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
FEDERICA SARRO, University College London, UK

Context: 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 vulnerabilities detection. A large number of studies poses challenges to the community to understand the current landscape.
Objective: We aim to summarize the current knowledge in the area of applied machine learning for source code analysis.
Method: We investigate 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 carried out an extensive literature search and identified 364 primary studies published between 2002 and 2021. We summarize our observations and findings with the help of the identified studies.
Results: Our findings suggest that the usage 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 the employed machine learning techniques. Additionally, we collate a comprehensive list of available datasets and tools useable in this context. Finally, we summarize the perceived challenges in this area that include 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, and Federica Sarro. 2021. A Survey on Machine Learning Techniques for Source Code Analysis. *ACM Trans. Softw. Eng. Methodol.* 0, 0, Article 0 (2021), 59 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 [162, 308], text [2, 178], and speech [115, 116, 267] processing. These advancements, coupled with a large amount of open-source code and associated artifacts, as well as the availability of accelerated hardware, have facilitated the use of ML and DL techniques to address software engineering problems [332, 358].

The software engineering community has employed ML and DL techniques for a variety of applications such as software testing [185, 236, 363], source code representation [13, 133], source code quality analysis [25, 35], program synthesis [170, 348], code completion [192], refactoring [30], code summarization [12, 175, 196], and vulnerability analysis [278, 287, 322] 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

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; Federica Sarro, f.sarro@ucl.ac.uk, University College London, London, UK.

2021. 1049-331X/2021/0-ART0 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

as tools and datasets, is also increasing. This poses a challenge, to both research and practice communities, to comprehend the landscape of the available resources and infer the potential directions that the field is taking.

Recently, there have been some attempts to summarize the application-specific knowledge in the form of surveys. For example, Allamanis et al. [13] present key methods to model source code using ML techniques. Shen and Chen [287] provide a summary of research methods associated with software vulnerability detection, software program repair, and software defect prediction. Durelli et al. [93] collect 48 primary studies focusing on software testing using machine learning. Alsolai and Roper [25] present a systematic review of 56 studies related to maintainability prediction using ML techniques. Recent surveys [9, 35, 317] summarize application of ML techniques on software code smells and technical debt identification. Similarly, literature reviews on program synthesis [170] and code summarization [227] have been attempted.

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 paper is an attempt to fill this gap.

The goal of this study is to summarize the current knowledge of 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 similar studies. Additionally, we aim to identify challenges in the 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, but also to focus their efforts on tackling the identified challenges.

This survey makes the following contributions to the field:

- It presents a summary of the applied machine learning studies attempted in the domain of source code analysis.
- 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 8, and conclude the paper in Section 9.

2 Methods

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 and questions

The study aims to provide a consolidated yet extensive overview of the source code analysis field. The survey covers aspects such as software engineering tasks and applications (*e.g.,* program comprehension and vulnerability analysis) for which ML techniques are employed, their classification, and various datasets and tools that are made available. We define the following research questions to explore in this study:

RQ1 *What specific tasks that involve source code analysis have been attempted using machine learning?*
In this research question, we would like to investigate different types of code analysis tasks that have been attempted using ML techniques. We would like to summarize how ML methods and techniques are helping in solving specific software engineering tasks.

RQ2 *What are the machine learning techniques used?*
This research question 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.

RQ3 *What datasets and tools are available?*
With this research question, we aim to provide a consolidated summary of available datasets and tools along with their purpose.

RQ4 *What are the challenges and perceived deficiencies?*
In this research question, we 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 and filtered the relevant studies in three phases. Figure 1 summarizes the search process. We elaborate on each of these phases in the rest of the section.

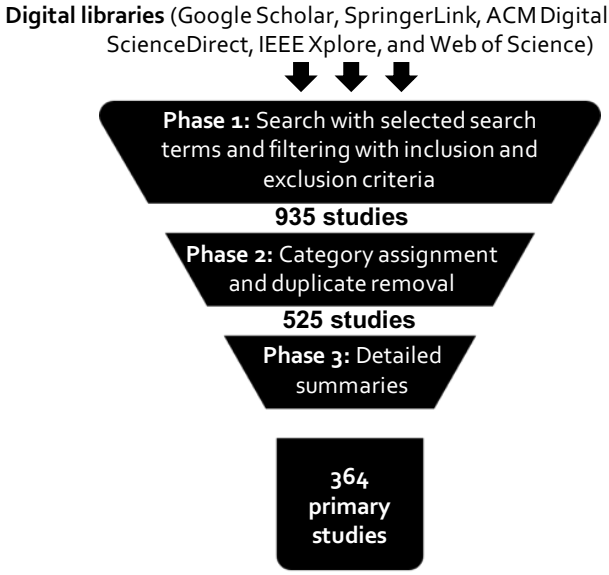


Fig. 1. Overview of the search process

2.3 Literature search — Phase 1

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 Nov-Dec 2020. 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. Next, we defined inclusion and exclusion criteria to filter out irrelevant studies.

2.3.1 *Inclusion criteria*

- Studies that discuss source code analysis using any of the ML techniques (including DL).
- Surveys discussing source code analysis using ML techniques.
- Resources revealing the deficiencies or challenges in the present set of methods, tools, and practices.

2.3.2 *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 containing 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 in any other language than English.

During the search, we documented studies that satisfy our search protocol in a spreadsheet including 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[1] to encourage extensions of our survey. 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 935 articles after completing *Phase 1* of the search process.

2.4 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 [13, 35, 101, 287]. 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 fitting in our inclusion criteria after reading their title and abstract. After this phase, we were left with 508 studies.

2.5 Literature search — Phase 3

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 study. The remaining 364 articles are the primary studies that we examine in detail. For each study, we extracted the

[1] https://github.com/tushartushar/ML4SCA/tree/main/replication

core idea and contribution, used ML techniques and tools as well as mentioned challenges and findings. Next, we present our observations corresponding to each considered research question.

3 RQ1. What specific tasks that involve source code analysis have been attempted using machine learning?

We tagged each selected article with one of the task categories based on the primary focus of the study. The categories represent the 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*. Three studies do not fall in these categories but are still relevant for our discussion (such as survey papers offering general discussion on the topic); we put such studies in the *general* category. Figure 2 presents a category-wise distribution of studies per year. It is evident that the topic in scope is attracting the research community and we observe a constant upward trend especially from year 2015.

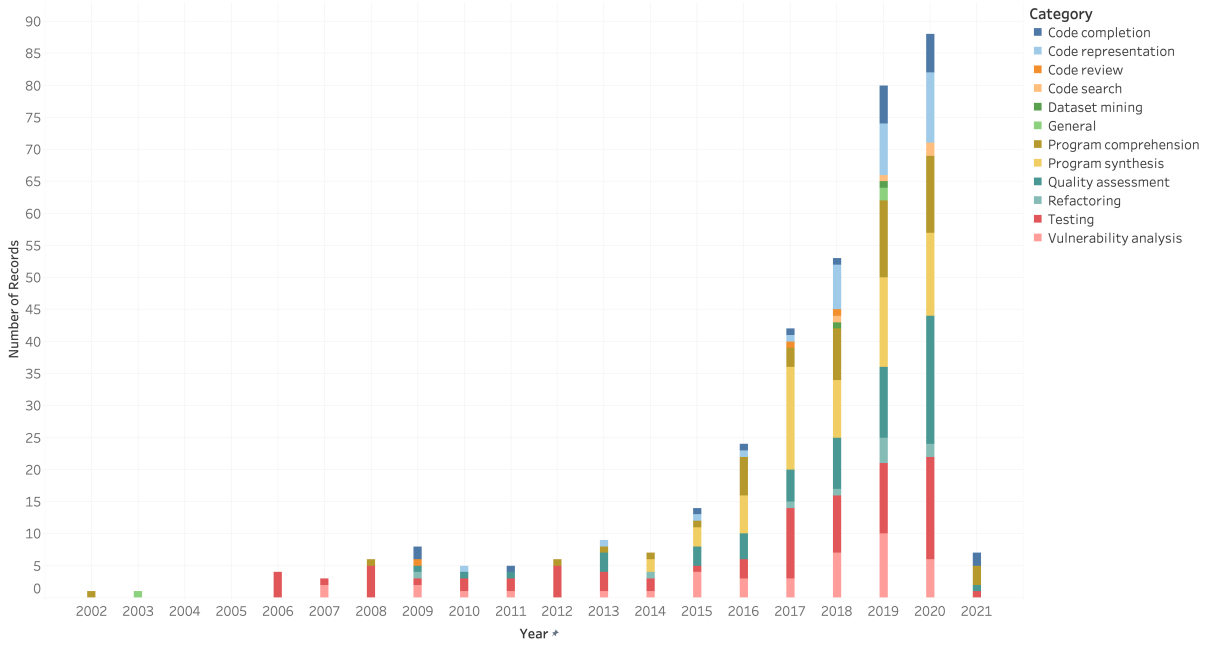


Fig. 2. Category-wise distribution of studies

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*, *effort prediction*, and *test data/case generation*. We assigned sub-categories to the studies wherever applicable; if none of the sub-categories are 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. *Refactoring* is a category in itself but also included as a sub-category of the *program synthesis* category. Studies that propose or identify refactoring operations to be applied using ML are included in the *refactoring* category; whereas the studies that generate refactored code are kept in the sub-category of *program synthesis* category.

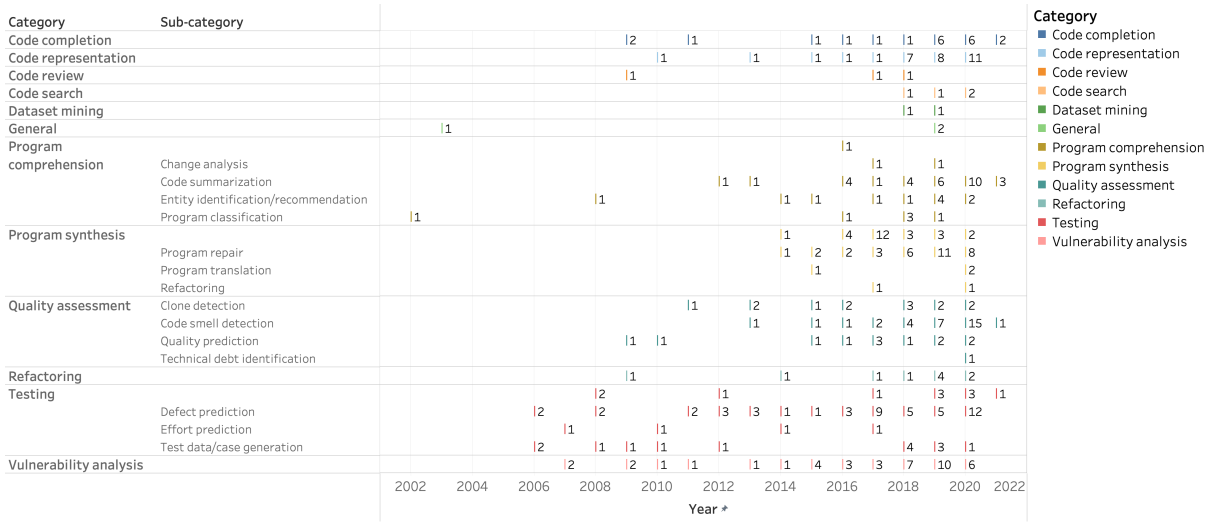


Fig. 3. Category- and sub-categories-wise distribution of studies

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 2002 and 2020. We observed that *quality assessment* and *testing* categories grew most aggressively with approximately 118% and 94% average growth each year, respectively. At sub-category level, *code smell detection* and *defect prediction* grew the most with approximately 88% and 70% average growth rate each year.

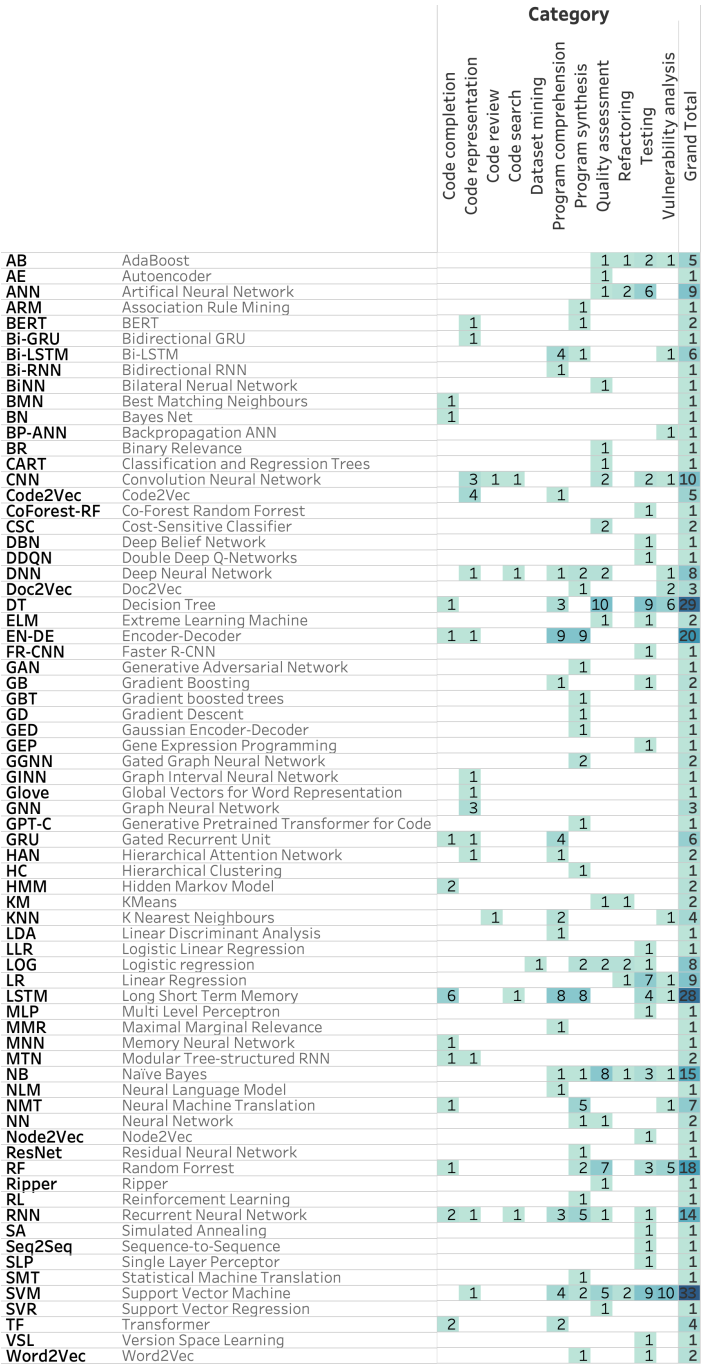
4 RQ2. What are the machine learning techniques used?

We document our observations per category and subcategory by providing a summary of the existing efforts. Figure 4 shows the frequency of various ML techniques per category used in the primary studies. Nevertheless, the figure illustrates commonly used acronyms for ML techniques along with corresponding expanded form; we utilize these acronyms throughout the paper. It is evident from the figure that SVM and DT are the most frequently employed ML techniques. From the DL side, 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 to emphasize that we structure the discussion around the crucial steps *e.g.,* model generation, data sampling, feature extraction, and model training for each category.

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 [13, 230]; the proposed techniques extensively utilize the syntax, structure, and semantics (such as type information from 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 [225, 312], method name prediction [23], and comment classification [333].



