathiou and Diomidis Spinellis. 2019. Semantic Source Code Models Using Identifier Embeddings. In Marco D’Ambros, Michele Lanza, and Romain Robbes. 2012. Evaluating Defect Prediction Approaches: A Benchmark and an Extensive Study. In Proceedings of the 3rd International Conference on Technical Debt, 31–40.

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

Daniel Cruz, Amanda Santana, and Eduardo Figueiredo. 2020. Detecting Bad Smells with Machine Learning Algorithms: An Empirical Study. In Proceedings of the 3rd International Conference on Technical Debt (Seoul, Republic of Korea) (TechDebt ’20). 31–40. https://doi.org/10.1145/3387906.3388618

Jianfeng Cui, Lixin Wang, Xin Zhao, and Hongyi Zhang. 2020. Towards predictive analysis of android vulnerability using statistical codes and machine learning for IoT applications. Computer Communications 155 (2020), 125 – 131. https://doi.org/10.1016/j.comcom.2020.02.078

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

Altino Dantas, Eduardo F. de Souza, Jerffeson Souza, and Celso G. Camilo-Junior. 2019. Code Naturalness to Assist Search Space Exploration in Search-Based Program Repair Methods. In Search-Based Software Engineering, Shiva Nejati and Gregory Gay (Eds.). 164–170.

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

Seema Dewangan, Rajwant Singh Rao, Alok Mishra, and Manjari Gupta. 2021. A Novel Approach for Code Smell Detection: An Empirical Study. IEEE Access 9 (2021), 162869–162883.

N. Dhamayanthi and B. Lavanya. 2019. Improvement in Software Defect Prediction Outcome Using Principal Component Analysis and Ensemble Machine Learning Algorithms. In International Conference on Intelligent Data Communication Technologies and Internet of Things (ICICT) 2018, Jude Hemanth, Xavier Fernando, Pavel Lafata, and Zubair Baig (Eds.). 397–406.

Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, and Federica Sarro. 2011. A Genetic Algorithm to Configure Support Vector Machines for Predicting Fault-Prone Components. In Product-Focused Software Process Improvement, Danilo Caivano, Markku Olivo, Maria Teresa Baldassarre, and Giuseppe Visaggio (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 247–261.

Altino Dantas, Eduardo F. de Souza, Jerffeson Souza, and Celso G. Camilo-Junior. 2019. Code Naturalness to Assist Search Space Exploration in Search-Based Program Repair Methods. In Search-Based Software Engineering, Shiva Nejati and Gregory Gay (Eds.). 164–170.

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

Seema Dewangan, Rajwant Singh Rao, Alok Mishra, and Manjari Gupta. 2021. A Novel Approach for Code Smell Detection: An Empirical Study. IEEE Access 9 (2021), 162869–162883.

N. Dhamayanthi and B. Lavanya. 2019. Improvement in Software Defect Prediction Outcome Using Principal Component Analysis and Ensemble Machine Learning Algorithms. In International Conference on Intelligent Data Communication Technologies and Internet of Things (ICICT) 2018, Jude Hemanth, Xavier Fernando, Pavel Lafata, and Zubair Baig (Eds.). 397–406.

Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, and Federica Sarro. 2011. A Genetic Algorithm to Configure Support Vector Machines for Predicting Fault-Prone Components. In Product-Focused Software Process Improvement, Danilo Caivano, Markku Olivo, Maria Teresa Baldassarre, and Giuseppe Visaggio (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 247–261.

D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. 2018. Detecting code smells using machine learning techniques: Are we there yet?. In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). 612–621. https://doi.org/10.1145/SANER.2018.8330266

Geanderson Esteves Dos Santos, E. Figueiredo, Adriano Veloso, Markos Viggioi, and N. Ziviani. 2020. Understanding machine learning software defect predictions. Autom. Softw. Eng. 27 (2020), 369–392.

Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. LEOPARD: Identifying Vulnerable Code through Program Metrics. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 60–71. https://doi.org/10.1109/ICSE.2019.00024

Yao Du, Xiaqing Wang, and Junfeng Wang. 2015. A Static Android Malicious Code Detection Method Based on Multi-Source Fusion. Sec. and Commun. Netw. 8, 17 (nov 2015), 3238–3246. https://doi.org/10.1002/sec.1248

V. H. S. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. C. Dias, and M. P. Guimarães. 2019. Machine Learning Applied to Software Testing: A Systematic Mapping Study. In Proceedings of the 17th International Conference on Mining Software Repositories (Seoul, Republic of Korea) (MSR ’19). 243–253. https://doi.org/10.1145/3379597.3387445

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

Seema Dewangan, Rajwant Singh Rao, Alok Mishra, and Manjari Gupta. 2021. A Novel Approach for Code Smell Detection: An Empirical Study. IEEE Access 9 (2021), 162869–162883.

N. Dhamayanthi and B. Lavanya. 2019. Improvement in Software Defect Prediction Outcome Using Principal Component Analysis and Ensemble Machine Learning Algorithms. In International Conference on Intelligent Data Communication Technologies and Internet of Things (ICICT) 2018, Jude Hemanth, Xavier Fernando, Pavel Lafata, and Zubair Baig (Eds.). 397–406.

Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, and Federica Sarro. 2011. A Genetic Algorithm to Configure Support Vector Machines for Predicting Fault-Prone Components. In Product-Focused Software Process Improvement, Danilo Caivano, Markku Olivo, Maria Teresa Baldassarre, and Giuseppe Visaggio (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 247–261.

D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. 2018. Detecting code smells using machine learning techniques: Are we there yet?. In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). 612–621. https://doi.org/10.1145/SANER.2018.8330266

Geanderson Esteves Dos Santos, E. Figueiredo, Adriano Veloso, Markos Viggioi, and N. Ziviani. 2020. Understanding machine learning software defect predictions. Autom. Softw. Eng. 27 (2020), 369–392.

Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. LEOPARD: Identifying Vulnerable Code for Vulnerability Assessment Through Program Metrics. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 60–71. https://doi.org/10.1109/ICSE.2019.00024

Yao Du, Xiaqing Wang, and Junfeng Wang. 2015. A Static Android Malicious Code Detection Method Based on Multi-Source Fusion. Sec. and Commun. Netw. 8, 17 (Nov 2015), 3238–3246. https://doi.org/10.1002/sec.1248

V. H. S. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. C. Dias, and M. P. Guimarães. 2019. Machine Learning Applied to Software Testing: A Systematic Mapping Study. IEEE Transactions on Reliability 68, 3 (2019), 1189–1212. https://doi.org/10.1109/TR.2019.2892517

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

Vasiliki Efstathiou and Diomidis Spinellis. 2019. Semantic Source Code Models Using Identifier Embeddings. In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR) ’19. 29–33. https://doi.org/10.1109/MSR.2019.00015

Yuval Elovici, Asaf Shabtai, Robert Moskovitch, Gil Tahan, and Chanan Glezer. 2007. Applying Machine Learning Techniques for Detection of Malicious Code in Network Traffic. In KI 2007: Advances in Artificial Intelligence, Joachim Hertzberg, Michael Beetz, and
Roman Englert (Eds.). 44–50.

[119] Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. 2019. DeepFault: Fault Localization for Deep Neural Networks. In Fundamental Approaches to Software Engineering, Reiner Hähnle and Wil van der Aalst (Eds.). Springer International Publishing, Cham, 171–191.

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

[121] Khashayar Etemadi and Martin Monperrus. 2020. 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. 470–475.

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

[123] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (Vasteras, Sweden) (ASE ’14). 313–324. https://doi.org/10.1145/2642937.2642982

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

[125] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingshui Shi. 2020. Functional Code Clone Detection with Syntax and Semantics Fusion Learning. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (Virtual Event, USA) (ISSTA 2020). 516–527. https://doi.org/10.1145/3395363.3397362

[126] Yong Fang, Yongchong Liu, Cheng Huang, and Liang Liu. 2020. FastEmbed: Predicting vulnerability exploitation possibility based on ensemble machine learning algorithm. PLoS ONE 15 (Feb. 2020), e0228439. https://doi.org/10.1371/journal.pone.0228439 ADS Bibcode: 2020PLoSO..1528439F.

[127] Ebubeogu Amarachukwu Felix and Sai Peck Lee. 2017. Integrated approach to software defect prediction. IEEE Access 5 (2017), 21524–21547.

[128] Rudolf Ferenc, Péter Hegedűs, Péter Gyimesi, Gábor Antal, Dénes Bán, and Tibor Gyimóthy. 2019. Challenging Machine Learning Approaches to Software Engineering Tactics?. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). 15–26. https://doi.org/10.1109/ICSE.2017.10
[189] Shuyao Jiang. 2019. Boosting neural commit message generation with code semantic analysis. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 1280–1282.

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

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

[192] Gong Jie, Kuang Xiao-Hui, and Liu Qiang. 2016. Survey on Software Vulnerability Analysis Method Based on Machine Learning. In 2016 IEEE First International Conference on Data Science in Cyberspace (DSC). 642–647. https://doi.org/10.1109/DSC.2016.33

[193] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. 2019. 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 (Tallinn, Estonia) (ESEC/FSE 2019). Association for Computing Machinery, New York, NY, USA, 695–705. https://doi.org/10.1145/3338906.3338941

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

[195] René Just, Darioush Jalali, and Michael D. Ernst. 2014. 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 (San Jose, CA, USA) (ISSTA 2014). Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[196] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119), Hal Daumé III and Aarti Singh (Eds.). PMLR, 5110–5121. https://proceedings.mlr.press/v119/kanade20a.html

[197] Hong Jin Kang, Tegawendé F. Bassyande, and David Lo. 2019. Assessing the Generalizability of Code2vec Token Embeddings. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). 1–12. https://doi.org/10.1109/ASE.2019.00011

[198] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (Seoul, South Korea) (ICSE ’20). 1073–1085. https://doi.org/10.1145/3377811.3380342

[199] A. Kaur, S. Jain, and S. Goel. 2017. A Support Vector Machine Based Approach for Code Smell Detection. In 2017 International Conference on Machine Learning and Data Science (MLDS). 9–14. https://doi.org/10.1109/MLDS.2017.8

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

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

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

[203] Muhammad Noman Khalid, Humera Farooq, Muhammad Talha Alam, and Kamran Rasheed. 2019. Predicting Web Vulnerabilities in Web Applications Based on Machine Learning. In Intelligent Technologies and Applications (Communications in Computer and Information Science). Imran Sarwar Bajwa, Fairouz Kamareddine, and Anna Costa (Eds.). Springer, Singapore, 473–484. https://doi.org/10.1007/978-981-33-6502-7_41

[204] Bilal Khan, Danish Iqbal, and Sher Badshah. 2020. Cross-Project Software Fault Prediction Using Data Leveraging Technique to Improve Software Quality. In Proceedings of the Evaluation and Assessment in Software Engineering (Trondheim, Norway) (EASE ’20). 434–438. https://doi.org/10.1145/3383219.3383281

[205] Junae Kim, David Hubchenko, and Paul Montague. 2019. Towards Attention Based Vulnerability Discovery Using Source Code Representation. In Artificial Neural Networks and Machine Learning – ICANN 2019: Text and Time Series, Igor V. Tetko, Věra Kůrková, Pavel Karpov, and Fabian Theis (Eds.). 731–746. https://doi.org/10.1007/s11109-019-06164-4

[206] J. Kim, M. Kwon, and S. Yoo. 2018. Generating Test Input with Deep Reinforcement Learning. In 2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST). 51–58.

[207] Sangwoo Kim, Seokmyung Hong, Jaesang Oh, and Heejo Lee. 2018. Obfuscated VBA Macro Detection Using Machine Learning. In 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 490–501. https://doi.org/10.1109/DSN.2018.00057

[208] Y. Kim, C. Jeong, A. Jeong, and H. S. Kim. 2009. Risky Module Estimation in Safety-Critical Software. In 2009 Eighth IEEE/ACIS International Conference on Computer and Information Science. 967–970. https://doi.org/10.1109/ICIS.2009.83

[209] Patrick Knab, Martin Pinzger, and Abraham Bernstein. 2006. Predicting Defect Densities in Source Code Files with Decision Tree Learners. In Proceedings of the 2006 International Workshop on Mining Software Repositories (Shanghai, China) (MSR ’06). 119–125. https://doi.org/10.1145/1137983.1138012
[233] Song-Mi Lee, Sang Min Yoon, and Heeryon Cho. 2017. Human activity recognition from accelerometer data using Convolutional Neural Network. In *Big Data and Smart Computing (BigComp)*, 2017 IEEE International Conference on. IEEE, 131–134.

