Predictive Models in Software Engineering: Challenges and Opportunities

YANMING YANG, Zhejiang University, China
XIN XIA *, Software Engineering Application Technology Lab, Huawei
DAVID LO, Singapore Management University, Singapore , Singapore
TINGTING BI, Monash University, Australia
JOHN GRUNDY, Monash University, Australia
XIAOHU YANG, Zhejiang University, China

Predictive models are one of the most important techniques that are widely applied in many areas of software engineering. There have been a large number of primary studies that apply predictive models and that present well-performed studies in various research domains, including software requirements, software design and development, testing and debugging and software maintenance. This paper is a first attempt to systematically organize knowledge in this area by surveying a body of 421 papers on predictive models published between 2009 and 2020. We describe the key models and approaches used, classify the different models, summarize the range of key application areas, and analyze research results. Based on our findings, we also propose a set of current challenges that still need to be addressed in future work and provide a proposed research road map for these opportunities.

CCS Concepts: • **Software and its engineering → Software development techniques**.

Additional Key Words and Phrases: Predictive models, machine learning, deep learning, software engineering, survey

ACM Reference Format:
Yanming Yang, Xin Xia, David Lo, Tingting Bi, John Grundy, and Xiaohu Yang . 2021. Predictive Models in Software Engineering: Challenges and Opportunities. 1, 1 (December 2021), 75 pages. https://doi.org/mmmm/mmmmmmmmm

1 INTRODUCTION

Researchers have developed automated methodologies to improve software engineering tasks. Key reasons are usually to save developer time and effort and to improve the software quality in terms of stability, reliability, and security. Many of such studies have resulted in great improvements in various tasks [36, 192, 309, 319, 355, 435].

A key technology, the *predictive model*, has been developed to solve a range of software engineering problems over several decades. The use of predictive models is in fact becoming increasingly

*Corresponding author: Xin Xia

Authors' addresses: Yanming Yang Zhejiang University, China, yanmingyang@zju.edu.cn; Xin Xia Software Engineering Application Technology Lab, Huawei, xin.xia@acm.org; David Lo Singapore Management University, Singapore, Singapore, davidlo@smu.edu.sg; Tingting Bi Monash University, Australia, Tingting.Bi@monash.edu; John Grundy Monash University, Australia, John.Grundy@monash.edu; Xiaohu Yang Zhejiang University, China, yangxh@zju.edu.cn.

XXXX-XXXX/2021/12-ART $15.00
https://doi.org/mmmm/mmmmmmmmm

popular in a wide range of software engineering research areas. Predictive models are built based on different types of datasets – such as software requirements, APIs, bug reports, source code and run-time data – and provide a final output according to distinct features found in the data. There are various predictive models commonly used in software engineering tasks that contribute to improving the efficiency of development processes and software quality. Common ones include defect prediction [355], API issue classification [190], and code smell detection [273].

Despite numerous studies on predictive models in software engineering, to the best of our knowledge, there has been no systematic study to analyze the use of and demonstrated the potential value of and current challenges of using predictive models in software engineering. There is no clear answer as to which software engineering tasks predictive models can be best applied and how to best go about leveraging the right predictive models for these tasks. Answering these questions would be beneficial for both practitioners and researchers, in order to make informed decisions to solve a problem or conduct research using predictive models.

This paper contributes to the research on predictive models bu performing a comprehensive systematic survey of the domain. The types and number of SE tasks involved are incredibly huge since the predictive model has a general definition. For example, a predictive model can be used to provide a result of forecasting for the regression, classification, recommendation, and generation task, respectively. Therefore, we are unable to summarize and analyze every related study due to the huge number of ones applying predictive models in SE. For a better analysis, we adopted two strategies to narrow the research scope as well as ensure the research quality is still at a high level. First, in this work, we only focused on the classification task as a majority of studies used a predictive model for classification, and classification involves more different SE tasks compared with another three problem types, i.e., regression, recommendation, and generation tasks. Meanwhile, to ensure the quality of our work while narrowing the scope of the research, we selected 15 high-quality and leading SE publication venues including nine conference proceedings and six journals. We collected relevant studies published between 2009 and 2020 to form a small SE research community that covers not only most of the important SE research directions but high-quality studies. The search process for collecting relevant studies includes three steps. First, we composed a search string that consists of several key search terms and used the search string to perform searches on the title and abstract of each paper. We collected over 2,000 papers containing our target search terms. We then filtered the irrelevant studies according to the inclusion and exclusion criteria and removed the duplicated studies. Finally, we carefully read the abstract and introduction of each paper to validate its relevance to our study. After the search and filtering process, we identified 421 relevant primary studies. We found that predictive models have been widely applied to a variety of software engineering tasks. Common ones include requirements classification, code change detection and malware detection. We grouped predictive model usage in software engineering into six research domains according to the software development lifecycle (SDLC) – software requirements, software design, software implementation, testing and debugging, software maintenance, and software professional practice knowledge area. We further summarized and analyzed these studies in each domain. After that, we identified a set of remaining research and practice challenges and new research directions that we think should be investigated in the future. To the best of our knowledge, we are the first to perform such a systematic review on the use of predictive models in software engineering domain.

This paper makes the following key contributions:

(1) we present a comprehensive survey on predictive models covering 421 primary study papers between 2009 and 2020;
(2) we analyze these 421 primary studies and characterize them;

(3) we conclude 148 different predictive models that have been used in SE between 2009 and 2020, and then summarize the top 14 common evaluation measures;
(4) we summarize the research domains in which the predictive model has been applied and also common measures for model evaluation in different SE tasks;
(5) we discuss distinct technical challenges of using predictive models in software engineering; and
(6) we outline key future avenues for research on predictive models in software engineering.

The remainder of this paper is organized as follows. Section 2 briefly introduces the workflow of predictive models. Section 3 presents our study methodology. Section 4 investigates the evolution and distribution of the selected primary studies using predictive models for software engineering tasks, and Section 5 gives a classification and research distribution of these predictive models and also summarizes the top 10 common evaluation measures for predictive models. Section 6 summarizes and analyzes the application of predictive models in software engineering, and a discussion of the threats that could affect the validity of our findings is presented in Section 7. Section 8 discusses the challenges that still need to be addressed in future work and give outlines a research road map of potential research opportunities in this domain. Section 9 provides a summary of the key conclusions of this work.

2 PREDICTIVE MODELS FOR CLASSIFICATION

In classification tasks, a predictive model is usually treated as a black box that automatically assigns a class label when presented with the feature set of an unknown record, illustrated in Fig. 1. Instances in the input dataset can be assigned to one of several predefined categories by building such a predictive model. The predictive model is usually trained with a representative input dataset, then applied to target input datasets. The workflow of using a predictive model can be described as a mathematical problem of learning a target function (f) that maps each feature set (x) in a dataset (X) to one class label (y). The job of predictive models is to find the best target function (f_m).

There are four critical components when building such a predictive model:

Datasets: As the most basic component of predictive models, the dataset has a large impact on a model's performance. Low-quality datasets with noise and mislabeling may lead predictive models to provide (very) wrong experimental results, even if the process of model selection and training is effective.

Different types of datasets are used when performing different software engineering tasks with predictive models. For instance, studies may use source code, bug reports, or requirement documents as datasets of predictive models for defect prediction, bug report classification, and requirements-related knowledge classification respectively. In general, different datasets have different properties. These include scale, distribution, bias, quality, representativeness, sparseness and so on.

Features: In building predictive models, features (or attributes) in datasets are essential in the model training phase. The goal of a predictive model is essentially to learn a target function by analyzing different feature sets in given input datasets. Thus a good feature set can allow predictive models to learn the potential patterns in datasets and thus to output correct labels effectively. In order to construct high-quality feature sets, feature selection is an important optimization technique in building predictive models. Better feature selection strategies can improve the accuracy of results, reduce overfitting, as well as reduce training time.

Model Building Algorithms: A predictive model can be implemented by using different algorithms. For instance, J48, C4.5 and CART are commonly used algorithms for building Decision Tree-based predictive models. A number of relatively new deep learning-based algorithms and architectures, such as RNN and CNN-based networks, have been applied as predictive models to

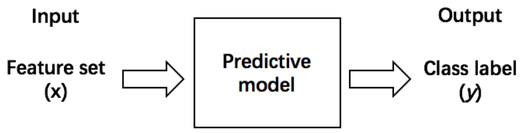


Fig. 1. Predictive model as the task of mapping an input feature set x into its class label y.

many software engineering tasks. Some predictive model algorithms have been introduced for specific software engineering problems, by enhancing existing algorithms, or providing new model capabilities. For example, many large and complex software-intensive systems widely used logs for troubleshooting, and thus Zhang et al. [437] present a tool, called LogRobust, to detect system anomalies by analyzing log messages. They adopted Bi-LSTM, a variant of LSTM model, to capture the contextual information in log sequences, and combined attention mechanism to strengthen the ability of automatically learning the importance of different log events. Liu et al. [198] concentrated on detecting feature envy, one of the most common code smells, and tailored a deep learning model to consider both textual input (method name or class name) and numerical input (the distance between a method and a class) by merging two CNN models.

Model Performance Measures: Many widely-used evaluation metrics, such as recall, precision, F-measure, accuracy, and AUC, have been introduced and applied in software engineering studies. This is because different tasks may require different metrics to evaluate the effectiveness of their proposed predictive models. On the other hand, some of the less common evaluation metrics, such as Balance, G-measure, and specificity, also appear in a few studies. We will present an analysis of various evaluation metrics for predictive models in Section 5.

3 METHODOLOGY

To perform a comprehensive systematic review of the use of predictive models in software engineering, we followed the systematic review guidelines provided by Kitchenham and Charters [155] and Petersen et al. [292].

3.1 Research Questions

The aim of this paper is to summarize, classify, analyze and propose research directions based on empirical evidence concerning the different predictive modeling techniques and task domains involved. To achieve this, we define three research questions as shown in Table 1 and give the motivation behind each question.

RQ1 will analyze the distribution of publications on predictive models over the last decade to give an overview of the trend in software engineering research. RQ2 will provide a classification and distribution of predictive models used in software engineering and identify what are the common evaluation metrics/measures. RQ3 will explore where and how predictive models have been applied for specific software engineering tasks.

3.2 Literature Search and Selection

To identify the relevant studies, we determined a search string which composes with a set of search terms that are helpful to identify SE tasks using predictive models. After considering the alternative spellings and synonyms for these search terms [125], they were combined with logical ORs, forming the complete search string. The search terms are listed as follows:

Table 1. Research Questions and Motivations

RQs	Research Question	Motivation
RQ1	What are the trends in the primary studies on use of predictive models in SE?	The basic information (e.g., authors, publication year, affiliations) of the primary study papers can be informative. We wanted to understand where primary study papers on SE uses of predictive models are being published and any trends that can be observed. The goal of this RQ is to investigate these publication trends and distribution of the primary studies.
RQ2	Which predictive models are applied to what software engineering tasks?	Various predictive models are used to in the context of different software engineering tasks. It is necessary to generalize about predictive models so that researchers can select the appropriate model when researching in certain domains. The goal of this question is to determine which predictive models are frequently applied in software engineering, and to develop a classification for them.
RQ3	In what software engineering domains and applications have predictive models been applied?	Although predictive models have been applied to broad application scenarios in software engineering, there does not exist any comprehensive study that summarizes these research domains and applications using different models. This RQ aims to find and report on such a domain analysis.

("predict" OR "predictive model" OR "prediction model")* [1]

When initially collecting relevant papers, we noticed that tens of thousands of studies that used predictive models to addressing SE tasks have been published in various venues. It is difficult for us to make a comprehensive analysis on each study. To solve this problem, we narrowed the number of publication venues for keeping the number of related papers within a reasonable and researchable range.

To cover the research directions and types of papers in SE as fully as possible and retain as many high-quality studies as possible, we extracted the subject terms describing the topics of papers published in each venue from DBLP from 2009 to 2020 and analyzed the research areas in SE the journal venues contribute to. We observed that leading conferences and journals have covered important SE activities and most SE research directions. The hot research directions or new technologies (e.g., deep learning) in SE are also widely discussed in top quality and leading journals and conferences. Besides, we noticed that the amount of studies in different SE activities and research directions conforms to the research trend in whole filed of SE. For instance, in most venues, the number of studies focusing on addressing specific problems in software testing and maintenance accounts for a large proportion, being consistent with the distribution of studies in SE.

[1]Since the Digital Library can support search terms that are not case sensitive and not "whole words only", we use the stemmed term to represent all forms related to them (e.g., "predict*" can search for "predict", "predicting", "prediction" as well as "predictive").

Therefore, we selected 15 high-quality and leading publication venues, including nine conferences and six journals, to form a small SE research community, covering most research directions and following the overall research trend in SE. We then used the search string to identify relevant studies among these SE publication venues. Table 5 details 15 leading publication venues our work focuses on, including nine conference proceedings and six journals.

Table 2. Publication venues for manual search.

No.	Acronym	Full name	No.	Acronym	Full name
1.	ASE	IEEE/ACM International Conference Automated Software Engineering	10.	TSE	IEEE Transactions on Software Engineering
2.	ESEC/FSE	ACM SIGSOFT Symposium on the Foundation of Software Engineering/European Software Engineering Conference	11.	TOSEM	ACM Transactions on Software Engineering and Methodology
3.	ICSE	ACM/IEEE International Conference on Software Engineering	12.	ESE	Empirical Software Engineering
4.	ESEM	ACM/IEEE International Symposium on Empirical Software Engineering and Measurement	13.	IST	Information and Software Technology
5.	ICPC	IEEE International Conference on Program Comprehension	14.	JSS	Journal of Systems and Software
6.	ICSME	IEEE International Conference on Software Maintenance and Evolution	15.	TRel	IEEE Transactions on Reliability
7.	MSR	IEEE Working Conference on Mining Software Repositories			
8.	ISSTA	ACM SIGSOFT International Symposium on Software Testing and Analysis			
9.	SANER	IEEE International Conference on Software Analysis, Evolution and Reengineering			

Following previous survey study approaches [125, 133], for each publication venue, we used our search string to perform search for related studies published between 2009 and 2020. We obtained 2410 candidate studies. Table 3 shows the initial search results in different publication venues after searching by using the search string between 2009 and September, 2021.

3.3 Inclusion and Exclusion Criteria

A number of low quality studies (e.g., non-published manuscripts and grey literature) are found when searching for the relevant papers with search engines. We also wanted only the most comprehensive or latest version of repeated studies in the candidate set. For example, we removed a conference paper if it was extended in a follow-on journal paper. Thus we applied a set of inclusion and exclusion criteria to select the studies with strong relevance, filter out irrelevant and duplicated studies.

The following inclusion and exclusion criteria were used:

✔ The paper must apply one or more predictive models to address software engineering tasks.

✔ The paper must be a peer reviewed full research paper published in a conference proceedings or a journal.

✘ Papers less than six pages are not considered.

✘ Short and workshop papers are discarded.

Table 3. The initial search results in different publication venues.

Conference venue		Journal venue	
Acronym	**# Studies**	**Acronym**	**# Studies**
ASE	101	TSE	131
ESEC/FSE	40	TOSEM	13
ICSE	174	ESE	981
ESEM	74	IST	145
ICPC	35	JSS	196
ICSME	108	TRel	225
ISSTA	13		
MSR	120		
SANER	54		

✘ The predictive models used in studies cannot be baseline approaches.
✘ Conference version of a study that has an extended journal version is not considered.

The inclusion and exclusion criteria were piloted and applied by the first and fourth authors by assessing 45 randomly selected papers from the initial set. We used Cohen's kappa coefficient to measure the reliability of the inclusion and exclusion decisions. The agreement rate in the pilot study was "substantial" (0.72). We also perform the assessment in the full list of identified papers, and Cohen's kappa score was "substantial" (0.68). The first and fourth authors reached a consensus through discussions when encountering disagreements. In case disagreements were not resolved, a third researcher who is not the author of the paper was invited to make the final decision.

After discarding irrelevant and duplicated studies according to inclusion and exclusion criteria, in the third step, we are careful to read the abstract and introduction section for each candidate study to double-check its relevance to the predictive model. Then, we only select the studies that utilized predictive modeling techniques for classification tasks.

In addition, to ensure that all relevant studies are involved in our dataset as much as possible, we also adopted the snowballing strategy in each publication venue to find out certain related studies but not found by the search string. Finally, we totally collected 421 relevant studies, including eight ones found by snowballing. The publication distribution in different venues is present in Section 4.

3.4 Data Extraction and Collection

After carefully reading all 190 papers in full, we extracted the required data and conducted detailed analysis to answer our three research questions. The detailed information is summarized in Table 4. Data collection mainly concentrated on three aspects: the fundamental information of each paper, some contents about predictive models and the research domain to which each study belongs. In order to prevent data loss and avoid mistakes as much as possible, data collection was performed by the first and fourth authors [2] and then the results were verified by other researchers who are not co-authors of this paper.

4 RQ1: WHAT ARE THE TRENDS IN THE PRIMARY STUDIES ON USE OF PREDICTIVE MODELS IN SE?

To discuss the emerging trends in predictive model use in SE, in this section we present an analysis of primary studies from three perspectives according to the fundamental information presented in these studies.

[2]Thanks to the two researchers who helped to collect the dataset. They are studying at Monash University and Dalian University of Technology, respectively.

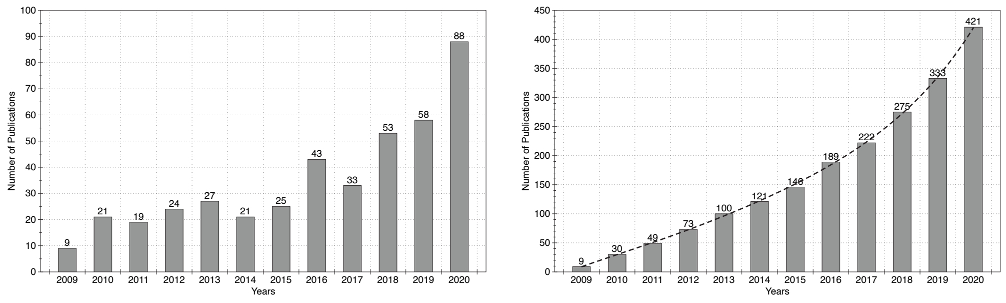
Table 4. Data Collection for Research Questions

RQs	Types of Data to be Extracted
RQ1	Fundamental information for each paper (publication year, author name, type of paper and affiliation).
RQ2	Description, predictive modeling techniques, evaluation measures.
RQ3	Background, motivation, application scenario, and research topic of each study.

4.1 Publication trends in research on predictive models in Software Engineering

To better understand the publication trends of studies related to predictive models, we summarized the publication number of relevant studies per year and analyzed the growth of publication number as time goes by.

Fig. 2(a) shows the number of papers published between January 1st, 2009 and December 31st, 2020. It is clear that only nine relevant studies were published in 2009, and it has the lowest number of publications compared with other years. A stable trend of the publication number appears between 2010 and 2015 since around 20 to 30 relevant studies were released in each of those years. The publication number shows a clear trend upwards from 2016 to 2020, despite a little decrease occurs in 2017. The potential reason for this increasing trend is that Deep Learning (DL) gradually becomes famous in 2016, causing more studies started to adopt DL techniques as predictive models for various classification tasks in SE.

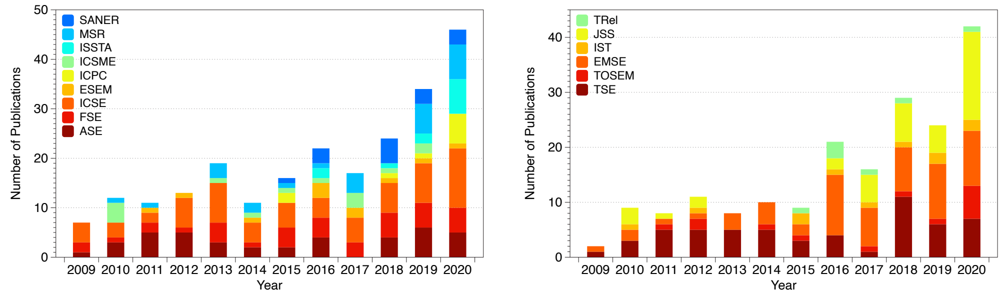


(a) Number of publications per year.

(b) Cumulative number of publications per year.

Fig. 2. Publication trends of predictive models between 2009 and 2020.

To clear the growth trend of relevant studies, an analysis of the cumulative publications is shown in Fig. 2(b). We used a polynomial function to fit the cumulative number of publications, revealing the publication trend between 2009 and 2020. It can be observed that The (R^2) amounts to 0.99953 and the slope of the curve increases remarkably between 2009 and 2020. This indicates that research studies using predictive models are likely to continue to experience a strong growth in the future. According to the trend of this curve, it can be foreseen that the cumulative number of publications may be near 600 by the end of 2021. We can also forecast from Fig. 2 that the use of predictive models is becoming more popular in software engineering and there will be more studies that use these models to tackle practical SE problems.



(a) Number of publications per year in different conference proceedings.

(b) Number of publications per year in different journals.

Fig. 3. Publication trends for conferences and journals between 2009 and 2020.

In addition, we also analyzed the publication distribution of conference and journal papers per year, shown in Fig. 3. In conference proceedings, we can notice that three top conferences, i.e., ASE, FSE, and ICSE, involve many relevant studies using predictive models. Besides, predictive models also frequently occur in ESEM, MSR, and SANER. For journal papers, two top journals, i.e., TSE and TOSEM, almost employ relevant studies per year, and JSS also involves lots of relevant studies between 2016 and 2020.

4.2 Distribution of Primary Studies in Selected Publication Venues

The 421 reviewed studies were published in various publication venues, including in six high-quality journals and nine leading conferences, well-known and highly regarded in the field of software engineering. The prevalence of papers on predictive models in these conferences and journals indicates construction of predictive models for software engineering purposes are considered of importance. Table 5 lists the number of papers published in each publication venue. ICSE (67) includes the highest number of primary study papers compared with other leading conferences. EMSE employs 60 relevant studies using predictive models to solve SE issues, followed by TSE and JSS that employ 56 and 41 related studies, respectively. There are only 14 relevant studies published in TOSEM.

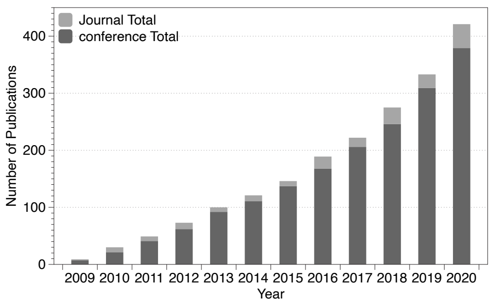


Fig. 4. Venue distribution for primary papers.

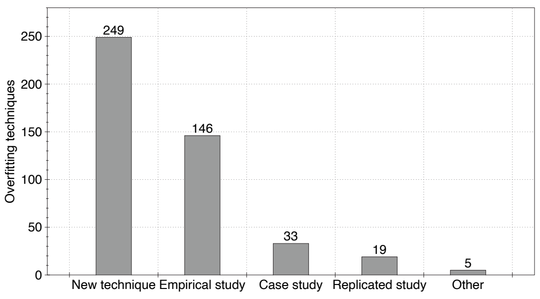


Fig. 5. Type of main contribution.

We checked the distribution between different publication venues. Fig. 4 gives the venue distribution per year. We can observe that compared with journal studies, the majority of publications

Table 5. The number of relevant studies in different publication venues.

Rank	Acronym	Full name	#Studies
1.	ICSE	ACM/IEEE International Conference on Software Engineering	67
2.	ASE	IEEE/ACM International Conference Automated Software Engineering	40
3.	ESEC/FSE	ACM SIGSOFT Symposium on the Foundation of Software Engineering/ European Software Engineering Conference	37
4.	MSR	IEEE Working Conference on Mining Software Repositories	26
5.	SANER	IEEE International Conference on Software Analysis, Evolution and Reengineering	15
6.	ICSME	IEEE International Conference on Software Maintenance and Evolution	14
7.	ISSTA	ACM SIGSOFT International Symposium on Software Testing and Analysis	12
8.	ESEM	ACM/IEEE International Symposium on Empirical Software Engineering and Measurement	11
9.	ICPC	IEEE International Conferences on Program Comprehensive	10
1.	ESE	Empirical Software Engineering	60
2.	TSE	IEEE Transactions on Software Engineering	56
3.	JSS	Journal of Software Systems	41
4.	TOSEM	ACM Transactions on Software Engineering and Methodology	14
5.	IST	Information and Software Technology	11
6.	TRel	Transactions of Reliability	7

appeared in conferences. One potential reason is that there are nine conference venues but only six journal ones we considered in our work. As Fig. 4 shows, in last decade, the publication trend of relevant studies shows a significant growing in terms of both conferences and journals.

4.3 Types of Contributions

We mainly summarized the main contribution of each primary study into five categories: New technique, Empirical study, Case study, Replicated study, and others (e.g., Survey). As some studies may contain more than one single type of contribution, we thus use a bar chart to depict the primary publications according to their main contribution and present the trend in Fig. 5.

The main contribution of 269 publications was to propose a novel predictive model technique / methodology. 146 studies focused on assessment and empirical studies, and 7.8% were case studies. The main contribution of 17 studies was replicated studies, and around 1.2% of primary studies were surveys on predictive models.

Summary of answers to RQ1:

(1) 421 relevant papers were identified from 15 high-quality publication venues.
(2) The predictive model has attracted sustained interest, with the topic showing significant increase in primary studies in 2009-2020.
(3) Most studies were published at conferences, compared with journals. But, in recent years, more related studies were published in leading journals.
(4) The main contribution of most studies was to present a new technique or methodology.

5 RQ2: WHICH PREDICTIVE MODELS ARE APPLIED TO SOFTWARE ENGINEERING TASKS?

In this section, we summarized the predictive modelling techniques used in every primary study and gave a systematic classification of these techniques based on their different functionalities. Finally, we analyzed the research distribution of these techniques from related studies to present an overview of predictive models in SE.