AB	AdaBoost						1	1	2	1	5
AE	Autoencoder						1				1
ANN	Artifical Neural Network						1	2	6		9
ARM	Association Rule Mining					1					1
BERT	BERT		1			1					2
Bi-GRU	Bidirectional GRU		1								1
Bi-LSTM	Bi-LSTM				4	1				1	6
Bi-RNN	Bidirectional RNN				1						1
BiNN	Bilateral Nerual Network						1				1
BMN	Best Matching Neighbours	1									1
BN	Bayes Net	1									1
BP-ANN	Backpropagation ANN									1	1
BR	Binary Relevance						1				1
CART	Classification and Regression Trees						1				1
CNN	Convolution Neural Network		3	1	1		2		2	1	10
Code2Vec	Code2Vec		4			1					5
CoForest-RF	Co-Forest Random Forrest								1		1
CSC	Cost-Sensitive Classifier						2				2
DBN	Deep Belief Network								1		1
DDQN	Double Deep Q-Networks								1		1
DNN	Deep Neural Network		1		1	1	2	2		1	8
Doc2Vec	Doc2Vec					1				2	3
DT	Decision Tree	1				3		10		9	6	29
ELM	Extreme Learning Machine						1		1		2
EN-DE	Encoder-Decoder	1	1			9	9					20
FR-CNN	Faster R-CNN								1		1
GAN	Generative Adversarial Network					1					1
GB	Gradient Boosting					1			1		2
GBT	Gradient boosted trees						1				1
GD	Gradient Descent						1				1
GED	Gaussian Encoder-Decoder						1				1
GEP	Gene Expression Programming								1		1
GGNN	Gated Graph Neural Network						2				2
GINN	Graph Interval Neural Network		1								1
Glove	Global Vectors for Word Representation		1								1
GNN	Graph Neural Network		3								3
GPT-C	Generative Pretrained Transformer for Code						1				1
GRU	Gated Recurrent Unit	1	1			4					6
HAN	Hierarchical Attention Network		1			1					2
HC	Hierarchical Clustering						1				1
HMM	Hidden Markov Model	2									2
KM	KMeans							1	1		2
KNN	K Nearest Neighbours			1		2				1	4
LDA	Linear Discriminant Analysis					1					1
LLR	Logistic Linear Regression								1		1
LOG	Logistic regression				1		2	2	2	1		8
LR	Linear Regression							1	7	1	9
LSTM	Long Short Term Memory	6			1	8	8			4	1	28
MLP	Multi Level Perceptron								1		1
MMR	Maximal Marginal Relevance					1					1
MNN	Memory Neural Network	1									1
MTN	Modular Tree-structured RNN	1	1								2
NB	Naïve Bayes					1	1	8	1	3	1	15
NLM	Neural Language Model					1					1
NMT	Neural Machine Translation	1					5			1	7
NN	Neural Network						1	1			2
Node2Vec	Node2Vec								1		1
ResNet	Residual Neural Network						1				1
RF	Random Forrest	1					2	7		3	5	18
Ripper	Ripper							1				1
RL	Reinforcement Learning						1					1
RNN	Recurrent Neural Network	2	1		1	3	5	1		1		14
SA	Simulated Annealing								1			1
Seq2Seq	Sequence-to-Sequence								1			1
SLP	Single Layer Perceptor								1			1
SMT	Statistical Machine Translation						1					1
SVM	Support Vector Machine		1			4	2	5	2	9	10	33
SVR	Support Vector Regression							1				1
TF	Transformer	2				2						4
VSL	Version Space Learning								1			1
Word2Vec	Word2Vec						1		1			2

Fig. 4. Usage of ML techniques in all the primary studies

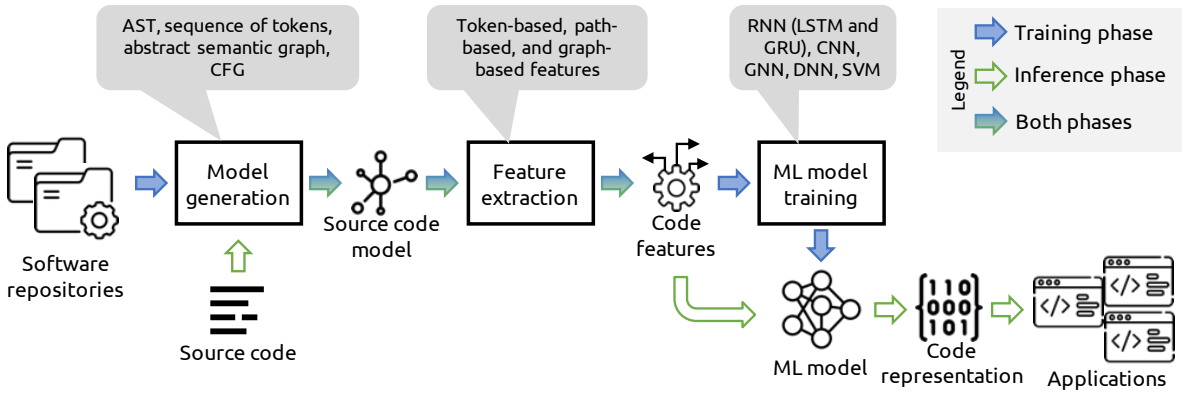


Fig. 5. Overview of the code representation process pipeline

Figure 5 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 [19, 21–23, 54, 69, 85, 230, 333, 339, 348, 360]. Some studies [17, 34, 286] parsed source code as tokens and prepared a sequence of tokens in this step. Hoang et al. [135] generated tokens representing only the code changes. Furthermore, Sui et al. [302] compiled a program into LLVM-IR. The inter-procedural value-flow graph (IVFG) is built on top of the intermediate representation. Thaller et al. [312] used abstract semantic graph as their code model. Finally, Brauckmann et al. [53] and Tufano et al. [319] 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. [230] prepared vectors of syntactic context (referred to as *syntaxeme*), type context (*sememes*), and lexical tokens. Shedko et al. [286] generated a stream of tokens corresponding to function calls and control flow expressions. Karampatsis et al. [152] 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. [22] used paths between AST nodes. Kovalenko et al. [161] extracted path context representing two tokens in code and a structural connection along with paths between AST nodes. Alon et al. [21] 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. [23] 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. [69] created AST node identified by an API name and attached each node to the corresponding AST node belonging to the identifier. Thaller et al. [312] proposed feature maps; feature maps are

human-interpretation, stacked, named subtrees extracted from abstract semantic graph. Brauckmann et al. [53] created a dataflow-enriched AST graph, where nodes are labeled as declarations, statements, and types as found in the Clang[2] AST. Cvitkovic et al. [85] augmented AST with semantic information by adding a graph-structured vocabulary cache. Finally, Zhang et al. [360] 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 to learn features automatically [135, 319].

ML model training: The majority of the studies rely on the RNN-based DL model. Among them, some of the studies [21, 53, 133, 333, 339] employed LSTM-based models; while others [54, 135, 152, 348, 360] used GRU-based models. Among the other kinds of ML models, studies employed GNN-based [85, 341], DNN [230], conditional random fields [22], SVM [184, 253], and CNN-based models [69, 225, 312]. Some of the studies rely on the combination of different DL models. For example, Tufano et al. [319] 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.

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 6 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 [46, 59, 63, 64, 66, 68, 78, 86, 89, 90, 92, 108, 155, 181, 204, 208, 211, 248, 272, 289, 294–296, 336, 338, 364] used the PROMISE dataset [273]. Xiao et al. [344] utilized a Continuous Integration (CI) dataset and Pradel and Sen [249] generated a synthetic dataset. Apart from using the existing datasets, some other studies prepared their own datasets by utilizing various GitHub projects [4, 131, 209, 210, 296] including Apache [51, 87, 183], Eclipse [87, 369] and Mozilla [159, 206] projects, or industrial data[51].

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

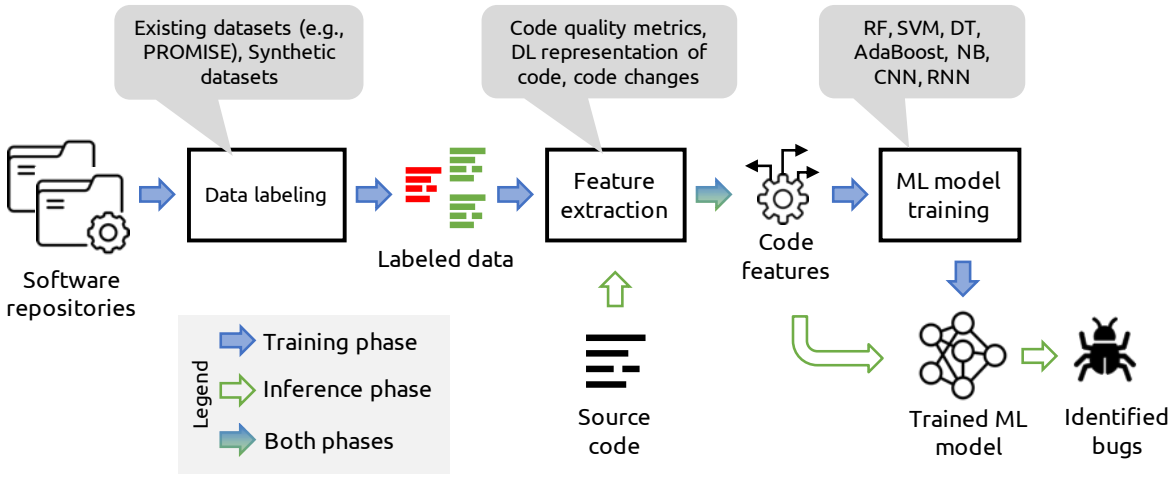


Fig. 6. Overview of the defect prediction process pipeline

Feature extraction: The most common features to train a defect prediction model are the source code metrics introduced by Halstead [126], Chidamber and Kemerer [76], and McCabe [213]. Most of the examined studies [59, 63, 66, 67, 78, 108, 155, 159, 204, 208–211, 276, 295, 338] used a large number of metrics such as Lines of Code, Number of Children, Coupling Between Objects, and Cyclomatic Complexity. In addition to the above, some authors [46, 64, 89, 248] suggested the use of dimensional space reduction techniques—such as Principal Component Analysis (PCA)—to limit the number of features. Pandey and Gupta [237] used Sequential Forward Search (SFS) to extract relevant source code metrics. Dos Santos et al. [92] suggested a sampling-based approach to extract source code metrics to train defect prediction models. Kaur et al. [154] suggested an approach to fetch entropy of change metrics. Bowes et al. [51] introduced a novel set of metrics constructed in terms of mutants and the test cases that cover and detect them.

Other authors [249, 364] used word embeddings as features to train models, while Li et al. [183] suggested an approach that constructs the AST for a selected method and extracts the paths along with AST's nodes and uses Word2Vec to feed them as vectors a model. Singh et al. [296] 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. [86]. 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. [336] employed the Deep Belief Network algorithm (DBN) to learn semantic features from token vectors, which are fetched from applications' ASTs. Shi et al. [289] used a DNN model to automate the features extraction from the source code. Xiao et al. [344] 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 [206] and Aggarwal [4] 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. [68] 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 [328]. 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 [46, 59, 63, 64, 66, 78, 89, 92, 108, 127, 154, 155, 208–211, 237, 248, 294–296, 336] used traditional ML algorithms such as *Decision Tree*, *Random Forest*, *Support Vector Machine*, and *AdaBoost*. In addition, authors [181, 204, 338] proposed changes to traditional ML algorithms to train their models. Specifically, Wang and Yao [338] suggested a dynamic version of *AdaBoost.NC* that adjusts its parameters automatically during training. Similarly, Li et al. [181] proposed ACoForest, an active semi-supervised learning method to sample the most useful modules to train defect prediction models. Ma et al. [204] 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 [68, 86, 183, 249, 276] used DL models such as CNN and RNN-based models for defect prediction. Moreover, by using DL approaches, authors achieved improved accuracy for defect prediction and they pointed out bugs in real-world applications [183, 249].

4.2.2 *Testing effort prediction*

In this section, we analyze studies that use ML methods to predict the required testing effort of a software system. Figure 7 provides an overview of predicting testing effort using ML techniques. As a first step of using ML techniques, a labeled dataset is required. Next, relevant features are extracted that, in turn, are fed to the ML models. The results of the process is the predicted effort required to test a software.

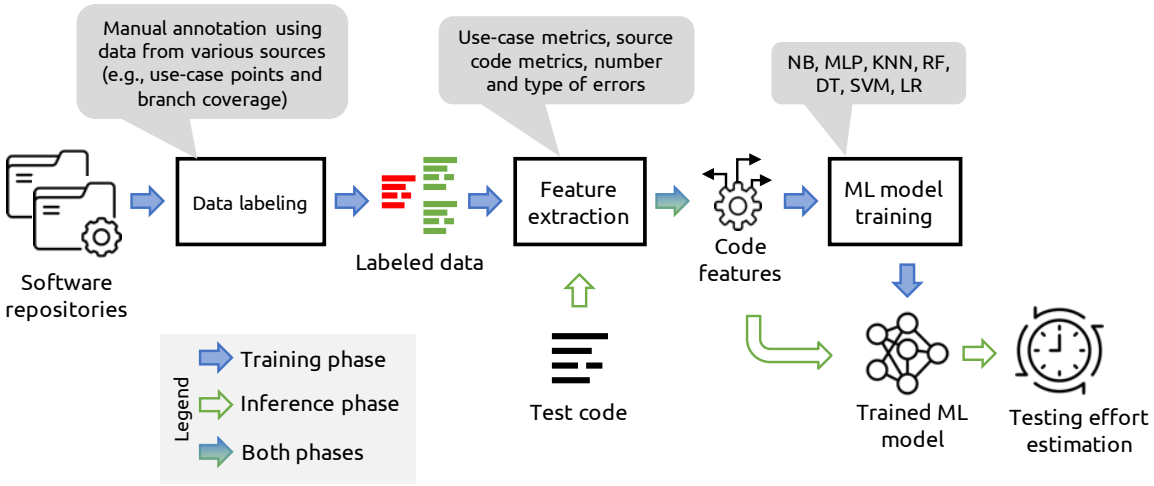


Fig. 7. Overview of the test effort prediction process pipeline

Data labeling: Badri et al. [37] used an approach to estimate the time required to test a software based on its use-cases in order to label their dataset. They chose five Java projects (*i.e.,* ATM, NextGen, CommonsExec, CommonsEmail, and CommonsIO) to prepare their dataset. Similarly, Tripathi and Sharma [316] utilized XML-security, a Java-based Apache project and label the dataset with change statistics such as change proneness. Grano et al. [114] used Apache Cassandra, Apache Ivy, Google Guava, and Google Dagger as their subject systems and collected branch coverage information, which they used to label their dataset. E. Silva et al. [94] employed two database projects from HARPIA in order to develop models for estimating the test suite execution time. They have manually labeled their dataset based on the time needed to execute test cases.

Feature extraction: From the above-mentioned datasets, researchers extracted relevant features to train their models. Particularly, Badri et al. [37] used use-case metrics such as the number of involved classes, external operations, and test scenarios. Some studies [94, 114, 316] employed source code metrics as features. Kumar and Singh [167] used the number of critical, major, and minor errors from a publicly available failure dataset as features.

ML model training : To train their ML models with the selected datasets and features, researchers used various algorithms. Most of them [37, 94, 114, 167] employed *Naive Bayes*, *Multilayer Perceptron*, *K Nearest Neighbors*, *Random Forest*, *Decision Tree*, *Support Vector Machine*, and *Linear Regression*. Two studies [94, 114] concluded that *Support Vector Machine* offers the best results for testing effort prediction. Tripathi and Sharma [316] pointed out that *AdaBoost* and *Random Forest* are the best performing algorithms in this context.

4.2.3 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. [52] recorded traces for applications to capture usage data. They sanitized any irrelevant information collected from the programs recording components. AppFlow [138] captures human-event sequences from a smart-phone screen in order to identify tests. Similarly, Nguyen et al. [231] suggested Shinobi, a framework that uses a fast R-CNN model to identify input data fields from multiple web-sites. Utting et al. [325] captured user and system execution traces to help generating missing API tests. To automatically identify metamorphic relations, Nair et al. [226] 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. Takagi et al. [309] manually classified the data to automatically classify the difference of actual and expected output.

Feature extraction: Function CFG is the feature extracted by the majority of the studies [43, 44, 130, 150, 151] in the sub-category. They aimed to train models for predicting the metamorphic relations for testing. Some authors [52, 325] used execution traces as features. Kim et al. [157] suggested an approach that replaces SBST's meta-heuristic algorithms with deep reinforcement learning to generate test cases based on branch coverage information.

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 [43, 44, 52, 130, 150, 151, 157, 226, 309, 325] used traditional ML algorithms, such as *Support Vector Machine*, *Naive Bayes*, *Decision Tree*, *Multilayer Perceptron*, *Random Forest*, *AdaBoost*, *Linear Regression*. Nguyen et al. [231] used the DL algorithm Fast R-CNN.