[234] Stanislav Levin and Amiram Yehudai. 2017. 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*, 97–106.

[235] Tomasz Lewowski and Lech Madeyski. 2022. Code smells detection using artificial intelligence techniques: A business-driven systematic review. In *Developments in Information and Communication Technologies (EICT)*. Springer, 285–319.

[236] Boao Li, Meng Yan, Xin Xia, Xing Hu, Ge Li, and David Lo. 2020. 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 Symposia on the Foundations of Software Engineering* (Virtual Event, USA) (ESEC/FSE 2020). 1571–1575. https://doi.org/10.1145/3368089.3417926

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

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

[239] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. EditSum: A Retrieve-and-Edit Framework for Source Code Summarization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 155–166.

[240] Yuan Cheng Li, Rong Ma, and Runhai Jiao. 2015. A Hybrid Malicious Code Detection Method based on Deep Learning. *International journal of security and its applications* 9 (2015), 205–216.

[241] Yi Li, Shaohua Wang, and Tian N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) (ICSE ’20). 602–614. https://doi.org/10.1145/3377811.3380345

[242] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. A Context-based Automated Approach for Method Name Consistency Checking and Suggestion. In *2021 IEEE/ACM 43rd International Conference on Software Engineering* (ICSE). IEEE, 574–586.

[243] Yi Li, Shaohua Wang, Tian N. Nguyen, and Son Van Nguyen. 2019. Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 162 (October 2019), 30 pages. https://doi.org/10.1145/3360588

[244] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin. 2019. A Comparative Study of Deep Learning-Based Vulnerability Detection System. *IEEE Access* 7 (2019), 103184–103197. https://doi.org/10.1109/ACCESS.2019.2930578

[245] Hongliang Liang, Yue Yu, Lin Jiang, and Zhousi Xie. 2019. Seml: A semantic LSTM model for software defect prediction. *IEEE Access* 7 (2019), 83812–83824.

[246] H. Lim. 2018. Applying Code Vectors for Presenting Software Features in Machine Learning. In *2018 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 01, 803–804. https://doi.org/10.1109/COMPSAC.2018.00128

[247] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin. 2019. A Comparative Study of Deep Learning-Based Vulnerability Detection System. *IEEE Access* 7 (2019), 103184–103197. https://doi.org/10.1109/ACCESS.2019.2930578

[248] Boao Li, Meng Yan, Xin Xia, Xing Hu, Ge Li, and David Lo. 2020. 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 Symposia on the Foundations of Software Engineering* (Virtual Event, USA) (ESEC/FSE 2020). 1571–1575. https://doi.org/10.1145/3368089.3417926

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

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

[251] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. EditSum: A Retrieve-and-Edit Framework for Source Code Summarization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 155–166.

[252] Yuan Cheng Li, Rong Ma, and Runhai Jiao. 2015. A Hybrid Malicious Code Detection Method based on Deep Learning. *International journal of security and its applications* 9 (2015), 205–216.

[253] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) (ICSE ’20). 602–614. https://doi.org/10.1145/3377811.3380345

[254] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. A Context-based Automated Approach for Method Name Consistency Checking and Suggestion. In *2021 IEEE/ACM 43rd International Conference on Software Engineering* (ICSE). IEEE, 574–586.

[255] Yi Li, Shaohua Wang, Tian N. Nguyen, and Son Van Nguyen. 2019. Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 162 (October 2019), 30 pages. https://doi.org/10.1145/3360588
[324] Miroslaw Ochodek, Regina Hebig, Wilhelm Meding, Gert Frost, and Miroslaw Staron. 2019. Recognizing lines of code violating company-specific coding guidelines using machine learning. Empirical Software Engineering 25 (2019), 220–265.

[325] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. 2015. Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation. In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). 574–584. https://doi.org/10.1109/ASE.2015.36

[326] Ahmet Okutan and Olcay Taner Yildiz. 2014. Software defect prediction using Bayesian networks. Empirical Software Engineering 19, 1 (2014), 154–181.

[327] Daniel Oliveira, Wesley K. G. Assunção, Leonardo Souza, William Oizumi, Alessandro Garcia, and Balduino Fonseca. 2020. Applying Machine Learning to Customized Smell Detection: A Multi-Project Study (SRES ’20). 233–242. https://doi.org/10.1145/3422392.3422427

[328] Safa Omri and Carsten Sinz. 2020. Deep Learning for Software Defect Prediction: A Survey. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (Seoul, Republic of Korea) (ICSEW’20). 209–214. https://doi.org/10.1145/3387940.3391463

[329] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan. 2015. Buffer Overflow Vulnerability Prediction from x86 Executables Using Static Analysis and Machine Learning. In 2015 IEEE 39th Annual Computer Software and Applications Conference, Vol. 2. 450–459. https://doi.org/10.1109/COMPSAC.2015.78

[330] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. 2016. 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). IEEE, 244–255.

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

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

[333] A. K. Pandey and Manjari Gupta. 2018. Software fault classification using extreme learning machine: a cognitive approach. Evolutionary Intelligence (2018), 1–8

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

[335] Y. Pang, X. Xue, and A. S. Namin. 2016. Early Identification of Vulnerable Software Components via Ensemble Learning. In 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA). 476–481. https://doi.org/10.1109/ICMLA.2016.0084

[336] Yulei Pang, Xiaozhen Xue, and Huoueng Gao. 2017. Predicting Vulnerable Software Components through Deep Neural Network. In Proceedings of the 2017 International Conference on Deep Learning Technologies (Chengdu, China) (ICDLT ’17). Association for Computing Machinery, New York, NY, USA, 6–10. https://doi.org/10.1145/3094243.3094245

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

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

[339] Kayur Patel, James Fogarty, James A. Landay, and Beverly Harrison. 2008. Investigating Statistical Machine Learning as a Tool for Software Development. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (Florence, Italy) (CHI ’08). 667–676. https://doi.org/10.1145/1357054.1357160

[340] Fabiano Pecorelli, Dario Di Nucci, Coen De Roover, and Andrea De Lucia. 2019. 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 (Tallinn, Estonia) (MalTeSQuE 2019). 19–24. https://doi.org/10.1145/3340482.3342744

[341] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia. 2019. Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection. In 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC). 93–104.