5.1 Predictive Model Classification

The predictive model, as a widely-used technology, has many different implementations [125]. We classified these predictive modelling techniques into six big categories based on different functionalities of these techniques: 1) Rule-based techniques, 2) Statistical models, 3) NLP techniques, 4) Machine Learning techniques, 5) Ensemble Learning techniques, and 6) Neural networks.

A Rule-based technique is a modelling approach that can leverage a set of rules to transform into a mathematical model or different equations to directly address specific problems. For example, Padhye et al. [271] proposed a novel approach to model code relevance and highlighted code changes by using a set of techniques, including Conjunctive Rule, a rule-based classifier.

A Statistical model can be considered as a mathematical model that embodies some statistical assumptions concerning the distribution of sample data. Statistical models can be classified into four categories based on different functionalities, including dimensionality reduction algorithms, regularization algorithms, state models, and probability models.

A NLP model is a novel technique that targets at processing natural languages and enables machines to read, decipher, understand, and make sense of natural languages. From language translation, sentiment analysis to speech recognition, diverse NLP techniques such as LSA, N-gram models can be used for emulating human intelligence and abilities impressively.

A Machine learning algorithm is usually generated from training data by a base learning algorithm [450]. These are multiple common machine learning algorithms in SE, such as logistic regression, naive bayes, decision tree, kNN, etc. In our study, we notice that the majority of primary studies used machine learners to build predictive models compared with other types of techniques. Most of them are classic algorithms which work well for many tasks.

An Ensemble learning technique uses a single base learning algorithm to produce homogeneous base learners. However there are also some methods which use multiple learning algorithms to produce heterogeneous learners for reducing bias (Boosting), variance (Bagging), or improving predictions. The well-known Random Forest algorithm, as a parallel ensemble algorithm, has become one of the most commonly used ensemble algorithms. A large number of studies (37) employed Random Forest to address different software engineering research tasks. These include defect prediction, vulnerability prediction, code quality prediction, software license exception detection, and fault localization [242, 287, 357, 373, 445].

There are 13 studies using a Boosting approach in their reported models ([80, 208, 211, 212, 264, 286, 411]). Most of these studies ([80, 212, 411]) utilized LogitBoost as a part of their approach. Yang et al. [411] implemented a prototype tool with LogitBoost to detecting nocuous coordination ambiguities in requirement documents, which pose a high risk of misunderstanding among different developerss. Falessiet al. [80] introduced a novel approach to estimate the number of remaining positive links in traceability recovery with LogitBoost. There are some studies that use other Boosting techniques, such as RankBoost and Gradient Boost. Perini et al. [286] employed RankBoost in order to give priority to requirements that were to be considered first. Machalica et al. [208] present a data-driven test selection strategy by training a gradient boosted decision tree classifier.

Bagging techniques were also applied by 8 studies to make predictions [163, 448]. Zhou et al. [448] leveraged six representative machine learning methods including bagging, AdaBoost and Random Forest to build fault prediction models to study the potentially confounding effect of class size. Kim et al. [163] evaluated the impact of data noise on defect prediction with adoption of basic learners and bagging learners. Malhotra and Khanna [212] evaluated the performance of LogitBoost, AdaBoost, Bagging, and other ML techniques for handling imbalanced datasets in software change prediction task.

Neural Network: Neural networks (NN) are a set of algorithms (e.g., perceptrons and deep learning techniques) designed to identify patterns. They interpret sensory data through a machine perception, labeling, or classifying raw data input. Therefore, NN can be considered as components of larger machine-learning applications involving algorithms for clustering, classification, or regression. Neural Network models can be classified into three categories, i.e., perceptron models, MP Neural models, and deep learning models, through the analysis of selected studies.

A perceptron is the simplified form of a neural network applied for supervised learning of binary classifiers, which consists of four main components including input values, net sum, weights and bias, and an activation function.

The MP neuron model is actually a simplified model constructed based on the structure and working principle of biological neurons, which consists of a function with a single parameter, taking binary input, and giving binary output according to a determined threshold value.

Deep learning is part of a broader family of machine learning methods based on artificial neural networks [324]. Deep learning architectures, such as deep belief networks, recurrent neural networks (RNN), and convolutional neural networks (CNN), have been applied to help various tasks across the software development life cycle. With the advent and development of deep learning techniques, 15 primary studies utilized deep learning techniques in their reported models [42, 191, 198, 401]. In this section, we briefly introduce several deep neural networks to give an overview of the fundamental principles. We also present how they have been applied to solve software engineering problems.

CNN has been used for SE tasks, e.g., [198]. The CNN model consists of tree layer, including an input layer, a hidden neurons layer and an output layer. CNN has achieved significant advances in recent years, with CNNs effectively increasing the flexibility and capacity of machine learning [198]. A CNN predictive model is made up of two parts, i.e., feature extraction and prediction. Compared to other models, the biggest strength of CNN lies in its convolution kernels, which focus only on local features and can fully extract the internal features of the data, boosting its accuracy. CNN have been applied to many SE research tasks. Liu et al. [198] exploited CNN to identify feature envy smells. The model they build had three layers with 128 kernels. The performance of their proposed approach outperformed the state-of-the-art significantly in feature envy detection and recommendation. Xu et al. [401] present a deep-learning based approach to predict semantically linkable units in developers' discussions in Stack Overflow. They formulated this problem as a multiclass classification problem, training a CNN to solve it. For classifying semantic relatedness, they used filters of five different window sizes, and each window size contained 128 filters to capture the most informative features.

Another approach is to use an LSTM model, a variant of the Recurrent Neural Network (RNN). This is specially designed for precessing sequential data. An LSTM model can capture the contextual information of sequences thanks to the recurrent nature of the RNN [437]. Computational units connected in each layer form a directed graph along a temporal sequence, which allows it to exhibit temporal dynamic behavior. Each LSTM unit contains an input gate, a memory cell, a forget gate, and an output gate. The gating mechanism of LSTM guarantees that the gradient of the long-term dependencies will not vanish [59].

Zhang et al. [437] utilized an attention-based Bi-LSTM model to detect anomalies in log events. Different from a standard LSTM, a Bi-LSTM can divide the hidden neuron layer into forward and backward, which allows it to capture more information of the log sequences. They confirmed the effectiveness of the proposed approach by comparing it with several traditional machine learning methods. Chen et al. [59] leveraged Bi-LSTM model as an emoji-powered learning approach for sentiment analysis. In their paper, the Bi-LSTM model contained two bi-directional LSTM layers and one attention layer, which treated the sentence vectors in Twitter and GitHub posts containing emojis as inputs and output the probabilities that instances contain each emoji.

5.2 Distribution of Different Predictive Models

In order to get a better understanding of predictive models, we summarize 148 predictive modelling techniques used in different software engineering tasks. Table 8 shows the number of cases for which different categories of predictive models and relevant studies in which they appear. It can be observed that 27 studies adopted rule-based predictive modeling techniques, where JRip and RIPPER are the top two common rule-based models. Statistical models can be classified into four categories according to different functionalities, and the dimensionality reduction algorithm (e.g., LDA) is the most frequently used one, followed by the state model. Some models are specialized for processing human languages, which are classified into the NLP family. Only three studies used related techniques as predictive models in our primary studies.

Comparing with another five categories, the machine learner is the most commonly used predictive model among all learners including seven different families, where over 100 studies adopted Logistic Regression, Naive Bayes, and SVM models to address software engineering problems. Among those ML-based predictive modeling techniques, the regression family involves 25 different regression algorithms, followed by SVM and Decision Tree families. There are 12 specific algorithms in the SVM and Decision Tree families, respectively. Apart from the above common ML algorithms, instance-based algorithms are also often applied in relevant studies, such as the kNN and its variants. Besides, the clustering technique and Learning To Rank (LTR) families also play essential roles in machine learning. It can be seen from Table 8 (See in Appendix) that machine learners have also been used widely in recent years after the occurrence of DL, indicating that these learners are still effective techniques in solving suitable problems, although they have been introduced some time ago.

A large number of studies applied ensemble algorithms to solve SE issues, and these different algorithms can be classified into three categories, involving random Forest, Bagging, and Boosting. Compared with the bagging and boosting techniques, over 65% of the studies selected Random Forest as the predictive model. 54 studies leveraged Boosting techniques in their experiments, in which AdaBoost and XGBoost are the top two popular predictive modeling techniques among other boosting-based algorithms. Bagging is another popular ensemble technique being used in 23 SE studies.

With the increasing popularity of DL technology, many studies adopted Neural Network techniques as predictive models, where almost 78% studies used deep learning techniques, and most of these studies were conducted between 2018 and 2020. This suggests the application of deep learning to software engineering is exhibiting a booming trend in the last few years. In Table 8, the family of DNN, i.e., deep learning techniques, is very popular with 113 studies applying them to software engineering problems. The most commonly used deep learning model is the CNN, with 24 studies using it, followed by ANN and LSTM. Deep Relief Network (DBN) was used only in five studies among all selected studies. 32 studies adopted perceptron models, where Multi-Layer Perceptron was used in 31 studies.

5.3 Evaluation Metrics for Predictive Models in classification tasks

To better understand how these predictive models were evaluated in SE tasks, we recorded the evaluation metrics in each primary study and summarized the most common metrics used to assess the performance of predictive models for classification tasks in SE.

Table 6 lists the most widely used evaluation metrics for SE classification tasks, including a description, definition and the some related references. As Table 6 shows, Recall, Precision and F-measure are the most commonly used to evaluate the performance of predictive models, followed by AUC and accuracy. Besides, we notice that many studies using Precision as one of evaluation metrics often adopted Recall and F-measure as another two metrics too. Some studies employed ROC to evaluate the performance of their predictive models and binary classification tasks would like to use MCC as their evaluation metric. The two metrics, FPR and TPR, have a close relationship, which causing they often occur in the same study. In classification tasks, there are some infrequently used evaluation metrics, such as G-measure, balance, pf, pd, and specificity.

Summary of answers to RQ2:

(1) Predictive models can be classified into six categories based on model architectures, i.e., Rule-based technique, Statistical model, NLP technique, Machine learning algorithm, Ensemble learning technique, and Neural network.
(2) 148 predictive models used in different SE activities are summarized in Table 8 and categorized them into multiple families according to diverse functionalities.
(3) Most of the primary studies applied machine learners to tackle software engineering problems, where Logistic Regression and Naive Bayes are the most popular predictive models.
(4) Using deep learning techniques in software engineering shows a thriving trend in recent years.
(5) Recall, precision and F-measure are the three most commonly used evaluation metrics to evaluate the performance of different predictive models in classification tasks.

Table 6. The definition and usage of common evaluation measures across the primary studies.

Measure	Description	Definition	Reference
Precision	The proportion of predicted positive instances that are correct.	$\frac{TP}{TP+FP}$	[2, 5, 6, 8, 10, 13, 17, 19–21, 24, 25, 28, 30, 31, 33, 38, 45–48, 50, 52–54, 56–58, 60, 61, 64, 66, 67, 70, 71, 75, 77, 78, 81, 85, 87, 88, 91, 95–97, 104, 106, 112–114, 117, 118, 121, 123, 130, 132, 135–137, 154, 157, 158, 165, 171, 173–175, 178, 204, 205, 207, 210, 213, 214, 217, 220, 224, 234, 240, 253, 255, 258, 262, 263, 266, 268, 269, 272, 274–276, 279–285, 287, 290, 293, 294, 296, 299, 301, 305, 307, 310, 311, 313, 315, 318, 321, 322, 326, 327, 334, 339, 343, 345, 352, 360, 361, 363, 366, 370–372, 372, 374, 377, 378, 382, 383, 386, 387, 392–394, 396, 398–400, 404, 406, 409, 415, 425, 426, 430, 434, 435, 438, 440, 446, 447]
Recall	The proportion of positive cases that were correctly predicted	$\frac{TP}{TP+FN}$	[2, 5, 6, 8, 10, 13, 17, 19–21, 24, 25, 28, 30, 31, 38, 45–47, 50, 52–54, 57, 58, 60, 61, 64, 66, 67, 70, 71, 75, 77, 81, 85, 87, 88, 91, 95–97, 104, 106, 112–114, 117, 118, 121, 123, 130, 132, 135–137, 154, 156–158, 163, 165, 171, 173–175, 178, 189, 204, 205, 207, 210, 214, 217, 224, 240, 253, 255, 258, 262, 263, 266, 268, 269, 272, 274–276, 279–285, 287, 290, 293, 294, 296, 299, 301, 305, 307, 310, 311, 313, 315, 318, 321, 322, 326, 327, 334, 343, 345, 350, 352, 360, 361, 363, 366, 370–372, 372, 374, 377, 378, 382, 383, 386, 387, 392–394, 396, 398, 399, 404–406, 409, 415, 425–427, 430, 434, 435, 438, 440, 446]
F-measure	Harmonic mean of precision and recall	$\frac{2 \times Precision \times Recall}{Precision + Recall}$	[2, 6, 8, 10, 13, 19–21, 24, 25, 28, 31, 38, 45–48, 50, 52–54, 57, 61, 64, 66, 71, 72, 77, 78, 81, 85, 87, 91, 97, 104, 113, 114, 117, 118, 123, 130, 132, 135–137, 154, 158, 163, 165, 171–175, 196, 204, 205, 207, 210, 213, 214, 217, 224, 240, 253, 255, 258, 262, 263, 266, 269, 272, 274, 275, 279–285, 287, 290, 294, 296, 299, 301, 305, 307, 315, 318, 321, 327, 330, 339, 343, 345, 352, 360, 363, 366, 370–372, 372, 374, 377, 378, 382, 383, 387, 392–394, 396, 398, 399, 402, 404, 405, 405, 406, 415, 425, 435, 439, 440, 446, 447]
Area Under the Curve (AUC)	The area under the receiver operating characteristics curve. Independent of the cutoff value		[24, 31, 45, 50, 60, 61, 64, 72, 77, 94–96, 117, 121, 123, 127, 136, 158, 167, 173, 183, 188, 205, 210, 233, 272, 274, 275, 279, 294, 302, 306, 313, 315, 343–345, 348, 360, 363, 366, 387, 392, 394, 399, 400, 409, 410, 421, 425, 427, 434, 438, 444]
Accuracy	Proportion of correctly classified instances	$\frac{TP+TN}{TP+FP+TN+FN}$	[5, 10, 13, 17, 20, 24, 25, 31, 32, 39, 45, 48, 52, 54, 60, 66, 71, 72, 88, 97, 100, 104, 113, 117, 132, 154, 172, 178, 189, 196, 213, 220, 239, 266, 274, 290, 293, 301, 305, 307, 322, 344, 361, 372, 372, 409, 410, 427, 434, 447]

Measure	Description	Definition	Reference
ROC	A comprehensive indicators reflecting sensitivity and specificity of continuous variables		[45, 58, 61, 72, 78, 127, 136, 158, 272, 274, 279, 301, 315, 343–345, 360, 366, 421]
MCC	is used in machine learning as a measure of the quality of binary (two-class) classifications	$\frac{TP\times TN-FP\times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$	[31, 45, 47, 77, 78, 167, 210, 282, 283, 315, 402, 438, 439]
False positive rate(FPR)	the ratio between the number of negative events wrongly categorized as positive and the total number of actual negative events	$\frac{FP}{FP+TN}$	[48, 207, 219, 219, 220, 269, 293, 334, 421]
True positive rate(TPR)	The probability that an actual positive will test positive	$\frac{TP}{TP+FN}$	[207, 219, 219, 220, 293, 334, 421]
G-measure	Harmonic mean of pd and (1-pf)	$\frac{2\times pd\times(1-pf)}{pd+(1-pf)}$	[213, 275, 289, 290, 403, 405]
Balance(bla)	The balance between pd (the probability of detection) and pf. The bigger the bal value is, the better the performance of learning model is.	$1-\frac{\sqrt{(1-pd)^2+(0-pd)^2}}{\sqrt{2}}$	[206, 213, 239, 339, 403, 405]
Probability of False Alarm (pf)	Proportion of negative instances incorrectly classified as positive	$\frac{FP}{FP+TN}$	[77, 203, 289, 326, 339, 365]
probability of detection (pd)	Probability of detection PD is the ratio of all possible targets in a given direction.	$\frac{detectedtargets}{allpossibletargets}$	[77, 289, 339, 365]
Specificity	The proportion of predicted negative instances that are correct	$\frac{TN}{FP+TN}$	[48, 66, 240, 447]

6 RQ3: IN WHAT DOMAINS AND APPLICATIONS HAVE PREDICTIVE MODELS BEEN APPLIED?

In this section, we conduct an analysis on the distribution of different research domains to which predictive models were applied, and analyse the application of predictive models in each research topics.

To better understand the application distribution of predictive models, we categorise the software development life cycle (SDLC) into the following stages according to [36]: (1) Software Requirements; (2) Software Design; (3) Software Development; (4) Software Testing; (5) Software Maintenance; and (6) Software management.

We referred to the definition of each category of the SDLC in [36] and conducted a comprehensive analysis of the problem addressed by each study and its practical application scenarios to determine which category the study belongs to.

6.1 Distribution in Different SDLC Domains

We categorize our selected primary studies based on research domains in Fig. 6 and illustrate the development trend of the main research domains in each year in Fig. 7. Table 7 lists the ranking of software engineering tasks that most studies have concentrated on.

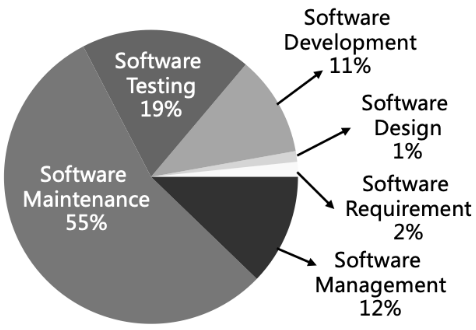


Fig. 6. Research domains.

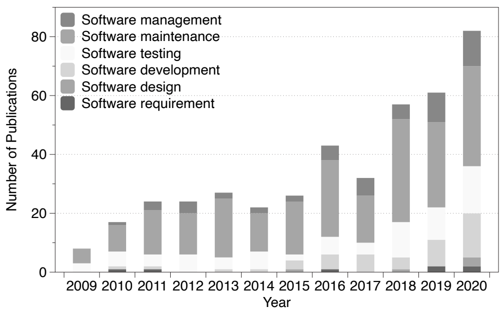


Fig. 7. Development trend of six main research domains.

As Fig. 6 shows, the primary research domain of the majority of the studies (55%) was software maintenance, followed by software testing (19%). Studies related to software management were 12%. Software development was the focus of 11% of the studies. We notice that most of the primary studies worked on specific tasks with respect to software testing and maintenance. One potential reason is that these two research directions involve many predictive model-applicable tasks, such as defect prediction, code smell detection, and bug report management. Only 2% and 1% of studies focused on specific tasks in software requirement and software design.

Fig.7 shows the development trend of the main research domains – Software requirements, Software design, Software development, Software testing, Software maintenance, and Software management – over the last decade. Apart from 2009, there are a few studies related to software management in each year. Only 7 software requirement related studies using predictive models appeared between 2009 and 2020. This indicates that software engineering professional practice shows a stable development trend, but there seems to be a lack of attention to predictive model research topics in software requirements. The number of studies in software maintenance occupied a large proportion in each of last decade, which means that software maintenance, as one of the most

Table 7. Top SE Topics with a Minimum of 10 Relevant Papers.

Rank	Specific Task	Research Domain	#Studies
1.	Defect Prediction	Software Maintenance	111
2.	Bug/Fault Prediction	Software Testing	42
3.	Software Quality Assessment	Software Maintenance	20
4.	Developer Behavior Analysis	Software Management	18
5.	Vulnerability Detection	Software Maintenance	14
5.	Software Repository Mining Detection	Software Management	14
7.	Performance Prediction	Software Maintenance	11
7.	Code Smell Detection	Software Maintenance	11

essential SE activities, attracted much attention of lots of researchers. 19% of studies concentrated on specific tasks in software development. We also notice from Fig.6 that predictive models have not yet been widely used in certain domains, such as software requirements and software design. This may perhaps motivate researchers to explore more scenarios in these SDLC domains suitable for application of predictive models.

Table 7 lists the ranking of specific topics where over 10 relevant studies have appeared. We observe that five of the eight research topics belong to software maintenance and two topics are in software management. 111 publications employed predictive models in defect prediction, a research topic that most commonly used predictive models as a solution. Bug detection is the second common research topic followed by software quality assessment and vulnerability detection. In software management, 20 studies focused on developer behavior analysis and 14 studies adopted predictive modeling techniques to mining software repositories, such as GitHub, Stack Overflow, and App store. The number of studies related to performance prediction is the same as that of code smell detection. Besides, no research topics in software requirement and design are listed in Table 7.

6.2 Software Requirements

Requirements engineering (RE) is the process of defining, documenting, and maintaining requirements in software design process [36] and play a crucial role throughout systems' lifecycle. A requirement with high quality can notably improve development efficiency by avoiding reconstruction and rework. Table 9 (See in Appendix) presents the details of relevant studies that used predictive models in software requirement.

6.2.1 Requirements Classification. An effective classification of software requirements enables analysts to perform well-focused communication with developers and users as well as prioritize the requirement documents according to their importance [1].

Non-functional requirements (NFRs) classification: Anish et al. [15] conducted a study to identify, document, and organize Probing Questions (PQs) for five different areas of functionality into structured flows, called PQ-flow. They used Naive Bayes to identify Architecturally Significant Functional Requirements (ASFRs), used random k labelsets classifier (RAkEL) to categorize ASFR by types, and finally recommended PQ-flows.

Other requirements classification: Abad et al. [1] proposed a novel approach that extracts and classifies requirements-related knowledge to support analysts familiarity with different domains. Based on the intuition behind utilizing lexical association, they built a dynamic generative model to extract the relevant terms in documents and applied the rational kernels method as well as SVMs when classifying requirements. Although their proposed method has been evaluated, some factors (e.g. human interface factors) might impact the performance of this technique in industrial contexts. Abualhaija et al. [30] used machine learning methods to demarcating requirements in free-form historical specifications from security assessments. Their automated method achieved an average recall of 94% and average precision of 63%.

6.2.2 Requirements Detection. In large software projects, participants engage in parallel and interdependent tasks, resulting in work dependence and coordination needs. When developers remain unaware or do not obtain timely awareness of the coordination that is required to manage work dependencies, there is potential for software productivity and quality problems. Yang et al. [411] implemented a prototype tool for Nocuous Ambiguity Identification (NAI) in text. In their earlier work, they had focused on identifying the coordination ambiguity and anaphora ambiguity in requirement documents, and thus this study tried to detect another two types of ambiguity (i.e., nocuous and innocuous ambiguity). After extracting ambiguity instances and recognizing coordination constituents, they defined a set of heuristics (e.g., coordination matching, distribution similarity, and collocation frequency) based on word distribution and collocation and trained a binary classifier using LogitBoost algorithm to classify the input as nocuous or innocuous. Blincoe et al. [35] investigated what work dependencies a software development team needs to consider when establishing the coordination needs. They present a list of properties helping to characterize task pairs that require coordination. They noticed that existing techniques for detecting coordination requirements find excessive dependencies. They applied k-nearest neighbor machine learning algorithm to identify and supplement task properties that are indicative of the crucial coordination needs, contributing in the selection of the most critical coordination needs.

6.2.3 Other Requirement Tasks. Apart from requirement classification and requirement detection, several studies [34, 259] built predictive models to predict and identify requirement changes and Falessi et al. [79] leverage the vector space model to conduct an industrial case study for similar requirement detection.

6.3 Software Design

A software design is a description of the architecture of the software to be implemented, and algorithms or code management used. Below we introduce the application of predictive models in software design and for code development. Table 10 (See in Appendix) presents the details of relevant studies that used predictive models in software design.

The architecture of a system describes the relationships between its main components and how they interact with each other. There are many factors involved in software architecture and design (e.g., quality, security, performance, etc.). It is a challenging task to balance different factors for designing a good architecture. To identify a set of quality concerns for maintaining security, reliability and performance of software systems, Mirakhorli et al. [226] proposed a cost-effective approach to automatically construct traceability links for architectural tactics. They applied a base learner as a tactic-classifier identifying all classes related to a given tactic. To reconstruct tactic-level traceability, these classes were mapped to their relevant tactic through two different classifiers, where one was trained using textual descriptions of each tactic and another using code snippets taken. Their experimental results showed that code-trained classifiers outperformed description-trained classifiers in terms of precision and recall.

Architects construct software architectures by making a variety of design decisions that can satisfy quality concerns (e.g., reliability, performance, and security). However, the architectural quality decreased when developers modified code without fully understanding design decisions. In order to address this problem, Mirakhorli et al. [225] proposed a solution for detecting, tracing and visualizing architectural tactics in code. They utilized six machine learning algorithms (i.e., SVM, Decision Tree (C.45), Bayesian Logistic Regressions (BLR), AdaBoost, SLIPPER, Bagging) to train classifiers for detecting the presence of architectural tactics in source code. They monitored the architecturally significant code by mapping those relevant code segments into Tactic Traceability Patterns (tTPs) and notified developers when those segments were modified. By analyzing the experimental results, they concluded that the six of classifiers performed equivalently in tactical detection tasks.

Gopalakrishnan et al. [99] present a bottom-up approach to identify architectural tactics. They adopted the Random Forest algorithm to capture relationships between latent topics and architectural tactics in source code of projects. They evaluated the ability of the proposed approach through a series of experiments on two large-scale datasets, i.e., Apache Hive and Hadoop. Reman et al. [309] researched software architecture in another perspective. They performed a case study to investigate whether and how Hands-on software architects benefit the projects by using data analytics-based techniques in five large-scale industrial systems. To better understand what type of code architects write, they employed a classification algorithm [226] to identify functional comments and analyzed architects' code contributions. They observed that architects write code for addressing quality concerns and shaping the initial structure of projects. They also noticed that some architects engaged in other activities in addition to writing tactic, functional code as well as tests.

Some studies [10, 234] helped developers to design the user interface screen (UI screen) through leveraging predictive models to identify UI components from n mobile development screenshots. Alahmad et al. [10] trained a CNN model, named UIScreens, to detect and extract the most representative UI screens from mobile screenshots. They evaluated their model in two empirical studies on Android and iOS screencasts, respectively, and their model achieved high performance in terms of accuracy.