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

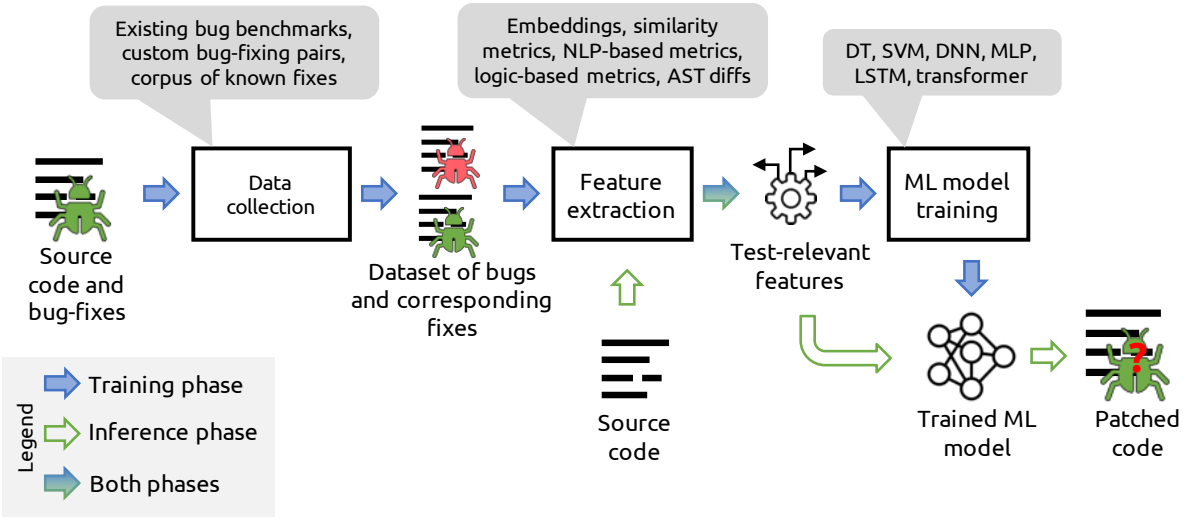


Fig. 8. Overview of the program repair process pipeline.

with a little or without human intervention [112]. 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[3] of that patch follows.

According to Goues et al. [112], 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 pre-processing step, such methods take source code of the buggy revision as an input, and those revisions that fixes the buggy revision. The revision with the fixes includes a patch carried out manually that corrects the buggy revision and a test case that checks whether the bug has been fixed. Learning-aided-based repair is mainly based on the hypothesis that similar bugs will have similar fixes. Therefore, during the training phase, such techniques can use features such as similarity metrics to match bug patterns to similar fixes. Then, the generated patches rely on those learnt patterns. Next, we elaborate upon the individual steps involved in the process of program repair using ML techniques.

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. [320] extracted millions of bug-fixing pairs from GitHub, Amorim et al. [29] leveraged the naturalness obtained from a corpus of known fixes, and Chen et al. [73] used natural language structures from source code. Furthermore, many studies develop their own large-scale bug benchmarks. Ahmed et al. [7] leveraged 4,500 erroneous C programs, Gopinath et al. [110] used a suite of programs and datasets stemmed from real-world applications, and Long and Rinard [197] used a set of successful manual patches from open-source software repositories. Le et al. [172]

[3]The term semantic correctness is a criterion for evaluating whether a generated patch is similar to the human fix for a given bug [194].

created an oracle for predicting which bugs should be delegated to developers for fixing and which should be fixed by repair tools.

Other studies use existing bug benchmarks, such as Defects4J [149] and IntroClass [173], which already include buggy revisions and human fixes, to evaluate their approaches. For instance, Saha et al. [266], Wang et al. [337], and Chen et al. [74] leveraged Defects4J for the evaluations of their approaches. Additionally, Dantas et al. [88] used the IntroClass benchmark and Majd et al. [207] conducted experiments using 119,989 C/C++ programs within Code4Bench. Wu et al. [343] used the DeepFix dataset that contains 46,500 correct C programs and 6,975 programs with errors for their graph-based DL approach for syntax error correction.

Some studies examine bugs in different programming languages. For instance, Svyatkovskiy et al. [306] used 1.2 billion lines of source code in Python, C#, JavaScript, and TypeScript programming languages. Also, Lutellier et al. [202] 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. [122] used 6,975 erroneous C programs with typographic errors, Santos et al. [270] used source code files with syntax errors, and Sakkas et al. [268] used a corpus of 4,500 ill-typed OCaml programs that lead to compile-time errors. Bhatia et al. [47] 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. [47], Gupta et al. [122], and Sakkas et al. [268] 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. [29], Santos et al. [270], Svyatkovskiy et al. [306], and Chen et al. [73], leveraged source-code naturalness and applied NLP-based metrics. Tian et al. [314] employed different representation learning approaches for code changes to derive embeddings for similarity computations. Ahmed et al. [7] used similar metrics for fixing compile-time errors. Additionally, Saha et al. [266] 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. [337] 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. [268] also applied similarity metrics.

There are several approaches that use logic-based metrics based on the relationships of the features used. Specifically, Van Thuy et al. [326] 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 [197] more efficient and precise. Alrajeh et al. [24] identified counterexamples and witness traces using model checking for logic-based learning to perform repair process automatically. Cai et al. [60] 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.

Many studies also extract and consider the context where the bugs are related to. For instance, Tufano et al. [320] 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 [98]. Additionally, Soto and Le Goues [297] 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. [217] 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. [182] used the prior bug fixes and the surrounding code contexts of the fixes for code transformation learning. Saha et al. [265] 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. Finally, Bader et al. [36] 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. [327] used results from localization of variable-misuse bugs. Wu et al. [343] developed an approach, GGF, for syntax-error correction that treats the code as a mixture of the token sequences and graphs.

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

Neural Machine Translation: This category includes papers that apply neural machine translation (NMT) for enhancing automated program repair. Such approaches can, for instance, include techniques that use examples of bug fixing for one programming language to fix similar bugs for other programming language. Lutellier et al. [202] 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. [320] used NMT techniques (Encoder-Decoder model) for learning bug-fixing patches for real defects, and generated repair patches. Mesbah et al. [217] introduced DeepDelta, which used NMT for learning to repair compilation errors.

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. [314] 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. [29] 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. [326] 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 [197]. Gupta et al. [122] used word embeddings and reinforcement learning to fix erroneous C student programs with typographic errors. Tian et al. [314] 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. [266] used similarity analysis for repairing bugs that may require applying a substantially similar patch at a number of locations. Additionally, Wang et al. [337] used also similarity metrics to compare the differences among machine-generated and human patches. Santos et al. [270] used n-grams and NNs to detect and correct syntax errors.

Logic-based rules: Alrajeh et al. [24] combined model checking and logic-based learning to support automated program repair. Cai et al. [60] also combined model-checking and ML for program repair. Shim et al. [291] 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. [268] 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 [197] 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 [297] 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. [110] 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. [265] developed ELIXIR repair approach that uses *Logistic Regression* models and similarity-score metrics. Bader et al. [36] 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. [346] 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. [111] used *Support Vector Machine* and ID3 with path exploration to repair bugs in complex data structures. Le et al. [172] 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.

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. [88] combined DOC2VEC and LSTM, to capture dependencies between source code statements, and improve the fault-localization step of program repair. Ahmed et al. [7] developed a repair approach (TRACER) for fixing compilation errors using RNNs. Recently, Li et al. [182] 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. [306] 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. [73] used the LSTM for synthesizing `if-then` constructs. Similarly, Vasic et al. [327] applied the LSTM in multi-headed pointer networks for jointly learning to localize and repair variable misuse bugs. Bhatia et al. [47] combined neural networks, and in particular RNNs, with constraint-based reasoning to repair syntax errors in buggy programs. Chen et al. [74] applied LSTM for sequence-to-sequence learning achieving end-to-end program repair through the SEQUENCER repair tool they developed. Majd et al. [207] developed SLDEEP, statement-level software defect prediction, which uses LSTM on static code features.

4.3.2 Refactoring

Here, we include related work that uses ML techniques for source-code refactoring. Stein et al. [300] used *Random Forest* for identifying valid source-code transformations via the use of paraphrases. The dataset used for the training is comprised of 27,300 C++ source code files, consisting of 273 topics each with 10 files that solved the same challenge question. This generates approximately 152,000 paraphrases. From these paraphrases, 11% produce valid code transformations. Tamarit et al. [310] applied a program transformation approach to convert procedural code into functionally equivalent code adapted to a given platform, by using classification trees and reinforcement learning. The approach learns from successful transformation sequences, and, then, it produces encodings of strategies. The approach has been evaluated by using a series of benchmarks, adapting standard C code to GPU execution via OPENCL.

4.3.3 Program translation

In this section, we list studies that use ML that can be used, for instance, for translating source code from one programming language to another by learning source-code patterns. Le et al. [171] presented a survey on DL techniques including machine translation algorithms and applications. Chakraborty et al. [65] 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. [234] 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 [285]. Extensive literature is available on detecting smells automatically [285]; 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.

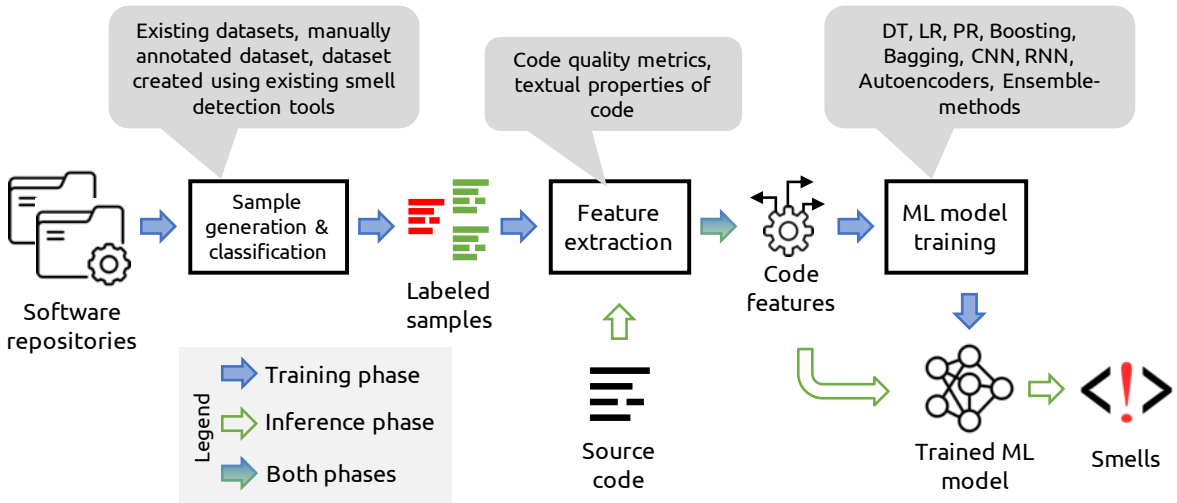


Fig. 9. Overview of the code smell detection process pipeline

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. [97] developed a manually validated oracle containing 1,700 instances of linguistic smells. Pecorelli et al. [241] created a dataset of 8.5 thousand samples of smells from 13 open-source projects. Some authors [8, 218] employed existing datasets (such as Landfill) in their studies. Oliveira et al. [235] relied on historical data and mined smell instances from history where the smells are refactored. Some efforts such as one by Sharma et al. [283] used CodeSplit [281, 282] first to split source code files into individual classes

and methods. Then, they used existing smell detection tools [280, 284] to identify smells in the subject systems. They used the output of both of these tasks to identify and segregate positive and negative samples.

Liu et al. [193] adopted an usual mechanism to identify their positive and negative samples. They assumed that popular well-known open-source projects are well-written and hence all of the classes/methods of these projects are by default considered free from smells. To obtain positive samples, they carried out *reverse refactoring e.g.,* moving a method from a class to another class to create an instance of feature envy smell.

Feature extraction: The majority of the articles [5, 28, 31, 41, 82, 84, 102, 103, 121, 124, 153, 166, 193, 218, 235, 251, 313] 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 [313] and Agnihotri and Chug [5] 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. [28] employed 62, Mhawish and Gupta [218] utilized 82, and Arcelli Fontana and Zanoni [31] 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. [200] used warnings generated from existing static analysis tools as features. Similarly, Ochodek et al. [233] 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). Furthermore, Sharma et al. [283] 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 [5, 82, 84, 91, 102, 103, 121, 166, 200, 235, 242, 251, 313] in this category compared the performance of various ML models. Some of the authors experimented with individual ML models; for example, Kaur et al. [153] and Amorim et al. [28] 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. [41] and Tummalapalli et al. [321] experimented with ensemble techniques such as *majority training ensemble* and *best training ensemble*.

DL-based models: Studies that use DL form the third category. Sharma et al. [283] used CNN, RNN (LSTM), and Autoencoders-based DL models. Hadj-Kacem and Bouassida [124] 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. Furthermore, Liu et al. [193] 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.

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* [5, 31, 41, 61, 82, 103, 117, 121, 124, 153, 200, 235], *feature envy* [5, 31, 41, 82, 102, 103, 124, 153, 283], *long method* [31, 35, 82, 102, 103, 117, 121, 124, 153], *data class* [31, 102, 103, 117, 153, 235], and *complex class* [121, 200, 235]. 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* [9, 35, 102, 121] and *Random Forest* [31, 35, 102, 166, 218] perform the best. Other methods that have been reported better than other ML models in their respective studies are *Support Vector Machine* [321], *Boosting* [199], and *Autoencoders* [283].

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.

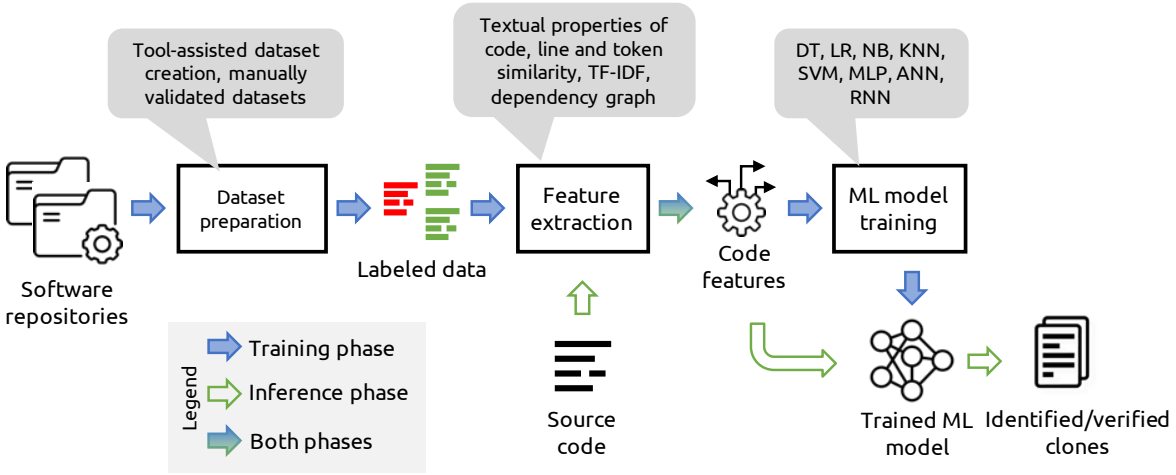


Fig. 10. Overview of the code clone detection/validation process pipeline

Dataset preparation: Manual annotation is a common way to prepare a dataset for applying ML to identify code clones [221, 224, 342]. Mostaeen et al. [224] 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. [349] used existing code clone detection tools to generate their training set. Some authors (such as Bandara and Wijayarathna [38]) relied on existing code-clone datasets. Bui et al. [57] 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 [38] identified features such as the number of characters and words, identifier count, identifier character count, and underscore count using ANTLR tool. Some studies [221, 223, 224] utilized line similarity and token similarity. Yang et al. [349] computed TF-IDF along with other metrics such as position of clones in the file. Cesare et al. [62] 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. Similarly, Sheneamer and Kalita [288] 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 [365] used CFG and DFG (Data Flow Graph) for clone detection. Some of the studies [57, 99, 342] 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 [38, 221, 223, 288] experimented with a number of ML approaches. For example, Mostaeen et al. [221] used *Bayes Network*, *Logistic Regression*, and *Decision Tree*; Bandara and Wijayarathna [38] employed *Naive Bayes*, *K Nearest Neighbors*, *AdaBoost*. Similarly, Sheneamer and Kalita [288] 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 [223, 224], DNN [99, 365], and RNN with *Reverse neural network* [342] are also employed extensively. Bui et al. [58] and Bui et al. [57] combined neural networks for ML models training. Specifically, Bui et al. [58] 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. [57] constructed BiTBCNNs—a combination layer of sub-networks to encode similarities and differences among code structures in different languages.