[342] J. D. Pereira, J. R. Campos, and M. Vieira. 2019. 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). 19–24. https://doi.org/10.1145/3342292.3342247

[343] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. 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 (Denver, Colorado, USA) (CCS ’15). 426–437. https://doi.org/10.1145/2810103.2813604

[344] Hung Phan and Ali Jannesari. 2020. 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 (Virtual, USA) (RL+SE&PL ’20). 3–12. https://doi.org/10.1145/3416506.3423576
5, 10 pages. https://doi.org/10.1145/3306446.3340828

[369] Guillermo Rodriguez, Cristian Mateos, Luciano Listorti, Brian Hammer, and Sanjay Misra. 2019. A Novel Unsupervised Learning Approach for Assessing Web Services Refactoring. In Information and Software Technologies, Robertas Damaševičius and Giedrė Vasiljevičiūnė (Eds.). 273–284.

[370] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). 757–762. https://doi.org/10.1109/ICMLA.2018.00120

[371] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). 757–762. https://doi.org/10.1109/ICMLA.2018.00120

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

[373] N. Saccente, J. Dehlinger, L. Deng, S. Chakraborty, and Y. Xiong. 2019. 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). 114–121. https://doi.org/10.1109/ASEW.2019.00040

[374] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on Source Code: A Neural Code Search. In Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (Philadelphia, PA, USA) (MAPL 2018). 31–41. https://doi.org/10.1145/3211346.3211353

[375] Priyadarshini Suresh Sagar, Eman Abdulah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Christian D. Newman. 2021. Comparing Commit Messages and Source Code Metrics for the Prediction Refactoring Activities. Algorithms 14, 10 (2021). https://doi.org/10.3390/733_Sagar2021

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

[377] S. Saha, R. k. Saha, and M. r. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 13–24. https://doi.org/10.1109/ICSE.2019.00020

[378] Isabel Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. 2020. Web Service API Anti-patterns Detection as a Multi-label Learning Problem. In International Conference on Web Services. Springer, 114–132.

[379] Tara N Sainath, Brian Kingsbury, George Saon, Hagen Soltou, Abdel-rahman Mohamed, George Dahl, and Bhuvana Ramabhadran. 2015. Deep convolutional neural networks for large-scale speech tasks. Neural Networks 64 (2015), 39–48.

[380] Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. 2020. Type Error Feedback via Analytic Program Repair. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (London, UK) (PLDI 2020). 16–30. https://doi.org/10.1145/3385412.3386005

[381] Anush Sankaran, Rahul Arialikatte, Senthil Mani, Shreyas Khare, naveen Panwar, and Neelamadhab Gantayat. 2017. DARVIZ: Deep Abstract Representation, Visualization, and Verification of Deep Learning Models. CoRR abs/1708.04915 (2017). arXiv:1708.04915 http://arxiv.org/abs/1708.04915

[382] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral. 2018. 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). 311–322. https://doi.org/10.1109/SANER.2018.830219

[383] Igor Santos, Jaime Devesa, Félix Brezo, Javier Nieves, and Pablo Garcia Bringas. 2013. OPEM: A Static-Dynamic Approach for Machine-Learning-Based Malware Detection. In International Joint Conference CISIS’12-ICEUTE ‘12-SOCO ’12 Special Sessions, Álvaro Herrero, Václav Snášel, Ajith Abraham, Ivan Zelinka, Bruno Baraque, Héctor Quintián, José Luis Calvo, Javier Sedano, and Emilio Corchado (Eds.). 271–280.

[384] F. Sarro, S. Di Martino, F. Ferrucci, and C. Gravinio. 2012. 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 (Trento, Italy) (SAC ’12). Association for Computing Machinery, New York, NY, USA, 1215–1220. https://doi.org/10.1145/2245276.2231967

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

[386] Max Eric Henry Schumacher, Kim Tuyen Le, and Artur Andriejev. 2020. Improving Code Recommendations by Combining Neural and Classical Machine Learning Approaches. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (Seattle, Republic of Korea) (ICSEW ’20). 476–482. https://doi.org/10.1109/ICSEW42.2020.00290

[387] R. Schuster, Congzheng Song, Erin Tromer, and Vitaly Smilatikov. 2021. You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. In 8th USENIX Security Symposium (USENIX Security 21).

[388] T. Sethi and Gagandep. 2016. Improved approach for software defect prediction using artificial neural networks. In 2016 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO). 480–485. https://doi.org/10.1109/ICRITO.2016.7785003

, Vol. 0, No. 0, Article 0. Publication date: 2022.
A Survey on Machine Learning Techniques for Source Code Analysis

[389] Burr Settles. 2009. Active learning literature survey. (2009).
[390] Asaf Shabtai, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. 2009. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. Information Security Technical Report 14, 1 (2009), 16–29. https://doi.org/10.1016/j.jistr.2009.03.003

[391] L. K. Shar, L. C. Briand, and H. B. K. Tan. 2015. Web Application Vulnerability Prediction Using Hybrid Program Analysis and Machine Learning. IEEE Transactions on Dependable and Secure Computing 12, 6 (2015), 688–707. https://doi.org/10.1109/TDSC.2014.2373377

[392] Tushar Sharma. 2018. DesigniteJava. https://doi.org/10.5281/zenodo.2566861 https://github.com/tushartushar/DesigniteJava.

[393] Tushar Sharma. 2019. CodeSplit for C#. https://doi.org/10.5281/zenodo.2566865 https://github.com/tushartushar/CodeSplitJava.

[394] Tushar Sharma, Vasiliki Efstatiou, Panos Louridas, and Diomidis Spinellis. 2021. Code smell detection by deep direct-learning and transfer-learning. Journal of Systems and Software 176 (2021), 110936. https://doi.org/10.1016/j.jss.2021.110936

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

[396] T. Sharma and M. Kessentini. 2021. QScored: A Large Dataset of Code Smells and Quality Metrics. In 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (MSR). IEEE Computer Society, Los Alamitos, CA, 590–594. https://doi.org/10.1109/MSR52588.2021.00080

[397] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. 2016. 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). https://doi.org/10.1145/2896955.2896958

[398] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. Journal of Systems and Software 138 (2018), 158–173. https://doi.org/10.1016/j.jss.2017.12.034

[399] Andrey Shedo, Ilya Palachev, Andrey Kvochko, Aleksandr Semenov, and Kwangwon Sun. 2020. Applying Probabilistic Models to C++ Code on an Industrial Scale. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (Seoul, Republic of Korea) (ICSEW’20). 595–602. https://doi.org/10.1145/3387940.3391477

[400] Zhidong Shen and S. Chen. 2020. A Survey of Automatic Software Vulnerability Detection, Program Repair, and Defect Prediction Techniques. Secur. Commun. Networks 2020 (2020), 8858010:1–8858010:16.