6.4 Software Development

Software development is the process that developers implement specific functions of a software system by coding according to some SE documentation, such as requirement specification and design documentation. A number of specific tasks and problems are involved in this SE activity, including code consistency related studies, program security, program analysis, code comments, code review, etc. Table 11 (See in Appendix) describes the details of all relevant studies in software development.

6.4.1 Code Consistency. Change management involves tracking and managing changes to artifacts, such as requirements, change requests, bug reports, source code files, and other digital assets. It's critical for effective application development. With the rapid evolution and change of software, effective change management has become a major challenge in the industry, and thus many studies have worked on identifying changes, learning changes, analyzing changes, tracing changes, assessing security or metrics using predictive models [81, 262, 383].

In Table 11, most of the studies focused on code consistency, where including six different families: code change prediction, co-change prediction, code change identification, change impact analysis, conflict prediction, and code change classification. For code change prediction, Malhotra et al. [212] conducted an empirical study to investigate the performance of the predictive models on predicting code changes. They adopted six different algorithms, including machine learning, neural network, and ensemble learning algorithms. There are seven studies using predictive models for co-change prediction, where three studies [210, 387, 410] employed Random Forest. Nguyen et al. [262] trained a RNN model with an encoder-decoder architecture to predict co-change in software systems. There are two relevant studies in three SE issues: code change identification, change impact analysis, and conflict prediction, respectively. Only one study worked on code change classification. Padhye et al. [271] present a prototype tool for highlighting code changes in order to improve development efficiency. They explored the use of various techniques including Naive approach (TOUCH-based subscription strategy) and machine learning algorithms (Conjunctive Rule, J48 and Naive Bayes) to model code relevance. Their proposed tool can highlight changes that developers may need to review in order to personalize a developers' change notification feed. They found that the best strategy can reduce the notification clutter by more than 90%. They then applied these methods at a different granularity. Misirli et al. [228] analyzed high impact fix-inducing changes (HIFCs). They used a Random Forest-based algorithm to identify HIFCs and determine the best indicators of HIFCs on six large open source projects. They also present a measure to evaluate the impact of fix-inducing changes. Falessi et al. [81] introduced new Requirements to the Requirements Set (R2RS) family of metrics based on some intuitions. In order to evaluate the usefulness of the proposed metrics, they applied five classifiers to predict the set of classes impacted by a requirement over 700,000 classes.

6.4.2 Program Analysis. In Table 11, a set of specific SE issues requires program analysis techniques to be solve. Among them, type inference is one of the most common tasks in this topic. Malik et al. [215] trained a DL model to infer JavaScript function types by using natural language information. Hora et al. [124] adopted random forest algorithm to predict whether and when the interface should used the "Public" attribute. There are other program analysis related issues in Table 11, such as program execution classification, type annotation, termination proofs detection, etc. Except for common program analysis tasks, some SE issues also need to the knowledge of program analysis, including commit code detection, program classification, and compile version prediction.

6.4.3 API-related. API-related SE tasks are the second most common research topic in software development as API is one of the most frequently used code elements during coding. There are six relevant studies in Table 11, where five studies used predictive models to identify APIs from code segments. For example, Zhong et al. [442] utilized the Hidden Markov Model to detect API in program and Santos et al. [321] performed a comparison study to investigate diverse predictive models for API identification. In order to identify API issue-related sentences in Stack Overflow (SO) Posts and to classify SO posts concerning API issues, Lin et al. [190] introduced a machine learning based approach called POME. POME classifies sentences related to APIs in Stack Overflow, using natural language parsing and pattern-matching and determined their polarity (positive vs negative). Evaluation results showed that POME exhibited a higher precision compared with a state-of-the-art approaches. Similarly, Ahasanuzzaman et al. [8] developed a supervised learning approach with a Conditional Random Field (CRF). They also performed an investigation to identify important features and test the performance of the proposed technique for classifying issue sentences. Nam et al. [250] conducted studies on boilerplate code and investigated what properties make code be considered as "boilerplate". They then proposed a novel approach automatically mine boilerplate code candidates from API client code repositories.

6.4.4 Code Comment. Code comment is an essential factor to help the developer understand the functionality of a code fragment quickly. Some studies conducted further analysis towards code comments. For example, Chen et al. [55] used machine learning techniques to classify code comments. Huang et al. [135] trained an LSTM model to predict the position of comments in the program. Besides, Chen et al. [52] leverage an ensemble learning model, Random Forest, to detect the scope of source code that the corresponding code comment stands for. Comments in source code explain the purpose of the code and help with code understanding during development and maintenance. Huang et al. [129] present a classification approach to detect self-admitted technical debt (SATD) in code comments. They applied feature selection to select valuable features and trained a composite classifier by combining multiple classifiers for identifying SATD comments in open source projects. Evaluation results showed that the proposed approach outperformed the state-of-the-art and baselines in terms of precision, recall and F-measure. For better understanding the goal and target audience of code comments, Pascarella et al. [278] investigated how diverse Java software projects used code comments, and applied machine learning algorithms to automatically classify code comments at line level. Chen et al. [55] analyzed the relationship between a code fragment and its code comment and used multiple predictive models to investigate to which extent this relationship can be exploited for improving code summarization performance.

6.4.5 Other SE Tasks. There are a few studies working on code review, program security, and code extraction. For example, Rahman et al. [305] used a set of machine learning algorithms to extract textual features for predicting useful code review comments according to developers' experience, and Fan et al. [86] used Random Forest to analyze the merged change code for identifying critical review tasks. To ensure the program security, Bettaieb et al. [30] used a set of machine learning algorithms to assist with the selection of security controls during code security assessment. Code impact analysis is an essential task for software development and aims to make clear the impact scope of a code fragment in a software system, which helps change code and avoids introducing new bugs. Falessi et al. [81] utilized multiple machine learning algorithms, i.e., Decision tree, Random forest, Naive bayes, logistical regression, and Bagging algorithms, to conduct impact analysis towards source code in software.

6.5 Software Testing

Software testing is the process of verifying and validating whether a software or application is bug free, and its design and development meet technical and user requirements. Debugging is the process of identifying, analyzing and remove errors after the software fails to execute properly. Effective testing and debugging techniques can improve software quality with reduced effort consumption and improved performance [362]. In this section, we summarized research topics in software testing, and also described some studies published before 2009 to help to introduce the development roadmap of different topics. Table 12 (See in Appendix) shows the details of relevant studies that used predictive models to solve specific issues in this SE activity.

6.5.1 Bug/Fault Prediction. In Table 12, a large number of studies adopted predictive models to predict bugs or faults in software systems. Failure prediction is critical to predictive maintenance since it has the ability to prevent maintenance costs and failure / bug occurrences [33, 50, 187, 348, 440]. Yilmaz and Porter [420] classify measured executions into successful and failed executions in order to apply the resulting models to systems with an unknown failure status. They further applied their technique to online environments. Ozcelik and Yilmaz [270] proposed a online failure prediction approach for combining hardware and software instrumentation. In the training phase, they applied global and frequency filtering to pick up candidate functions from historical data as the input of the proposed approach and created a predictive model for identifying the best performing functions that best distinguish failing executions from passing executions. These best performing functions were referred to as seer functions. Then their proposed model makes a binary prediction for each seer function into passing or failing during monitoring phase. To address the understudied problem of inferring concurrency-related documentations, Habib et al. [108] developed a novel tool, TSFinder, to automatically classify classes as thread-safe or thread-unsafe with a combination of static analysis and graph-based classification. Yilmaz and Porter [420] present a hybrid instrumentation approach to distinguish failed executions from successful executions with consideration of potential cost-benefit tradeoffs. After collecting program spectra and augmenting the underlying data with low cost, they trained a classification model with J48 algorithm to classify program executions.

6.5.2 Test Case. In software testing, test case quality is a critical factor. Good test cases can effectively improve software quality. On the contrary, constructing and running bad test cases may cost significant time and effort while still not finding defects or faults. There are many studies that aim to improve test case quality using predictive models.

They used a decision tree to classify whether a fault is representative or not based on software complexity metrics. Hajri et al. [112] used a logistical progression model to automatically classify the test cases for use case-driven testing in product lines. Yu et al. [423] proposed a technique that can be used to distinguish failing tests that executed a single fault from those that executed multiple faults. The technique combines information from a set of fault localization ranked lists, each produced for a certain failing test, and the distance between a failing test and the passing test that most resembles it.

6.5.3 Bug Classification. Some studies [138, 263, 322, 351, 352] used predictive models to classify bugs from perspectives. Tan et al. [351] used libSVM and BayesNet models to extract software bug characteristics and performed bug classification based on their characteristics. Ni et al. [263] combined TBCNN and SVM to analyze fixed bugs and corresponding fixing methods to automatically classify bugs according to their causes. Tan et al. [352] trained multiple machine learning approaches to analyze the severity of bugs and predict high-level bugs in software.

6.5.4 Fault Localization. There are four tasks using predictive models for fault localization [175, 176, 297, 445]. Le et al. [175] adopted SVM to evaluate the effectiveness of bug localization tools and they then [176] improved the information retrieval-based bug localization tools by mitigating the impact of the unreliability of information. Pradel et al. [297] present a novel model to localize the bug among the large scale of code by training an RNN model. Zhou et al. [445] leveraged the log information and used multiple machine learning methods to perform error prediction and bug localization.

6.5.5 Testing Applications. To reduce expensive calculation of mutation testing, Zhang et al. [435] proposed predictive mutation testing (PMT), that can predict the testing results without executing mutants. After identifying execution features, infection features and propagation features of tests and mutants, they selected Random Forest algorithm to construct a classification model for predicting whether a mutant can survived or be killed without mutant execution. Cotroneo et al. [63] proposed a method based on machine learning to adaptively combine testing techniques during the testing process. Their method contains an offline learning phase and an online learning phase. During offline learning, they first defined the features of a testing session potentially related to the techniques performance, and then used several machine learning approaches (i.e., Decision Trees, Bayesian Network, Naive Bayes, Logistic Regression) to predict the performance of a testing technique. During online learning, they adapt the selection of test cases to the data observed as the testing proceeds.

6.5.6 Test Report Classification. Wang et al. [376] introduced a Local-based Active ClassiFication (LOAF) approach to classify crowdsourced test reports that reveal true faults. They adopted active learning to train a classification model with as few labeled inputs as possible in order to reduce the onerous burden of manual labeling. Compared with existing supervised machine learning methods, LOAF exhibited promising results on one of the largest Chinese crowdsourced testing platforms. Chen et al. [57] proposed a framework to classify and assess the quality of crowdsourced test reports by using predictive models.

6.5.7 Other Tasks. A few studies focused on test alarm identification, test reports, and fault injection. For example, Natella et al. [256] proposed a new approach to refine the “fault load" by removing faults that are not representative of residual software faults. Wang et al. [376] propsoed a new apporach to assist crowdsourced testing to classify test reports by using SVM, Decision tree, Naive Bayes, and Logistic regression models. Jiang et al. [141] used kNN model to conduct an automatic cause analysis for investigating the reason of a test alarm.

6.6 Software Maintenance

Software maintenance is an integral stage of the software life cycle. No software product is ever completely finished and all need some form of ongoing maintenance. We introduce representative applications of predictive models for software maintenance. The relevant studies in software maintenance are listed in Table 13 (See in Appendix).

6.6.1 Software Defect Prediction. Software defect prediction techniques are employed to help prioritize software testing and debugging. These techniques can recommend software components that are likely to be defective to developers. Rahman et al. compared static bug finders and defect predictive models [302]. They found that in some cases, the performance of certain static bug-finders can be enhanced using information provided by statistical defect prediction models. Therefore, applying prediction models to defect prediction has become a popular research direction [154].

Lessmann et al. [181] proposed a framework for comparative software defect prediction experiments. They conducted a large-scale empirical comparison of 22 classifiers over 10 public domain data sets. Their results indicated that the importance of the particular classification algorithm may be less than previously assumed. This is because no significant performance differences could be detected among the top 17 classifiers. However, Ghotra et al. [94] doubted this conclusion and they pointed that the datasets Lessmann et al. used were both noisy and biased. Therefore, they replicated the prior study with original datasets as well as new datasets after cleansing. Their new experimental results demonstrated that some classification techniques were more suitable for building predictive models. They showed that Logistic Model Tree when combined with ensemble methods (i.e., bagging, random subspace, and rotation forest) achieves top-rank performance. Furthermore, clustering techniques (i.e., Expectation Maximization and K-means), rule-based techniques (Repeated Incremental Pruning to Produce Error Reduction and Ripple Down Rules), and support vector machine perform worse than other predictive models. Herzig et al. [120] conducted an analysis of the impact of tangled code changes in defect prediction. They used a multi-predictive model to identify tangled changes and they found that untangling tangled code changes can achieve significant accuracy improvements on defect prediction.

Many predictive model algorithms have several tunable parameters and their values may have a large impact on the model’s prediction performance. Song et al. [342] proposed and evaluated a general framework for defect prediction that supports unbiased and comprehensive comparison between competing prediction systems. They first evaluated and chose a good learning scheme, consisting of a data preprocessor, an attribute selector and a learning algorithm. They then used the scheme to build a predictor. Their framework proposes three key elements for defect prediction models, i.e., datasets, features and algorithms. Tantithamthavorn et al. [355] conducted a case study on 18 datasets to investigate the performance of an automated parameter optimization technique, Caret, for defect prediction. They tried 26 classification techniques that require at least one parameter setting and concluded that automated parameter optimization techniques like Caret yield substantially benefits in terms of performance improvement and stability, while incurring a manageable additional computational cost.

Features, metrics or attributes are crucial for the building of a well-performed defect predictive model. A number of studies researched a lot about feature extraction for defect prediction [301, 335, 379]. Kim et al. [162] introduced a new technique for change-level defect prediction by using support vector machine. They are the first to classify file changes as buggy or clean leveraging change information features. An obvious advantage of change-level classification is that predictions can be performed immediately upon the completion of changes. Shivaji et al. [335] investigated multiple feature selection techniques that are generally applicable to classification-based bug prediction methods. They found that binary features are better, and that between 3.12% and 25% of the total feature set yielded optimal classification results. Lee et al. [179] proposed 56 new micro interaction metrics (MIMs) by using developers' interaction information (e.g., file editing and selection events) in an Eclipse plug-in, Mylyn. To evaluate the effectiveness of MIMs, they built defect predictive models using traditional metrics, MIMS and their combinations. The evaluation results showed that MIMs allowed the performance of those models improve significantly. Yu et al. [422] initially focused on defect prediction for concurrent programs and proposed ConPredictor, a prototype tool that used four classification techniques to identify defects by applying a set of static and dynamic code metrics based on unique features of concurrent programs. Lin et al. [192] present an approach to identifying buggy videos from a set of gameplay videos. They applied three classification models – logistic regression, neural network and random forests – to determine the probability that a video showcases a bug. They observed that random forests achieved the best performance among classifiers. Afterwards, they identified key features in game videos and used random forest classifier to prioritize game videos according to the likelihood of each video containing bugs.

A good predictive model relies heavily on the dataset it learns from, which is also the case for models for software defect prediction. Many studies investigate the quality, such as bias and size, of datasets for defect prediction. Rahman et al. [304] investigated the effect of size and bias of datasets on performance of defect prediction using logistic regression model. They investigated 12 open source projects and their results suggested that the type of bias has limited impact on prediction results, and the effect of bias is strongly confounded by size. Tantithamthavorn et al. [354] used random forest to investigate the impact of mislabelling on the performance and interpretation of defect models. They found that precision is rarely impacted by mislabelling while recall is impacted much by mislabelling. Moreover, the most influential features are generally robust to mislabelling.

For datasets of poor quality, researchers also have proposed several approaches to address their issues. Kim et al. [163] proposed an approach, named CLNI, to deal with the noise in defect prediction datasets. CLNI can effectively identify and eliminate noise. The noise eliminated from the training sets produced by CLNI was shown to improve the defect prediction model's performance. Nam et al. [252] proposed novel approaches CLA and CLAMI, which can work well for defect prediction on unlabeled datasets in an automated manner without any manual effort. Gong et al. [98] investigated the impact of class overlap and class imbalance problems on defect prediction, and then present an improved approach (IKMCCA) to solve those problems in order to improve defect prediction performance for within project defect prediction (WPDP) and cross project defect prediction (CPDP). Tantithamthavorn et al. [237, 353] conducted an empirical study to investigate how class rebalancing techniques impact the performance and interpretation of defect prediction. They also explored experimental settings that can help rebalancing techniques achieve the best performance in defect prediction. They concluded that using different metrics and classification algorithms allowed the performance of predictive models to vary, indicating that researchers should avoid class rebalancing techniques when deriving understandings and knowledge from models.

Datasets from many studies are unfortunately not always available due to privacy policies and other factors. To address the privacy problem, Peters et al. [287–289] measured the utility of privatized datasets empirically using Random Forests, Naive Bayes and Logistic Regression. Through this they showed the usefulness of their proposed privacy algorithm MORPH [287]. MORPH is a data mutator that moves the data a random distance, while not across the class boundaries. In a later work [288], they improved MORPH by proposing CLIFF+MORPH to enable effective defect prediction from shared data while preserving privacy. CLIFF is an instance pruner that deletes irrelevant examples. Recently, they again extended MORPH to propose LACE2 [289].

Recently, deep learning, as an advanced machine learning algorithm, has become widely discussed and applied including in Software Engineering. Several researchers have tried to improve the performance of defect prediction via deep learning [53, 378, 379]. Jing et al. [147] proposed a cost-sensitive discriminative dictionary learning (CDDL) approach for software defect prediction. CDDL is based on sparse coding which can transform the initial features into more representative code. Their results showed that CDDL is superior to five representative methods, i.e., support vector machine, Compressed C4.5 decision tree, weighted Naive Bayes, coding based ensemble learning (CEL), and cost-sensitive boosting neural network. Wang et al. [379] leveraged Deep Belief Network (DBN) to automatically learn semantic features from token vectors extracted from programs Abstract Syntax Trees. Their evaluation on ten open source projects showed that learned semantic features significantly improve both within-project defect prediction and cross-project defect prediction compared to traditional features. Chen et al. [53] trained a CNN model to perform defect prediction on source code. In this model, they applied the self-attention mechanism to extract features and used transfer learning technique to reduce the difference in sample distributions between different software products.

Cross project defect prediction: Cross-project defect prediction is a topic of growing research interest. It uses data from one project to build the predictive model and predicts defects in another project based on the trained model so that it can solve the problem that there is no sufficient amount of data available to train within a project, such as a new project. Some studies concentrated on cross-project defect predictive models on a large scale [183, 451].

To improve the performance of cross-project defect prediction, researches have tried several techniques [145, 251]. Nam et al. [253] proposed a novel transfer defect learning approach, TCA+, extending a transfer learning approach Transfer Component Analysis (TCA) . TCA+ can provide decision rules to select suitable normalization options for TCA of a given source-target project pair. In a later work, they addressed the limitation that cross-project defect prediction cannot be conducted across projects with heterogeneous metric sets by proposing a heterogeneous defect prediction approach. Jiang et al. [145] proposed an approach CCA+ for heterogeneous cross-company defect prediction. CCA+ combines unified metric representation and canonical correlation analysis and can achieve the best prediction results with the nearest neighbor classifier.

Zhang et al. [432] found that connectivity-based unsupervised classifiers (via spectral clustering) offer a viable solution for cross-project defect prediction. Their spectral classifier ranks as one of the top classifiers among five widely-used supervised classifiers and five unsupervised classifiers in cross-project defect prediction. Due to the impact of heterogeneous metric sets, current defect prediction techniques are difficult to use in CPDP. To address this problem, Nam and Kim [251] trained a predictive model for heterogeneous defect prediction (HDP) with heterogeneous metric sets by applying metric selection and metric matching techniques. Zhou et al. [449] noticed that most CPDP models were not compared against simple defect predictive models, and thus they aimed to investigate whether CPDP models really performed well compared against simple models, ManualDown and ManualUp. To their surprise, they found that the performance of ManualDown and ManualUp were superior to most of existing CPDP models.

Just-in-time (JIT) defect prediction: Compared with traditional defect predictions at class or file level, Just-in-Time (JIT) defect prediction is of more practical value for participants, which aims to identify defect-inducing changes. Many studies focused on JIT defect prediction by employing

SZZ approach [83, 151, 152, 167, 335]. In order to identify bug-introducing changes, SZZ first detects the bug-fixing changes whose the change log contains bug identifier. Among the bug-fixing changes, SZZ identifies the modified the buggy lines so that the bug can be removed. SZZ then traces the code change history to search for potential bug-introducing changes that may introduce buggy lines. Finally, the remaining changes are deemed as bug-introducing after SZZ eliminates the incorrect ones from the initial set of potential bug-introducing changes. Yang et al. [416] used the most common change metrics from various source to build unsupervised and supervised models to predict defect-inducing changes. Comparing these models under cross-validation and time-wise-cross-validation, their results showed that the simple unsupervised models performed better than state-of-the-art supervised models for JIT defect prediction.

6.6.2 Bug Report Management. Bug reports are essential for any software development, which typically contain a detailed description of the failure and occasionally hint at the location of the fault in the source code in the form of patches or stack traces. They also allow users to inform developers of problems encountered while applying software applications.

Considering the impact of bug report assignment on the efficiency of development process, making a proper and effective assignment of each bug report is a crucial task. This can save much time and effort on bug-fixing activity. Jeong et al. [138] introduced a graph model based on Markov chains, which captures bug tossing history. They showed that the accuracy of bug assignment prediction is improved using naive bayes with tossing graph. Anvik et al. [16] presented a machine learning approach to create recommenders that assist with a variety of decisions aimed at streamlining the development process. They built three different development-oriented recommenders by applying six predictive models to suggest which developers might fix errors in the bug reports, which product components a report might pertain to, and which developers on the project might be interested in following the report. Jonsson et al.[148] conducted a study to evaluate machine learning classification based automated bug assignment techniques. They combined an ensemble learner Stacked Generalization (SG) with several classifiers and evaluated their performance with over 50,000 bug reports from two companies. Peters et al. [291] present a framework to filter and remove non-security bug reports containing security related keywords by training five predictive models – Random Forest, Naive Bayes, Logistic Regression, Multilayer Perceptron, and K Nearest Neighbor – on Four Apache projects and Chromium involving 45,940 bug reports. They found that their framework mitigates the class imbalance issue and reduces the number of mislabelled security bug reports by 38%. Proper bug report assignment also relies on proper bug report categorization. There are a number of studies that propose techniques to categorize bug reports. Among them a popular research area is reopened bug report prediction. bhattacharya et al. [32] characterized and predicted which bugs get reopened. They first qualitatively identified causes for bug reopens, and then built lmultiple machine learning approaches, e.g., Naive Bayes, Bayesian Network, C4.5, to predict the probability that a bug will be reopened. Xia et al. [397] proposed a novel approach called ReopenPredictor, which extract more textual features from the bug reports and combines decision tree and multinomial Naive Bayes to yield better performance for reopened bug prediction.

To reduce redundant effort, there has been much research into identification of duplicate bug reports [349, 350]. Sun et al. [350] used discriminative models to identify duplicates more accurately. They used a support vector machine (SVM) with linear kernel based on 54 text features. In a later work [349], they more fully utilized the information available in bug reports including not only textual content and description fields, but also non-textual fields (e.g., product, component, version) and proposed a retrieval function REP to identify duplicated reports. A two-round stochastic gradient descent was applied to automatically optimize REP in a supervised learning manner.

Zanetti et al. [426] proposed an efficient and practical method to identify valid bug reports i.e., the bug reports that refer to an actual software bug, are not duplicates, and contain enough information to be processed right away. They used support vector machine to identify valid bug reports based on nine network measures using a comprehensive data set of more than 700,000 bug reports obtained from the BUGZILLA installation of four major OSS communities. Fa et al. [84] also proposed a model to determine the valid bug reports by characterizing the features of the bug reports. To effectively analyze anomaly bug reports, Lucia et al. [203] proposed an approach to automatically identify and refine bug reports by the incorporation of user feedback. They first listed the top few anomaly reports from the list of reports generated by a tool in its default ordering. Based on feedback of users who either accepted or rejected each of the reports, their tool adopted nearest neighbor with non-nested generalization (NNGe) to automatically and iteratively refine a classification model for anomalies and resorted the rest of the reports.

A number of studies leverage bug report contents to address other bug fixing related tasks. Guo et al. [106] performed an empirical study to characterize factors that affect which bugs get fixed in Windows Vista and Windows 7. They concentrated on investigating factors related to bug report edits and relationships between people involved in fixing bugs, and built a statistical model using logistic regression to predict the probability that a new bug will be fixed. Kim et al. [159] proposed a two phase predictive model to identify the files likely to be fixed by analyzing bug reports contents. In order to collect predictable bug reports, they checked whether the given bug report contains sufficient information for prediction (predictable or deficient) in the first phase. Zhang et al. [433] proposed a Markov-based method for predicting the number of bugs that will be fixed in future. For a given bug report, they also constructed a KNN classification model to predict bug-fixing effort based on the intuition of similar bugs requiring similar time cost.

6.6.3 Software Quality Assessment. With the continuous release of new software versions, developers have been committed to maintaining a high level of software quality during the whole of SDLC. Generally, Software quality can be defined as “the degree to which a software product meets established requirements" and it involves many aspects, such as desirable characteristics of software products and process, techniques, or tools used to achieve those characteristics. In this section, we introduce the studies using predictive models for assessing software quality.

High quality software products can improve user experience and reduce developers’ effort in maintenance. Quality assessment is an important task to discover and fix new bugs that appear in products timely. Müller et al. [242] investigated whether biometrics are able to determine code quality concerns. They conducted a field study with ten professional developers and further adopted machine learning classifiers to predict developers’ perceived difficulty of code elements while working on a change task. Experimental results showed that biometrics are indeed able to predict quality concerns of parts of the code, and lower development and evaluation costs, improving upon a naive classifier by more than 26% and outperforming classifiers based on traditional metrics. Mills et al. [223] proposed a solution to the problem of predicting query quality by training a machine learning classifier based on 28 query measures. They applied a Random Forest algorithm to 12 open source systems implemented in Java and C++, defining seven post-retrieval query quality properties, which extended 21 pre-retrieval properties proposed by previous work [109, 111]. In order to improve the efficiency of developers in handling test reports, Chen et al. [57] present a framework to assess the quality of crowdsourced test reports. They trained a classifier using logistic regression to predict the quality of reports. The proposed framework achieved good performance in terms of precision, recall and F-measure.