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. [134] 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. [259] 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. [11] used the Understand tool to detect various metrics, and employed them on the Juliet test suite for C++.

Feature extraction: Heo et al. [134] 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. [259] generated features only from the warnings (such as redundancy level and number of warnings in the same file). Some studies [11, 158] used source code metrics as features.

ML model training: Kim et al. [158] used *Support Vector Machine* to identify risky modules from a software system. Alikhashashneh et al. [11] 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. The study by Ribeiro et al. [259] claimed that ensemble methods such as *AdaBoost* works superior than standalone ML methods. Anomaly-detection techniques such as *One-class Support Vector Machine* have been used by Heo et al. [134]. They applied their method on taint analysis and buffer overflow detection to improve the recall of static analysis. Whereas, some other studies [11, 259] aimed to rank and classify static analysis warnings. Kim et al. [158] 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 [55]. The latest generation of auto-completion methods uses NLP and advanced ML models, trained on publicly available software repositories, to suggest source-code completions, given the current context of the software-projects under examination.

Data collection: The majority of the studies mined a large number of repositories to construct their own dataset. Specifically, Gopalakrishnan et al. [109] 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. [128], Han et al. [129] trained and tested their system by sampling 4,919 source code lines from open-source projects. Raychev et al. [257] used large codebases from GitHub to make predictions for JavaScript and Python code completion. Svyatkovskiy et al. [307] used 2,700 Python open-source software GitHub repositories for the evaluation of their novel approach, Pythia.

The rest of the approaches employed existing benchmarks and datasets. Rahman et al. [255] trained their proposed model using the data extracted from Aizu Online Judge (AOJ) system. Liu et al. [191], Liu et al. [192] 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. [180] 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 [245] used three corpus for their experiments—a large-scale corpus of English-German translation in NLP [201], the Conala corpus [356], which contains Python software documentation as 116,000 English sentences, and the MSR 2013 corpus [18]. Schuster et al. [275] used a public archive of GitHub from 2020 [1].

Feature extraction: Studies in this category extract source code information in variety of forms. Gopalakrishnan et al. [109] extracted relationships between topical concepts in the source code and the use of specific architectural developer tactics in that code. Phan and Jannesari [245] 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 code. Liu et al. [191], Liu et al. [192] 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. [180] used locally repeated terms in program source code to predict out-of-vocabulary (OoV) words that restrict the code completion. Chen and Wan [70] proposed a tree-to-sequence (Tree2Seq) model that captures the structure information of source code to generate comments for source code. Raychev et al. [257] used ASTs and performed prediction of a program element on a dynamically computed context. Svyatkovskiy et al. [307] 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 [311] used LSTM for code completion to facilitate programming education. Rahman et al. [255] also used LSTM. Wang et al. [335] 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. [367] 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. [180] 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. [142] presented CodeGRU that uses GRU for capturing source codes contextual, syntactical, and structural dependencies. Yang et al. [351] 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. [274] 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. [109] developed a recommender system using prediction models including neural networks for latent topics. Han et al. [128], Han et al. [129] 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. [252] used *Bayesian Networks* for intelligent code completion. Raychev et al. [257] utilized a probabilistic model for code in any programming language with *Decision Tree*. Svyatkovskiy et al. [307] 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. [191], Liu et al. [192] applied Multi-Task Learning (MTL) for suggesting code elements. Lee et al. [177] developed MergeLogging, a DLbased merged network that uses code representations for automated logging decisions. Chen and Wan [70] applied Tree2Seq model with NMT techniques for code comment generation. Phan and Jannesari [245] 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.

4.6 Program Comprehension

Program comprehension techniques attempt to understand the theory of comprehension process of developers as well as the tools, techniques, and processes that influence the comprehension activity [301]. 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.

4.6.1 Code summarization

Code summarization techniques attempt to provide a consolidated summary of the source code entity (typically a method). A variety of attempts has been made in this direction. The majority of the studies [6, 72, 139, 144, 174, 175, 179, 189, 290, 334, 340, 353, 355, 359] produces a summary for a small block (such as a method). This category also includes studies that summarize small code fragments [228], code folding within IDEs [329], commit message generation [147, 196], and title generation for online posts from code [105]. Figure 11 provides an overview of the mechanism used by code summarization techniques.

Data collection and processing: The majority of the studies [6, 16, 71, 72, 139, 175, 179, 189, 331, 334, 340] in this category prepares pairs of code snippets and their corresponding natural language description. Specifically, Chen and Zhou [72] 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. [6] 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. [139] and Li et al. [179] prepared a large dataset from 9,714 GitHub projects. Similarly, Wang et al. [334] mined code snippets and corresponding `javadoc` comments for their experiment. Chen et al. [71] 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. Choi et al. [77] collected and refined more than 114 thousand pairs of methods and corresponding code annotations from 100 open-source Java projects. Iyer et al. [144] mined StackOverflow and extracted title and code snippet from posts that contain exactly one code snippet. Similarly, Gao et al. [105] used a dump of StackOverflow dataset. They tokenized code snippets with respect to each programming language for pre-processing. The common steps in preprocessing identifiers include making them lower case, splitting the camel-cased and underline identifiers into sub-tokens, and normalizing the code with special tokens such as "VAR" and "NUMBER". Nazar et al. [228] used human annotators to summarize 127 code fragments retrieved from Eclipse and NetBeans official frequently

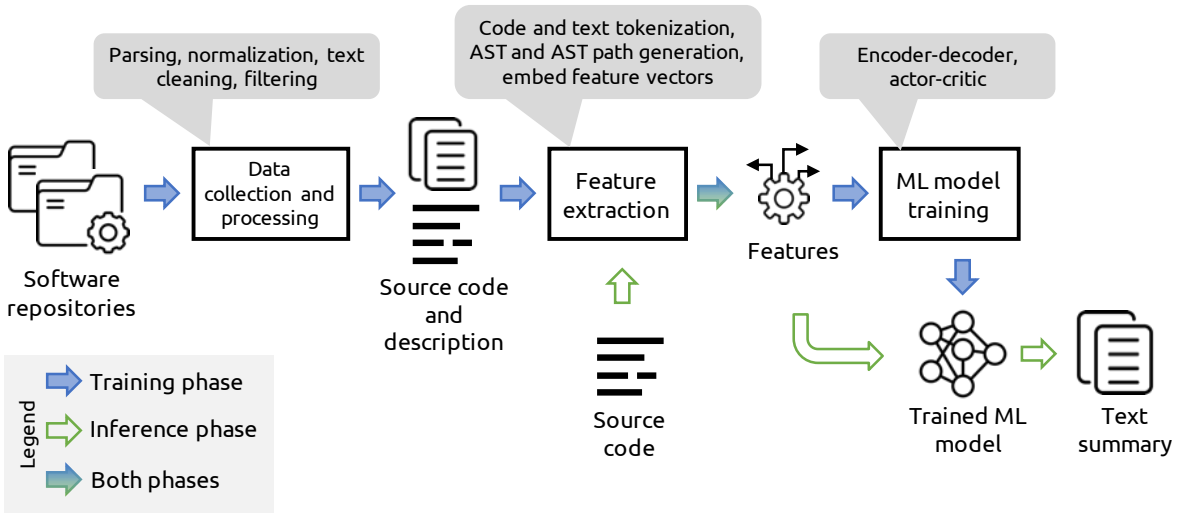


Fig. 11. Overview of the code summarization process pipeline

asked questions. Yang et al. [352] built a dataset with over 300K pairs of method and comment to evaluate their approach.

Apart from source code, some of the studies used additional information generated from source code. For example, LeClair et al. [175] used AST along with code and their corresponding summaries belonging to more than 2 million Java methods. Likewise, Shido et al. [290] and Zhang et al. [359] also generated ASTs of the collected code samples. Liu et al. [189] utilized call dependencies along with source code and corresponding comments from more than a thousand GitHub repositories. LeClair et al. [174] employed AST along with adjacency matrix of AST edges.

Some of the studies used existing datasets such as StaQC [354] and the dataset created by Jiang et al. [147]. Specifically, Liu et al. [196] utilized a dataset of commits provided by Jiang et al. [147] that contains two million commits from one thousand popular Java projects. Yao et al. [353] and Ye et al. [355] used StaQC dataset [354]; it contains more than 119 thousand pairs of question title and code snippet related to SQL mined from StackOverflow. Xie et al. [345] utilized two existing datasets—one each for Java [176] and Python [42]. Bansal et al. [40] evaluated their code summarization technique using a Java dataset of 2.1M Java methods from 28K projects created by LeClair and McMillan [176].

Efforts in the direction of automatic code folding also utilize techniques similar to code summarization. Viuginov and Filchenkov [329] 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. [147] tokenized their code and text. Liu et al. [189] 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 [329] 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. [139] and Li et al. [179] transformed AST into Structure-Based Traversal (SBT). Yang et al. [352] 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.

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. [175] used embeddings layer for code, text, as well as for AST. Similarly, Choi et al. [77] transformed each of the tokenized source code into a vector of fixed length through an embedding layer. Wang et al. [334] extracted the functional keyword from the code and perform positional encoding. Yao et al. [353] 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. [355] 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. [71] 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. [174] used a single embedding layer for both the source code and AST node inputs to exploit a large overlap in vocabulary. Wang et al. [340] prepared a large-scale corpus of training data where each code sample is represented by three sequences—code (in text form), AST, and CFG. These sequences are encoded into vector forms using work2vec. Studies also explored other mechanisms to encode features. For example, Liu et al. [196] 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. [359] parsed code snippets into ASTs and calculated their similarity using ASTs. Allamanis et al. [16] and Ahmad et al. [6] employed attention-based mechanism to encode tokens.

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. [105] 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. [147] trained an NTM algorithm to "translate" from diffs to commit messages. Similarly, Chen et al. [71] and Hu et al. [139] employed LSTM-based *Encoder-Decoder* model to generate summaries. Zhang et al. [359] 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. [144] 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. [77] 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. [6] 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. [175] 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. [352] developed a multi-modal transformer-based code summarization approach for smart contracts. Xie et al. [345] designed a novel multi-task learning (MLT) approach for code summarization through mining the relationship between method-code summaries and method names. Bansal et al. [40] introduced a project-level encoder DL model for code summarization.

Extended encoder-decoder models: Many studies extended the traditional *Encoder-Decoder* mechanism in a variety of ways. Liu et al. [189] 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. [179] 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. [290] 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. [334] 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. [174] 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. [355] 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.

Reinforcement learning models: Some of the studies exploited reinforcement learning techniques for code summary generation. In particular, Yao et al. [353] 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. [331] and Wang et al. [340] 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.

Other techniques: For automated code folding, Viuginov and Filchenkov [329] used *Random Forest* and *Decision Tree* to classify whether a code block needs to be folded. Similarly, Nazar et al. [228] used *Support Vector Machine* and *Naive Bayes* classifiers to generate summaries from the extracted features.

4.6.2 Program classification

Studies targeting this category classify software artifacts based on programming language [324], application domain [324], and type of commits (such as buggy and adaptive) [145, 216]. We summarize these efforts below from dataset preparation, feature extraction, and ML model training perspective.

Dataset and benchmarks: Ma et al. [203] identified more than 91 thousand open-source repositories from GITHUB as subject systems. They created an oracle by manually classifing software artifacts from 383 sample projects. Shimonaka et al. [292] 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. [145] and Meqdadi et al. [216] analyzed the GITHUB commit history. Ugurel et al. [324] relied on C and C++ projects from Ibiblio and the Sourceforge archives.

Feature extraction: Features in this category of studies belong to either source code features category or repository features. A subset of studies [203, 292, 324] relies on features extracted from source code token including language specific keywords and other syntactic information. Other studies [145, 216] collect repository metrics (such as number of changed statements, methods, hunks, and files) to classify commits. Ben-Nun et al. [45] leveraged both the underlying data- and control-flow of a program to learn code semantics performance prediction.

ML model training: A variety of ML approaches have been applied. Specifically, Ma et al. [203] used *Support Vector Machine*, *Decision Tree*, and *Bayes Network* for artifact classification. Meqdadi et al. [216] employed *Naive Bayes*, *Ripper*, as well as *Decision Tree* and Ugurel et al. [324] used *Support Vector Machine* to classify specific commits. Ben-Nun et al. [45] proposed an approach based on an RNN architecture and fixed INST2VEC embeddings for code analysis tasks.

4.6.3 Change analysis

Researchers have explored applications of ML techniques to identify or predict relevant code changes [315, 318]. 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. [315] performed their study on two industrial projects. Tufano et al. [318] extracted 236K pairs of code snippets identified before and after the implementation of the changes provided in the pull requests.

Feature extraction: Tollin et al. [315] extracted features related to the code quality from the issues of two industrial projects. Tufano et al. [318] used features from pull requests to investigate the ability of a NMT modes.

ML model training: Tollin et al. [315] employed *Decision Tree*, *Random Forest*, and *Naive Bayes* ML algorithms for their prediction task. Tufano et al. [318] used *Encoder-Decoder* architecture of a typical NMT model to learn the changes introduced in pull requests.

4.6.4 Entity identification/recommendation

This category represents studies that recommend source code entities (such as method and class names) [12, 132, 146, 212, 347] or identify entities such as design patterns [104] in code using ML [323]. Specifically, Linstead et al. [188] 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. [141] found commenting position in code using ML techniques. Uchiyama et al. [323] identified design patterns and Abuhamad et al. [3] recommended code authorship. Similar approaches include recommending method name [12, 146, 347], method signature [212], class name [12], and type inference [132]. 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. [188] used two large, open source Java projects, Eclipse and ArgoUML in their experiments to apply unsupervised statistical topic models. Similarly, Hellendoorn et al. [132] downloaded 1,000 open-source TypeScript projects and extracted identifiers with corresponding type information. Uchiyama et al. [323] performed experimental evaluations with five programs to evaluate their approach on predicting design patterns. Abuhamad et al. [3] 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. [12] mined 20 software projects from GitHub to predict method and class names. Jiang et al. [146] used the Code2Seq dataset containing 3.8 million methods as their experimental data.

Feature extraction: Several studies generated embeddings from their feature set. Specifically, Huang et al. [141] used embeddings generated from *Word2vec* capturing code semantics. Similarly, Jiang et al. [146] employed *Code2vec* embeddings and Allamanis et al. [12] used embeddings that contain semantic information about sub-tokens of a method name to identify similar embeddings utilized in similar contexts. Zhang et al. [361] utilized knowledge graph embeddings to extract interrelations of code for bug localization. In addition, Uchiyama et al. [323] used several source-code metrics as features to detect design patterns in software programs. Abuhamad

et al. [3] extracted code authorship attributes from samples of code. Malik et al. [212] used function names, formal parameters, and corresponding comments as features.

ML model training: The majority of studies in this category use RNN-based DL models. In particular, Huang et al. [141] and Hellendoorn et al. [132] used bidirectional RNN models. Similarly, Abuhamad et al. [3] and Malik et al. [212] also employed RNN models to identify code authorship and function signatures respectively. Zhang et al. [361] created a bug-localization tool, KGBugLocator utilizing knowledge graph embeddings and bi-directional attention models. Xu et al. [347] employed the GRU-based *Encoder-Decoder* model for method name prediction. Uchiyama et al. [323] used a hierarchical neural network as their classifier. Allamanis et al. [12] utilized neural language models for predicting method and class names.

4.7 Code review

Code Review is the process of systematically check the code written by a developer performed by one or more different developers. A very small set of studies explore the role of ML in the process of code review. Specifically, Lal and Pahwa [169] 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. [33] 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*.