[401] A. Sheneamer and J. Kalita. 2016. Semantic Clone Detection Using Machine Learning. In 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA). 1024–1028. https://doi.org/10.1109/ICMLA.2016.0185

[402] Ke Shi, Yang Lu, Jingfei Chang, and Zhen Wei. 2020. PathPair2Vec: An AST path pair-based code representation method for defect prediction. Journal of Computer Languages 59 (2020), 100979. https://doi.org/10.1016/j.jolar.2020.100979

[403] Y. Shido, Y. Kobayashi, A. Yamamoto, A. Miyamoto, and T. Matsumura. 2019. Automatic Source Code Summarization with Extended Tree-LSTM. In 2019 International Joint Conference on Neural Networks (IJCNN). 1–8. https://doi.org/10.1109/IJCNN.2019.8851751

[404] S. Shim, P. Patil, R. R. Yadav, A. Shinde, and V. Devale. 2020. DeeperCoder: Code Generation Using Machine Learning. In 2020 10th Annual Computing and Communication Workshop and Conference (CCWC). 0194–0199. https://doi.org/10.1109/CCWC47524.2020.9031149

[405] K. Shimonaka, S. Sumi, Y. Higo, and S. Kusumoto. 2016. Identifying Auto-Generated Code by Using Machine Learning Techniques. In 2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP). 18–23. https://doi.org/10.1109/IWESEP.2016.18

[406] L. Shiqi, T. Shengwei, Y. Long, Y. Jing, and S. Hua. 2018. Android malicious code Classification using Deep Belief Network. KSII Transactions on Internet and Information Systems 12 (Jan. 2018), 454–475. https://doi.org/10.3837/tiis.2018.01.022

[407] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Transition of Machine Learning with Co-Attentive Representation. In Proceedings of the 28th International Conference on Program Comprehension (Seoul, Republic of Korea) (ICPC ’20). 196–207. https://doi.org/10.1145/3387904.3389269

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

[409] Ajmer Singh, Rajesh Bhatia, and Anita Singhroha. 2018. Taxonomy of machine learning algorithms in software fault prediction using object oriented metrics. Procedia computer science 132 (2018), 993–1001.

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

[411] P. Singh and R. Malhotra. 2017. 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). 204–209. https://doi.org/10.1109/ICRITO.2017.8542425

[412] R. Singh, J. Singh, M. S. Gill, R. Malhotra, and Garima. 2020. Transfer Learning Code Vectorizer based Machine Learning Models for Software Defect Prediction. In 2020 International Conference on Computational Performance Evaluation (ComPE). 497–502. https://doi.org/10.1109/ComPE49325.2020.9200076
[414] Behjat Soltanifar, Shirin Akbarinasaji, Bora Caglayan, Ayse Basar Bener, Asli Filiz, and Bryan M Kramer. 2016. Software analytics in practice: a defect prediction model using code smells. In Proceedings of the 20th International Database Engineering & Applications Symposium. 148–155.

[415] Qinbao Song, Yuchen Guo, and Martin Shepperd. 2019. A Comprehensive Investigation of the Role of Imbalanced Learning for Software Defect Prediction. IEEE Transactions on Software Engineering 45, 12 (2019), 1253–1269. https://doi.org/10.1109/TSE.2018.2836442

[416] M. Soto and C. Le Goues. 2018. Common Statement Kind Changes to Inform Automatic Program Repair. In 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR). 102–105.

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

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

[419] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A Human Study of Comprehension and Code Summarization. In Proceedings of the 28th International Conference on Program Comprehension (Seoul, Republic of Korea) (ICPC ’20). 2–13. https://doi.org/10.1145/3387904.3389258

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

[421] Yulei Sui, Xiao Cheng, Guangqin Zhang, and Haoyu Wang. 2020. Flow2Vec: Value-Flow-Based Precise Code Embedding. Proc. ACM Program. Lang. 4, OOPSLA, Article 233 (November 2020), 27 pages. https://doi.org/10.1145/3428301

[422] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In Proceedings of the 25th international conference on compiler construction. ACM, 265–266.

[423] Kazi Zakia Sultana. 2017. 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). 1022–1025. https://doi.org/10.1109/ASE.2017.8115724

[424] Kazi Zakia Sultana, Vaibhav Anu, and Tai-Yin Chong. 2021. Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach. Journal of Software: Evolution and Process 33, 3 (2021), e2303. https://doi.org/10.1002/smr.2303

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

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

[427] Girish Suryanarayana, Ganesh Samarthym, and Tushar Sharma. 2014. Refactoring for Software Design Smells: Managing Technical Debt (1 ed.). Morgan Kaufmann.

[428] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In 2014 IEEE International Conference on Software Maintenance and Evolution. 476–480. https://doi.org/10.1109/ICSME.2014.77

[429] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. 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 (Virtual Event, USA) (ESEC/FSE 2020). 1433–1443. https://doi.org/10.1145/3368089.3417058

[430] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. In 2021 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE, 329–340.

[431] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: AI-Assisted Code Completion System. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery &amp; Data Mining (Anchorage, AK, USA) (KDD ’19). 2727–2735. https://doi.org/10.1145/3292500.3330699

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

[433] Akiyoshi Takahashi, Hiromitsu Shiina, and Nobuyuki Kobayashi. 2019. Automatic Generation of Program Comments for Computational Thinking. In 2019 8th International Workshop on Study of Comprehension and Code Summarization. In Proceedings of the IEEE conference on computer vision and pattern recognition. 181–191. https://doi.org/10.1109/ICME.2014.77

[434] K. Terada and Y. Watanobe. 2019. Code Completion for Programming Education based on Recurrent Neural Network. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). 207–217. https://doi.org/10.1109/SANER.2019.8667978

[435] P. Thongkum and S. Mekruksavanich. 2020. 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). 79–82. https://doi.org/10.1109/ECTIDAMTNCON48261.
[437] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé. 2020. 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). 981–992.