6.6.4 Vulnerability Detection. An indicator being commonly used for evaluating software quality is software vulnerability. There are many studies on vulnerability detection using predictive models [65, 161, 319, 323, 326, 332, 436]. Shin et al. [332] conducted an empirical case study to investigate the impact of software metrics obtained in the early SDLC on predicting vulnerable code. They collected complexity, code churn, and developer activity metrics from two large-scale projects (Mozilla Firefox and the Linux kernel) and trained five machine learning models to identify vulnerabilities by applying these metrics. The experimental results showed that Naive Bayes achieve the best performance among other classifiers. Scandariato et al. [323] present an approach to predict which components of a software application contain security vulnerabilities by training five machine learning models. Based on [326], Zhang et al. [436] utilized deep learning techniques to detecting SQL injection vulnerabilities in PHP code. They observed that a classifier trained using CNN outperformed another one trained by Multilayer Perceptron (MLP). Saccente et al. [319] developed a prototype tool for identify method-level vulnerabilities within source code. The tool employed various data preparation methods to be independent of coding style and to automate the process of extracting methods, labeling data, and partitioning datasets. Dam et al [65] built a Long Short Term Memory deep learning-based model to automatically identify key features that predict vulnerable software components. Their experiments on Firefox and 18 Android apps showed that LSTM performs better than other predictive models for both within-project and cross-project vulnerability detection.

6.6.5 Performance Prediction. There are several studies focusing on the effect of parameters in predictive models or which parameter values lead to the best performance [340, 341, 358].

Thomas et al. [358] investigated the effectiveness of a large space of classifier configurations, 3,172 in total, and present a framework for combining the results of multiple classifier configurations since classifier combination has shown promise in certain domains. The evaluation results demonstrated that parameters of a classifier has a significant impact on the model's performance. They found that combining multiple classifiers improves the performance of even the best individual classifiers. Due to several limitations of the combinatorial interaction testing (CIT) approach, Song et al. [340] implemented an iterative learning algorithm called iTree, which effectively searched for a small part of configurations that closely approximated the effective configuration space of a system. Based on their previous work [341], the key improvements of are based on the use of composite proto-interactions, i.e. a construct that improves iTrees ability to correctly learn key configuration option combinations. This in turn significantly improves iTrees running time, without sacrificing effectiveness. Nair et al. [249] present a rank-based approach to rank software configurations and identify the optimal value without requiring exact performance values. Experiments were conducted on 21 scenarios for evaluating effectiveness of their strategy, and results showed that rank-based approach allows building an accurate performance model with very few data instances, helping developers to significantly reduce the cost of building models.

6.6.6 Code Smell Detection. Code smell refers to the symptoms of poor design and implementation choices in source code, which possibly indicate deeper problems for further development, maintenance, and evolution of software. A large number of studies have been proposed to automatically detect various types of code smells since it is tedious and time consuming to manually identify code smells [46, 266].

Fontana et al. [90] applied 16 different machine learning techniques on large-scale code smell instances. By comparing the performance of various machine learning algorithms, they found that all algorithms performed well on cross-validation datasets. Their experimental results showed that the highest performance was achieved by J48 and Random Forest, while SVMs had the worst performance in code smell detection. Palomba et al. [273] performed an empirical study to investigate the relationship between community smells and code smells. Community smells reflect the presence of organizational and socio-technical issues within a software community that may lead to additional project costs. Prior studies provided pieces of evidence that are often connected to circumstances such as code smells. Thus they proposed a code smell intensity predictive model and conducted a fine-grained analysis on 117 datasets for exploring how and the extent to which community smells impact code smell intensity. Their conclusion was that community smells result in the increasing intensity of code smells in projects. Liu et al. [198] proposed a neural network-based classifier that detects feature envy without the need for manually selecting features. Evaluation results showed that the proposed approach significantly improved the state-of-the-art in identifying feature envy smells.

6.6.7 Traceability Prediction. In Table 13, there are eight studies focusing on traceability detection, where over a half of studies used predictive models to recover traceability links. For example, Mills et al. [224] combined active learning and Random Forest to perform classification-based traceability link recovery. Panichella et al. [276] constructed a novel framework, which can automatically recover traceability links and identify bug fixing patches by using Latent Dirichlet Allocation as well as Genetic Algorithms. Hirao et al. [122] selected a set of machine learning and a neural netwok model (i.e., MLP) for link classification.

6.6.8 Code Clone Detection. Code clones have always been a double-edged sword in software development. On one hand, developers reuse the existing code snippets to complete development tasks, largely reducing coding effort. On the other hand, creating clones in source code may introduce new defects or bugs, which can lead to extra maintenance effort to ensure consistency among cloned code snippets. Since software clones have a clear influence on software maintenance and evolution, many studies [247, 320, 359, 380, 381] proposed various approaches to detect and management code clones in source code.

Saini et al. [320] implemented a novel code clone detection approach referred as to Oreo, which can be capable of detecting harder-to-detect clones (i.e., semantic clones) in the Twilight Zone. They employed five different models to detect clones and selected siamese architecture neural network as the final choice. They evaluated the recall of Oreo onvBigCloneBench, and Oreo achieved both high recall and precision. Nafi et al. [247] proposed a framework which can detect cross language clones (CLCs) with no need to generate the intermediate representation of source code. Their framework analyzed the different syntactic features of source code across different programming languages and applied siamese architecture neural network to detect CLCs. Their framework outperformed state-of-the-art approaches in detecting cross language clones in terms of precision, recall, and F-measure. Wu et al. [393] implemented a tool called SCDetector, which can identify functional code clones with high scalability and time-saving by training a Siamese Network. Since only a small part of code clones experience consistent changes during software evolution history, Wang et al. [381] defined a code cloning operation as consistency-maintenance-required if a code clone experience consistent changes in their evolution history, and leveraged Bayesian Networks to automatically predict whether a code cloning operation needs consistency maintenance when developers performing copy-and-paste operations. They evaluated the effectiveness of their approach on four projects under two usage scenarios. In order to perform the efficiency of clone management efforts, Thongtanunam et al. [359] conducted an empirical study on six open-source Java systems to investigate the life expectancy of clones. They found that 30% to 87% of clones were short-lived, and these short-lived clones were changed more frequently than long-lived clones throughout their lifetime. They also applied Random Forest classifier to predict the life expectancy of newly-introduced clones. Furthermore, they noticed that several features can influence the life expectancy of newly-introduced clones, such as the size and complexity of clones. Since many clone detectors return code fragments that are not considered clones by users, it is necessary to perform manual validation of the reported possible clones. Mostaeen et al. [238] present a tool called CloneCognition that can automate the laborious manual validation process by using Artificial Neural Network (ANN). Their tool showed promising performance when compared with state-of-the-art clone validation techniques.

6.6.9 Issue Detection. In Table 13, there are seven studies related to issue detection, and each study targets to identify different types of issues. Li et al. [186] used a clustering algorithm, i.e., DBSCAN clustering algorithm to identify and diagnose energy related issues from mobile applications. Choetkiertikul et al. [61] used a sparse logistic regression model to predict issue-related risks from software systems. Besides, Kikas et al. [158] extracted dynamic and textual features and utilized Random Forest to predict the lifetime of issues in GitHub projects.

6.6.10 SATD Detection. Five studies [129, 343, 382, 408, 425] concentrated on Self-Admitted Technical Debt(SATD) detection. Yan et al. [408] applied Random Forest to automatically detect change-level SATDs from source code, and Zampetti et al. [425] trained a deep learning model to learn patterns for remove useless SATD comments. Huang et al. [129] used multiple predictive models, including SVM, kNN, and NLP algorithms for SATD detection.

6.6.11 Malware Detection. Some software, especially mobile apps, can include significant malware. They are likely to implement functionalities which contradict user and organisational interests [18]. Generally, malware includes viruses, worms, trojans, key loggers and spyware, and are harmful at diverse severity scales. Malware can lead to damages of varying severity, ranging from spurious app crashes to financial losses with malware sending premium-rate SMS, and to private data leaks. A number of studies have aimed to better detect malware by using predictive models [20**?**]. Chandramohan et al. [49] proposed and evaluated a bounded feature space behavior modeling (BOFM) framework for scalable malware detection. They first extracted a feature vector that is bounded by an upper limit N using BOFM, and then trained an SVM to detect malware. Since there is no studies to detect malicious and aggressive push notification advertisement, Liu et al. [200] provided a taxonomy of push notifications and present a crowdsourcing-based approach to automatically detect aggressive push notifications in Android apps. Their approach used a guided testing approach to record push notifications and flagged the malicious ones by analyzing runtime information.

6.6.12 Log Analysis. Logging provides visibility into the health and performance of an application and infrastructure stack. Some studies concentrated on log analysis in order to enable development teams and system administrators to more easily diagnose and rectify issues. Russo et al. [316] proposed an approach for classification and prediction of defective log sequences. They used three well-known SVMs – radial basis function, multilayer perceptron and linear kernels – to predict and fit log sequences containing defects with high probability. The approach achieves comparable accuracy as other log analysis techniques. Li et al. [182] studied reasons of log changes and proposed an approach to determine whether log change suggestions are required while committing a code change. Zhang et al. [437] present LogRobust, a log-based anomaly detection approach, using an attention-based Bi-LSTM model. They evaluated LogRobust on Hadoop and the results demonstrated that LogRobust can identify and handle stable log events and sequences with high accuracy.

6.7 Software Management

Software management is related to the skills, knowledge and attitudes that software engineers need to possess to practice software engineering in a professional, responsible and ethical manner. The study of professional practices includes the areas of technical communication, group dynamics and human aspects including psychology, emotions, team dynamics, social and professional responsibilities [325]. On the other hand, the goal of software development is to meet user desires or needs [36]. We introduce some key related studies [131] into these developers' and users' perspectives that use predictive models in Table 14 (See in Appendix).

6.7.1 Developer Behavior Analysis. Fritz et al. [91] implemented a novel approach to detect when software developers are experiencing difficulty while they work on their programming tasks, and stop developers' behavior before they introduce bugs into the code. They classify the difficulty of code comprehension tasks using data from psycho-physiological sensors and chose Naive Bayes as the classification algorithm because its training can be updated on-the-fly. Some papers used predictive models to recognize developers' emotions [60, 97, 241]. Muller et al. [241] investigated developers emotions, progress and applied biometric measures to classify them in the context of software change tasks. They trained a J48 decision tree to distinguish between positive and negative emotions based on biometric measurements. Bacchelli et al. [19] presented an approach to classify email content at line level. Their technique fused naive Bayes algorithm with island parsing to perform automatic classification of the content of development emails into five language categories: natural language text, source code fragments, stack traces, code patches, and junk. Their technique can help developers subsequently apply ad hoc analysis techniques for each category. Later, Sorbo et al. [73] proposed a semi-supervised approach named DECA (Development Emails Content Analyzer) to mine intention from developer emails. DECA uses Natural Language Parsing to classify the content of development emails according to their different purposes (e.g., feature request, opinion asking, problem discovery, solution proposal, or information giving), identifying email elements that can be used for specific tasks. They showed the superiority of DECA to traditional machine learning techniques. Egelman et al. [75] used a logistic progression model to identify developers' negative feelings from code review logs. They found suggested that they can cause negative repercussions although negative experiences are relatively rare in practice.

Zhou et al. [443] measured, understood, and predicted how the newcomers involvement and environment in issue tracking systems affects their odds of becoming a long term contributor. They constructed nine measures of involvement and environment based on events recorded in an issue tracking system and used logistic regression model to predict long term contributors. Wood et al. [389] conducted an empirical study with 30 professional programmers and trained a supervised learning algorithm to identify speech act types in developers' conversations in order to obtain useful information for bug repair.

6.7.2 Software Repository Mining. Software repositories, such as GitHub and Q&A sites, contain a large amount of knowledge related to software development. Mining these repositories is able to recover valuable knowledge that can help developers address difficulties when programming [8, 31, 434]. However, the quality of a high proportion of questions and answers in repositories is still a concern. To ensure that researchers reach realistic and accurate conclusions, Munaiah et al. [243] trained a Random Forest classifier for sieving out the noise in GitHub repositories. They evaluated their framework on 200 repositories with known ground truth classification. Xu et al. [401] applied deep learning techniques to predict Semantically Linkable Knowledge units in Stack Overflow. They formulated the problem of identifying knowledge units as a multi-class classification task and trained a convolutional neural network (CNN) model by exploiting informative word-level and document-level features. Prana et al. [298] manually annotated 4,226 README files from GitHub repositories and designed a multi-label classifier to classify these files into eight different categories. Bao et al. [25] proposed a method named psc2code to extract source code from programming screencasts. They first trained a CNN model to classify programming images for filtering non-code or noisy-code frames, and then identified the screen region that is most likely to be a code editor based on edge detection and clustering-based image segmentation technologies.

In order to capture the meaning of a sentence at a higher level of abstraction, Jha and Mahmoud [139] applied frame semantics to generate low-dimensional representations of text and used SVM and Naive Bayes to classify app store reviews into three categories of actionable software maintenance requests (i.e., bug reports, feature requests, and otherwise), enhancing the predictive capabilities and reducing the chances of overfitting. Martens et al. [216] performed a survey of 43 fake review providers to study the significant differences between fake reviews and non-fake reviews. They then implemented seven classifiers to automatically detect fake reviews in app stores. The experimental results showed that the Random Forest algorithm achieved the best performance compared to other models.

6.7.3 Other Tasks. In Table 14 (See in Appendix), some research topics only involves several related studies, including software classification, software release management, license prediction, energy efficiency, and behavior detection. For software classification, Linares et al. [194] used a set of machine learning algorithms (i.e., Support Vector Machines, Naive Bayes, Decision Tree, RIPPER, and IBK) to automatically classify software applications into different domain categories. Nayebi et al. [258] employed Decision tree, SVM and Random Forest to predict the released versions of applications. Mcintosh et al. [218] selected a set of predictive models for predicting software energy consumption. Martens et al. [216] adopted several machine learning and neural network models (i.e., Naive Bayes, Random Forest (RF), Decision Tree (DT), Support Vector Machine (SVM), Linear support vector classification(LinearSVC), Multilayer perceptron (MLP)) to detect fake reviews in App store.

Summary of answers to RQ3:

(1) This section presents the mapping between every study and the corresponding predictive models.
(2) The application of predictive models in 43 research topics throughout the SDLC were summarized.
(3) Software maintenance is the domain in which 55% of predictive models have been applied in the selected primary studies.
(4) 111 studies use predictive models for software defect prediction.
(5) Only 8 of the studies worked on requirements classification using predictive models.
(6) Researchers may want to focus more attention on using predictive models in analyzing software requirements and design.

7 THREATS TO VALIDITY

7.1 Publication Bias

Publication bias is the issue of publishing more positive results over negative ones. Claims to reject or support a hypothesis will be biased if the original publication is suffering from bias. A tendency toward positive study outcomes rather than others leads to biased and even possibly incorrect conclusions, while some preferences in publishing are useful. Studies with a null result might not always be worse than studies with significant positive results, although significant results have a statistically higher chance of getting published [164]. In this paper, we select six top and prevalent software engineering venues. Thus, to some extent, the publications included in these venues are high-quality, which can reduce or eliminate the influence of publication bias on conclusions.

7.2 Search Terms

Finding all relevant primary studies is still a challenge to any survey or literature review-based study. To tackle this issue, a detailed search strategy was prepared and performed in our research. Search terms were constructed with different strings identified by checking titles and keywords from relevant publications already known to the authors. Alternative synonyms and spellings for search terms were then modified by consulting an expert. These procedures provided a high confidence that the majority of the key studies were identified.

7.3 Study Selection Bias

The publication selection process was carried out in two stages. In the first stage, studies were excluded based on the title and abstract independently by two researchers with extensive experience in software engineering. We conducted a pilot study of publication selection process to place a foundation in order to better understand the inclusion/ exclusion criteria, and evaluated inter-rater reliability to mitigate the threat emerged from the researchers' personal subjective judgment. When two researchers could not reach an agreement on a study, a third researcher was consulted. In the second stage, studies were eliminated based on the full paper. Thus there is little possibility to miss relevant studies through this well-established study selection process.

7.4 Data Extraction

Data extraction was conducted by two of the researchers, and the data extracted from the relevant studies was rechecked by the other researchers. Disagreements in the data extraction process were discussed and solved after the pilot data extraction so that the researchers could complete the data extraction process following the refinement of the criteria, improving the validity of our analysis.

8 CHALLENGES AND OPPORTUNITIES

A number of research challenges and opportunities relating to the use of predictive models in software engineering remain, requiring further investigation. In this section, we discuss the limitations of using predictive models in different SE research areas and conclude four key findings pertaining to challenges when building these models. We also present several recommendations for future work.

8.1 Challenges

8.1.1 Challenge 1: Quality of Datasets. Source datasets for training and testing are critcial for predictive model-based approaches. The most significant challenge for most studies is whether the information in the datasets available reflect ground truth about the related software engineering tasks. With different research areas, there are a large number of datasets that can be used for experiments. However, the datasets vary much in quality, including bias, noise, size, imbalance and mislabelling. Any of these will influence the effectiveness of the predictive models used. For example, a dataset with much noise may cause underfitting in building predictive models. A dataset with small size may cause overfitting in building predictive models. A heavily imbalanced dataset may even fail to build useful models at all.

Data imbalance: The datasets are usually not distributed evenly [146]. In some cases, over 90% of the total data have data imbalance problem. Some datasets even contain a large proportion of false positive instances. This issue causes the class imbalance problem, which has a large influence on the performance of predictive models. Learning a minority class is difficult when a classifier is trained on imbalanced data. Therefore, the classifier is skewed toward the majority class resulting in a lower rate of detection. Several studies proposed different methods to address this problem, such as instance re-weighting [248], data re-sampling [230, 418], and selective learning strategy [126].

Noise in datasets: Compared with the acceptable range of the feature spaces, data instances having excessive deviation can be referred to as outliers [125]. Classifiers trained on noisy data with many outliers (e.g., incorrect, missing information) are expected to be less accurate. Since the problem of noisy data is inevitable from time to time when dealing with certain datasets such as software repositories, the usage of such noisy data is likely to pose a threat to the validity of many studies [163]. Prior studies demonstrated that noisy may creep into datasets and impact predictive models when collecting data carelessly [163]. There are two lines of related studies focusing on this problem. The first conducted a comprehensive investigation of the impact of noisy data on models performance [168]. The second line of studies proposed different textual similarity approaches to address this challenge [168].

Size of datasets: Most of selected studies mentioned the problem of generality of their proposed approaches as an external threat due to lack of enough data. Using representative datasets is one of crucial factors towards performing experiments [304]. For instance, some studies trained their predictive models using source code in a single programming language such as C, Java or Python. This means the training classes cannot comprehensively cover all possible patterns and may prevent those algorithms from addressing corresponding tasks in all cases. In addition, overfitting may affect models' ability when training models without large-scale datasets. A number of studies have worked on alleviating this problem by constructing large-scale datasets from various software repositories [199]. However, a sufficiently large scale volume of data in some communities is not yet available to be used.

Data Privacy: Data privacy issues arise when the confidentiality of the data is a concern for the project owners. This issue in turn may cause the owners to not contribute to the pool of available data even though such data might contribute to further research efforts. Privacy is considered by Peters et al. [288] in the context of Cross-Company Defect Prediction (CCDP). They obfuscate the data in order to hide the project details when they are shared among multiple parties.

Data heterogeneity: The outcome of predictive models can be easy influenced by the similarity of source and target data distributions, and this problem is called data heterogeneity. As expressed by Canfora et al. [43], most software projects are heterogeneous and have various metric distributions. Moreover, certain context factors (e.g., size, domain, and programming language) have a huge impact on data heterogeneity. Machine learning techniques can work well under the assumption that source and target data have the same distribution. Some studies [369, 451] present different approaches such as filtering (F), transformation (DT), normalization (N) and feature matching to tackle this problem.

8.1.2 Challenge 2: Feature selection. As many candidate features are introduced to better represent various research domains, many studies extracted and used a greater variety of features to train predictive models, increasing the size of research space of features. However, not all features used are beneficial to improving a predictive model's performance [65]. First, some interdependent features do not fit when applied to build models. Another reason is that using too many features may result in overfitting. Therefore, how to select the most suitable features in a huge research space has become a critical challenge. Feature selection techniques ensure that useless features are removed. Some studies [301, 335] employed such techniques in order to optimize performance and scalability. A good feature extractor should meet two main principles. First of all, it should be automatic so that it won't cost too much manual effort. Second, it can reduce feature dimension while keeping feature quality. That is, given a specific task, the feature extractor can identify and preserve relevant features and remove irrelevant features to improve the performance of predictive models. Automatic feature selection may even be possible in some research directions [65].

8.1.3 *Challenge 3: Evaluation metric selection.*

Diverse evaluation metrics have been used to evaluate the effectiveness of different models. However, evaluation metrics may introduce bias. For example, [129, 449] highlight the problem of the suitability of the chosen evaluation metrics. Most primary studies solve this challenge by utilizing standard metrics. Instead of using standard metrics such as precision, recall and F-measure, many studies [91, 203, 248, 443] have defined new evaluation measures to evaluate the performance of their proposed frameworks when addressing the same problem. This makes it difficult to evaluate the performance of their frameworks. On the other hand, the measures used for evaluation models keep changing thanks to new techniques being found. Therefore, it is a challenging task to achieve optimal selection of evaluation measures.

8.1.4 *Challenge 4: Guidelines for the selection of suitable predictive models.*

From our survey, we see that there are various software engineering tasks leveraging predictive models. However, different predictive models fit better for different tasks. With various tasks and various predictive models, there is a need to have reliable guidelines for how to select a good predictive model for a specific tasks. Among the reviewed papers, some studies have proposed frameworks for specific tasks to make comparison of various predictive models [68, 94, 229, 237, 291]. However, almost all of them are based on experiments, which always have threats to validity. There ideally should be some theoretical guidelines that can help researchers select the right predictive models according to the characteristics of the specific SE tasks.

The selection of classifiers is another possible source of bias. Given the variety of available learning algorithms, there are still others that could have been considered. An appropriate selection might be guided by the aim of finding a meaningful balance ,or trade-off, between established techniques and novel approaches.

8.2 Opportunities

8.2.1 *Opportunity 1: Leveraging the power of big data.*

The scale of available SE data has rapidly become larger and larger. A growing number of studies tend to be large-scale. The biggest advantage of big data is that it generally leads to robust predictive models, improving the practicability and generality. If a study only uses several small datasets to build predictive models it may not be generalisable and its results may have serious threats to validity. On the contrary, results achieved using big data are usually more convincing and more likely to have generality. In addition, deep learning techniques usually require a large amount of data to learn their predictive models, since there are many more parameters to learn than when using traditional predictive models. With the trend to using big data, reducing the time and space cost for data computation can also become a challenging issue.

8.2.2 *Opportunity 2: Neural network based predictive models.*

Studies that apply predictive models to software engineering tasks occurred over several decades. Although researchers have achieved significant improvement from simple linear regression to complex ensemble learning, they have been bottlenecks for addressing many software engineering tasks. Since 2016, deep learning, as an advanced predictive model, has become more and more popular. Some studies that tried to leverage deep learning for software engineering tasks have achieved even better performance than all other start-of-the-art techniques.

The biggest advantage of deep learning is that it can automatically generate more expressive features that are better for learning predictive models, which can't be done when using traditional predictive models [65]. As mentioned above, many software engineering tasks face the feature extraction challenge, i.e., either it is hard to manually infer proper features or there are too many features needing to be carefully selected for building their predictive models. In recent years, Some studies [92, 222] call into question whether the performance of DL techniques can surpass that of machine learning or other traditional algorithms. Therefore, a potential challenge is to conduct a wide range of comparative studies to investigate the performance of DL techniques and other based predictive models in diverse SE tasks, respectively. In this way, we can deeply recognize the differences between the characteristics of DL techniques and traditional predictive models, which can lead us to correctly select appropriate algorithms facing specific SE issues.

8.2.3 Opportunity 3: Assessment and selection of predictive models. Clearly, computational efficiency and transparency are desirable features of candidate classifiers and appear to be a promising area for future research to formalize these concepts, e.g., by developing a multidimensional classifier assessment system [181].

Consequently, the assessment and selection of a predictive model should not be based on predictive accuracy alone but should be comprised of several additional criteria. These include computational efficiency, ease of use, and especially comprehensibility. Comprehensible models reveal the nature of the detected relationships and help improve our overall understanding of software failures and their sources, which, in turn, may enable the development of novel predictors in various research directions.

Efforts to design new software metrics and other explanatory variables appear to be a particularly promising area for future research. They have the potential to achieve general accuracy improvements across all types of classifiers.

8.2.4 Opportunity 4: predictive models in specific research domains. **Predictive models in software requirements and testing:** According to our analysis we found that predictive models were rarely applied in certain domains, such as software requirements, software design and implementation. There are many different practical problems that are able to be tackled by leveraging predictive models in these two topics. We believe that researchers can try to apply state-of-the-art models on classification, prioritization and prediction tasks in both software requirements and testing.

Predictive models in identifying different defects: As mentioned above, a significant number of studies have applied predictive models to identify various types of defects, such as logical defects, syntax defects, interface defects, security defects, and performance defects. However, there are no studies that investigate on which defect types the existing defect predictive models work the best. In future work, researchers can conduct studies to investigate the performance of different categories of predictive models on predicting different types of defects. We may also add specific contexts when performing experiments.