4.8 Code search

Code search is an activity of searching a code snippet based on individual's need typically in Q&A sites such as StackOverflow [264, 293, 330]. 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. [293] utilized commented code as input. Wan et al. [330] used source code in the the form of tokens, AST, and CFG. Sachdev et al. [264] employed a simple tokenizer to extract all tokens from source code by removing non–alphanumeric tokens. Ling et al. [187] 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. [293] 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. [330] employed a multi-modal code representation, where they learnt the representation of each modality via LSTM, Tree-LSTM and GGNN, respectively. Sachdev et al. [264] identified words from source code and transformed the extracted tokens into a natural language documents. Similarly, Ling et al. [187] 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. [293] 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. [330] 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. [264] utilized word and documentation embeddings and performed code search using the learned embeddings. Similarly, Ling et al. [187] 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. [293] used the given query and code sample to measure the semantic similarity using cosine similarity. Wan et al. [330] ranked all the code snippets by their similarities with the input query. Similarly, Sachdev et al. [264] were able to answer almost 43% of the collected StackOverflow questions directly from code.

4.9 Refactoring

Refactoring transformations are intended to improve code quality (specifically maintainability), while preserving the program behavior (functional requirements) from users' perspective [304]. This section summarizes the studies that identify refactoring candidates 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. [260] and Amal et al. [27] 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. [160] 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. [232] analyzed 43 open-source repositories with 13.5 thousand commits to prepare their dataset. Similarly, Aniche et al. [30] created a dataset comprising over two million refactorings from more than 11 thousand open-source repositories. Finally, Kurbatova et al. [168] 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 [32, 164, 165], 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. [160] computed cyclomatic complexity along with 25 other code quality metrics. Similarly, Kumar et al. [164] 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 [32, 165] calculated a large number of metrics. Specifically, Kumar and Sureka [165] computed 102 metrics and then applied PCA to reduce the number of features to 31, while Aribandi et al. [32] used 125 metrics.

Some other studies did not limit themselves to only code quality metrics. Particularly, Yue et al. [357] collected 34 features belonging to code, evolution history, *diff* between commits, and co-change. Similarly, Aniche et al. [30] extracted code quality metrics, process metrics, and code ownership metrics.

In addition, Nyamawe et al. [232] carried out standard NLP preprocessing and generated TF-IDF embeddings for each sample. Along the similar lines, Kurbatova et al. [168] used *code2vec* to generate embeddings for each method.

ML model training: All the studies in this category utilized traditional ML techniques. Rodriguez et al. [260] proposed a method to identify web-service groups for refactoring using *K-means*, COBWEB, and expectation

maximization. Kosker et al. [160] trained a *Naive Bayes*-based classifier to identify classes that need refactoring. Kumar and Sureka [165] 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. [232] 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. [357] 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. [164] 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. [27] proposed the use of ANN to generate a sequence of refactoring. Aribandi et al. [32] 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. [168] proposed an approach to recommend move method refactoring based on a path-based presentation of code using *Support Vector Machine*. Similarly, Aniche et al. [30] used *Linear Regression*, *Naive Bayes*, *Support Vector Machine*, *Decision Tree*, *Random Forest*, and *Neural Network* to predict applicable refactoring operations.

4.10 Vulnerability analysis

The studies in this domain analyze source code to identify potential security vulnerabilities. In this section, we point out the state-of-the-art in software vulnerability detection using ML techniques. 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.

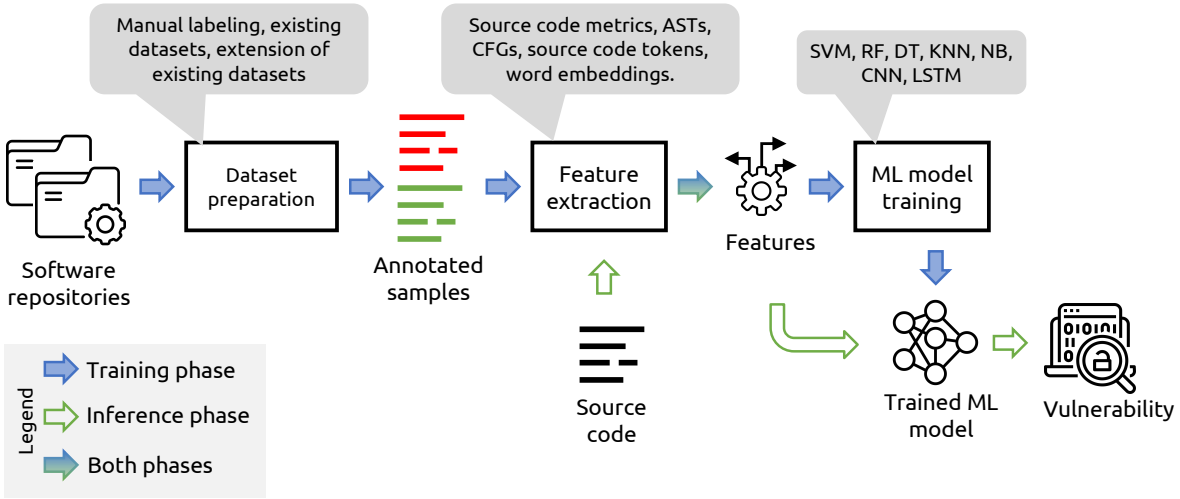


Fig. 12. Overview of vulnerability detection process pipeline

Dataset preparation: Authors used existing labeled datasets as well as created their own datasets to train ML models. Specifically, a set of studies [48, 156, 219, 243, 254, 263, 298] used available labeled datasets for PHP, Java, C, C++, and Android applications to train vulnerability detection models. In other cases, Russell et al. [261] 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 [10, 26, 75, 79, 83, 95, 100, 120, 137, 148, 163, 205, 215, 220, 229, 238, 244, 246, 271, 279, 350, 366] created their own datasets. Ali Alatwi et al. [10], Cui et al. [83], Ma et al. [205], and Gupta et al. [120] created datasets to train vulnerability detectors for Android applications. In particular, Ma et al. [205] decompiled and generated CFGs of approximately 10 thousand, both benign and vulnerable, Android applications from *AndroZoo* and *Android Malware* datasets; Ali Alatwi et al. [10] collected 5,063 Android applications where 1,000 of them were marked as benign and the remaining as malware; Cui et al. [83] 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. [120] 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. [215] collected 35 open-source PHP projects and intentionally injected 76 vulnerabilities in their dataset. Shar et al. [279] 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. [229] 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 [244, 254, 350] 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 [246] manually label their dataset. Hou et al. [137], Moskovitch et al. [220] and Santos et al. [271] created their datasets by collecting web-page samples from StopBadWare and VxHeavens.

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 [26, 79, 83, 100, 120, 215, 246, 254] 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. [48], Kim et al. [156], Kronjee et al. [163], Ma et al. [204] used CFGs, ASTs, or data flow analysis as features. More specifically, Ma et al. [205] 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. [156] extracted ASTs and GFCs which they tokenized and fed into ML models, while Bilgin et al. [48] extracted ASTs and translated their representation of source code into a one-dimensional numerical array to fed them to a model. Kronjee et al. [163] used data-flow analysis to extract features, while Spreitzenbarth et al. [298] used static, dynamic analysis, and information collected from `ltrace` to collect features and train a linear vulnerability detection model.

Repository and file metrics: Perl et al. [244] 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 [95, 243] used file meta-data such as *files' creation and modification time, machine type, file size,* and *linker version.*

Code tokens: Chernis and Verma [75] 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. [137] 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 [220, 238, 261, 263, 271, 279, 350, 366] tokenized parts of the source code 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.

Other features: Ali Alatwi et al. [10], Ndichu et al. [229] and Milosevic et al. [219] extracted permission-related features. In some cases, authors [261] used CNN and GRU to extract features from source code representations.

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

Traditional ML techniques: One set of studies [10, 220, 229, 238, 243, 244, 261, 279] used traditional ML algorithms such as *Support Vector Machine*, *Linear Regression*, *Decision Tree*, and *Random Forest* to train their models. Specifically, Ali Alatwi et al. [10], Perl et al. [244], Russell et al. [261] 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. [229] used *Support Vector Machine* to train their model with linear kernel. Pereira et al. [243] used *Decision Tree*, *Linear Regression*, and *Lasso* to train their models. Compared to the above studies, Shar et al. [279] 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.

Other studies [26, 75, 79, 83, 100, 120, 137, 163, 215, 219, 246, 254, 271] used up to 32 different ML algorithms to train models and compared their performance. Specifically, Medeiros et al. [215] 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. [219] and Rahman et al. [254] 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. [100] showed that *K Nearest Neighbors* offers the best performance for their dataset after experimenting with DNN, *K Nearest Neighbors*, *Support Vector Machine*, *Linear Regression*, *Decision Tree*, *Random Forest*, and *Naive Bayes*. In order to find out which is the best model for the SWAN tool, Piskachev et al. [246] 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. [163], Cui et al. [83], and Gupta et al. [120] compared different ML algorithms and found *Decision Tree* and *Random Forest* as the best performing algorithms.

DL techniques: A few number of studies [156, 263, 350] used DL methods such as CNN, RNN, and ANN to train models. In more details, Yang et al. [350] utilized the BP-ANN algorithm to train vulnerability detectors. For the project *Achilles*, Saccente et al. [263] 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. [156] 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. [366] 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.

5 RQ3. What datasets and tools are available?

This research question aims to explore and provide 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 (August 2021).
- 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 1 lists all the tools that we found in this exploration. We provide the original reference of the resource along with name and link to access the resource in the first column, category, a short description.

Table 1. A list of tools useful for analyzing source code and applying machine learning techniques

Category	Name	Description
Code Representation	ncc [45]	Learns representations of code semantics
	Code2vec [23]	Generates distributed representation of code
	Code2seq [21]	Generates sequences from structured representation of code
	Vector representation for coding style [161]	Implements vector representation of individual coding style
	CC2Vec [135]	Implements distributed representation of code changes
	AutoenCODE [319]	Encodes source code fragments into vector representations
	Graph-based code modeling [15]	Generates code modeling with graphs
	Vocabulary learning on code [85]	Generates an augmented AST from Java source code
	User2code2vec [34]	Generates embeddings for developers based on distributed representation of code
Code Search	Deep Code Search [118]	Searches code by using code embeddings
Program Comprehension	Obfuscated-code2vec [80]	Embeds Java Classes with Code2vec
	DeepTyper [132]	Annotates types for JavaScript and TypeScript
	CallNN [189]	Implements a code summarization approach by using call dependencies
	NeuralCodeSum [6]	Implements a code summarization method by using transformers
	Summarization_tf [290]	Summarizes code with Extended Tree-lstm
	CoaCor [353]	Explores the role of rich annotation for code retrieval
	DeepCom [179]	Generates code comments
	Rencos [359]	Generates code summary by using both neural and retrieval-based techniques
	CODES [239]	Extracts method description from StackOverflow discussions
	CFS	Summarizes code fragments using SVM and NB
	TASSAL	Summarizes code using autofolding

	ChangeScribe [81]	Generates commit messages
	CodeInsight [256]	Recommends insightful comments for source code
	CodeNN [144]	Summarizes code using neural attention model
	Code2Que [105]	Suggests improvements in question titles from mined code in StackOverflow
	BI-TBCNN [57]	Implements a BI-TBCNN model to classify algorithms
	DeepSim [365]	Implements a DL approach to measure code functional similarity
	FCDetector [99]	Proposes a fine-grained granularity of source code for functionality identification
Quality Assessment	SonarQube	Analyzes code quality
	SVF [303]	Enables inter-procedural dependency analysis for LLVM-based languages
	Designite [284]	Detects code smells and computes quality metrics in Java and C# code
	CloneCognition [222]	Proposes a ML framework to validate code clones
	SMAD [41]	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
	ML Clone Validation Framework [223]	Implements a ML framework for automatic code clone validation
	py-ccflex [233]	Mimics code metrics by using ML
	Deep learning smells [283]	Implements DL (CNN, RNN, and Autoencoder-based models) to identify four smells
	CREC [357]	Recommends clones for refactoring
	ML for software refactoring [30]	Recommends refactoring by using ML
Program Synthesis	CoCoNuT [202]	Repairs Java programs
	DeepFix [123]	Fixes common C errors
Testing	AppFlow [138]	Automates UI tests generation
	DeepFuzz [195]	Grammar fuzzer that generates C programs
	Agilika [325]	Generates tests from execution traces
	BugDetection [183]	Trains models for defect prediction
	DTLDP [68]	Implements a deep transfer learning framework
	DeepBugs [249]	Implements a framework for learning name-based bug detectors
	Randoop	Generates tests automatic for Java code
	TestDescriber	Implements test case summary generator and evaluator

Vulnerability Analysis	WAP [214]	Detects and corrects input validation vulnerabilities
	SWAN[246]	Identifies vulnerabilities
	VCCFinder [244]	Finds potentially dangerous code in repositories
General	BERT	NLP pre-trained models
	BC3 Annotation Framework	Annotates emails/conversations easily
	JGibLDA	Implements Latent Dirichlet Allocation
	Stanford NLP Parser	A statistical NLP parser
	srcML	Generates XML representation of sourcecode
	CallGraph	Generates static and dynamic call graphs for Java code
	ML for programming	Offers various tools such as JSNice, Nice2Predict, and DEBIN

The list of datasets found in this exploration are presented in Table 2. We provide the original reference of the dataset along with name, category, a short description, and link to access the dataset.

Table 2. A list of datasets useful for analyzing source code and applying machine learning techniques

Category	Name	Description
Code Representation	Code2seq [23]	Sequences generated from structured representation of code
	GHTorrent [113]	Meta-data from GitHub repositories
Code Completion	Neural Code Completion	Dataset and code for code completion with neural attention and pointer networks
	TL-CodeSum [140]	Dataset for code summarization
Program Synthesis	CoNaLa corpus [356]	Python snippets and corresponding natural language description
	IntroClass [173]	Program repair dataset of C programs
Program Comprehension	Program comprehension dataset [299]	Contains code for a program comprehension user survey
	CommitGen [147]	Commit messages and the diffs from 1,006 Java projects
	StaQC [354]	148K Python and 120K SQL question-code pairs from StackOverflow
Quality Assessment	src-d datasets	Various labeled datasets (commit messages, duplicates, DockerHub, and Nuget)
	BigCloneBench [305]	Known clones in the IJaDataset source repository
	Multi-label smells [119]	A dataset of 445 instances of two code smells and 82 metrics
	Deep learning smells [283]	A dataset of four smells in tokenized form from 1,072 C# and 100 Java repositories
	ML for software refactoring [30]	Dataset for applying ML to recommend refactoring
Testing	Defects4J [149]	Java reproducible bugs
	PROMISE [273]	Various datasets including defect prediction and cost estimation

	BugDetection [183]	A bug prediction dataset containing 4.973M methods belonging to 92 different Java project versions
	DAMT [226]	Metamorphic testing dataset
	DTLDP [68]	Dataset for deep transfer learning for defect prediction
	DeepBugs [249]	A JavaScript code corpus with 150K code snippets
Vulnerability Analysis	WPScan	a PHP dataset for WordPress plugin vulnerabilities
	Genome [368]	1,200 malware samples covering the majority of existing malware families
	Juliet [50]	81K synthetic C/C++ and Java programs with known flaws
	AndroZoo [20]	15.7M APKs from Google's Play Store
	TRL [186]	Vulnerabilities in six C programs
	Draper VDISC [262]	1.27 million functions mined from C and C++ applications
	SAMATE [49]	A set of known security flaws from NIST for C, C++, and Java programs
	JSVulner [100]	JavaScript Vulnerability Analysis dataset
	SWAN [246]	A Vulnerability Analysis collection of 12 Java applications
	Project-KB [247]	A Manually-Curated dataset of fixes to vulnerabilities of open-source software
General	GitHub Java Corpus [17]	A large collection of Java repositories
	150k Python dataset [257]	Contains parsed AST for 150K Python files
	UCI source code dataset [198]	Various large scale source code analysis datasets

6 RQ4. What are the challenges and perceived deficiencies?

The aim of this research question is to focus on the perceived deficiencies, challenges, and opportunities in applying ML techniques in the context of source code analysis observed from the collected articles. We documented 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 [25, 41, 68, 93, 106, 108, 111, 148, 167, 193, 268, 287, 297, 306, 314, 317, 322, 332]. 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. [91] 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 [91]; carrying out a performance evaluation based on a balanced dataset is flawed and obviously the model will perform significantly inferior when it is put at work in a real-world context. We noted many studies [5, 84, 102, 103, 119, 235, 313] 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 [332]. Phases of ML development are very exploratory in nature and highly domain and problem dependent [332]. Identifying the most appropriate ML model, their appropriate parameters, and configuration is largely driven by *trial and error* manner [35, 287, 332]. 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 [107, 240, 332]. 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 [107].
- ***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 [56, 362], 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 [56, 107].
- ***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 [143, 240, 287, 317, 332]. Specifically, software engineering researchers have notified that identifying and extracting relevant features beyond code quality metrics is non-trivial. For example, Ivers et al. [143] 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. [283] 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. [332] 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 [125] also reports a serious lack of ML expertise in academic software engineering efforts. Other authors [240] 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) [107, 332]. A user-survey study [332] 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 venue label where at least three articles per category appear in a venue. Each label includes a number indicating the number of articles a same venue in that category.