[438] Irene Tolić, Francesca Arcelli Fontana, Marco Zanoni, and Riccardo Roveda. 2017. Change Prediction through Coding Rules Violations. In Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering (Karlskrona, Sweden) (EASE’17). 61–64. https://doi.org/10.1145/3084226.3084282.

[439] Nikolaos Triantafilis, Ameya Ketkar, and Danny Díg. 2020. RefactoringMiner 2.0. IEEE Transactions on Software Engineering (2020), 21 pages. https://doi.org/10.1109/TSE.2020.3007722.

[440] Angeliki-Agathi Tsintzira, Elvira-Maria Arvanitou, Apostolos Ampatzoglou, and Alexander Chatzigeorgiou. 2020. Applying Machine Learning in Technical Debt Management: Future Opportunities and Challenges. In Quality of Information and Communications Technology, Martin Shepperd, Fernando Brito e Abreu, Alberto Rodrigues da Silva, and Ricardo Pérez-Castillo (Eds.). 53–67.

[441] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshvyanyk. 2019. On Learning Meaningful Code Changes Via Neural Machine Translation. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 25–36. https://doi.org/10.1109/ICSE.2019.00021.

[442] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshvyanyk. 2018. Deep Learning Similarities from Different Representations of Source Code (MSR ’18). 542–553. https://doi.org/10.1145/3196398.3196431.

[443] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshvyanyk. 2019. An Empirical Study on Learning Bug-Fixing Patterns in the Wild via Neural Machine Translation. ACM Trans. Softw. Eng. Methodol. 28, 4, Article 19 (September 2019), 29 pages. https://doi.org/10.1145/3340544.

[444] Sahithi Tummalapalli, Lov Kumar, NL Bhanu Murthy, and Aneesh Krishna. 2022. Detection of Web Service Anti-Patterns Using Weighted Extreme Learning Machine. Computer Standards & Interfaces (2022). 103621.

[445] Sahithi Tummalapalli, Lov Kumar, and N. L. Bhanu Murthy. 2020. 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 (Jabalpur, India) (ISEC’20). Article 8, 11 pages. https://doi.org/10.1145/3385032.3385042.

[446] Sahithi Tummalapalli, Lov Kumar, Lalitha Bhanu Murthy Neti, Vipul Kocher, and Srinivas Padmanabhan. 2021. A Novel Approach for the Detection of Web Service Anti-Patterns Using Word Embedding Techniques. In International Conference on Computational Science and Its Applications. Springer, 217–230.

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

[448] Sahithi Tummalapalli, Jihi Mittal, Lov Kumar, Lalitha Bhanu Murthy Neti, and Santanu Kumar Rath. 2021. 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. Springer, 263–276.

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

[450] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. 2019. Survey of machine learning techniques for malware analysis. Computers & Security 81 (2019), 123 – 147. https://doi.org/10.1016/j.cose.2018.11.001.

[451] S. Uchiyama, A. Kubo, H. Washizaki, and Y. Fukazawa. 2014. Detecting Design Patterns in Object-Oriented Program Source Code by Using Metrics and Machine Learning. Journal of Software Engineering and Applications 07 (2014), 983–998.

[452] Anderson Uchoa, Caio Barbosa, Daniel Coutinho, Willian Oizumi, Wesley KG Assunção, Silvia Regina Vergilio, Juliana Alves Pereira, Anderson Oliveira, and Alessandro Garcia. 2021. 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). IEEE, 471–482.

[453] Secil Ugurel, Robert Krovetz, and C. Lee Giles. 2002. 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 (Edmonton, Alberta, Canada) (KDD ’02). 652–658. https://doi.org/10.1145/775047.775141.

[454] M. Utting, B. Legeard, F. Dadeau, F. Tamagnan, and F. Bouquet. 2020. Identifying and Generating Missing Tests using Machine Learning on Execution Traces. In 2020 IEEE International Conference On Artificial Intelligence Testing (AITest). 83–90. https://doi.org/10.1109/AITEST49225.2020.9090717.

[455] Angeliki-Agathi Tsintzira, Elvira-Maria Arvanitou, Apostolos Ampatzoglou, and Alexander Chatzigeorgiou. 2020. Applying Machine Learning in Technical Debt Management: Future Opportunities and Challenges. In Quality of Information and Communications Technology, Martin Shepperd, Fernando Brito e Abreu, Alberto Rodrigues da Silva, and Ricardo Pérez-Castillo (Eds.). 53–67.
[457] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabhb Singh. 2019. Neural Program Repair by Jointly Learning to Localize and Repair.

[458] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, L ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In Advances in Neural Information Processing Systems, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dec91fd853c1e4a845aa-Paper.pdf

[459] B. A. Vishnu and K. P. Jevitha. 2014. Prediction of Cross-Site Scripting Attack Using Machine Learning Algorithms. In Proceedings of the 2014 International Conference on Interdisciplinary Advances in Applied Computing (Amritapuri, India) (ICONIAAC ’14). Association for Computing Machinery, New York, NY, USA, Article 55, 5 pages. https://doi.org/10.1145/2660859.2660969

[460] Nickolay Viniginov and Andrey Filchenkov. 2019. 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 (Tallinn, Estonia) (MaTuQaSoft 2019). 31–36. https://doi.org/10.1109/Ase.2019.00012

[461] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-Modal Attention Network Learning for Semantic Source Code Retrieval. In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (San Diego, California) (ASE ’19). 13–25. https://doi.org/10.1109/Ase.2019.00012

[462] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (Montpellier, France) (ASE 2018). 397–407. https://doi.org/10.1109/3238147.3238206

[463] Z. Wan, X. Xia, D. Lo, and G. C. Murphy. 2019. How does Machine Learning Change Software Development Practices? IEEE Transactions on Software Engineering (2019). 1–1. https://doi.org/10.1109/TSE.2019.2937083

[464] Deze Wang, Wei Dong, and Shanshan Li. 2020. 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 (Virtual, USA) (RL-SE&PL 2020). 1–2. https://doi.org/10.1145/3416506.3423575

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

[466] R. Wang, H. Zhang, G. Lu, L. Lyu, and C. Lyu. 2020. Fret: Functional Reinforced Transformer With BERT for Code Summarization. IEEE Access 8 (2020), 135591–135604. https://doi.org/10.1109/ACCESS.2020.3011744