8.2.5 Opportunity 5: The usability analysis of prediction tasks. There is a significant difference between the accuracy and usefulness of predictions across projects. Projects with a very low percentage of positives and a very high number of classes are intuitively hard to predict for most approaches. However, this task is even harder for humans to perform manually. User studies are therefore needed to provide insights into when and where automated techniques are useful to humans, expecting to find a break even point where automation becomes more accurate than fully manual techniques.

Another potential direction is understanding the utility of change impact analysis. The ability to predict which classes are impacted by a new requirement can potentially support a range of tasks including refactoring decisions, defect prediction, and effort estimation [81]. An increased understanding of such tasks could enable developers to build more effective prediction algorithms.

9 CONCLUSION

We wanted to analyse the use of predictive models in software engineering. A comprehensive analysis was performed to answer our defined research questions using a systematic review covering 190 selected papers published between 2009 and 2020 in top SE venues. These studies were identified by following a systematic series of steps and assessing the quality of the studies. Our key findings from this study are summarized below:

(1) The cumulative number of predictive model related studies shows an increasing trend between 2009 and 2020, and most of the selected primary studies focus on proposing novel approaches.
(2) We found 148 different predictive models were employed in software engineering tasks. These models can be classified into six categories – rule-based techniques, statistical models, NLP models, machine learning algorithms, ensemble techniques, and neural networks.
(3) Naive Bayes and Logistic Regression are the most widely used learning techniques to build predictive models for SE tasks to date. Several machine learning models are also popular models for addressing specific problems, including SVM and decision trees.
(4) Precision, recall and F-measure are the most frequently used performance metrics for evaluating the effectiveness of predictive models.
(5) The application of deep learning techniques has seen an increasing trend in recent years. CNN- and RNN-based models are the two most popular deep learning models used in the studies for various SE tasks.
(6) Most predictive models have been applied to the defect prediction task and 5 of 8 research topics where predictive models are frequently used belong to software maintenance, 2 topics are in the domain of software management.
(7) We identified a number of challenges when using these predictive models for SE tasks, including issues with dataset quality, feature and evaluation metric selection, and lack of model selection guidelines.

The main objective of this survey was to analyze and classify the use of existing predictive models and their related studies in SE to date. A range of future studies should include more applications of predictive models to requirements engineering and testing, and conduct further research to investigate the characteristics of different predictive models, including deep learning techniques, for providing more useful guidelines during model selection.

ACKNOWLEDGEMENTS

This research is supported by ARC Laureate Fellowship (FL190100035) and the National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] Zahra Shakeri Hossein Abad, Vincenzo Gervasi, Didar Zowghi, and Behrouz H Far. 2019. Supporting analysts by dynamic extraction and classification of requirements-related knowledge. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 442–453.

[2] Hani Abdeen, Khaled Bali, Houari Sahraoui, and Bruno Dufour. 2015. Learning dependency-based change impact predictors using independent change histories. *Information and Software Technology* 67 (2015), 220–235.

[3] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2018. Testing vision-based control systems using learnable evolutionary algorithms. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1016–1026.

[4] Surafel Lemma Abebe, Nasir Ali, and Ahmed E Hassan. 2016. An empirical study of software release notes. *Empirical Software Engineering* 21, 3 (2016), 1107–1142.

[5] Sallam Abualhaija, Chetan Arora, Mehrdad Sabetzadeh, Lionel C Briand, and Michael Traynor. 2020. Automated demarcation of requirements in textual specifications: a machine learning-based approach. *Empirical Software Engineering* 25, 6 (2020), 5454–5497.
[6] Petar Afric, Lucija Sikic, Adrian Satja Kurdija, and Marin Silic. 2020. REPD: Source code defect prediction as anomaly detection. *Journal of Systems and Software* 168 (2020), 110641.
[7] Amritanshu Agrawal and Tim Menzies. 2018. Is" Better Data" Better Than" Better Data Miners"?. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1050–1061.
[8] Md Ahasanuzzaman, Muhammad Asaduzzaman, Chanchal K Roy, and Kevin A Schneider. 2020. CAPS: a supervised technique for classifying Stack Overflow posts concerning API issues. *Empirical Software Engineering* 25, 2 (2020), 1493–1532.
[9] Jehad Al Dallal and Lionel C Briand. 2012. A precise method-method interaction-based cohesion metric for object-oriented classes. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21, 2 (2012), 1–34.
[10] Mohammad Alahmadi, Abdulkarim Khormi, and Sonia Haiduc. 2020. UI screens identification and extraction from mobile programming screencasts. In *Proceedings of the 28th International Conference on Program Comprehension*. 319–330.
[11] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 281–293.
[12] Kevin Allix, Tegawendé F Bissyandé, Quentin Jérome, Jacques Klein, Yves Le Traon, et al. 2016. Empirical assessment of machine learning-based malware detectors for Android. *Empirical Software Engineering* 21, 1 (2016), 183–211.
[13] Kamel Alrashedy, Dhanush Dharmaretnam, Daniel M German, Venkatesh Srinivasan, and T Aaron Gulliver. 2020. SCC++: predicting the programming language of questions and snippets of Stack Overflow. *Journal of Systems and Software* 162 (2020), 110505.
[14] Anunay Amar and Peter C Rigby. 2019. Mining historical test logs to predict bugs and localize faults in the test logs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 140–151.
[15] Preethu Rose Anish, Balaji Balasubramaniam, Abhishek Sainani, Jane Cleland-Huang, Maya Daneva, Roel J Wieringa, and Smita Ghaisas. 2016. Probing for requirements knowledge to stimulate architectural thinking. In *Proceedings of the 38th International Conference on Software Engineering*. 843–854.
[16] John Anvik and Gail C Murphy. 2011. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20, 3 (2011), 1–35.
[17] Erik Arisholm, Lionel C Briand, and Eivind B Johannessen. 2010. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software* 83, 1 (2010), 2–17.
[18] Ömer Aslan Aslan and Refik Samet. 2020. A Comprehensive Review on Malware Detection Approaches. *IEEE Access* 8 (2020), 6249–6271.
[19] Alberto Bacchelli, Tommaso Dal Sasso, Marco D'Ambros, and Michele Lanza. 2012. Content classification of development emails. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 375–385.
[20] Yude Bai, Zhenchang Xing, Xiaohong Li, Zhiyong Feng, and Duoyuan Ma. 2020. Unsuccessful story about few shot malware family classification and siamese network to the rescue. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1560–1571.
[21] Liang Bao, Qian Li, Peiyao Lu, Jie Lu, Tongxiao Ruan, and Ke Zhang. 2018. Execution anomaly detection in large-scale systems through console log analysis. *Journal of Systems and Software* 143 (2018), 172–186.
[22] Liang Bao, Xin Liu, Ziheng Xu, and Baoyin Fang. 2018. Autoconfig: Automatic configuration tuning for distributed message systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 29–40.
[23] Lingfeng Bao, Xin Xia, David Lo, and Gail C Murphy. 2019. A large scale study of long-time contributor prediction for GitHub projects. *IEEE Transactions on Software Engineering* (2019).
[24] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, and Shanping Li. 2017. Who will leave the company?: a large-scale industry study of developer turnover by mining monthly work report. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 170–181.
[25] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, Minghui Wu, and Xiaohu Yang. 2020. psc2code: Denoising Code Extraction from Programming Screencasts. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 3 (2020), 1–38.
[26] Robert M Bell, Thomas J Ostrand, and Elaine J Weyuker. 2013. The limited impact of individual developer data on software defect prediction. *Empirical Software Engineering* 18, 3 (2013), 478–505.
[27] Kwabena Ebo Bennin, Jacky W Keung, and Akito Monden. 2019. On the relative value of data resampling approaches for software defect prediction. *Empirical Software Engineering* 24, 2 (2019), 602–636.
[28] Tristan Benoit, Jean-Yves Marion, and Sébastien Bardin. 2021. Binary level toolchain provenance identification with graph neural networks. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering*

(SANER). IEEE, 131–141.

[29] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 309–321.

[30] Seifeddine Bettaieb, Seung Yeob Shin, Mehrdad Sabetzadeh, Lionel C Briand, Michael Garceau, and Antoine Meyers. 2020. Using machine learning to assist with the selection of security controls during security assessment. *Empirical Software Engineering* 25, 4 (2020), 2550–2582.

[31] Stefanie Beyer, Christian Macho, Massimiliano Di Penta, and Martin Pinzger. 2020. What kind of questions do developers ask on Stack Overflow? A comparison of automated approaches to classify posts into question categories. *Empirical Software Engineering* 25, 3 (2020), 2258–2301.

[32] Pamela Bhattacharya, Iulian Neamtiu, and Christian R Shelton. 2012. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. *Journal of Systems and Software* 85, 10 (2012), 2275–2292.

[33] Pan Bian, Bin Liang, Jianjun Huang, Wenchang Shi, Xidong Wang, and Jian Zhang. 2020. SinkFinder: harvesting hundreds of unknown interesting function pairs with just one seed. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1101–1113.

[34] Kelly Blincoe, Ali Dehghan, Abdoul-Djawadou Salaou, Adam Neal, Johan Linaker, and Daniela Damian. 2019. High-level software requirements and iteration changes: a predictive model. *Empirical Software Engineering* 24, 3 (2019), 1610–1648.

[35] Kelly Blincoe, Giuseppe Valetto, and Daniela Damian. 2013. Do all task dependencies require coordination? the role of task properties in identifying critical coordination needs in software projects. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 213–223.

[36] Pierre Bourque, Richard E Fairley, et al. 2014. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0.* IEEE Computer Society Press.

[37] 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*. 330–341.

[38] Caius Brindescu, Iftekhar Ahmed, Rafael Leano, and Anita Sarma. 2020. Planning for untangling: Predicting the difficulty of merge conflicts. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 801–811.

[39] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. Bilateral dependency neural networks for cross-language algorithm classification. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 422–433.

[40] Raymond PL Buse and Westley Weimer. 2009. The road not taken: Estimating path execution frequency statically. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 144–154.

[41] Raymond PL Buse and Westley R Weimer. 2009. Learning a metric for code readability. *IEEE Transactions on Software Engineering* 36, 4 (2009), 546–558.

[42] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974.

[43] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. 2015. Defect prediction as a multiobjective optimization problem. *Software Testing, Verification and Reliability* 25, 4 (2015), 426–459.

[44] Casey Casalnuovo, Prem Devanbu, Abilio Oliveira, Vladimir Filkov, and Baishakhi Ray. 2015. Assert use in github projects. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 755–766.

[45] Gemma Catolino, Fabio Palomba, Andrea De Lucia, Filomena Ferrucci, and Andy Zaidman. 2018. Enhancing change prediction models using developer-related factors. *Journal of Systems and Software* 143 (2018), 14–28.

[46] Gemma Catolino, Fabio Palomba, Francesca Arcelli Fontana, Andrea De Lucia, Andy Zaidman, and Filomena Ferrucci. 2020. Improving change prediction models with code smell-related information. *Empirical Software Engineering* 25, 1 (2020), 49–95.

[47] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software* 152 (2019), 165–181.

[48] Narciso Cerpa, Matthew Bardeen, Cesar A Astudillo, and June Verner. 2016. Evaluating different families of prediction methods for estimating software project outcomes. *Journal of Systems and Software* 112 (2016), 48–64.

[49] Mahinthan Chandramohan, Hee Beng Kuan Tan, Lionel C Briand, Lwin Khin Shar, and Bindu Madhavi Padmanabhuni. 2013. A scalable approach for malware detection through bounded feature space behavior modeling. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 312–322.

[50] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. 2020. BUILDFAST: History-Aware Build Outcome Prediction for Fast Feedback and Reduced Cost in Continuous Integration. In *2020 35th IEEE/ACM International Conference on*

Automated Software Engineering (ASE). IEEE, 42–53.

[51] Guibin Chen, Chunyang Chen, Zhenchang Xing, and Bowen Xu. 2016. Learning a dual-language vector space for domain-specific cross-lingual question retrieval. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 744–755.

[52] Huanchao Chen, Yuan Huang, Zhiyong Liu, Xiangping Chen, Fan Zhou, and Xiaonan Luo. 2019. Automatically detecting the scopes of source code comments. *Journal of Systems and Software* 153 (2019), 45–63.

[53] 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*. 578–589.

[54] Jinfu Chen, Patrick Kwaku Kudjo, Solomon Mensah, Selasie Aformaley Brown, and George Akorfu. 2020. An automatic software vulnerability classification framework using term frequency-inverse gravity moment and feature selection. *Journal of Systems and Software* 167 (2020), 110616.

[55] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why My Code Summarization Model Does Not Work: Code Comment Improvement with Category Prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–29.

[56] Tse-Hsun Chen, Stephen W Thomas, Hadi Hemmati, Meiyappan Nagappan, and Ahmed E Hassan. 2017. An empirical study on the effect of testing on code quality using topic models: A case study on software development systems. *IEEE Transactions on Reliability* 66, 3 (2017), 806–824.

[57] Xin Chen, He Jiang, Xiaochen Li, Liming Nie, Dongjin Yu, Tieke He, and Zhenyu Chen. 2020. A systemic framework for crowdsourced test report quality assessment. *Empirical Software Engineering* (2020), 1–37.

[58] Yang Chen, Andrew E Santosa, Ang Ming Yi, Abhishek Sharma, Asankhaya Sharma, and David Lo. 2020. A machine learning approach for vulnerability curation. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 32–42.

[59] Zhenpeng Chen, Yanbin Cao, Xuan Lu, Qiaozhu Mei, and Xuanzhe Liu. 2019. SEntiMoji: an emoji-powered learning approach for sentiment analysis in software engineering. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 841–852.

[60] Zhenpeng Chen, Yanbin Cao, Huihan Yao, Xuan Lu, Xin Peng, Hong Mei, and Xuanzhe Liu. 2021. Emoji-powered Sentiment and Emotion Detection from Software Developers' Communication Data. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–48.

[61] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. 2015. Characterization and prediction of issue-related risks in software projects. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 280–291.

[62] Deshan Cooray, Sam Malek, Roshanak Roshandel, and David Kilgore. 2010. RESISTing reliability degradation through proactive reconfiguration. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 83–92.

[63] Domenico Cotroneo, Roberto Pietrantuono, and Stefano Russo. 2013. A learning-based method for combining testing techniques. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 142–151.

[64] 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 *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 46–57.

[65] Hoa Khanh Dam, Truyen Tran, Trang Thi Minh Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2018. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering* (2018).

[66] Andre B De Carvalho, Aurora Pozo, and Silvia Regina Vergilio. 2010. A symbolic fault-prediction model based on multiobjective particle swarm optimization. *Journal of Systems and Software* 83, 5 (2010), 868–882.

[67] Ali Dehghan, Adam Neal, Kelly Blincoe, Johan Linaker, and Daniela Damian. 2017. Predicting likelihood of requirement implementation within the planned iteration: an empirical study at ibm. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 124–134.

[68] Karel Dejaeger, Wouter Verbeke, David Martens, and Bart Baesens. 2011. Data mining techniques for software effort estimation: a comparative study. *IEEE transactions on software engineering* 38, 2 (2011), 375–397.

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

[70] Tapajit Dey, Sara Mousavi, Eduardo Ponce, Tanner Fry, Bogdan Vasilescu, Anna Filippova, and Audris Mockus. 2020. Detecting and characterizing bots that commit code. In *Proceedings of the 17th international conference on mining software repositories*. 209–219.

[71] Dario Di Nucci, Fabio Palomba, Sandro Siravo, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. 2015. On the role of developer's scattered changes in bug prediction. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 241–250.

[72] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea 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)*. IEEE, 612–621.

[73] Andrea Di Sorbo, Sebastiano Panichella, Corrado A Visaggio, Massimiliano Di Penta, Gerardo Canfora, and Harald C Gall. 2015. Development emails content analyzer: Intention mining in developer discussions (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 12–23.

[74] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M Rao, RP Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 118–128.

[75] Carolyn D Egelman, Emerson Murphy-Hill, Elizabeth Kammer, Margaret Morrow Hodges, Collin Green, Ciera Jaspan, and James Lin. 2020. Predicting developers' negative feelings about code review. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 174–185.

[76] Beyza Eken, RiFat Atar, Sahra Sertalp, and Ayşe Tosun. 2019. Predicting Defects with Latent and Semantic Features from Commit Logs in an Industrial Setting. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, 98–105.

[77] Beyza Eken and Ayse Tosun. 2021. Investigating the performance of personalized models for software defect prediction. *Journal of Systems and Software* 181 (2021), 111038.

[78] Sarah Fakhoury, Venera Arnaoudova, Cedric Noiseux, Foutse Khomh, and Giuliano 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)*. IEEE, 602–611.

[79] Davide Falessi, Giovanni Cantone, and Gerardo Canfora. 2011. Empirical principles and an industrial case study in retrieving equivalent requirements via natural language processing techniques. *IEEE Transactions on Software Engineering* 39, 1 (2011), 18–44.

[80] Davide Falessi, Massimiliano Di Penta, Gerardo Canfora, and Giovanni Cantone. 2017. Estimating the number of remaining links in traceability recovery. *Empirical Software Engineering* 22, 3 (2017), 996–1027.

[81] Davide Falessi, Justin Roll, Jin LC Guo, and Jane Cleland-Huang. 2018. Leveraging Historical Associations between Requirements and Source Code to Identify Impacted Classes. *IEEE Transactions on Software Engineering* (2018).

[82] Yuanrui Fan, D Alencar da Costa, D Lo, AE Hassan, and L Shanping. 2020. The impact of mislabeled changes by szz on just-in-time defect prediction. *IEEE Transactions on Software Engineering* (2020).

[83] Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E Hassan, and Shanping Li. 2019. The Impact of Changes Mislabeled by SZZ on Just-in-Time Defect Prediction. *IEEE Transactions on Software Engineering* (2019).

[84] Yuanrui FAN, Xin XIA, David LO, and Ahmed E HASSAN. [n.d.]. Chaff from the wheat: Characterizing and determining valid bug reports.(2018). *IEEE Transactions on Software Engineering* ([n. d.]), 1–30.

[85] Yuanrui Fan, Xin Xia, David Lo, and Ahmed E Hassan. 2018. Chaff from the wheat: Characterizing and determining valid bug reports. *IEEE transactions on software engineering* 46, 5 (2018), 495–525.

[86] Yuanrui Fan, Xin Xia, David Lo, and Shanping Li. 2018. Early prediction of merged code changes to prioritize reviewing tasks. *Empirical Software Engineering* 23, 6 (2018), 3346–3393.

[87] 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*. 516–527.

[88] Henning Femmer, Dharmalingam Ganesan, Mikael Lindvall, and David McComas. 2013. Detecting inconsistencies in wrappers: A case study. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1022–1031.

[89] Antonio Filieri, Giordano Tamburrelli, and Carlo Ghezzi. 2015. Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. *IEEE Transactions on Software Engineering* 42, 1 (2015), 75–99.

[90] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21, 3 (2016), 1143–1191.

[91] Thomas Fritz, Andrew Begel, Sebastian C Müller, Serap Yigit-Elliott, and Manuela Züger. 2014. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th international conference on software engineering*. 402–413.

[92] Wei Fu and Tim Menzies. 2017. Easy over hard: A case study on deep learning. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 49–60.

[93] Wei Fu and Tim Menzies. 2017. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 72–83.

[94] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. 2015. Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software*

Engineering, Vol. 1. IEEE, 789–800.

[95] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. 2012. Method-level bug prediction. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 171–180.

[96] Emanuel Giger, Martin Pinzger, and Harald C Gall. 2011. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 83–92.

[97] Daniela Girardi, Nicole Novielli, Davide Fucci, and Filippo Lanubile. 2020. Recognizing developers' emotions while programming. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 666–677.

[98] Lina Gong, Shujuan Jiang, Rongcun Wang, and Li Jiang. 2019. Empirical Evaluation of the Impact of Class Overlap on Software Defect Prediction. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 698–709.

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

[100] 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*. 243–253.

[101] Swapna Gottipati, David Lo, and Jing Jiang. 2011. Finding relevant answers in software forums. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 323–332.

[102] Giovanni Grano, Fabio Palomba, and Harald C Gall. 2019. Lightweight assessment of test-case effectiveness using source-code-quality indicators. *IEEE Transactions on Software Engineering* (2019).

[103] Alex Groce, Todd Kulesza, Chaoqiang Zhang, Shalini Shamasunder, Margaret Burnett, Weng-Keen Wong, Simone Stumpf, Shubhomoy Das, Amber Shinsel, Forrest Bice, et al. 2013. You are the only possible oracle: Effective test selection for end users of interactive machine learning systems. *IEEE Transactions on Software Engineering* 40, 3 (2013), 307–323.

[104] Yongfeng Gu, Jifeng Xuan, Hongyu Zhang, Lanxin Zhang, Qingna Fan, Xiaoyuan Xie, and Tieyun Qian. 2019. Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. *Journal of Systems and Software* 148 (2019), 88–104.

[105] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 301–311.

[106] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. 2010. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 495–504.

[107] Huong Ha and Hongyu Zhang. 2019. Deepperf: performance prediction for configurable software with deep sparse neural network. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1095–1106.

[108] Andrew Habib and Michael Pradel. 2018. Is this class thread-safe? inferring documentation using graph-based learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 41–52.

[109] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 842–851.

[110] Sonia Haiduc, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Andrian Marcus. 2012. Automatic query performance assessment during the retrieval of software artifacts. In *Proceedings of the 27th IEEE/ACM international conference on Automated Software Engineering*. 90–99.

[111] Sonia Haiduc, Gabriele Bavota, Rocco Oliveto, Andrian Marcus, and Andrea De Lucia. 2012. Evaluating the specificity of text retrieval queries to support software engineering tasks. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1273–1276.

[112] Ines Hajri, Arda Goknil, Fabrizio Pastore, and Lionel C Briand. 2020. Automating system test case classification and prioritization for use case-driven testing in product lines. *Empirical Software Engineering* 25, 5 (2020), 3711–3769.

[113] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. 2017. Learning to predict severity of software vulnerability using only vulnerability description. In *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 125–136.

[114] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. 2014. Finding patterns in static analysis alerts: improving actionable alert ranking. In *Proceedings of the 11th working conference on mining software repositories*. 152–161.

[115] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2010. Fault-prone module detection using large-scale text features based on spam filtering. *Empirical Software Engineering* 15, 2 (2010), 147–165.

[116] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2012. Bug prediction based on fine-grained module histories. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 200–210.

[117] Jianjun He, Ling Xu, Meng Yan, Xin Xia, and Yan Lei. 2020. Duplicate bug report detection using dual-channel convolutional neural networks. In *Proceedings of the 28th International Conference on Program Comprehension.* 117–127.

[118] Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. 2015. An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology* 59 (2015), 170–190.

[119] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering.* 152–162.

[120] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering* 21, 2 (2016), 303–336.

[121] Kim Herzig and Nachiappan Nagappan. 2014. The impact of test ownership and team structure on the reliability and effectiveness of quality test runs. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* 1–10.

[122] Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. 2019. The review linkage graph for code review analytics: a recovery approach and empirical study. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 578–589.

[123] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR).* IEEE, 34–45.

[124] Andre Hora, Marco Tulio Valente, Romain Robbes, and Nicolas Anquetil. 2016. When should internal interfaces be promoted to public?. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 278–289.

[125] Seyedrebvar Hosseini, Burak Turhan, and Dimuthu Gunarathna. 2017. A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering* 45, 2 (2017), 111–147.

[126] Seyedrebvar Hosseini, Burak Turhan, and Mika Mäntylä. 2018. A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction. *Information and Software Technology* 95 (2018), 296–312.

[127] Wei Hu and Kenny Wong. 2013. Using citation influence to predict software defects. In *2013 10th Working Conference on Mining Software Repositories (MSR).* IEEE, 419–428.

[128] Jeff Huang, Qingzhou Luo, and Grigore Rosu. 2015. GPredict: Generic predictive concurrency analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 847–857.

[129] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* 23, 1 (2018), 418–451.

[130] Qiao Huang, Xin Xia, and David Lo. 2017. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 159–170.

[131] Qiao HUANG, Xin XIA, David LO, and Gail C MURPHY. [n.d.]. Automating intention mining.(2018). *IEEE Transactions on Software Engineering* ([n. d.]), 1–22.

[132] Qiao Huang, Xin Xia, David Lo, and Gail C Murphy. 2018. Automating intention mining. *IEEE Transactions on Software Engineering* 46, 10 (2018), 1098–1119.

[133] Rubing Huang, Weifeng Sun, Yinyin Xu, Haibo Chen, Dave Towey, and Xin Xia. 2019. A Survey on Adaptive Random Testing. *IEEE Transactions on Software Engineering* (2019).

[134] Yi Huang, Chunyang Chen, Zhenchang Xing, Tian Lin, and Yang Liu. 2018. Tell them apart: distilling technology differences from crowd-scale comparison discussions. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 214–224.

[135] 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.

[136] Zijie Huang, Zhiqing Shao, Guisheng Fan, Jianhua Gao, Ziyi Zhou, Kang Yang, and Xingguang Yang. 2021. Predicting Community Smells' Occurrence on Individual Developers by Sentiments. *arXiv preprint arXiv:2103.07090* (2021).

[137] Md Shariful Islam, Abdelwahab Hamou-Lhadj, Korosh Koochekian Sabor, Mohammad Hamdaqa, and Haipeng Cai. 2021. EnHMM: On the Use of Ensemble HMMs and Stack Traces to Predict the Reassignment of Bug Report Fields. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 411–421.

[138] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. 2009. Improving bug triage with bug tossing graphs. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.* 111–120.

[139] Nishant Jha and Anas Mahmoud. 2018. Using frame semantics for classifying and summarizing application store reviews. *Empirical Software Engineering* 23, 6 (2018), 3734–3767.

[140] Zhouyang Jia, Shanshan Li, Tingting Yu, Xiangke Liao, and Ji Wang. 2019. Automatically detecting missing cleanup for ungraceful exits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 751–762.

[141] He Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. 2017. What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 712–723.

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

[143] Jiajun Jiang, Weihai Lu, Junjie Chen, Qingwei Lin, Pu Zhao, Yu Kang, Hongyu Zhang, Yingfei Xiong, Feng Gao, Zhangwei Xu, et al. 2020. How to mitigate the incident? an effective troubleshooting guide recommendation technique for online service systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1410–1420.