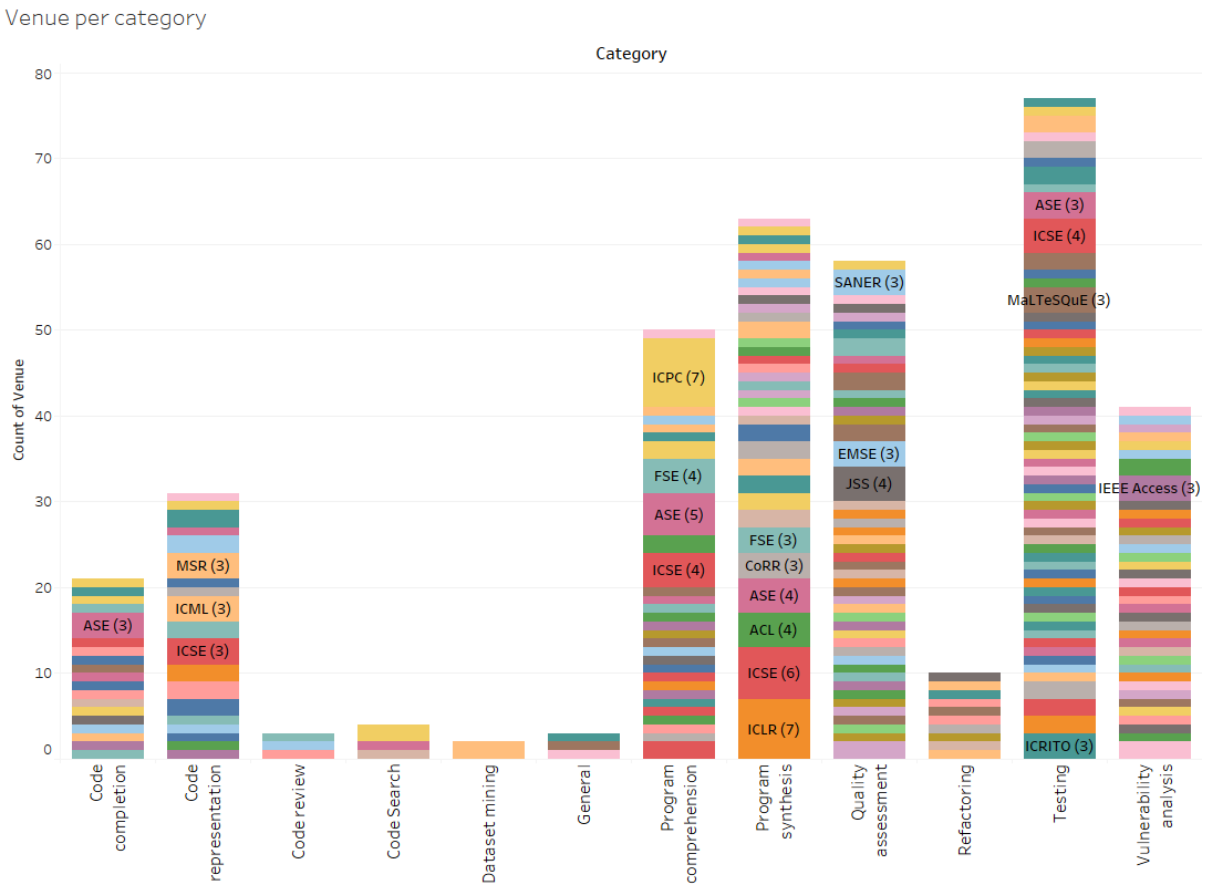


Fig. 13. Top venues for each considered category

We can observe that ICSE and ASE are among the top venues, appearing to four categories each. The journals EMSE and JSS are the top venues for the category *quality assessment*. Machine learning conferences ICML and ICLR are also appearing as the top venues for one category each (i.e., *code representation* and *program synthesis*, respectively).

The categories of *program comprehension* and *program synthesis* exhibit the highest concentration of articles to a relatively small list of top venues where 40% and 42% 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 Mitigating the challenges

7.2.1 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 MSR data showcase track. Dataset search engines such as Google dataset search[4] 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 [250, 258, 277] 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. [148] 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.

7.2.2 Reproducibility and replicability: The importance of reproducibility and replicability has been emphasized and understood by the software engineering community [190]. It has lead to a concrete artifact evaluation mechanism adopted by leading software engineering conferences. For example, FSE artifact evaluation divides artifacts into five categories—*functional, reusable, available, results reproduced,* and *results replicated.*[5] 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 [39]) are being made to support ML research reproducible and replicable.

7.2.3 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 AWS Sagemaker and Google Notebooks attempt to make ML development easier, they essentially do not offer much help in reducing the ad-hoc nature of the development. A significant research push from the community would make ML development relatively systematic and organized.

Recent advancements in the form of available tools not only help a developer to comprehend the process but also let them effectively manage code, data, and experimental results. Examples of such tools and methods include DARVIZ [269] for DL model visualization, MLFlow[6] for managing the ML lifecycle, and DeepFault [96] for identifying faults in DL programs. Such efforts are expected to address the challenge.

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

7.2.4 Hardware resources: ML development is resource hungry. Certain DL models (such as models based on RNN) consume excessive hardware resources. The need for a large-scale hardware infrastructure is increasing with the increase in size of the captured features and the training samples. To address the challenge, infrastructure at

[4]https://datasetsearch.research.google.com/
[5]https://2021.esec-fse.org/track/fse-2021-artifacts
[6]https://mlflow.org/

institution and country level are maintained in some countries; however, a generic and widely-applicable solution is needed for more globally-inclusive research.

8 Threats to validity

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.

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[7] containing all the metadata for all the articles selected in each of the three phases of article selection. In addition, inspired by previous surveys [14, 136], we have developed a website[8] as a *living documentation and literature survey* to facilitate easy navigation, exploration, and extension. The website can be easily extended as the new studies emerge in the domain; we have made the repository[9] open-source to allow the community to extend the living literature survey.

9 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 364 primary studies spanning from 2002 to 2020 (and to some extent 2021) 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.gharchive.org/

[2] Osama Abdeljaber, Onur Avci, 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.

[3] 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

[4] 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

[7] https://github.com/tushartushar/ML4SCA/tree/main/replication

[8] http://www.tusharma.in/ML4SCA

[9] https://github.com/tushartushar/ML4SCA

[5] 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

[6] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-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

[7] 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

[8] H. A. Al-Jamimi and M. Ahmed. 2013. Machine Learning-Based Software Quality Prediction Models: State of the Art. In *2013 International Conference on Information Science and Applications (ICISA)*. 1–4. https://doi.org/10.1109/ICISA.2013.6579473

[9] 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.

[10] Huda Ali Alatwi, 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

[11] 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

[12] 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

[13] 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

[14] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

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

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

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

[18] 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

[19] Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. 2015. 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.

[20] Kevin Allix, Tegawendé F. Bissyandé, 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

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

[22] 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/10.1145/3296979.3192412

[23] 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

[24] 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

[25] 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

[26] 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

[27] Boukhdhir Amal, Marouane Kessentini, Slim Bechikh, Josselin Dea, and Lamjed Ben Said. 2014. On the Use of Machine Learning and Search-Based Software Engineering for Ill-Defined Fitness Function: A Case Study on Software Refactoring. In *Search-Based Software Engineering*, Claire Le Goues and Shin Yoo (Eds.). 31–45.

[28] 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

[29] 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.2018.8489079

[30] 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

[31] 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

[32] 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.

[33] 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*.

[34] David Azcona, Piyush 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) *(LAK19)*. 86–95. https://doi.org/10.1145/3303772.3303813

[35] 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

[36] 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

[37] Mourad Badri, Linda Badri, William Flageol, and Fadel Toure. 2017. Investigating the Accuracy of Test Code Size Prediction Using Use Case Metrics and Machine Learning Algorithms: An Empirical Study. In *Proceedings of the 2017 International Conference on Machine Learning and Soft Computing* (Ho Chi Minh City, Vietnam) *(ICMLSC '17)*. 25–33. https://doi.org/10.1145/3036290.3036323

[38] 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.

[39] Vishnu Banna, Akhil Chinnakotla, Zhengxin Yan, Anirudh Vegesana, Naveen Vivek, Kruthi Krishnappa, Wenxin 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

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

[41] 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

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

[43] Nicolas Baskiotis and Michele Sebag. 2008. Structural Statistical Software Testing with Active Learning in a Graph. In *Inductive Logic Programming*, Hendrik Blockeel, Jan Ramon, Jude Shavlik, and Prasad Tadepalli (Eds.). 49–62.

[44] Nicolas Baskiotis, Michèle Sebag, Marie-Claude Gaudel, and Sandrine-Dominique Gouraud. 2007. A Machine Learning Approach for Statistical Software Testing. *Proc. International Joint Conference on Artificial Intelligence*, 2274–2279.

[45] 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* (Montréal, Canada) *(NIPS'18)*. 3589–3601.

[46] 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/CCCS.2018.8586805

[47] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-Symbolic Program Corrector for Introductory Programming Assignments. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. 60–70. https://doi.org/10.1145/3180155.3180219

[48] 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

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

[50] 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.2012.345

[51] 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* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 330–341. https://doi.org/10.1145/2931037.2931039

[52] 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/10.1145/3266237.3266273

[53] Alexander Brauckmann, Andrés 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.

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

[55] 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) *(ESEC/FSE '09)*. 213–222. https://doi.org/10.1145/1595696.1595728

[56] 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) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 754–759. https://doi.org/10.1145/3236024.3264838

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

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

[59] 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

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

[61] 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

[62] 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.

[63] 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/ICCCNT49239.2020.9225352

[64] E. Ceylan, F. O. Kutlubay, and A. B. Bener. 2006. Software Defect Identification Using Machine Learning Techniques. In *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*. 240–247. https://doi.org/10.1109/EUROMICRO.2006.56

[65] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray. 2020. CODIT: Code Editing with Tree-Based Neural Models. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.3020502

[66] VENKATA UDAYA B. CHALLAGULLA, FAROKH B. BASTANI, I-LING YEN, and RAYMOND A. PAUL. 2008. EMPIRICAL ASSESSMENT OF MACHINE LEARNING BASED SOFTWARE DEFECT PREDICTION TECHNIQUES. *International Journal on Artificial Intelligence Tools* 17, 02 (2008), 389–400. https://doi.org/10.1142/S0218213008003947 arXiv:https://doi.org/10.1142/S0218213008003947

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

[68] Jinyin Chen, Keke Hu, Yue Yu, Zhuangzhi Chen, Qi Xuan, Yi Liu, and Vladimir Filkov. 2020. Software Visualization and Deep Transfer Learning for Effective Software Defect Prediction. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. 578–589. https://doi.org/10.1145/3377811.3380389

[69] Long Chen, Wei Ye, and Shikun Zhang. 2019. Capturing Source Code Semantics via Tree-Based Convolution over API-Enhanced AST. In *Proceedings of the 16th ACM International Conference on Computing Frontiers* (Alghero, Italy) *(CF '19)*. 174–182. https://doi.org/10.1145/3310273.3321560

[70] M. Chen and X. Wan. 2019. Neural Comment Generation for Source Code with Auxiliary Code Classification Task. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. 522–529. https://doi.org/10.1109/APSEC48747.2019.00076

[71] Qiuyuan Chen, Han Hu, and Zhaoyi Liu. 2019. Code Summarization with Abstract Syntax Tree. In *Neural Information Processing*, Tom Gedeon, Kok Wai Wong, and Minho Lee (Eds.). 652–660.

[72] Q. Chen and M. Zhou. 2018. A Neural Framework for Retrieval and Summarization of Source Code. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 826–831. https://doi.org/10.1145/3238147.3240471

[73] Xinyun Chen, Chang Liu, Richard Shin, Dawn Song, and Mingcheng Chen. 2016. Latent Attention for If-Then Program Synthesis. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. 4581–4589.

[74] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus. 2019. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2940179

[75] Boris Chernis and Rakesh Verma. 2018. Machine Learning Methods for Software Vulnerability Detection. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics* (Tempe, AZ, USA) *(IWSPA '18)*. 31–39. https://doi.org/10.1145/3180445.3180453

[76] S. R. Chidamber and C. F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transaction of Software Engineering* 20, 6 (June 1994), 476–493. https://doi.org/10.1109/32.295895

[77] Y. Choi, S. Kim, and J. Lee. 2020. Source Code Summarization Using Attention-Based Keyword Memory Networks. In *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*. 564–570. https://doi.org/10.1109/BigComp48618.2020.00011

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

[79] C. J. Clemente, F. Jaafar, and Y. Malik. 2018. Is Predicting Software Security Bugs Using Deep Learning Better Than the Traditional Machine Learning Algorithms?. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 95–102. https://doi.org/10.1109/QRS.2018.00023

[80] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. 2020. Embedding Java Classes with Code2vec: Improvements from Variable Obfuscation. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. 243–253. https://doi.org/10.1145/3379597.3387445

[81] Luis Fernando Cortes-Coy, M. Vásquez, Jairo Aponte, and D. Poshyvanyk. 2014. On Automatically Generating Commit Messages via Summarization of Source Code Changes. *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation* (2014), 275–284.

[82] 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

[83] 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

[84] 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.

[85] Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. 2019. Open Vocabulary Learning on Source Code with a Graph-Structured Cache *(Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). 1475–1485.

[86] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. 2019. Lessons Learned from Using a Deep Tree-Based Model for Software Defect Prediction in Practice. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. 46–57. https://doi.org/10.1109/MSR.2019.00017

[87] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2012. Evaluating Defect Prediction Approaches: A Benchmark and an Extensive Comparison. *Empirical Softw. Engg.* 17, 4–5 (Aug. 2012), 531–577. https://doi.org/10.1007/s10664-011-9173-9

[88] 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.

[89] 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 (ICICI) 2018*, Jude Hemanth, Xavier Fernando, Pavel Lafata, and Zubair Baig (Eds.). 397–406.

[90] 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 Oivo, Maria Teresa Baldassarre, and Giuseppe Visaggio (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 247–261.

[91] 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/10.1109/SANER.2018.8330266

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

[93] 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

[94] D. G. E. Silva, M. Jino, and B. T. d. Abreu. 2010. Machine Learning Methods and Asymmetric Cost Function to Estimate Execution Effort of Software Testing. In *2010 Third International Conference on Software Testing, Verification and Validation*. 275–284. https://doi.org/10.1109/ICST.2010.46

[95] 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.

[96] 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.

[97] 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

[98] 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

[99] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai 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

[100] Rudolf Ferenc, Péter Hegedundefineds, Péter Gyimesi, Gábor Antal, Dénes Bán, and Tibor Gyimóthy. 2019. Challenging Machine Learning Algorithms in Predicting Vulnerable JavaScript Functions. In *Proceedings of the 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering* (Montreal, Quebec, Canada) *(RAISE '19)*. 8–14. https://doi.org/10.1109/RAISE.2019.00010

[101] Fabio Ferreira, Luciana Lourdes Silva, and Marco Tulio Valente. 2021. Software Engineering Meets Deep Learning: A Mapping Study. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (Virtual Event, Republic of Korea) *(SAC '21)*. Association for Computing Machinery, New York, NY, USA, 1542–1549. https://doi.org/10.1145/3412841.3442029

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

[103] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä. 2013. Code Smell Detection: Towards a Machine Learning-Based Approach. In *2013 IEEE International Conference on Software Maintenance*. 396–399.