[467] Shuai Wang, Jinyang Liu, Ye Qiu, Zhiyi Ma, Junfei Liu, and Zhonghai Wu. 2019. Deep Learning Based Code Completion Models for Programming Codes. In Proceedings of the 2019 3rd International Symposium on Computer Science and Intelligent Control (Amsterdam, Netherlands) (ISCSCIC 2019). Article 16, 9 pages. https://doi.org/10.3386/3389083

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

[469] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically Learning Semantic Features for Defect Prediction. In Proceedings of the 38th International Conference on Software Engineering Conference (Austin, Texas) (ICSE ’16). 297–308. https://doi.org/10.1109/ICSE.2016.7556082

[470] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao. 2019. 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). 1–12. https://doi.org/10.1109/ESEM.2019.8870172

[471] S. Wang and X. Yao. 2013. Using Class Imbalance Learning for Software Defect Prediction. IEEE Transactions on Reliability 62, 2 (2013), 434–443. https://doi.org/10.1109/TR.2013.2259203

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

[473] Wei Wang and Michael W. Godfrey. 2014. Recommending Clones for Refactoring Using Design, Context, and History. In 2014 IEEE International Conference on Software Maintenance and Evolution. 331–340. https://doi.org/10.1109/ICSME.2014.55

[474] Wenhuan Wang, Ge Li, Sijie Shen, Xin Xia, and Zhi Jin. 2020. Modular Tree Network for Source Code Representation Learning. ACM Trans. Softw. Eng. Methodol. 29, 4 (September 2020), 31 pages. https://doi.org/10.1145/3409331

[475] W. Wang, Y. Zhang, Y. Sui, Y. Wu, Z. Zhao, J. Wu, P. Yu, and G. Xu. 2020. Reinforcement-Learning-Guided Source Code Summarization via Hierarchical Attention. IEEE Transactions on Software Engineering (2020). 1–1. https://doi.org/10.1109/TSE.2020.2979701

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

[477] Xinda Wang, Shu Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2020. A Machine Learning Approach to Classify Security Patches into Vulnerability Types. In 2020 IEEE Conference on Communications and Network Security (CNS). 1–9. https://doi.org/10.1109/CNS48642.2020.9162337

[478] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhong Wang. 2020. Learning Semantic Program Embeddings with Graph Interval Neural Network. Proc. ACM Program. Lang. 4, OOPSLA, Article 137 (November 2020), 27 pages. https://doi.org/10.1145/3428205
[500] Awad A. Younis and Yashkwant K. Malaiya. 2014. Using Software Structure to Predict Vulnerability Exploitation Potential. In 2014 IEEE Eighth International Conference on Software Security and Reliability-Companion. 13–18. https://doi.org/10.1109/SERE-C.2014.17

[501] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler. 2018. Automatic Clone Recommendation for Refactoring Based on the Present and the Past. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). 115–126. https://doi.org/10.1109/ICSME.2018.00021

[502] Marco Zanoni, Francesca Arcelli Fontana, and Fabio Stella. 2015. On applying machine learning techniques for design pattern detection. Journal of Systems and Software 103 (2015), 102–117.

[503] Du Zhang and Jeffrey J. P. Tsai. 2003. Machine Learning and Software Engineering. Software Quality Journal 11, 2 (June 2003), 87–119. https://doi.org/10.1023/A:1023760326768

[504] Fanlong Zhang and Siau-cheng Khoo. 2021. An empirical study on clone consistency prediction based on machine learning. Information and Software Technology 136 (2021), 106575.

[505] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-Based Neural Source Code Summarization. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20). 1385–1397. https://doi.org/10.1145/3377811.3380383

[506] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 783–794. https://doi.org/10.1109/ICSE.2019.00086

[507] Jinglei Zhang, Rui Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. 2020. Exploiting Code Knowledge Graph for Bug Localization via Bi-Directional Attention. In Proceedings of the 28th International Conference on Program Comprehension (ICPC ’20). Association for Computing Machinery, 219–229. https://doi.org/10.1145/3387904.3389281

[508] Jie M. Zhang and Mark Harman. 2021. "Ignorance and Prejudice" in Software Fairness. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). 1436–1447. https://doi.org/10.1109/ICSE43992.2021.00129

[509] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. 2020. Machine Learning Testing: Survey, Landscapes and Horizons. IEEE Transactions on Software Engineering (2020), 1–1. https://doi.org/10.1109/TSE.2019.2962027

[510] Q. Zhang and B. Wu. 2020. Software Defect Prediction via Transformer. In 2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC). Vol. 1. 874–879. https://doi.org/10.1109/ITNEC48623.2020.9084745

[511] Yang Zhang and Chunhao Dong. 2022. MARS: Detecting brain class/method code smell based on metric–attention mechanism and residual network. Journal of Software: Evolution and Process (2021), e2403.

[512] Yu Zhang and Bingle Li. 2020. Malicious Code Detection Based on Code Semantic Features. IEEE Access 8 (2020), 176728–176737. https://doi.org/10.1109/ACCESS.2020.3026052

[513] Gang Zhao and Jeff Huang. 2018. 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 (Lake Buena Vista, FL, USA) (ESEC/FSE 2018). 141–151. https://doi.org/10.1145/3236024.3236068

[514] Wei Zheng, Jiali Gao, Xiaoxue Wu, Fengyu Liu, Yuxing Xiong, Guoliang Liu, and Xiang Chen. 2020. The impact factors on the J. M. Zhang, M. Harman, L. Ma, and Y. Liu. 2020. Machine Learning Testing: Survey, Landscapes and Horizons. IEEE Transactions on Software Engineering (2020), 1–1. https://doi.org/10.1109/TSE.2019.2962027

[515] Q. Zhang and B. Wu. 2020. Software Defect Prediction via Transformer. In 2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC). Vol. 1. 874–879. https://doi.org/10.1109/ITNEC48623.2020.9084745

[516] Yang Zhang and Chunhao Dong. 2022. MARS: Detecting brain class/method code smell based on metric–attention mechanism and residual network. Journal of Software: Evolution and Process (2021), e2403.