[144] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Ieee, 279–289.

[145] Xiaoyuan Jing, Fei Wu, Xiwei Dong, Fumin Qi, and Baowen Xu. 2015. Heterogeneous cross-company defect prediction by unified metric representation and CCA-based transfer learning. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 496–507.

[146] Xiao-Yuan Jing, Fei Wu, Xiwei Dong, and Baowen Xu. 2016. An improved SDA based defect prediction framework for both within-project and cross-project class-imbalance problems. *IEEE Transactions on Software Engineering* 43, 4 (2016), 321–339.

[147] Xiao-Yuan Jing, Shi Ying, Zhi-Wu Zhang, Shan-Shan Wu, and Jin Liu. 2014. Dictionary learning based software defect prediction. In *Proceedings of the 36th International Conference on Software Engineering*. 414–423.

[148] Leif Jonsson, Markus Borg, David Broman, Kristian Sandahl, Sigrid Eldh, and Per Runeson. 2016. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering* 21, 4 (2016), 1533–1578.

[149] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, Mark D Syer, and Ahmed E Hassan. 2018. Examining the stability of logging statements. *Empirical Software Engineering* 23, 1 (2018), 290–333.

[150] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-based sampling of software configuration spaces. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1084–1094.

[151] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21, 5 (2016), 2072–2106.

[152] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773.

[153] Edward YY Kan, Wing Kwong Chan, and TH Tse. 2012. EClass: an execution classification approach to improving the energy-efficiency of software via machine learning. *Journal of Systems and Software* 85, 4 (2012), 960–973.

[154] Ritu Kapur and Balwinder Sodhi. 2020. A defect estimator for source code: Linking defect reports with programming constructs usage metrics. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 2 (2020), 1–35.

[155] Staffs Keele et al. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report. Technical report, Ver. 2.3 EBSE Technical Report. EBSE.

[156] Foutse Khomh, Brian Chan, Ying Zou, Anand Sinha, and Dave Dietz. 2011. Predicting post-release defects using pre-release field testing results. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 253–262.

[157] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2011. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software* 84, 4 (2011), 559–572.

[158] Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. 2016. Using dynamic and contextual features to predict issue lifetime in GitHub projects. In *2016 ieee/acm 13th working conference on mining software repositories (msr)*. IEEE, 291–302.

[159] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering* 39, 11 (2013), 1597–1610.

[160] Dongsun Kim, Xinming Wang, Sunghun Kim, Andreas Zeller, Shing-Chi Cheung, and Sooyong Park. 2011. Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Transactions on Software Engineering* 37, 3 (2011), 430–447.

[161] I Luk Kim, Yunhui Zheng, Hogun Park, Weihang Wang, Wei You, Yousra Aafer, and Xiangyu Zhang. 2020. Finding client-side business flow tampering vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on*

Software Engineering. 222–233.

[162] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196.

[163] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. 2011. Dealing with noise in defect prediction. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 481–490.

[164] Barbara A Kitchenham, Tore Dyba, and Magne Jorgensen. 2004. Evidence-based software engineering. In *Proceedings. 26th International Conference on Software Engineering*. IEEE, 273–281.

[165] Kenichi Kobayashi, Akihiko Matsuo, Katsuro Inoue, Yasuhiro Hayase, Manabu Kamimura, and Toshiaki Yoshino. 2011. ImpactScale: Quantifying change impact to predict faults in large software systems. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 43–52.

[166] Masanari Kondo, Cor-Paul Bezemer, Yasutaka Kamei, Ahmed E Hassan, and Osamu Mizuno. 2019. The impact of feature reduction techniques on defect prediction models. *Empirical Software Engineering* 24, 4 (2019), 1925–1963.

[167] Masanari Kondo, Daniel M German, Osamu Mizuno, and Eun-Hye Choi. 2020. The impact of context metrics on just-in-time defect prediction. *Empirical Software Engineering* 25, 1 (2020), 890–939.

[168] SB Kotsiantis, Dimitris Kanellopoulos, and PE Pintelas. 2006. Data preprocessing for supervised leaning. *International Journal of Computer Science* 1, 2 (2006), 111–117.

[169] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2011. Design evolution metrics for defect prediction in object oriented systems. *Empirical Software Engineering* 16, 1 (2011), 141–175.

[170] Rahul Krishna, Tim Menzies, and Wei Fu. 2016. Too much automation? The bellwether effect and its implications for transfer learning. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 122–131.

[171] Patrick Kwaku Kudjo and Jinfu Chen. 2019. A cost-effective strategy for software vulnerability prediction based on bellwether analysis. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 424–427.

[172] Lov Kumar, Sai Krishna Sripada, Ashish Sureka, and Santanu Ku Rath. 2018. Effective fault prediction model developed using least square support vector machine (LSSVM). *Journal of Systems and Software* 137 (2018), 686–712.

[173] Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. 2010. Predicting the severity of a reported bug. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 1–10.

[174] Tien-Duy B Le, Mario Linares-Vásquez, David Lo, and Denys Poshyvanyk. 2015. Rclinker: Automated linking of issue reports and commits leveraging rich contextual information. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 36–47.

[175] Tien-Duy B Le and David Lo. 2013. Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 310–319.

[176] Tien-Duy B Le, Ferdian Thung, and David Lo. 2017. Will this localization tool be effective for this bug? Mitigating the impact of unreliability of information retrieval based bug localization tools. *Empirical Software Engineering* 22, 4 (2017), 2237–2279.

[177] Claire Le Goues and Westley Weimer. 2011. Measuring code quality to improve specification mining. *IEEE Transactions on Software Engineering* 38, 1 (2011), 175–190.

[178] Sangho Lee, Changhee Jung, and Santosh Pande. 2014. Detecting memory leaks through introspective dynamic behavior modelling using machine learning. In *Proceedings of the 36th International Conference on Software Engineering*. 814–824.

[179] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. 2011. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 311–321.

[180] Taek Lee, Jaechang Nam, Donggyun Han, Sunghun Kim, and Hoh Peter In. 2016. Developer micro interaction metrics for software defect prediction. *IEEE Transactions on Software Engineering* 42, 11 (2016), 1015–1035.

[181] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering* 34, 4 (2008), 485–496.

[182] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E Hassan. 2017. Towards just-in-time suggestions for log changes. *Empirical Software Engineering* 22, 4 (2017), 1831–1865.

[183] Ke Li, Zilin Xiang, Tao Chen, and Kay Chen Tan. 2020. BiLO-CPDP: Bi-Level Programming for Automated Model Discovery in Cross-Project Defect Prediction. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 573–584.

[184] Wanchun Li, Mary Jean Harrold, and Carsten Görg. 2010. Detecting user-visible failures in AJAX web applications by analyzing users' interaction behaviors. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 155–158.

[185] Xiang Li, Chetan Mutha, and Carol S Smidts. 2016. An automated software reliability prediction system for safety critical software. *Empirical Software Engineering* 21, 6 (2016), 2413–2455.

[186] Xueliang Li, Yuming Yang, Yepang Liu, John P Gallagher, and Kaishun Wu. 2020. Detecting and diagnosing energy issues for mobile applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 115–127.

[187] Yangguang Li, Zhen Ming Jiang, Heng Li, Ahmed E Hassan, Cheng He, Ruirui Huang, Zhengda Zeng, Mian Wang, and Pinan Chen. 2020. Predicting Node Failures in an Ultra-large-scale Cloud Computing Platform: an AIOps Solution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 2 (2020), 1–24.

[188] Zhiqiang Li, Xiao-Yuan Jing, Xiaoke Zhu, and Hongyu Zhang. 2017. Heterogeneous defect prediction through multiple kernel learning and ensemble learning. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 91–102.

[189] Soo Ling Lim, Daniele Quercia, and Anthony Finkelstein. 2010. StakeNet: using social networks to analyse the stakeholders of large-scale software projects. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 295–304.

[190] Bin Lin, Fiorella Zampetti, Gabriele Bavota, Massimiliano Di Penta, and Michele Lanza. 2019. Pattern-based mining of opinions in Q&A websites. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 548–559.

[191] Bin Lin, Fiorella Zampetti, Gabriele Bavota, Massimiliano Di Penta, Michele Lanza, and Rocco Oliveto. 2018. Sentiment Analysis for So ware Engineering: How Far Can We Go?(2018). (2018).

[192] Dayi Lin, Cor-Paul Bezemer, and Ahmed E Hassan. 2019. Identifying gameplay videos that exhibit bugs in computer games. *Empirical Software Engineering* 24, 6 (2019), 4006–4033.

[193] Qingwei Lin, Ken Hsieh, Yingnong Dang, Hongyu Zhang, Kaixin Sui, Yong Xu, Jian-Guang Lou, Chenggang Li, Youjiang Wu, Randolph Yao, et al. 2018. Predicting node failure in cloud service systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 480–490.

[194] Mario Linares-Vásquez, Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. 2014. On using machine learning to automatically classify software applications into domain categories. *Empirical Software Engineering* 19, 3 (2014), 582–618.

[195] 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).

[196] Hui Liu, Mingzhu Shen, Jiahao Jin, and Yanjie Jiang. 2020. Automated classification of actions in bug reports of mobile apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 128–140.

[197] Han Liu, Chengnian Sun, Zhendong Su, Yu Jiang, Ming Gu, and Jiaguang Sun. 2017. Stochastic optimization of program obfuscation. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 221–231.

[198] Hui Liu, Zhifeng Xu, and Yanzhen Zou. 2018. Deep learning based feature envy detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 385–396.

[199] Pei Liu, Li Li, Yanjie Zhao, Xiaoyu Sun, and John Grundy. 2018. AndroZooOpen: Collecting Large-scale Open Source Android Apps for the Research Community. *Star* 1, 800 (2018), 1300.

[200] Tianming Liu, Haoyu Wang, Li Li, Guangdong Bai, Yao Guo, and Guoai Xu. 2019. DaPanda: Detecting Aggressive Push Notifications in Android Apps. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 66–78.

[201] Xiaoyu Liu, LiGuo Huang, Jidong Ge, and Vincent Ng. 2019. Predicting Licenses for Changed Source Code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 686–697.

[202] Yibin Liu, Yanhui Li, Jianbo Guo, Yuming Zhou, and Baowen Xu. 2018. Connecting software metrics across versions to predict defects. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 232–243.

[203] Lucia, David Lo, Lingxiao Jiang, Aditya Budi, et al. 2012. Active refinement of clone anomaly reports. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 397–407.

[204] Sicheng Luo, Hui Xu, Yanxiang Bi, Xin Wang, and Yangfan Zhou. 2021. Boosting symbolic execution via constraint solving time prediction (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 336–347.

[205] Ping Ma, Hangyuan Cheng, Jingxuan Zhang, and Jifeng Xuan. 2020. Can this fault be detected: A study on fault detection via automated test generation. *Journal of Systems and Software* 170 (2020), 110769.

[206] Wanwangying Ma, Lin Chen, Yibiao Yang, Yuming Zhou, and Baowen Xu. 2016. Empirical analysis of network measures for effort-aware fault-proneness prediction. *Information and Software Technology* 69 (2016), 50–70.

[207] 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.

[208] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, Helen Sharp and Mike Whalen (Eds.). IEEE / ACM, 91–100. https://doi.org/10.1109/ICSE-SEIP.2019.00018

[209] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 91–100.

[210] Christian Macho, Shane McIntosh, and Martin Pinzger. 2016. Predicting build co-changes with source code change and commit categories. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 541–551.

[211] Chandra Maddila, Chetan Bansal, and Nachiappan Nagappan. 2019. Predicting pull request completion time: a case study on large scale cloud services. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 874–882.

[212] Ruchika Malhotra and Megha Khanna. 2017. An empirical study for software change prediction using imbalanced data. *Empirical Software Engineering* 22, 6 (2017), 2806–2851.

[213] Ruchika Malhotra and Megha Khanna. 2019. Dynamic selection of fitness function for software change prediction using particle swarm optimization. *Information and Software Technology* 112 (2019), 51–67.

[214] Haroon Malik, Hadi Hemmati, and Ahmed E Hassan. 2013. Automatic detection of performance deviations in the load testing of large scale systems. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 1012–1021.

[215] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 304–315.

[216] Daniel Martens and Walid Maalej. 2019. Towards understanding and detecting fake reviews in app stores. *Empirical Software Engineering* 24, 6 (2019), 3316–3355.

[217] Mohammad Jafar Mashhadi and Hadi Hemmati. 2020. Hybrid deep neural networks to infer state models of black-box systems. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 299–311.

[218] Andrea McIntosh, Safwat Hassan, and Abram Hindle. 2019. What can Android mobile app developers do about the energy consumption of machine learning? *Empirical Software Engineering* 24, 2 (2019), 562–601.

[219] Collin McMillan, Mario Linares-Vasquez, Denys Poshyvanyk, and Mark Grechanik. 2011. Categorizing software applications for maintenance. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 343–352.

[220] Ibéria Medeiros, Nuno Neves, and Miguel Correia. 2015. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability* 65, 1 (2015), 54–69.

[221] Andrew Meneely, Pete Rotella, and Laurie Williams. 2011. Does adding manpower also affect quality? an empirical, longitudinal analysis. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 81–90.

[222] Tim Menzies, Suvodeep Majumder, Nikhila Balaji, Katie Brey, and Wei Fu. 2018. 500+ times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow). In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 554–563.

[223] Chris Mills, Gabriele Bavota, Sonia Haiduc, Rocco Oliveto, Andrian Marcus, and Andrea De Lucia. 2017. Predicting query quality for applications of text retrieval to software engineering tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 26, 1 (2017), 1–45.

[224] Chris Mills, Javier Escobar-Avila, Aditya Bhattacharya, Grigoriy Kondyukov, Shayok Chakraborty, and Sonia Haiduc. 2019. Tracing with Less Data: Active Learning for Classification-Based Traceability Link Recovery. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 103–113.

[225] Mehdi Mirakhorli and Jane Cleland-Huang. 2015. Detecting, tracing, and monitoring architectural tactics in code. *IEEE Transactions on Software Engineering* 42, 3 (2015), 205–220.

[226] Mehdi Mirakhorli, Yonghee Shin, Jane Cleland-Huang, and Murat Cinar. 2012. A tactic-centric approach for automating traceability of quality concerns. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 639–649.

[227] Ayse Tosun Misirli and Ayse Basar Bener. 2014. Bayesian networks for evidence-based decision-making in software engineering. *IEEE Transactions on Software Engineering* 40, 6 (2014), 533–554.

[228] Ayse Tosun Misirli, Emad Shihab, and Yasukata Kamei. 2016. Studying high impact fix-inducing changes. *Empirical Software Engineering* 21, 2 (2016), 605–641.

[229] Nikolaos Mittas and Lefteris Angelis. 2012. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *IEEE Transactions on software engineering* 39, 4 (2012), 537–551.

[230] Audris Mockus. 2009. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 11–20.

[231] Facundo Molina, Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre, and Marcelo Frias. 2019. Training binary classifiers as data structure invariants. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 759–770.

[232] Akito Monden, Takuma Hayashi, Shoji Shinoda, Kumiko Shirai, Junichi Yoshida, Mike Barker, and Kenichi Matsumoto. 2013. Assessing the cost effectiveness of fault prediction in acceptance testing. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1345–1357.

[233] Joao Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. 2019. Identifying experts in software libraries and frameworks among github users. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 276–287.

[234] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering* 46, 2 (2018), 196–221.

[235] Kevin Moran, David N Palacio, Carlos Bernal-Cárdenas, Daniel McCrystal, Denys Poshyvanyk, Chris Shenefiel, and Jeff Johnson. 2020. Improving the effectiveness of traceability link recovery using hierarchical bayesian networks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 873–885.

[236] Laura Moreno, Gabriele Bavota, Sonia Haiduc, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, and Andrian Marcus. 2015. Query-based configuration of text retrieval solutions for software engineering tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 567–578.

[237] Toshiki Mori and Naoshi Uchihira. 2019. Balancing the trade-off between accuracy and interpretability in software defect prediction. *Empirical Software Engineering* 24, 2 (2019), 779–825.

[238] 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*. 1105–1109.

[239] Rebecca Moussa and Danielle Azar. 2017. A PSO-GA approach targeting fault-prone software modules. *Journal of Systems and Software* 132 (2017), 41–49.

[240] Debdoot Mukherjee and Malika Garg. 2013. Which work-item updates need your response?. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 12–21.

[241] Sebastian C Müller and Thomas Fritz. 2015. Stuck and frustrated or in flow and happy: sensing developers' emotions and progress. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 688–699.

[242] Sebastian C Müller and Thomas Fritz. 2016. Using (bio) metrics to predict code quality online. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 452–463.

[243] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253.

[244] Alessandro Murgia, Marco Ortu, Parastou Tourani, Bram Adams, and Serge Demeyer. 2018. An exploratory qualitative and quantitative analysis of emotions in issue report comments of open source systems. *Empirical Software Engineering* 23, 1 (2018), 521–564.

[245] Syed Shariyar Murtaza, Nazim Madhavji, Mechelle Gittens, and Zude Li. 2011. Diagnosing new faults using mutants and prior faults (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*. 960–963.

[246] Pradeep K Murukannaiah and Munindar P Singh. 2015. Platys: An active learning framework for place-aware application development and its evaluation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 3 (2015), 1–32.

[247] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K Roy, and Kevin A Schneider. 2019. CLCDSA: cross language code clone detection using syntactical features and API documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1026–1037.

[248] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*. 284–292.

[249] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using bad learners to find good configurations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 257–267.

[250] Daye Nam, Amber Horvath, Andrew Macvean, Brad Myers, and Bogdan Vasilescu. 2019. Marble: Mining for boilerplate code to identify API usability problems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 615–627.

[251] Jaechang Nam, Wei Fu, Sunghun Kim, Tim Menzies, and Lin Tan. 2017. Heterogeneous defect prediction. *IEEE Transactions on Software Engineering* 44, 9 (2017), 874–896.

[252] Jaechang Nam and Sunghun Kim. 2015. Clami: Defect prediction on unlabeled datasets (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 452–463.

[253] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 382–391.

[254] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, and Yang Liu. 2018. A multi-view context-aware approach to Android malware detection and malicious code localization. *Empirical Software Engineering* 23, 3 (2018), 1222–1274.

[255] Ali Rezaei Nasab, Mojtaba Shahin, Peng Liang, Mohammad Ehsan Basiri, Seyed Ali Hoseyni Raviz, Hourieh Khalajzadeh, Muhammad Waseem, and Amineh Naseri. 2021. Automated identification of security discussions in microservices systems: Industrial surveys and experiments. *Journal of Systems and Software* 181 (2021), 111046.

[256] Roberto Natella, Domenico Cotroneo, Joao A Duraes, and Henrique S Madeira. 2012. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering* 39, 1 (2012), 80–96.

[257] Maleknaz Nayebi, Henry Cho, and Guenther Ruhe. 2018. App store mining is not enough for app improvement. *Empirical Software Engineering* 23, 5 (2018), 2764–2794.

[258] Maleknaz Nayebi, Homayoon Farahi, and Guenther Ruhe. 2017. Which version should be released to app store?. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 324–333.

[259] Shiva Nejati, Mehrdad Sabetzadeh, Chetan Arora, Lionel C Briand, and Felix Mandoux. 2016. Automated change impact analysis between SysML models of requirements and design. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 242–253.

[260] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 263–272.

[261] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. 2012. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 70–79.

[262] Son Nguyen, Hung Phan, Trinh Le, and Tien N Nguyen. 2020. Suggesting natural method names to check name consistencies. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1372–1384.

[263] Zhen Ni, Bin Li, Xiaobing Sun, Tianhao Chen, Ben Tang, and Xinchen Shi. 2020. Analyzing bug fix for automatic bug cause classification. *Journal of Systems and Software* 163 (2020), 110538.

[264] Haoran Niu, Iman Keivanloo, and Ying Zou. 2017. Learning to rank code examples for code search engines. *Empirical Software Engineering* 22, 1 (2017), 259–291.

[265] Aditya V Nori and Rahul Sharma. 2013. Termination proofs from tests. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 246–256.

[266] Miroslaw Ochodek, Regina Hebig, Wilhelm Meding, Gert Frost, and Miroslaw Staron. 2020. Recognizing lines of code violating company-specific coding guidelines using machine learning. *Empirical Software Engineering* 25, 1 (2020), 220–265.

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

[268] Ahmet Okutan and Olcay Taner Yildiz. 2016. A novel kernel to predict software defectiveness. *Journal of Systems and Software* 119 (2016), 109–121.

[269] Moein Owhadi-Kareshk, Sarah Nadi, and Julia Rubin. 2019. Predicting merge conflicts in collaborative software development. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–11.

[270] Burcu Ozcelik and Cemal Yilmaz. 2015. Seer: a lightweight online failure prediction approach. *IEEE Transactions on Software Engineering* 42, 1 (2015), 26–46.

[271] Rohan Padhye, Senthil Mani, and Vibha Singhal Sinha. 2014. NeedFeed: taming change notifications by modeling code relevance. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 665–676.

[272] Fabio Palomba and Damian Andrew Tamburri. 2021. Predicting the emergence of community smells using socio-technical metrics: a machine-learning approach. *Journal of Systems and Software* 171 (2021), 110847.

[273] Fabio Palomba, Damian Andrew Andrew Tamburri, Francesca Arcelli Fontana, Rocco Oliveto, Andy Zaidman, and Alexander Serebrenik. 2018. Beyond technical aspects: How do community smells influence the intensity of code smells? *IEEE transactions on software engineering* (2018).

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

[275] Cong Pan and Michael Pradel. 2021. Continuous test suite failure prediction. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 553–565.

[276] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimilano Di Penta, Denys Poshynanyk, and Andrea De Lucia. 2013. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 522–531.

[277] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 537–548.

[278] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. 2019. Classifying code comments in Java software systems. *Empirical Software Engineering* 24, 3 (2019), 1499–1537.

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

[280] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. Fine-grained just-in-time defect prediction. *Journal of Systems and Software* 150 (2019), 22–36.

[281] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2020. On the performance of method-level bug prediction: A negative result. *Journal of Systems and Software* 161 (2020), 110493.

[282] Fabiano Pecorelli, Dario Di Nucci, Coen De Roover, and Andrea De Lucia. 2020. A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. *Journal of Systems and Software* 169 (2020), 110693.

[283] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea 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)*. IEEE, 93–104.

[284] Jiayan Pei, Yimin Wu, Zishan Qin, Yao Cong, and Jingtao Guan. 2021. Attention-based model for predicting question relatedness on Stack Overflow. *arXiv preprint arXiv:2103.10763* (2021).

[285] Daniel Perez and Shigeru Chiba. 2019. Cross-language clone detection by learning over abstract syntax trees. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 518–528.

[286] Anna Perini, Angelo Susi, and Paolo Avesani. 2012. A machine learning approach to software requirements prioritization. *IEEE Transactions on Software Engineering* 39, 4 (2012), 445–461.

[287] Fayola Peters and Tim Menzies. 2012. Privacy and utility for defect prediction: Experiments with morph. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 189–199.

[288] Fayola Peters, Tim Menzies, Liang Gong, and Hongyu Zhang. 2013. Balancing privacy and utility in cross-company defect prediction. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1054–1068.

[289] Fayola Peters, Tim Menzies, and Lucas Layman. 2015. LACE2: Better privacy-preserving data sharing for cross project defect prediction. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 801–811.

[290] Fayola Peters, Tim Menzies, and Andrian Marcus. 2013. Better cross company defect prediction. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 409–418.

[291] Fayola Peters, Thein Tun, Yijun Yu, and Bashar Nuseibeh. 2017. Text filtering and ranking for security bug report prediction. *IEEE Transactions on Software Engineering* (2017).

[292] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology* 64 (2015), 1–18.

[293] Teerat Pitakrat, Dušan Okanović, André van Hoorn, and Lars Grunske. 2018. Hora: Architecture-aware online failure prediction. *Journal of Systems and Software* 137 (2018), 669–685.

[294] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2021. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. *arXiv preprint arXiv:2103.07068* (2021).

[295] Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2011. Ecological inference in empirical software engineering. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 362–371.

[296] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 209–220.

[297] Michael Pradel, Vijayaraghavan Murali, Rebecca Qian, Mateusz Machalica, Erik Meijer, and Satish Chandra. 2020. Scaffle: bug localization on millions of files. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 225–236.

[298] Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. 2019. Categorizing the content of GitHub README files. *Empirical Software Engineering* 24, 3 (2019), 1296–1327.

[299] Rahul Premraj and Kim Herzig. 2011. Network versus code metrics to predict defects: A replication study. In *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, 215–224.

[300] Yu Qu, Qinghua Zheng, Jianlei Chi, Yangxu Jin, Ancheng He, Di Cui, Hengshan Zhang, and Ting Liu. 2019. Using K-core Decomposition on Class Dependency Networks to Improve Bug Prediction Model's Practical Performance. *IEEE Transactions on Software Engineering* (2019).

[301] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 432–441.