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

[105] Zhipeng Gao, Xin Xia, John Grundy, David Lo, and Yuan-Fang Li. 2020. Generating Question Titles for Stack Overflow from Mined Code Snippets. *ACM Trans. Softw. Eng. Methodol.* 29, 4, Article 26 (September 2020), 37 pages. https://doi.org/10.1145/3401026

[106] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. *ACM Comput. Surv.* 50, 4, Article 56 (August 2017), 36 pages. https://doi.org/10.1145/3092566

[107] Görkem Giray. 2021. A software engineering perspective on engineering machine learning systems: State of the art and challenges. *Journal of Systems and Software* 180 (2021), 111031. https://doi.org/10.1016/j.jss.2021.111031

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

[109] R. Gopalakrishnan, P. Sharma, M. Mirakhorli, and M. Galster. 2017. Can Latent Topics in Source Code Predict Missing Architectural Tactics?. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 15–26. https://doi.org/10.1109/ICSE.2017.10

[110] Divya Gopinath, Sarfraz Khurshid, Diptikalyan Saha, and Satish Chandra. 2014. Data-Guided Repair of Selection Statements. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. 243–253. https://doi.org/10.1145/2568225.2568303

[111] D. Gopinath, K. Wang, J. Hua, and S. Khurshid. 2016. Repairing Intricate Faults in Code Using Machine Learning and Path Exploration. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 453–457. https://doi.org/10.1109/ICSME.2016.75

[112] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (November 2019), 56–65. https://doi.org/10.1145/3318162

[113] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236. http://dl.acm.org/citation.cfm?id=2487085.2487132

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

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

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

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

[118] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 933–944. https://doi.org/10.1145/3180155.3180167

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

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

[121] H. Gupta, L. Kumar, and L. B. M. Neti. 2019. An Empirical Framework for Code Smell Prediction using Extreme Learning Machine*. In *2019 9th Annual Information Technology, Electromechanical Engineering and Microelectronics Conference (IEMECON)*. 189–195. https://doi.org/10.1109/IEMECONX.2019.8877082

[122] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Deep Reinforcement Learning for Syntactic Error Repair in Student Programs. *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (07 2019), 930–937. https://doi.org/10.1609/aaai.v33i01.3301930

[123] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning.. In *AAAI*. 1345–1351.

[124] Mouna Hadj-Kacem and Nadia Bouassida. 2018. A Hybrid Approach To Detect Code Smells using Deep Learning.. In *ENASE*. 137–146.

[125] T. Hall and D. Bowes. 2012. The State of Machine Learning Methodology in Software Fault Prediction. In *2012 11th International Conference on Machine Learning and Applications*, Vol. 2. 308–313. https://doi.org/10.1109/ICMLA.2012.226

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

[127] Awni Hammouri, Mustafa Hammad, Mohammad Alnabhan, and Fatima Alsarayrah. 2018. Software Bug Prediction using Machine Learning Approach. *International Journal of Advanced Computer Science and Applications* 9 (01 2018). https://doi.org/10.14569/IJACSA.2018.090212

[128] S. Han, D. R. Wallace, and R. C. Miller. 2009. Code Completion from Abbreviated Input. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 332–343. https://doi.org/10.1109/ASE.2009.64

[129] Sangmok Han, David R. Wallace, and Robert C. Miller. 2011. Code Completion of Multiple Keywords from Abbreviated Input. *Automated Software Engg.* 18, 3–4 (December 2011), 363–398. https://doi.org/10.1007/s10515-011-0083-2

[130] Bonnie Hardin and Upulee Kanewala. 2018. Using Semi-Supervised Learning for Predicting Metamorphic Relations. In *Proceedings of the 3rd International Workshop on Metamorphic Testing* (Gothenburg, Sweden) *(MET '18)*. 14–17. https://doi.org/10.1145/3193977.3193985

[131] Mark Harman, Syed Islam, Yue Jia, Leandro L. Minku, Federica Sarro, and Komsan Srivisut. 2014. Less is More: Temporal Fault Predictive Performance over Multiple Hadoop Releases. In *Search-Based Software Engineering*, Claire Le Goues and Shin Yoo (Eds.). Springer International Publishing, Cham, 240–246.

[132] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference *(ESEC/FSE 2018)*. 152–162. https://doi.org/10.1145/3236024.3236051

[133] Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are Deep Neural Networks the Best Choice for Modeling Source Code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. 763–773. https://doi.org/10.1145/3106237.3106290

[134] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-Learning-Guided Selectively Unsound Static Analysis. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. 519–529. https://doi.org/10.1109/ICSE.2017.54

[135] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed Representations of Code Changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. 518–529. https://doi.org/10.1145/3377811.3380361

[136] Max Hort, Maria Kechagia, Federica Sarro, and Mark Harman. 2021. A Survey of Performance Optimization for Mobile Applications. *IEEE Transactions on Software Engineering (TSE)* (2021).

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

[138] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. 269–282. https://doi.org/10.1145/3236024.3236055

[139] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. 2018. Deep Code Comment Generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. 200–20010.

[140] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing Source Code with Transferred API Knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on

Artificial Intelligence Organization, 2269–2275. https://doi.org/10.24963/ijcai.2018/314

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

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

[143] J. Ivers, I. Ozkaya, and R. L. Nord. 2019. Can AI Close the Design-Code Abstraction Gap?. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. 122–125. https://doi.org/10.1109/ASEW.2019.00041

[144] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083. https://doi.org/10.18653/v1/P16-1195

[145] T. Ji, J. Pan, L. Chen, and X. Mao. 2018. Identifying Supplementary Bug-fix Commits. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 01. 184–193. https://doi.org/10.1109/COMPSAC.2018.00031

[146] Lin Jiang, Hui Liu, and He Jiang. 2019. Machine Learning Based Recommendation of Method Names: How Far Are We. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. 602–614. https://doi.org/10.1109/ASE.2019.00062

[147] 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

[148] 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

[149] 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

[150] U. Kanewala and J. M. Bieman. 2013. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. 1–10. https://doi.org/10.1109/ISSRE.2013.6698899

[151] Upulee Kanewala, James M. Bieman, and Asa Ben-Hur. 2016. Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels. *Software Testing, Verification and Reliability* 26, 3 (2016), 245–269. https://doi.org/10.1002/stvr.1594 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1594

[152] 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

[153] 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

[154] 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

[155] 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

[156] Junae Kim, David Hubczenko, 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.

[157] 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.

[158] 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

[159] 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

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

[161] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. 2020. Building Implicit Vector Representations of Individual Coding Style. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW'20)*. 117–124. https://doi.org/10.1145/3387940.3391494

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

[163] Jorrit Kronjee, Arjen Hommersom, and Harald Vranken. 2018. Discovering Software Vulnerabilities Using Data-Flow Analysis and Machine Learning. In *Proceedings of the 13th International Conference on Availability, Reliability and Security* (Hamburg, Germany) *(ARES 2018)*. Article 6, 10 pages. https://doi.org/10.1145/3230833.3230856

[164] Lov Kumar, Shashank Mouli Satapathy, and Lalita Bhanu Murthy. 2019. Method Level Refactoring Prediction on Five Open Source Java Projects Using Machine Learning Techniques. In *Proceedings of the 12th Innovations on Software Engineering Conference (Formerly Known as India Software Engineering Conference)* (Pune, India) *(ISEC'19)*. Article 7, 10 pages. https://doi.org/10.1145/3299771.3299777

[165] L. Kumar and A. Sureka. 2017. Application of LSSVM and SMOTE on Seven Open Source Projects for Predicting Refactoring at Class Level. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 90–99. https://doi.org/10.1109/APSEC.2017.15

[166] L. Kumar and A. Sureka. 2018. An Empirical Analysis on Web Service Anti-pattern Detection Using a Machine Learning Framework. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 01. 2–11. https://doi.org/10.1109/COMPSAC.2018.00010

[167] Pradeep Kumar and Yogesh Singh. 2012. Assessment of Software Testing Time Using Soft Computing Techniques. *SIGSOFT Softw. Eng. Notes* 37, 1 (January 2012), 1–6. https://doi.org/10.1145/2088883.2088895

[168] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. 2020. Recommendation of Move Method Refactoring Using Path-Based Representation of Code. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW'20)*. 315–322. https://doi.org/10.1145/3387940.3392191

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

[170] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *ACM Comput. Surv.* 53, 3, Article 62 (June 2020), 38 pages. https://doi.org/10.1145/3383458

[171] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *ACM Comput. Surv.* 53, 3, Article 62 (June 2020), 38 pages. https://doi.org/10.1145/3383458

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

[173] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[174] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. In *Proceedings of the 28th International Conference on Program Comprehension* (Seoul, Republic of Korea) *(ICPC '20)*. 184–195. https://doi.org/10.1145/3387904.3389268

[175] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. 795–806. https://doi.org/10.1109/ICSE.2019.00087

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

[177] Suin Lee, Youngseok Lee, Chan-Gun Lee, and Honguk Woo. 2021. Deep Learning-Based Logging Recommendation Using Merged Code Representation. In *IT Convergence and Security*, Hyuncheol Kim and Kuinam J. Kim (Eds.). 49–53.

[178] 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.

[179] 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 Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. 1571–1575. https://doi.org/10.1145/3368089.3417926

[180] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (Stockholm, Sweden) *(IJCAI'18)*. 4159–25.

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

[182] Yi Li, Shaohua Wang, and Tien 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

[183] Yi Li, Shaohua Wang, Tien 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

[184] H. Lim. 2018. Applying Code Vectors for Presenting Software Features in Machine Learning. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 01. 803–804. https://doi.org/10.1109/COMPSAC.2018.00128

[185] R. Lima, A. M. R. da Cruz, and J. Ribeiro. 2020. Artificial Intelligence Applied to Software Testing: A Literature Review. In *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)*. 1–6. https://doi.org/10.23919/CISTI49556.2020.9141124

[186] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Yang Xiang, Olivier De Vel, and Paul Montague. 2018. Cross-Project Transfer Representation Learning for Vulnerable Function Discovery. *IEEE Transactions on Industrial Informatics* 14, 7 (2018), 3289–3297. https://doi.org/10.1109/TII.2018.2821768

[187] Chunyang Ling, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2020. Adaptive Deep Code Search. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC '20)*. Association for Computing Machinery, 48—-59. https://doi.org/10.1145/3387904.3389278

[188] E. Linstead, C. Lopes, and P. Baldi. 2008. An Application of Latent Dirichlet Allocation to Analyzing Software Evolution. In *2008 Seventh International Conference on Machine Learning and Applications*. 813–818. https://doi.org/10.1109/ICMLA.2008.47

[189] Bohong Liu, Tao Wang, Xunhui Zhang, Qiang Fan, Gang Yin, and Jinsheng Deng. 2019. A Neural-Network Based Code Summarization Approach by Using Source Code and Its Call Dependencies. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware* (Fukuoka, Japan) *(Internetware '19)*. Article 12, 10 pages. https://doi.org/10.1145/3361242.3362774

[190] Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2020. On the Replicability and Reproducibility of Deep Learning in Software Engineering. arXiv:2006.14244 [cs.SE]

[191] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2020. A Self-Attentional Neural Architecture for Code Completion with Multi-Task Learning. In *Proceedings of the 28th International Conference on Program Comprehension* (Seoul, Republic of Korea) *(ICPC '20)*. 37–47. https://doi.org/10.1145/3387904.3389261

[192] F. Liu, G. Li, Y. Zhao, and Z. Jin. 2020. Multi-task Learning based Pre-trained Language Model for Code Completion. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 473–485.

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

[194] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the Efficiency of Test Suite Based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. 615?–627. https://doi.org/10.1145/3377811.3380338

[195] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (Jul. 2019), 1044–1051. https://doi.org/10.1609/aaai.v33i01.33011044

[196] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-Machine-Translation-Based Commit Message Generation: How Far Are We?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. 373–384. https://doi.org/10.1145/3238147.3238190

[197] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. 298–312. https://doi.org/10.1145/2837614.2837617

[198] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi. 2010. UCI Source Code Data Sets. http://www.ics.uci.edu/$\sim$lopes/datasets/

[199] Frederico Caram Luiz, Bruno Rafael de Oliveira Rodrigues, and Fernando Silva Parreiras. 2019. Machine Learning Techniques for Code Smells Detection: An Empirical Experiment on a Highly Imbalanced Setup. In *Proceedings of the XV Brazilian Symposium on Information Systems* (Aracaju, Brazil) *(SBSI'19)*. Article 65, 8 pages. https://doi.org/10.1145/3330204.3330275

[200] Savanna Lujan, Fabiano Pecorelli, Fabio Palomba, Andrea De Lucia, and Valentina Lenarduzzi. 2020. A Preliminary Study on the Adequacy of Static Analysis Warnings with Respect to Code Smell Prediction. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation* (Virtual, USA) *(MaLTeSQuE 2020)*. 1–6. https://doi.org/10.1145/3416505.3423559

[201] Minh-Thang Luong, Eugene Brevdo, and Rui Zhao. 2017. Neural Machine Translation (seq2seq) Tutorial. *https://github.com/tensorflow/nmt* (2017).

[202] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. 101–114. https://doi.org/10.1145/3395363.3397369

[203] Yuzhan Ma, Sarah Fakhoury, Michael Christensen, Venera Arnaoudova, Waleed Zogaan, and Mehdi Mirakhorli. 2018. Automatic Classification of Software Artifacts in Open-Source Applications. In *Proceedings of the 15th International Conference on Mining Software*

Repositories (Gothenburg, Sweden) *(MSR '18)*. 414–425. https://doi.org/10.1145/3196398.3196446

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

[205] Z. Ma, H. Ge, Y. Liu, M. Zhao, and J. Ma. 2019. A Combination Method for Android Malware Detection Based on Control Flow Graphs and Machine Learning Algorithms. *IEEE Access* 7 (2019), 21235–21245. https://doi.org/10.1109/ACCESS.2019.2896003

[206] Janaki T. Madhavan and E. James Whitehead. 2007. Predicting Buggy Changes inside an Integrated Development Environment. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology EXchange* (Montreal, Quebec, Canada) *(eclipse '07)*. 36–40. https://doi.org/10.1145/1328279.1328287

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

[208] Ruchika Malhotra. 2014. Comparative analysis of statistical and machine learning methods for predicting faulty modules. *Applied Soft Computing* 21 (2014), 286 – 297. https://doi.org/10.1016/j.asoc.2014.03.032

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

[210] R. Malhotra and Rupender Jangra. 2017. Prediction & Assessment of Change Prone Classes Using Statistical & Machine Learning Techniques. *Journal of Information Processing Systems* 13 (01 2017), 778–804. https://doi.org/10.3745/JIPS.04.0013

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

[212] R. S. Malik, J. Patra, and M. Pradel. 2019. NL2Type: Inferring JavaScript Function Types from Natural Language Information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 304–315. https://doi.org/10.1109/ICSE.2019.00045

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

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

[215] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. 2014. Automatic Detection and Correction of Web Application Vulnerabilities Using Data Mining to Predict False Positives. In *Proceedings of the 23rd International Conference on World Wide Web* (Seoul, Korea) *(WWW '14)*. 63–74. https://doi.org/10.1145/2566486.2568024

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

[217] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning to Repair Compilation Errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. 925–936. https://doi.org/10.1145/3338906.3340455

[218] Mohammad Y. Mhawish and Manjari Gupta. 2020. Predicting Code Smells and Analysis of Predictions: Using Machine Learning Techniques and Software Metrics. *J. Comput. Sci. Technol.* 35 (2020), 1428–1445.