[517] Yu Zhang and Bingle Li. 2020. Malicious Code Detection Based on Code Semantic Features. IEEE Access 8 (2020), 176728–176737. https://doi.org/10.1109/ACCESS.2020.3026052

[518] Gang Zhao and Jeff Huang. 2018. 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 (Lake Buena Vista, FL, USA) (ESEC/FSE 2018). 141–151. https://doi.org/10.1145/3236024.3236068

[519] Wei Zheng, Jiali Gao, Xiaoxue Wu, Fengyu Liu, Yuxing Xiong, Guoliang Liu, and Xiang Chen. 2020. The impact factors on the J. M. Zhang, M. Harman, L. Ma, and Y. Liu. 2020. Machine Learning Testing: Survey, Landscapes and Horizons. IEEE Transactions on Software Engineering (2020), 1–1. https://doi.org/10.1109/TSE.2019.2962027

[520] Q. Zhang and B. Wu. 2020. Software Defect Prediction via Transformer. In 2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC). Vol. 1. 874–879. https://doi.org/10.1109/ITNEC48623.2020.9084745

[521] Yang Zhang and Chunhao Dong. 2022. MARS: Detecting brain class/method code smell based on metric–attention mechanism and residual network. Journal of Software: Evolution and Process (2021), e2403.
### Abbreviations of ML techniques

| Acronym | Full form |
|---------|-----------|
| A3C     | Asynchronous Advantage Actor-Critic |
| AB      | AdaBoost |
| AE      | Autoencoder |
| AIS     | Artificial Immune Systems |
| ANFIS   | Adaptive Neuro-Fuzzy Inference System |
| ANN     | Artificial Neural Network |
| ARM     | Association Rule Mining |
| B       | Bagging |
| BDT     | Boosted Decision Tree |
| BERT    | Bidirectional Encoder Representations from Transformers |
| Bi-GRU  | Bidirectional Gated Recurrent Unit |
| Bi-LSTM | Bi-Long Short-Term Memory |
| Bi-RNN  | Bidirectional Recurrent Neural Network |
| BiNN    | Bilateral Neural Network |
| BMN     | Best Matching Neighbours |
| BN      | Bayes Net |
| BNB     | Bernoulli Naive Bayes |
| BOW     | Bag of Words |
| BP-ANN  | Back-propagation Artificial Neural Network |
| BR      | Binary Relevance |
| CART    | Classification and Regression Trees |
| CC      | Classifier Chain |
| CCN     | Cascade Correlation Network |
| CNN     | Convolution Neural Network |
| COBWEB  | COBWEB |
| Code2Vec | Code2Vec |
| CoForest-RF | Co-Forest Random Forest |
| CSC     | Cost-Sensitive Classifier |
| DBN     | Deep Belief Network |
| DDQN    | Double Deep Q-Networks |
| DNN     | Deep Neural Network |
| Doc2Vec | Doc2Vec |
| DR      | Diverse Rank |
| DS      | Decision Stump |
| DT      | Decision Tree |
| EL      | Ensemble Learning |
| ELM     | Extreme Learning Machine |
| EM      | Expectation Minimization |
| EN-DE   | Encoder-Decoder |
| Acronym | Description |
|---------|-------------|
| FIS     | Fuzzy Inference System |
| FL      | Fuzzy Logic |
| FR-CNN  | Faster R-Convolutional Neural Network |
| GAN     | Generative Adversarial Network |
| GB      | Gradient Boosting |
| GBDT    | Gradient-Boosted Decision Tree |
| GBM     | Gradient Boosting Machine |
| GBT     | Gradient boosted trees |
| GCN     | Graph convolutional networks |
| GD      | Gradient Descent |
| GED     | Gaussian Encoder-Decoder |
| GEP     | Gene Expression Programming |
| GGNN    | Gated Graph Neural Network |
| GINN    | Graph Interval Neural Network |
| Glove   | Global Vectors for Word Representation |
| GNB     | Gaussian Naïve Bayes |
| GNN     | Graph Neural Network |
| GPT-C   | Generative Pre-trained Transformer for Code |
| GRASSHOPPER | Graph Random-walk with Absorbing StateS that HOPs among PEaks for Ranking |
| GRU     | Gated Recurrent Unit |
| HAN     | Hierarchical Attention Network |
| HC      | Hierarchical Clustering |
| HMM     | Hidden Markov Model |
| KM      | KMeans |
| KNN     | K Nearest Neighbours |
| KS      | Kstar |
| LB      | LogitBoost |
| LC      | Label Combination |
| LCM     | Log-bilinear Context Model |
| LDA     | Linear Discriminant Analysis |
| LLR     | Logistic Linear Regression |
| LMSR    | Least Median Square Regression |
| LOG     | Logistic regression |
| LR      | Linear Regression |
| LSTM    | Long Short Term Memory |
| MLP     | Multi Level Perceptron |
| MMR     | Maximal Marginal Relevance |
| MNB     | Multinomial Naïve Bayes |
| MNN     | Memory Neural Network |
| MTN     | Modular Tree-structured Recurrent Neural Network |
| MVE     | Majority Voting Ensemble |
| NB      | Naïve Bayes |
| NLM     | Neural Language Model |
| NMT     | Neural Machine Translation |
| Acronym | Description |
|---------|-------------|
| NNC    | Neural Network for Continuous goal |
| NND    | Neural Network for Discrete goal |
| Node2Vec | Node2Vec |
| OCC    | One Class Classifier |
| OR     | OneRule |
| PN     | Pointer Network |
| PNN    | Probabilistic Neural Network |
| POLY   | Polynomial regression |
| PR     | Pace Regression |
| PSO    | Particle Swarm Optimization |
| ReNN   | Reverse NN |
| ResNet | Residual Neural Network |
| RF     | Random Forrest |
| RGNN   | Regression Neural Network |
| Ripper | Ripper |
| RL     | Reinforcement Learning |
| RNN    | Recurrent Neural Network |
| RT     | RandomTree |
| SA     | Simulated Annealing |
| Seq2Seq | Sequence-to-Sequence |
| SMO    | Sequential Minimal Optimization |
| SMT    | Statistical Machine Translation |
| SOM    | Self Organizing Map |
| SVE    | Soft Voting Ensemble |
| SVLR   | Support Vector Logistic Regression |
| SVM    | Support Vector Machine |
| SVR    | Support Vector Regression |
| TF     | Transformer |
| TNB    | Transfer Naïve Bayes |
| V      | Voting |
| VSL    | Version Space Learning |
| Word2Vec | Word2Vec |
| XG     | XGBoost |