[302] Foyzur Rahman, Sameer Khatri, Earl T Barr, and Premkumar Devanbu. 2014. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*. 424–434.
[303] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. 2012. Recalling the" imprecision" of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.
[304] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar Devanbu. 2013. Sample size vs. bias in defect prediction. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. 147–157.
[305] Mohammad Masudur Rahman, Chanchal K Roy, and Raula G Kula. 2017. Predicting usefulness of code review comments using textual features and developer experience. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 215–226.
[306] Gopi Krishnan Rajbahadur, Shaowei Wang, Yasutaka Kamei, and Ahmed E Hassan. 2017. The impact of using regression models to build defect classifiers. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 135–145.
[307] Sandra L Ramírez-Mora, Hanna Oktaba, and Helena Gómez-Adorno. 2020. Descriptions of issues and comments for predicting issue success in software projects. *Journal of Systems and Software* 168 (2020), 110663.
[308] Michael Rath, Jacob Rendall, Jin LC Guo, Jane Cleland-Huang, and Patrick Mäder. 2018. Traceability in the wild: automatically augmenting incomplete trace links. In *Proceedings of the 40th International Conference on Software Engineering*. 834–845.
[309] Inayat Rehman, Mehdi Mirakhorli, Meiyappan Nagappan, Azat Aralbay Uulu, and Matthew Thornton. 2018. Roles and impacts of hands-on software architects in five industrial case studies. In *Proceedings of the 40th International Conference on Software Engineering*. 117–127.
[310] Luyao Ren, Shurui Zhou, Christian Kästner, and Andrzej Wąsowski. 2019. Identifying redundancies in fork-based development. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 230–241.
[311] Peter C Rigby and Martin P Robillard. 2013. Discovering essential code elements in informal documentation. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 832–841.
[312] Paige Rodeghero, Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Detecting user story information in developer-client conversations to generate extractive summaries. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 49–59.
[313] Daniele Romano and Martin Pinzger. 2011. Using source code metrics to predict change-prone java interfaces. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 303–312.
[314] Arunava Roy and Hoang Pham. 2018. Toward the development of a conventional time series based web error forecasting framework. *Empirical Software Engineering* 23, 2 (2018), 570–644.
[315] Devjeet Roy, Sarah Fakhoury, John Lee, and Venera Arnaoudova. 2020. A Model to Detect Readability Improvements in Incremental Changes. In *Proceedings of the 28th International Conference on Program Comprehension*. 25–36.
[316] Barbara Russo, Giancarlo Succi, and Witold Pedrycz. 2015. Mining system logs to learn error predictors: a case study of a telemetry system. *Empirical Software Engineering* 20, 4 (2015), 879–927.
[317] Duksan Ryu, Okjoo Choi, and Jongmoon Baik. 2016. Value-cognitive boosting with a support vector machine for cross-project defect prediction. *Empirical Software Engineering* 21, 1 (2016), 43–71.
[318] Arash Sabbaghi, Mohammad Reza Keyvanpour, and Saeed Parsa. 2020. FCCI: A fuzzy expert system for identifying coincidental correct test cases. *Journal of Systems and Software* 168 (2020), 110635.
[319] Nicholas Saccente, Josh Dehlinger, Lin Deng, Suranjan Chakraborty, and Yin 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)*. IEEE, 114–121.
[320] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. 2018. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 354–365.
[321] Fabio Santos, Igor Wiese, Bianca Trinkenreich, Igor Steinmacher, Anita Sarma, and Marco Gerosa. 2021. Can I Solve It? Identifying APIs Required to Complete OSS Task. *arXiv preprint arXiv:2103.12653* (2021).
[322] Aindrila Sarkar, Peter C Rigby, and Béla Bartalos. 2019. Improving bug triaging with high confidence predictions at ericsson. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 81–91.
[323] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering* 40, 10 (2014), 993–1006.
[324] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural networks* 61 (2015), 85–117.
[325] Linda Shafer. 2014. (CSDP) Software Engineering Professional Practice. (2014).
[326] Lwin Khin Shar, Hee Beng Kuan Tan, and Lionel C Briand. 2013. Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis. In *2013 35th International Conference on Software Engineering (ICSE)*.

IEEE, 642–651.

[327] Lin Shi, Mingzhe Xing, Mingyang Li, Yawen Wang, Shoubin Li, and Qing Wang. 2020. Detection of hidden feature requests from massive chat messages via deep siamese network. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 641–653.

[328] Ying Shi, Ming Li, Steven Arndt, and Carol Smidts. 2017. Metric-based software reliability prediction approach and its application. *Empirical Software Engineering* 22, 4 (2017), 1579–1633.

[329] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M Ibrahim, Masao Ohira, Bram Adams, Ahmed E Hassan, and Ken-ichi Matsumoto. 2013. Studying re-opened bugs in open source software. *Empirical Software Engineering* 18, 5 (2013), 1005–1042.

[330] Junji Shimagaki, Yasutaka Kamei, Naoyasu Ubayashi, and Abram Hindle. 2018. Automatic topic classification of test cases using text mining at an Android smartphone vendor. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[331] Yonghee Shin, Robert M Bell, Thomas J Ostrand, and Elaine J Weyuker. 2012. On the use of calling structure information to improve fault prediction. *Empirical Software Engineering* 17, 4 (2012), 390–423.

[332] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. 2010. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering* 37, 6 (2010), 772–787.

[333] Yonghee Shin and Laurie Williams. 2013. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering* 18, 1 (2013), 25–59.

[334] Thomas Shippey, David Bowes, and Tracy Hall. 2019. Automatically identifying code features for software defect prediction: Using AST N-grams. *Information and Software Technology* 106 (2019), 142–160.

[335] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. 2012. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering* 39, 4 (2012), 552–569.

[336] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 284–294.

[337] Marin Silic, Goran Delac, and Sinisa Srbljic. 2013. Prediction of atomic web services reliability based on k-means clustering. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 70–80.

[338] Joengju Sohn and Shin Yoo. 2019. Empirical evaluation of fault localisation using code and change metrics. *IEEE Transactions on Software Engineering* (2019).

[339] Behjat Soltanifar, Atakan Erdem, and Ayse Bener. 2016. Predicting defectiveness of software patches. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[340] Charles Song, Adam Porter, and Jeffrey S Foster. 2013. iTree: efficiently discovering high-coverage configurations using interaction trees. *IEEE Transactions on Software Engineering* 40, 3 (2013), 251–265.

[341] Charles Song, Adam A. Porter, and Jeffrey S. Foster. 2012. iTree: Efficiently discovering high-coverage configurations using interaction trees. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 903–913. https://doi.org/10.1109/ICSE.2012.6227129

[342] Qinbao Song, Zihan Jia, Martin Shepperd, Shi Ying, and Jin Liu. 2010. A general software defect-proneness prediction framework. *IEEE transactions on software engineering* 37, 3 (2010), 356–370.

[343] Murali Sridharan, Mika Mantyla, Leevi Rantala, and Maelick Claes. 2021. Data Balancing Improves Self-Admitted Technical Debt Detection. *arXiv preprint arXiv:2103.13165* (2021).

[344] Kamonphop Srisopha, Daniel Link, Devendra Swami, and Barry Boehm. 2020. Learning Features that Predict Developer Responses for iOS App Store Reviews. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11.

[345] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. 2020. Misbehaviour prediction for autonomous driving systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 359–371.

[346] Jaymie Strecker and Atif M Memon. 2012. Accounting for defect characteristics in evaluations of testing techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21, 3 (2012), 1–43.

[347] Yihsiung Su, Pin Luarn, Yue-Shi Lee, and Show-Jane Yen. 2017. Creating an invalid defect classification model using text mining on server development. *Journal of Systems and Software* 125 (2017), 197–206.

[348] Alexander Suh. 2020. Adapting bug prediction models to predict reverted commits at Wayfair. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1251–1262.

[349] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. Towards more accurate retrieval of duplicate bug reports. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 253–262.

[350] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 45–54.

[351] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19, 6 (2014), 1665–1705.

[352] Youshuai Tan, Sijie Xu, Zhaowei Wang, Tao Zhang, Zhou Xu, and Xiapu Luo. 2020. Bug severity prediction using question-and-answer pairs from Stack Overflow. *Journal of Systems and Software* 165 (2020), 110567.

[353] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering* (2018).

[354] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto. 2015. The impact of mislabelling on the performance and interpretation of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 812–823.

[355] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering*. 321–332.

[356] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering* 43, 1 (2016), 1–18.

[357] Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. 2015. Approximating attack surfaces with stack traces. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 199–208.

[358] Stephen W Thomas, Meiyappan Nagappan, Dorothea Blostein, and Ahmed E Hassan. 2013. The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1427–1443.

[359] Patanamon Thongtanunam, Weiyi Shang, and Ahmed E Hassan. 2019. Will this clone be short-lived? Towards a better understanding of the characteristics of short-lived clones. *Empirical Software Engineering* 24, 2 (2019), 937–972.

[360] Ferdian Thung, Xuan-Bach D Le, and David Lo. 2015. Active semi-supervised defect categorization. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 60–70.

[361] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying linux bug fixing patches. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 386–396.

[362] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.

[363] Haonan Tong, Bin Liu, and Shihai Wang. 2018. Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. *Information and Software Technology* 96 (2018), 94–111.

[364] Nuria Torrado, Michael P Wiper, and Rosa E Lillo. 2012. Software reliability modeling with software metrics data via Gaussian processes. *IEEE Transactions on Software Engineering* 39, 8 (2012), 1179–1186.

[365] Ayşe Tosun, Ayşe Bener, Burak Turhan, and Tim Menzies. 2010. Practical considerations in deploying statistical methods for defect prediction: A case study within the Turkish telecommunications industry. *Information and Software Technology* 52, 11 (2010), 1242–1257.

[366] Parastou Tourani and Bram Adams. 2016. The impact of human discussions on just-in-time quality assurance: An empirical study on openstack and eclipse. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 189–200.

[367] Christoph Treude and Markus Wagner. 2019. Predicting good configurations for github and stack overflow topic models. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 84–95.

[368] Feifei Tu, Jiaxin Zhu, Qimu Zheng, and Minghui Zhou. 2018. Be careful of when: an empirical study on time-related misuse of issue tracking data. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 307–318.

[369] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. 2009. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering* 14, 5 (2009), 540–578.

[370] Anderson Uchôa, Caio Barbosa, Daniel Coutinho, Willian Oizumi, Wesley KG Assunçao, Silvia Regina Vergilio, Juliana Alves Pereira, Anderson Oliveira, and Alessandro Garcia. 2021. Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 471–482.

[371] Qasim Umer, Hui Liu, and Inam Illahi. 2019. CNN-based automatic prioritization of bug reports. *IEEE Transactions on Reliability* 69, 4 (2019), 1341–1354.

[372] Harold Valdivia Garcia and Emad Shihab. 2014. Characterizing and predicting blocking bugs in open source projects. In *Proceedings of the 11th working conference on mining software repositories*. 72–81.

[373] Christopher Vendome, Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Daniel German, and Denys Poshyvanyk. 2017. Machine learning-based detection of open source license exceptions. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 118–129.

[374] Haoren Wang and Huzefa Kagdi. 2018. A conceptual replication study on bugs that get fixed in open source software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 299–310.

[375] Hongbing Wang, Lei Wang, Qi Yu, Zibin Zheng, Athman Bouguettaya, and Michael R Lyu. 2016. Online reliability prediction via motifs-based dynamic Bayesian networks for service-oriented systems. *IEEE Transactions on Software Engineering* 43, 6 (2016), 556–579.

[376] Junjie Wang, Song Wang, Qiang Cui, and Qing Wang. 2016. Local-based active classification of test report to assist crowdsourced testing. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 190–201.

[377] Song Wang, Chetan Bansal, and Nachiappan Nagappan. 2021. Large-scale intent analysis for identifying large-review-effort code changes. *Information and Software Technology* 130 (2021), 106408.

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

[379] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 297–308.

[380] Xiaoyin Wang, Yingnong Dang, Lu Zhang, Dongmei Zhang, Erica Lan, and Hong Mei. 2012. Can I clone this piece of code here?. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 170–179.

[381] Xiaoyin Wang, Yingnong Dang, Lu Zhang, Dongmei Zhang, Erica Lan, and Hong Mei. 2014. Predicting consistency-maintenance requirement of code clonesat copy-and-paste time. *IEEE Transactions on Software Engineering* 40, 8 (2014), 773–794.

[382] Xin Wang, Jin Liu, Li Li, Xiao Chen, Xiao Liu, and Hao Wu. 2020. Detecting and explaining self-admitted technical debts with attention-based neural networks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 871–882.

[383] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2018. How well do change sequences predict defects? sequence learning from software changes. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1155–1175.

[384] Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh Farahbod. 2012. Automated inference of goal-oriented performance prediction functions. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 190–199.

[385] Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. 2010. Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering* 15, 3 (2010), 277–295.

[386] Thomas D White, Gordon Fraser, and Guy J Brown. 2019. Improving random GUI testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 307–317.

[387] Igor Scaliante Wiese, Reginaldo Ré, Igor Steinmacher, Rodrigo Takashi Kuroda, Gustavo Ansaldi Oliva, Christoph Treude, and Marco Aurélio Gerosa. 2017. Using contextual information to predict co-changes. *Journal of Systems and Software* 128 (2017), 220–235.

[388] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. 2009. Predicting build failures using social network analysis on developer communication. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 1–11.

[389] Andrew Wood, Paige Rodeghero, Ameer Armaly, and Collin McMillan. 2018. Detecting speech act types in developer question/answer conversations during bug repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 491–502.

[390] Qian Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. 2011. Iterative mining of resource-releasing specifications. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 233–242.

[391] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 15–25.

[392] Xiaoxue Wu, Wei Zheng, Xiang Chen, Yu Zhao, Tingting Yu, and Dejun Mu. 2021. Improving high-impact bug report prediction with combination of interactive machine learning and active learning. *Information and Software Technology* 133 (2021), 106530.

[393] Yueming Wu, Deqing Zou, Shihan Dou, Siru Yang, Wei Yang, Feng Cheng, Hong Liang, and Hai Jin. 2020. SCDetector: Software Functional Clone Detection Based on Semantic Tokens Analysis. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 821–833.

[394] Xin Xia, David Lo, Shane McIntosh, Emad Shihab, and Ahmed E Hassan. 2015. Cross-project build co-change prediction. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 311–320.

[395] Xin Xia, David Lo, Sinno Jialin Pan, Nachiappan Nagappan, and Xinyu Wang. 2016. Hydra: Massively compositional model for cross-project defect prediction. *IEEE Transactions on software Engineering* 42, 10 (2016), 977–998.

[396] Xin Xia, David Lo, Emad Shihab, and Xinyu Wang. 2015. Automated bug report field reassignment and refinement prediction. *IEEE Transactions on Reliability* 65, 3 (2015), 1094–1113.

[397] Xin Xia, David Lo, Emad Shihab, Xinyu Wang, and Bo Zhou. 2015. Automatic, high accuracy prediction of reopened bugs. *Automated Software Engineering* 22, 1 (2015), 75–109.

[398] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. 2016. Collective personalized change classification with multiobjective search. *IEEE Transactions on Reliability* 65, 4 (2016), 1810–1829.

[399] Xin Xia, Emad Shihab, Yasutaka Kamei, David Lo, and Xinyu Wang. 2016. Predicting crashing releases of mobile applications. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[400] Peng Xiao, Bin Liu, and Shihai Wang. 2018. Feedback-based integrated prediction: Defect prediction based on feedback from software testing process. *Journal of Systems and Software* 143 (2018), 159–171.

[401] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. 2016. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 51–62.

[402] Zhou Xu, Li Li, Meng Yan, Jin Liu, Xiapu Luo, John Grundy, Yifeng Zhang, and Xiaohong Zhang. 2021. A comprehensive comparative study of clustering-based unsupervised defect prediction models. *Journal of Systems and Software* 172 (2021), 110862.

[403] Zhou Xu, Shuai Li, Yutian Tang, Xiapu Luo, Tao Zhang, Jin Liu, and Jun Xu. 2018. Cross version defect prediction with representative data via sparse subset selection. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 132–13211.

[404] Zhou Xu, Shuai Li, Jun Xu, Jin Liu, Xiapu Luo, Yifeng Zhang, Tao Zhang, Jacky Keung, and Yutian Tang. 2019. LDFR: Learning deep feature representation for software defect prediction. *Journal of Systems and Software* 158 (2019), 110402.

[405] Zhou Xu, Jin Liu, Xiapu Luo, and Tao Zhang. 2018. Cross-version defect prediction via hybrid active learning with kernel principal component analysis. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 209–220.

[406] Zhou Xu, Jifeng Xuan, Jin Liu, and Xiaohui Cui. 2016. MICHAC: Defect prediction via feature selection based on maximal information coefficient with hierarchical agglomerative clustering. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 370–381.

[407] Meng Yan, Yicheng Fang, David Lo, Xin Xia, and Xiaohong Zhang. 2017. File-level defect prediction: Unsupervised vs. supervised models. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 344–353.

[408] Meng Yan, Xin Xia, Emad Shihab, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Automating change-level self-admitted technical debt determination. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1211–1229.

[409] Ruibo Yan, Xi Xiao, Guangwu Hu, Sancheng Peng, and Yong Jiang. 2018. New deep learning method to detect code injection attacks on hybrid applications. *Journal of Systems and Software* 137 (2018), 67–77.

[410] Aidan ZH Yang, Daniel Alencar da Costa, and Ying Zou. 2019. Predicting co-changes between functionality specifications and source code in behavior driven development. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 534–544.

[411] Hui Yang, Alistair Willis, Anne De Roeck, and Bashar Nuseibeh. 2010. Automatic detection of nocuous coordination ambiguities in natural language requirements. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 53–62.

[412] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. 2015. Classification model for code clones based on machine learning. *Empirical Software Engineering* 20, 4 (2015), 1095–1125.

[413] Xiaoxing Yang, Ke Tang, and Xin Yao. 2014. A learning-to-rank approach to software defect prediction. *IEEE Transactions on Reliability* 64, 1 (2014), 234–246.

[414] Xiaoxing Yang and Wushao Wen. 2018. Ridge and lasso regression models for cross-version defect prediction. *IEEE Transactions on Reliability* 67, 3 (2018), 885–896.

[415] Ye Yang, Muhammad Rezaul Karim, Razieh Saremi, and Guenther Ruhe. 2016. Who should take this task? Dynamic decision support for crowd workers. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[416] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 157–168.

[417] Yibiao Yang, Yuming Zhou, Hongmin Lu, Lin Chen, Zhenyu Chen, Baowen Xu, Hareton Leung, and Zhenyu Zhang. 2014. Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? An empirical study. *IEEE Transactions on Software Engineering* 41, 4 (2014), 331–357.

[418] Bee Wah Yap, Khatijahhusna Abd Rani, Hezlin Aryani Abd Rahman, Simon Fong, Zuraida Khairudin, and Nik Nik Abdullah. 2014. An application of oversampling, undersampling, bagging and boosting in handling imbalanced datasets. In *Proceedings of the first international conference on advanced data and information engineering (DaEng-2013)*. Springer, 13–22.

[419] Mohamad Yazdaninia, David Lo, and Ashkan Sami. 2021. Characterization and Prediction of Questions without Accepted Answers on Stack Overflow. *arXiv preprint arXiv:2103.11386* (2021).

[420] Cemal Yilmaz and Adam Porter. 2010. Combining hardware and software instrumentation to classify program executions. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. 67–76.

[421] Qiao Yu, Shujuan Jiang, and Yanmei Zhang. 2017. A feature matching and transfer approach for cross-company defect prediction. *Journal of Systems and Software* 132 (2017), 366–378.

[422] Tingting Yu, Wei Wen, Xue Han, and Jane Huffman Hayes. 2018. Conpredictor: concurrency defect prediction in real-world applications. *IEEE Transactions on Software Engineering* 45, 6 (2018), 558–575.

[423] Zhongxing Yu, Chenggang Bai, and Kai-Yuan Cai. 2015. Does the failing test execute a single or multiple faults? An approach to classifying failing tests. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 924–935.

[424] Zhe Yu, Christopher Theisen, Laurie Williams, and Tim Menzies. 2019. Improving vulnerability inspection efficiency using active learning. *IEEE Transactions on Software Engineering* (2019).

[425] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2020. Automatically learning patterns for self-admitted technical debt removal. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 355–366.

[426] Marcelo Serrano Zanetti, Ingo Scholtes, Claudio Juan Tessone, and Frank Schweitzer. 2013. Categorizing bugs with social networks: a case study on four open source software communities. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1032–1041.

[427] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep just-in-time defect prediction: how far are we?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 427–438.

[428] Feng Zhang, Ahmed E Hassan, Shane McIntosh, and Ying Zou. 2016. The use of summation to aggregate software metrics hinders the performance of defect prediction models. *IEEE Transactions on Software Engineering* 43, 5 (2016), 476–491.

[429] Feng Zhang, Iman Keivanloo, and Ying Zou. 2017. Data transformation in cross-project defect prediction. *Empirical Software Engineering* 22, 6 (2017), 3186–3218.

[430] Fanlong Zhang, Siau-cheng Khoo, and Xiaohong Su. 2017. Predicting change consistency in a clone group. *Journal of Systems and Software* 134 (2017), 105–119.

[431] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2016. Towards building a universal defect prediction model with rank transformed predictors. *Empirical Software Engineering* 21, 5 (2016), 2107–2145.

[432] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. 2016. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 309–320.

[433] Hongyu Zhang, Liang Gong, and Steve Versteeg. 2013. Predicting bug-fixing time: an empirical study of commercial software projects. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1042–1051.

[434] Haoxiang Zhang, Shaowei Wang, Tse-Hsun Chen, and Ahmed E Hassan. 2021. Are comments on Stack Overflow well organized for easy retrieval by developers? *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–31.

[435] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. 2018. Predictive mutation testing. *IEEE Transactions on Software Engineering* 45, 9 (2018), 898–918.

[436] Kevin Zhang. 2019. A Machine Learning Based Approach to Identify SQL Injection Vulnerabilities. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1286–1288.

[437] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

807–817.
[438] Yuwei Zhang, Dahai Jin, Ying Xing, and Yunzhan Gong. 2020. Automated defect identification via path analysis-based features with transfer learning. *Journal of Systems and Software* 166 (2020), 110585.
[439] Kunsong Zhao, Jin Liu, Zhou Xu, Li Li, Meng Yan, Jiaojiao Yu, and Yuxuan Zhou. 2021. Predicting Crash Fault Residence via Simplified Deep Forest Based on A Reduced Feature Set. *arXiv preprint arXiv:2104.01768* (2021).
[440] Nengwen Zhao, Junjie Chen, Zhou Wang, Xiao Peng, Gang Wang, Yong Wu, Fang Zhou, Zhen Feng, Xiaohui Nie, Wenchi Zhang, et al. 2020. Real-time incident prediction for online service systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 315–326.
[441] Wujie Zheng, Haochuan Lu, Yangfan Zhou, Jianming Liang, Haibing Zheng, and Yuetang Deng. 2019. iFeedback: Exploiting User Feedback for Real-Time Issue Detection in Large-Scale Online Service Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 352–363.
[442] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring resource specifications from natural language API documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 307–318.
[443] Minghui Zhou and Audris Mockus. 2014. Who will stay in the floss community? modeling participant's initial behavior. *IEEE Transactions on Software Engineering* 41, 1 (2014), 82–99.
[444] Tianchi Zhou, Xiaobing Sun, Xin Xia, Bin Li, and Xiang Chen. 2019. Improving defect prediction with deep forest. *Information and Software Technology* 114 (2019), 204–216.
[445] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 683–694.
[446] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald Gall. 2016. Combining text mining and data mining for bug report classification. *Journal of Software: Evolution and Process* 28, 3 (2016), 150–176.
[447] Yuming Zhou, Baowen Xu, and Hareton Leung. 2010. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software* 83, 4 (2010), 660–674.
[448] Yuming Zhou, Baowen Xu, Hareton Leung, and Lin Chen. 2014. An in-depth study of the potentially confounding effect of class size in fault prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 1 (2014), 1–51.
[449] Yuming Zhou, Yibiao Yang, Hongmin Lu, Lin Chen, Yanhui Li, Yangyang Zhao, Junyan Qian, and Baowen Xu. 2018. How far we have progressed in the journey? an examination of cross-project defect prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 1 (2018), 1–51.
[450] Zhi-Hua Zhou. 2009. Ensemble Learning. *Encyclopedia of biometrics* 1 (2009), 270–273.
[451] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 91–100.
[452] Thomas Zimmermann, Nachiappan Nagappan, Philip J Guo, and Brendan Murphy. 2012. Characterizing and predicting which bugs get reopened. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1074–1083.

APPENDIX

Table 8. The number of predictive models applied in per year.