[219] Nikola Milosevic, Ali Dehghantanha, and Kim-Kwang Raymond Choo. 2017. Machine learning aided Android malware classification. *Computers & Electrical Engineering* 61 (2017), 266 – 274. https://doi.org/10.1016/j.compeleceng.2017.02.013

[220] Robert Moskovitch, Nir Nissim, and Yuval Elovici. 2009. Malicious Code Detection Using Active Learning. In *Privacy, Security, and Trust in KDD*, Francesco Bonchi, Elena Ferrari, Wei Jiang, and Bradley Malin (Eds.). 74–91.

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

[222] Golam Mostaeen, Jeffrey Svajlenko, Banani Roy, Chanchal Roy, and Kevin Schneider. 2018. [Research Paper] On the Use of Machine Learning Techniques Towards the Design of Cloud Based Automatic Code Clone Validation Tools. 155–164. https://doi.org/10.1109/SCAM.2018.00025

[223] G. Mostaeen, J. Svajlenko, B. Roy, C. K. Roy, and K. A. Schneider. 2018. [Research Paper] On the Use of Machine Learning Techniques Towards the Design of Cloud Based Automatic Code Clone Validation Tools. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 155–164. https://doi.org/10.1109/SCAM.2018.00025

[224] Golam Mostaeen, Jeffrey Svajlenko, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. CloneCognition: Machine Learning Based Code Clone Validation Tool. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. 1105–1109. https://doi.org/10.1145/3338906.3341182

[225] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona) *(AAAI'16)*. 1287–1293.

[226] Aravind Nair, Karl Meinke, and Sigrid Eldh. 2019. Leveraging Mutants for Automatic Prediction of Metamorphic Relations Using Machine Learning. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation* (Tallinn, Estonia) *(MaLTeSQuE 2019)*. 1–6. https://doi.org/10.1145/3340482.3342741

[227] N. Nazar, Y. Hu, and He Jiang. 2016. Summarizing Software Artifacts: A Literature Review. *Journal of Computer Science and Technology* 31 (2016), 883–909.

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

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

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

[231] Duc-Man Nguyen, Hoang-Nhat Do, Quyet-Thang Huynh, Dinh-Thien Vo, and Nhu-Hang Ha. 2019. Shinobi: A Novel Approach for Context-Driven Testing (CDT) Using Heuristics and Machine Learning for Web Applications. In *Industrial Networks and Intelligent Systems*, Trung Q Duong and Nguyen-Son Vo (Eds.). 86–102.

[232] A. S. Nyamawe, H. Liu, N. Niu, Q. Umer, and Z. Niu. 2019. Automated Recommendation of Software Refactorings Based on Feature Requests. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*. 187–198. https://doi.org/10.1109/RE.2019.00029

[233] 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.

[234] 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

[235] Daniel Oliveira, Wesley K. G. Assunção, Leonardo Souza, Willian Oizumi, Alessandro Garcia, and Baldoino Fonseca. 2020. Applying Machine Learning to Customized Smell Detection: A Multi-Project Study *(SBES '20)*. 233–242. https://doi.org/10.1145/3422392.3422427

[236] 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

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

[238] 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

[239] 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

[240] 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

[241] 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

[242] 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.

[243] 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)*. 1–10. https://doi.org/10.1109/LADC48089.2019.8995685

[244] 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

[245] 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 2020)*. 3–12. https://doi.org/10.1145/3416506.3423576

[246] Goran Piskachev, Lisa Nguyen Quang Do, and Eric Bodden. 2019. Codebase-Adaptive Detection of Security-Relevant Methods. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. 181–191.

https://doi.org/10.1145/3293882.3330556

[247] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. 383–387. https://doi.org/10.1109/MSR.2019.00064

[248] C. L. Prabha and N. Shivakumar. 2020. Software Defect Prediction Using Machine Learning Techniques. In *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)*. 728–733. https://doi.org/10.1109/ICOEI48184.2020.9142909

[249] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 147 (October 2018), 25 pages. https://doi.org/10.1145/3276517

[250] Michael Prince. 2004. Does active learning work? A review of the research. *Journal of engineering education* 93, 3 (2004), 223–231.

[251] N. Pritam, M. Khari, L. Hoang Son, R. Kumar, S. Jha, I. Priyadarshini, M. Abdel-Basset, and H. Viet Long. 2019. Assessment of Code Smell for Predicting Class Change Proneness Using Machine Learning. *IEEE Access* 7 (2019), 37414–37425. https://doi.org/10.1109/ACCESS.2019.2905133

[252] Sebastian Proksch, Johannes Lerch, and Mira Mezini. 2015. Intelligent Code Completion with Bayesian Networks. *ACM Trans. Softw. Eng. Methodol.* 25, 1, Article 3 (December 2015), 31 pages. https://doi.org/10.1145/2744200

[253] Md Rafiqul Islam Rabin, Arjun Mukherjee, Omprakash Gnawali, and Mohammad Amin Alipour. 2020. Towards Demystifying Dimensions of Source Code Embeddings. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages* (Virtual, USA) *(RL+SE&PL 2020)*. 29–38. https://doi.org/10.1145/3416506.3423580

[254] Akond Rahman, Priysha Pradhan, Asif Partho, and Laurie Williams. 2017. Predicting Android Application Security and Privacy Risk with Static Code Metrics. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems* (Buenos Aires, Argentina) *(MOBILESoft '17)*. 149–153. https://doi.org/10.1109/MOBILESoft.2017.14

[255] M. Rahman, Yutaka Watanobe, and K. Nakamura. 2020. A Neural Network Based Intelligent Support Model for Program Code Completion. *Sci. Program.* 2020 (2020), 7426461:1–7426461:18. https://doi.org/10.1155/2020/7426461

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

[257] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. *SIGPLAN Not.* 51, 10 (October 2016), 731–747. https://doi.org/10.1145/3022671.2984041

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

[259] Athos Ribeiro, Paulo Meirelles, Nelson Lago, and Fabio Kon. 2019. Ranking Warnings from Multiple Source Code Static Analyzers via Ensemble Learning. In *Proceedings of the 15th International Symposium on Open Collaboration* (Skövde, Sweden) *(OpenSym '19)*. Article 5, 10 pages. https://doi.org/10.1145/3306446.3340828

[260] 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 Vasiljevienė (Eds.). 273–284.

[261] 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

[262] 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

[263] 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

[264] 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

[265] 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

[266] 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

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

[268] 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

[269] Anush Sankaran, Rahul Aralikatte, Senthil Mani, Shreya Khare, Naveen Panwar, and Neelamadhav 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

[270] 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.2018.8330219

[271] 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 Baruque, Héctor Quintián, José Luis Calvo, Javier Sedano, and Emilio Corchado (Eds.). 271–280.

[272] F. Sarro, S. Di Martino, F. Ferrucci, and C. Gravino. 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

[273] 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

[274] Max Eric Henry Schumacher, Kim Tuyen Le, and Artur Andrzejak. 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* (Seoul, Republic of Korea) *(ICSEW'20)*. 476–482. https://doi.org/10.1145/3387940.3391489

[275] R. Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. In *30th USENIX Security Symposium (USENIX Security 21)*.

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

[277] Burr Settles. 2009. Active learning literature survey. (2009).

[278] 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.istr.2009.03.003 Malware.

[279] 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

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

[281] Tushar Sharma. 2019. CodeSplit for C#. https://doi.org/10.5281/zenodo.2566905

[282] Tushar Sharma. 2019. CodeSplitJava. https://doi.org/10.5281/zenodo.2566865 https://github.com/tushartushar/CodeSplitJava.

[283] Tushar Sharma, Vasiliki Efstathiou, 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

[284] 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/10.1145/2896935.2896938

[285] 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

[286] Andrey Shedko, 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

[287] 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.

[288] 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

[289] 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.cola.2020.100979

[290] 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

[291] 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

[292] 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

[293] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving Code Search with Co-Attentive Representation Learning. 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

[294] 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/CONFLUENCE.2017.7943255

[295] 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.2017.8342425

[296] 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

[297] 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.

[298] 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.

[299] 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

[300] A. J. Stein, L. Kapllani, S. Mancoridis, and R. Greenstadt. 2020. Exploring Paraphrasing Techniques on Formal Language for Generating Semantics Preserving Source Code Transformations. In *2020 IEEE 14th International Conference on Semantic Computing (ICSC)*. 242–248. https://doi.org/10.1109/ICSC.2020.00051

[301] 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

[302] Yulei Sui, Xiao Cheng, Guanqin 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

[303] 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.

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

[305] 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

[306] 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

[307] 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 & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. 2727–2735. https://doi.org/10.1145/3292500.3330699

[308] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, 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.

[309] Tomohiko Takagi, Takeshi Utsumi, and Zengo Furukawa. 2013. Back-to-Back Testing Framework Using a Machine Learning Method. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing 2012*, Roger Lee (Ed.). 27–36.

[310] Salvador Tamarit, Guillermo Vigueras, Manuel Carro, and Julio Mari
textasciitilde no. 2017. Machine Learning-Driven Automatic Program Transformation to Increase Performance in Heterogeneous Architectures. In *Tools for High Performance Computing 2016*, Christoph Niethammer, José Gracia, Tobias Hilbrich, Andreas Knüpfer, Michael M. Resch, and Wolfgang E. Nagel (Eds.). 115–140.

[311] K. Terada and Y. Watanobe. 2019. Code Completion for Programming Education based on Recurrent Neural Network. In *2019 IEEE 11th International Workshop on Computational Intelligence and Applications (IWCIA)*. 109–114. https://doi.org/10.1109/IWCIA47330.2019.8955090

[312] H. Thaller, L. Linsbauer, and A. Egyed. 2019. Feature Maps: A Comprehensible Software Representation for Design Pattern Detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 207–217. https://doi.org/10.1109/SANER.2019.8667978

[313] 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/ECTIDAMTNCON48261.2020.9090717

[314] 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.

[315] Irene Tollin, 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

[316] A. K. Tripathi and K. Sharma. 2014. Optimizing testing efforts based on change proneness through machine learning techniques. In *2014 6th IEEE Power India International Conference (PIICON)*. 1–4. https://doi.org/10.1109/POWERI.2014.7117742

[317] 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.

[318] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk. 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

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

[320] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches 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

[321] 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 2020)*. Article 8, 11 pages. https://doi.org/10.1145/3385032.3385042

[322] 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

[323] 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.

[324] 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)*. 632–638. https://doi.org/10.1145/775047.775141

[325] 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/AITEST49225.2020.00020

[326] Hoang Van Thuy, Phan Viet Anh, and Nguyen Xuan Hoai. 2018. Automated Large Program Repair Based on Big Code. In *Proceedings of the Ninth International Symposium on Information and Communication Technology* (Danang City, Viet Nam) *(SoICT 2018)*. 375?–381. https://doi.org/10.1145/3287921.3287958

[327] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural Program Repair by Jointly Learning to Localize and Repair.

[328] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł 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/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[329] Nickolay Viuginov 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) *(MaLTeSQuE 2019)*. 31–36. https://doi.org/10.1145/3340482.3342746

[330] 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

[331] 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/10.1145/3238147.3238206

[332] 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

[333] 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

[334] 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

[335] 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) *(ISCSIC 2019)*. Article 16, 9 pages. https://doi.org/10.1145/3386164.3389083

[336] 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* (Austin, Texas) *(ICSE '16)*. 297–308. https://doi.org/10.1145/2884781.2884804

[337] 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

[338] 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

[339] Wenhan 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, Article 31 (September 2020), 23 pages. https://doi.org/10.1145/3409331

[340] W. Wang, Y. Zhang, Y. Sui, Y. Wan, 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

[341] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang 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

[342] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. 87–98. https://doi.org/10.1145/2970276.2970326

[343] Liwei Wu, Fei Li, Youhua Wu, and Tao Zheng. 2020. GGF: A Graph-Based Method for Programming Language Syntax Error Correction. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC '20)*. Association for Computing Machinery, 139—-148. https://doi.org/10.1145/3387904.3389252

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

[345] R. Xie, W. Ye, J. Sun, and S. Zhang. 2021. Exploiting Method Names to Improve Code Summarization: A Deliberation Multi-Task Learning Approach. In *2021 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) (ICPC)*. 138–148. https://doi.org/10.1109/ICPC52881.2021.00022

[346] Yingfei Xiong, Bo Wang, Guirong Fu, and Linfei Zang. 2018. Learning to Synthesize. In *Proceedings of the 4th International Workshop on Genetic Improvement Workshop* (Gothenburg, Sweden) *(GI '18)*. 37–44. https://doi.org/10.1145/3194810.3194816

[347] Sihan Xu, Sen Zhang, Weijing Wang, Xinya Cao, Chenkai Guo, and Jing Xu. 2019. Method Name Suggestion with Hierarchical Attention Networks. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Cascais, Portugal) *(PEPM 2019)*. 10–21. https://doi.org/10.1145/3294032.3294079

[348] Eran Yahav. 2018. From Programs to Interpretable Deep Models and Back. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). 27–37.

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

[350] Mutian Yang, Jingzheng Wu, Shouling Ji, Tianyue Luo, and Yanjun Wu. 2018. Pre-Patch: Find Hidden Threats in Open Software Based on Machine Learning Method. In *Services – SERVICES 2018*, Alvin Yang, Siva Kantamneni, Ying Li, Awel Dico, Xiangang Chen, Rajesh Subramanyan, and Liang-Jie Zhang (Eds.). 48–65.

[351] Yixiao Yang, Xiang Chen, and Jiaguang Sun. 2019. Improve Language Modeling for Code Completion Through Learning General Token Repetition of Source Code with Optimized Memory. *International Journal of Software Engineering and Knowledge Engineering* 29, 11n12 (2019), 1801–1818. https://doi.org/10.1142/S0218194019400229 arXiv:https://doi.org/10.1142/S0218194019400229

[352] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, and M. Zhang. 2021. A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts. In *2021 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) (ICPC)*. 1–12. https://doi.org/10.1109/ICPC52881.2021.00010

[353] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. CoaCor: Code Annotation for Code Retrieval with Reinforcement Learning. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. 2203–2214. https://doi.org/10.1145/3308558.3313632

[354] Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. 2018. StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1693–1703. https://doi.org/10.1145/3178876.3186081

[355] Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. 2020. Leveraging Code Generation to Improve Code Retrieval and Summarization via Dual Learning. In *Proceedings of The Web Conference 2020* (Taipei, Taiwan) *(WWW '20)*. 2309–2319. https://doi.org/10.1145/3366423.3380295

[356] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 476–486. https://doi.org/10.1145/3196398.3196408

[357] 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

[358] 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

[359] 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* (Seoul, South Korea) *(ICSE '20)*. 1385–1397. https://doi.org/10.1145/3377811.3380383

[360] 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

[361] 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

[362] 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/ICSE43902.2021.00129

[363] 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

[364] 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

[365] 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

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

[367] Chaoliang Zhong, Ming Yang, and Jun Sun. 2019. JavaScript Code Suggestion Based on Deep Learning. In *Proceedings of the 2019 3rd International Conference on Innovation in Artificial Intelligence* (Suzhou, China) *(ICIAI 2019)*. 145–149. https://doi.org/10.1145/3319921.3319922

[368] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. 95–109. https://doi.org/10.1109/SP.2012.16

[369] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. 9–9. https://doi.org/10.1109/PROMISE.2007.10

A Abbreviations of ML techniques

Table 3. Machine learning techniques

Acronym	Full form
A3C	Asynchronous Advantage Actor-Critic
AB	AdaBoost
AE	Autoencoder
AIS	Artificial Immune Systems
ANFIS	Adaptive Neuro-Fuzzy Inference System
ANN	Artifical Neural Network
ARM	Association Rule Mining
B	Bagging
BERT	Bidirectional Encoder Representations from Transformers
Bi-GRU	Bidirectional Gated Recurrent Unit
Bi-LSTM	Bi-Long Sort-Term Memory
Bi-RNN	Bidirectional Recurrent Neural Network
BiNN	Bilateral Neural Network
BMN	Best Matching Neighbours
BN	Bayes Net
BP-ANN	Back-propagation Artificial Neural Network
BR	Binary Relevance
BTE	Best-in-training-based ensemble
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
ELM	Extreme Learning Machine
EM	Expectation Minimization
EN-DE	Encoder-Decoder
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
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
GPT-C	Generative Pre-trained Transformer for Code
GRASSHOPER	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
Lasso	Lasso
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 Naive 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
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	Recuurent 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