Category	Family	Model Name	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	Total
Rule-based predictive models	Rule-based	Propositional rule (JRip)						1		3	1	1	1	2	9
		RIPPER		1				1	1	3		1			7
		Zero-R						1		1	1		1		4
		One rule/One-R		1				1		1					3
		Conjunctive Rule			1			1							2
		Fuzzy Lattice Reasoning (FLR)									1				1
		Ridor							1						1
Statistical model	Dimensionality reduction algorithm	LDA			1	1	2			2	3		1		10
		Subclass Discriminant Analysis (SDA)								1					1
		Flexible Discriminant Analysis (FDA)								1					1
		PCA				1	1	1	1			2			6
		Kernel PCA										1			1
	Regularization algorithm	Ridge regression			1			1				1		2	5
		Bayesian Ridge Regression											1	1	2
		Lasso										1		1	2
	State model	HIdden Markov model(HMM)	1	1	1									1	4
		Markov Chains	1						1						2
		Markov chain Monte Carlo method (MCMC)				1					1				2
		Extended Finite State Machine (EFSM)								1	1				2
	Probability model	Conditional Random Field (CRF)											1	2	3
		ARIMA algorithm				1									1
		BM25F algorithm										1			1
		canonical correlation analysis (CCA)							1						1
		generalized Additive Model												1	1
NLP	NLP	LSA			1										1
		LSI					1								1
		N-gram						1							1
Machine learning	Regression	Logistic Regression	2	4	7	5	11	3	7	12	9	12	14	25	112
		Multinomial Logistic Regression				1		1	1	1		2	2		8
		Binary Logistic Regression		1				1	1				1	1	5

	Univariate Logistic Regression						1		1					2
	Bayesian Logistic Regression							1						1
	Binomial Logistic Regression												1	1
	Sparse Logistic Regression							1						1
	Weighted Logistic Regression												1	1
	Linear Regression (LR)	2		3		2	2	1	1	2		1	1	15
	Multiple Linear Regression			1				1						2
	Ordinary least squares (OLS)			1	2									3
	Least Median Squares (LMS)			1	2									3
	Least Trimmed Squares (LTS)				2									2
	Partial Least Squares (PLS)								1					1
	Simple Regression (SL)				1			1	4	1			2	9
	MARS			1	1	1			1					4
	Negative Binomial Regression		1	1	1	1								4
	Decision Tree Regression (DTR)								1			1		2
	Gaussian process		1										1	2
	Bayesian additive regression tree		1											1
	Penalized Multinomial Regression								1					1
	Robust M-estimator Regression (RobMM)				1									1
	Poisson Regression			1										1
	SGD Regression (SDGR)											1		1
	Support Vector Regression (SVR)											1		1
Bayesian	Naive Bayes	1	5	8	5	6	11	5	15	10	15	17	17	115
	Bayesian Network	2	2	3	4	1	5	3	5	3	2		5	35
	Multinomial Naive Bayes (MNB)								1		1	1	3	6
	Gaussian Naive Bayes												4	4
	Bayesian Belief Networks (BBNs)			1										1
	Bayesian Inference										1			1
	Bernoulli Naive Bayes												1	1
	Hierarchical Bayesian Network												1	1
SVM	SVM	1	3	6	2	7	8	6	8	5	19	15	21	101
	SMO						1	1	2	1	2	2		9
	Linear SVM								1			3	2	6
	LibSVM			1			1		1					3

		Least Square Support Vector Machine (LSSVM)			1							1			2
		SVM-LR												2	2
		SVM-Ploynomial				1		1							2
		SVM-RBF				1					1				2
		KernelSVM								1					1
		Nu-SVM												1	1
		Ranking SVM											1		1
		Weighted-SVM												1	1
	Decision Tree	Decision Tree	1	1	5	2	3	6	1	2	3	4	8	12	48
		J48		1	2	1	1	3	5	5	3	7	5	5	37
		C4.5		5	1	1		1	2	2	2	1	2		17
		CART			1	3	2			2	2		1	3	14
		ADTree					1	2	1	2	1	2	1	1	12
		Logistic Model Tree (LMT)							2	5	1		1	1	10
		Decision Table Majority						1	5	2		2		2	9
		PART		1				1		1				2	5
		C5.0								1		2		1	4
		Recursive Partitioning		1	1			1		1					4
		Exhaustive CHAID			1										1
		Tree-Disc Classification Tree						1							1
	Instance-based algorithm	K-nearest Neighbor	1	1	3		3	3	3	4	8	5	11	11	53
		IBk						2		2	1	1	1	1	8
		KStar								1				1	2
		MLkNN								1				1	2
		Exclusion KNN (EKNN)											1		1
		Nearest neighbor with non-nested generalization (NNGe)				1									1
		Locally weighted learning (LWL)		1				1		1					3
	Clustering technique	K-means				1	3		3	2		2	2	2	15
		Expectation Maximization(EM)			1	1			1				1		4
		DBSCAN clustering												1	1
		Fuzzy C-means (FCM)								1					1
		Partitioning Around Medoids (PAM)								1					1
		Spectral Clustering								1					1
	Learning to Rank (LTR)	LTR						1					1		2
		LambdaRank											1		1
		ListNet											1		1
		Rank Net											1		1
Ensemble learning	Random forest	Random Forest		1	1	3	2	6	5	22	16	21	28	31	136
		Random Tree								1	1	1			3
		Rotation Forest							1	1		1			3

		Balance Rnadom Forest (BRF)											1	1
		Extra Tree											1	1
		Ensemble Multiple KernelCorrelation Alignment (EMKCA)								1				1
	Bagging	Bagging	2	1	1		2	1	1	4	3	2	1	16
		Random Subspace						1	1			1		3
		Tree Bagging		1			1		1					3
		BaggedCART							1					1
	Boosting	AdaBoost					2	3	2	2		3	5	17
		XGBoost									1	2	6	9
		LogitBoost	1						2	1		1		5
		Tree Boosting									2	1	2	5
		Gradient boosting									1		3	4
		Boosting									1	2		3
		Gradient Boosting Decision tree										1	1	2
		Graussian Process Regression (GPR)										2		2
		AdaRank										1		1
		Boost C4.5	1											1
		eXtreme Gradient Boosting Tree (xGBTree)							1					1
		Gradient Boosting Regression (GBR)										1		1
		Generalized linear and Additive Models Boosting (GAMBoost)							1					1
		Light Gradient Boosting Machine (LightGBM)											1	1
		RandBoost										1		1
Neural network	Perceptron	Multi-Layer Perceptron (MLP)	1	2		1	1	3	4	2	2	4	11	31
		Single-Layer Perception									1			1
	DNN	ANN	2	1	2	1	1		1	3	3	7	1	21
		Radial basis functions network (RBFNet)		1				2	3	2		1		9
		AVNNet							1		1			2
		Deep Forest										1	1	2
		Deep Fusion Learning Model											1	1
		HAN											1	1
	CNN	CNN							2	1	7	4	10	24
		TBCNN										1	1	2
		R-CNN											1	1
	RNN	LSTM									3	6	9	18
		RNN									6	1	4	11
		Bi-LSTM										1	4	5
		GRU										1		1
		Tree-LSTM											1	1
		Recurrent Highway Network									1			1

Deep Belief Network (DBN)	DBN	1		2	1	1	5
GNN	GNN					1	1
	GGNN				1		1
Transformer	BERT					1	1
Architectures	Deep Siamese Network			1		3	4
	Stacked Denoising Autoencoder (SDA)			1			1

Table 9. Relevant studies in software requirements.

Research topic	**Predictive model**	**Ref.**
requirement classification	SVM	[1]
	random k labelsets	[15]
requirement change detection	regression tree	[259]
	Random Forest	[34]
requirement detection	LogitBoost algorithm	[411]
	Deep Siamese Network	[327]
	Logistic Regression and Support Vector Machine (mathematical category), Decision Tree and Random Forest (hierarchical category), and Feedforward Neural Network (layered category)	[5]
	k-nearest neighbor algorithm	[35]

Table 10. Relevant studies in software design.

Research topic	**Predictive model**	**Ref.**
Architectural tactic detection	support vector machine (SVM), C.45 decision tree (implemented as J48 in Weka), bayesian logistic regression (BLR), AdaBoost	[225]
	Random Forest	[99]
UI screen detection	CNN	[10]
	K-nearest-neighbors algorithm, CNN, Support Vector Machine (SVM)	[234]
Design change detection	Logistic Regression, Naive Bayes,SVM, Decision Tree, Random Forest, and Gradient Boosting	[370]

Table 11. Relevant studies in software development.

Research topic	**Family**	**Predictive model**	**Ref.**
Code consistency	Change prediction	Multilayer Perceptron (MLP), Random Forests (RF), Naive Bayes (NB), Adaboost (AB), Logitboost (LB) and Bagging (BG)	[212]
		AD tree, decision table majority, logistic regression, multilayer perception, SVM, Naive Bayes	[45]
		Random Forests (RF), Bagging (BG), Adaptive Boosting (AB) and Logitboost (LB)	[213]
	Co-change prediction	Bayesian network	[430]

		random forest	[387]
		random forest	[210]
		AdaBoost	[394]
		encoder-decoder RNN	[262]
		random forest, Naive Bayes,and logistic regression	[410]
		k-nearest neighbors algorithm	[88]
	Code change identification	Decision Tree (ADTree), Logistic Regression (Logistic), Naive Bayes (NB), Sup-port Vector Machine (SVM), and Random Forest (RF),	[377]
		binary logistic regression, Naive Bayes, Conjunctive Rule, J48	[271]
	Change impact analysis	random forest	[228]
		Bayesian classification	[2]
	Conflict prediction	decision tree, random forest	[269]
		BayesNet, Binomial Logistic Regression, Support Vector Machine (SVM), Multi-Layer Perceptron, Bagging	[38]
	Code change classification	ADTree	[398]
Program analysis	Type inference	LSTM	[215]
		random forest classifier	[124]
		RNN	[119]
	Termination proofs detection	linear regression	[265]
	Constraint solving time prediction, symbolic execution	LSTM, Tree-LSTM, KNN	[204]
	Program execution classification	decision tree	[420]
	State model inference	RNN, CNN	[217]
	Type annotations	LSTM	[296]
	Commit code detection	linear and logistic regression, generalized additive models, support vector machines,and random forest	[70]
	Program classification	Tree-based convolutional neural networks (TBCNN), LSTM, GGNN	[39]
	Compile version prediction	GNN	[39]
API-related	API identification	Decision Tree, Random Forest (ensemble clas-sifier), MPLC Classifier (neural network multilayer percep-tron), MLkNN (multi-label lazy learning approach basedon the traditional K-nearest neighbor algorithm),and Logistic Regression	[321]
		expectation-maximization (EM)	[250]
		Hidden Markov Model	[442]
		Latent Dirichlet Allocation (LDA)	[142]
		supervised CRF model	[8]
	Change-prone API prediction	SupportVector Machine (LibSVM),Naive Bayes Network (NBayes)andNeural Nets (NN)	[313]
Code comment	Code comment classification	Random forest, LigthGBM, Decision Tree, Naive Bayes, BiLSTM	[55]
		decision tree	[278]
	Comment position prediction	LSTM	[135]
	Comment scope detection	Random forest	[52]
Code review	Code review	Random Forest,Logistic Regression and Naive Bayes	[305]
		Random forest	[86]
Program security	Program obfuscation	Markov chain Monte Carlo method	[197]

	Security control selection	Naive Bayes, Logistic Regression, J48, CART, JRip, and PART	[30]
Code extraction	Code extraction	CNN	[25]
		CNN	[51]
		decision tree classifiers	[311]
Framework development	Framework development	logistic regression and SVM	[246]
Code impact analysis	Code impact analysis	Decision Tree , Random Forest, Naive Bayes, multinomial logistic regression, Bagging	[81]
Code redundancy	Redundant code detection	AdaBoost, logistic regressions, neural network,decision Trees, random forest, and k-Nearest Neighbor	[310]

Table 12. Relevant studies in software testing.

Research topic	**Family**	**Predictive model**	**Ref.**
Bug/Fault prediction	Bug/Fault prediction	Multilayer Perceptron, ADTree, Naive Bayes, LogisticRegression, Decision Table Majority, and Simple Logistic	[274]
		Poisson regression, logistic regression	[165]
		ADTree, Decision Table Majority, Logistic Regression, Multilayer Perceptron and Naive Bayes	[71]
		decision trees, naive Bayes, support vector machine, logistic regression, random forest, and multi layer perceptron	[275]
		univariate logistic regression, multivariate logistic regression	[206]
		random forest, linear regression model and CART (Classification and Regression Trees)	[232]
		Least Square Support Vector Machine (LSSVM)	[172]
		NN, SVM, Bagging, Boosting, Logistic regression, KNN, Part Random Forest, C4.5,	[239]
		NaiveBayes, Logistic Regression, J48, Random Forest	[37]
		BayesNet, SVM, and Random Forest	[205]
		negative binomial regression, recursive partitioning, random forests and Bayesian additive regression trees	[385]
		naive Bayes classifier and a logistic regression model	[115]
		negative binomial regression (NBR) models	[331]
		logistic regression model	[106]
		decision tree	[245]
		logistic regression model	[452]
		Random Forest	[116]
		Support Vector Machines(polynomial kernel)	[178]
		CNN, SVM	[74]
		Exclusive K Nearest Neighbour (EKNN)	[14]
		logistic regression , multivariate regression analysis	[9]
		Naive Bayes or Logistic Regression	[335]
		principal component analysis, Univariate Logistic Regression, Multivariate Logistic Regression Analysis	[417]
		Support Vector Machine (SVM), K-Nearest Neighbors (KNN), and Gradient Boosting Decision Tree (GBDT)	[348]
		XGBoost	[50]
		linear Support Vector Machine (SVM)	[33]
		random forest, LSTM	[187]
		Binary logistic regression model	[447]

		C4.5, C4.5 No Pruning, Rule learner Nearest-Neighbor-like algorithm, RIPPER, Bayesian Networks (BN), Naive Bayes(NB), Neural Network (NN), SVM	[66]
		Principal Component Analysis	[314]
	Anomaly detection	Bi-LSTM model	[437]
		PCA, CLSTR clustring algorithm	[21]
	Crash prediction	Naive Bayes (NB) and multilayer perceptron (MLP)	[160]
		Naive Bayes	[399]
	Online failure prediction	J48 classification tree algorithm	[270]
		LSTM, random forest	[193]
	Erroneous benavior detection	CNN, RNN	[362]
		Autoencoders lstm	[345]
	Incident prediction	the gradient boosting tree based model (XGBoost)	[440]
		textCNN	[143]
	Effort-aware bug prediction	Random Forest (RF) and Logistic Regression (LR)	[300]
	Build failure prediction	Bayesian classifier	[388]
Tese case	Test case effectiveness	Artificial Neural Networks (ANNs)	[231]
		k-means clustering	[318]
		RANDOM FOREST (RFC), K-NEIGHBORS (KNN), and SUPPORT VECTOR MACHINES (SVM)	[102]
		KNN, logistic regression, recursive partitioning, support vector machine, tree bagging, random forest	[121]
	Test case selection	gradient boosted decision trees classifier	[209]
		Naive Bayes (NB) classifier and a support vector machine	[103]
	Test case classification	KNN	[330]
		logistic regression	[112]
Bug classification	Bug classification	Markov chains	[138]
		Logistic Regression, NaiveBayes, linear SVM, and KNN	[322]
		libSVM, Bayes Net	[351]
		Naive Bayes, SVM, Logistic Regression, and Random Forest	[47]
	Bug cuase classification	TBCNN, Bayesian learning, SVM	[263]
	Bug severity classification	logistic repression, Naïve Bayesian, k-Nearest Neighbor algorithm (KNN), and Long Short-Term Memory (LSTM)	[352]
Fault localization	Fault localization	Latent Dirichlet Allocation (LDA) , radial basis function (RBF) network	[176]
		SVM	[175]
		RNN	[297]
		Random Forests (RF), K-Nearest Neighbors (KNN), Multi-Layer Perceptron (MLP)	[445]
		Deep forest	[439]
		LDA, Support Vector Machine (SVM)	[260]
		VSM, LSI, LDA	[358]
		Gaussian Process Modelling, Support Vector Machine, and Random Forest	[338]
		decision tree algorithm	[104]
Test application	Mutation testing	Logistic regression	[277]
		Random Forest	[435]
	Random testing	CNN	[386]

	Automotive software testing	Decision tree	[3]
	Testing performance prediction	Decision Trees, Bayesian Network, Naive Bayes, Logistic Regression	[63]
Test alarm	Test alarm identification	kNN	[141]
		decision tree (ADTree), naiveBayes and Bayesian network (BayesNet)	[114]
Test report	Test report classification	SVM, Decision Tree, Naive Bayes, and Logistic Regression	[376]
		Logistic regression	[112]
	Test report assessment	Logistic regression	[57]
Fault injection	Fault injection	decision trees, k-means clustering	[256]
		CNN, LSTM	[409]

Table 13. Relevant studies in software maintenance.

Research topic	**Family**	**Predictive model**	**Ref.**
Defect prediction	Defect prediction	Naive Bayes	[431]
		Naive Bayes classifier	[237]
		k-means	[360]
		Random Forest and the C4.5 Decision tree classifiers , Support Vector Machines (SVM), Neural Networks model , K-Nearest Neighbor model	[27]
		negative binomial regression model	[156]
		decision tree (J48 and LMT), Bayes classifier (Naive Bayes (NB) and BayesNet (BN)), instance- based algorithm (IBk and KStar), rule-based algorithm (PART and JRip), function-based model (Simple Logistic (SL) and SMO) and ensemble learning method (Vote and Random Forest (RF))	[438]
		logistic regression, SVM	[374]
		kernel SVM, KNN, NB	[268]
		KernelPCA	[403]
		FNN	[404]
		Naive Bayes (NB), Random Forest (RF), and Repeated Incremental Pruning to Produce Error Reduction (RIPPER)	[406]
		random forest (RF), and gradient boost regression tree (GBRT)	[400]
		Transfer learning	[421]
		decision tree C4.5, Naive bayes, bayes net, logistic regression, neural network	[347]
		KernelPCA	[405]
		RNN, Naive Bayes (NB), Logistic Regression (LR), k-Nearest Neighbor (KNN), Random Forest (RF), C5.0 decision tree(C5.0), standard Neural Network (NN) and C4.5-like decisiontrees (J48)	[202]
		Random Forest,Support Vector Machine,Bayesian Network, and J48	[279]
		k-Nearest Neighbor (k-NN), naive Bayes	[369]

linear regression, simple logistic(SL), radial basis functions network(RBFNet), sequential minimal optimization(SMO), KNN, Propositional rule(JRip), Ripple down rules (Ridor), Naive Bayes (NB), J48, Logistic model tree(LMT), Random Forest(RF), Bagging (BG+LMT, BG+NB, BG+SL, BG+SMO, BG+J48), adaboost (AB+LMT<AB+NB, AB+SL, AB+SMO, AB+J48), ROtation Forest(RF+LMT, RF+NB, RF+SL,RF+SMO, RF+J48), Random subspace(RS+LMT, RS+NB, RS+SL, RS+SMSO, RS+J48)	[407]
linear/logistic regression, random forest, KNN, SVM, CART, and neural networks	[306]
Neural network, C4.5, SVM, logistic regression, Boost C4.5, PART, C4.5 + PART, Decorate C4.5	[17]
Logistic Regression, Navie Bayes, and Bayesian Network model	[339]
LSTM	[64]
Bayesian networks	[267]
negative binomial regression (NBR)	[26]
deep belief network (DBN)	[378]
change classification (The Bayes Net) and buggy file prediction(The Bayes Net) Naïve Bayes, Support Vector Machines (SVM) and Bagging learners	[163]
Random Forests (RF), Naive Bayes (NB), Logistic Regression (LR)	[287]
logistic regression	[253]
logistic regression	[302]
Multivariate Adaptive Regression Splines (MARS) Naive Bayes and Simple Logistic statistical techniques K-means and Expectation Maximization clustering techniques, Ripper and Ridor rule-based techniques, Radial Basis Functions neural network technique, KNN nearest neighbour tech- nique, Sequential Minimal Optimization (SMO) SVM technique	[94]
logistic regression	[44]
Deep Belief Network (DBN)	[379]
logistic regression, decision tree	[451]
decision tree, Bayesian Network, J48 decision tree, and logistics linear regression	[179]
Logistic regression	[303]
Logistic Regression	[304]
IBk (KNN), J48 and Random Forests methods	[93]
Bayesian model	[184]
logistic regression	[295]
decision tree	[390]
Alternating Decision Tree (ADTree), Naive Bayes and Logistic Regression MARS	[144]
Logistic regression Bayesian network, J48 decision tree, Logistic model tree, Naive Bayesian, Random forest, and Support vector machine	[252]
Random Forests	[170]
kmeans	[402]
Naive Bayes (NB) and Random Forest (RF)	[77]
J48 Logistic Naive Bayes	[334]
deep forest, DBN, NB, LR, RF, SVM	[444]
logistic regression, SVM, ANN, Stacked denoising autoencoders (SDA)	[363]
J48, Logistic Regression (LR), Naive Bayes (NB), DecisionTable (DT), Support Vector Machine (SVM) and BayesianNetwork (BN)	[118]
Three learning algorithms. Naive Bayes (NB), J48,6 and OneR	[342]
Logistic regression	[152]

NB, SVMs, and NNs	[288]
autoencoder neural network, Gaussian Naive Bayes, logistic regression, k-nearest-neighbors, decision tree, and Hybrid SMOTE-Ensemble	[6]
random forest	[180]
Subclass discriminant analysis (SDA)	[146]
random forest algorithm	[428]
naive bayes classifiers, logistic regression classifiers, random forest classifiers	[356]
Recurrent Neural Network (RNN)	[383]
Random Forest (RF), Logistic Regression (LR), Naive Bayes (NB), neural network (AVNNet), C5.0 Boosting (C5.0), extreme gradient boosting (xGBTree), and gradi- ent boosting method (GBM)	[353]
Bayesian Network, J48 Decision Tree, Logistic Regression, and Random Forest	[422]
RNN	[383]
a deep belief network (DBN)	[378]
Linear SVM (LSVM), Supervised Deep Belief Network=, SVM, (DBN) Classification, Nu-SVM (NSVM), Logistic Regression, Gaussian Process (Gauss) classifier, Bernoulli Naive Bayes, K Nearest Neighbors (KNN) classifier, Multinomial Naive Bayes, Random Forest (RF) classifier, Gaussian Naive Bayes, Multi-Layer Perceptron (MLP) classifier	[154]
Transfer Naive Bayes	[207]
Naive Bayes	[365]
Ridge regression (RR), Lasso regression(LAR)	[414]
logistic regression	[130]
Bayesian Belief Networks (BBNs)	[157]
learning-to-rank	[413]
Decision tree, Naive Bayes, KNN, Random Forest, Zero-R	[372]
Random Forest (RF), Naive Bayes, Logistic regression, knn	[290]
logistical regression	[127]
Logis-tic Regression(LReg), J48(C 4.5 Decision Tree), Random-Forest(RFor), Bayesian Network(BNet), Exhaustive CHAID, Support Vector Machine, Naive Bayes Network(NBayes) and Neural Nets(NN)	[96]
Random Forest (RndFor), Bayesian Network(BN), Support Vector Machine (SVM), and the J48 decision tree	[95]
KNN, logistic regression, naive bayes, recursive partitioning, SVM, tree bagging	[299]
Bayesian network	[69]
Naive Bayes, XGBoost	[76]
Bayesian Network	[380]
Ensemble Multiple Kernel Correlation Alignment (EMKCA)	[188]
Logistic regression	[251]
canonical correlation analysis (CCA)	[145]
logistic regression model	[169]
Logistic Regression (LR), Decision Tree (J48), Random Forest (RF), Naive Bayes (NB), Logistic Model Tree (LMT)	[166]
logistic regression	[40]
Random forest, Logistic regression, K-means, Naive Bayes, decision trees, support vector machines	[7]
k-means	[98]
logistic regression model	[128]
Logistic Regression, J48, SVM, and Naive Bayes	[301]
logistic-regression model	[346]

		Naive Bayes (NB), MultiLayer Perception (MLP), Locally Weighted Learning (LWL), Random Forest (RF), AdaBoost, bagging, linear regression	[448]
		kNN, Logistic regression, Recursive partitioning, SVM, Tree Bagging, Random forest	[120]
		Naive bayes, kNN, Regression, Partial Least Square, Nueral Network, Discrimination Analysis, Rule-based classifier, Decision tree, SVM, Bagging, Boosting	[355]
		Random forest	[192]
		Random forest	[354]
	Cross-project defect prediction	Bayes net (BN), k-nearest neighbours (IBk), decision tree (J48), naive Bayes (NB), random forest (RF), and random tree (RT)	[429]
		SVM	[317]
		k-Nearest Neighbor (k-NN)	[289]
		random forest (RF), naive Bayes (NB), logistic regression (LR), decision tree (J48), and logistic model tree (LMT) k-means clustering (KM), partition around medoids (PAM), fuzzy C- means (FCM), and neural-gas (NG) spectral clustering (SC)	[432]
		TCA+ with logistic regression	[395]
		Extra Trees Classifier, Random Forest, Passive, Ridge, KNearest Neighbor(KNN), adaBoost Aggressive Classifier (PAC)	[183]
	JIT-defect predict	Random forest	[151]
		binary Logistic Regression, J-48, ADTree, Multilayer Perceptron, Naive Bayes, and Random Forest	[280]
		HAN	[427]
		Random Forest (RF), Logistic Regression (LR),Support Vector Machine (SVM), k-Nearest Neighbours (kNN),and AdaBoost	[294]
		CNN	[123]
		random forest, logistic regression and naive Bayes classifiers	[82]
		Linear regression, simple regression, RBFNet, SMO, IBk, JRip, Ridor, NB, J48, LMT, RF, Bagging, AdaBoost, Rotation Forest, Random Subspace	[416]
		CNN	[53]
		Logistic Regression (LR), Random Forest	[167]
		Random forest, Logistic regression, Naive Bayes	[83]
Bug report management	Bug report assignment	Naïve Bayes Classifier, Bayesian Network, C4.5, SVM-Polynomial, SVM-RBF	[32]
		Bayes Net, Naive Bayes, Logistic regression, Multilayer Perceptron, Simple logostic regression, SMO, IBk, KStar, LWL, ZeroR, Decision Table, JRip, OneR, PART, J48, LMT, Random Forest, Random Tree	[148]
		Decision tree	[397]
		Random Forest, Naive Bayes, Logistic Regression, Multilayer Perceptron, and K Nearest Neighbor	[291]
		Markov chains	[138]
		ensemble Hidden Markov Models (HMMs)	[137]
	Duplicated bug report detection	discriminative model, Support Vector Machines	[350]
		CNN	[117]
		LDA	[261]
		SVM	[349]
	Bug report prediction	Random Forest, Naive Bayes, Logistic Regression, Multilayer Perceptron and K-Nearest Neighbor	[291]
		Naive bayes	[173]
		Naive Bayes Multinomial (NBM)	[392]
		kNN	[433]

	Bug report classification	Bayesian Net Classifier	[446]
		SVM	[426]
		Naive Bayes, Support Vector Machines, C4.5, Expectation Maximization , conjunctive rules, and nearest neighbour	[16]
		random forest and SVM	[85]
		bert, logistic regression model, Naive Bayesian, Random Forest, Support Vector Machine, AdaBoost, Multi-Layer Perceptron, Convolutional Neural Network, Long Short-Term Memory, and Bidirectional Long Short-Term Memory	[196]
		CNN	[371]
		SVM	[84]
	Bug report refinement	nearest neighbor with non-nested generalization (NNGe)	[203]
		MLKNN	[396]
	Bug fix-related	decision tree, SVM	[100]
		Naive Bayes	[159]
		decision trees	[329]
Software quality assessment	Software reliability prediction	EFSM	[328]
		Bayesian network	[227]
		K-Means Clustering	[337]
		Dynamic Bayesian Networks	[375]
		Extended Finite State Machine (EFSM)	[185]
		Discrete-Time Markov model	[89]
		Hidden Markov Models (HMMs)	[62]
		Markov chain Monte Carlo method, Bayesian	[364]
		SVM	[108]
	Quality assessment	linear regression	[177]
		LDA	[56]
		C4.5 algorithm, Naive Bayes, KNN, Random Forest, Zero-R	[372]
		linear regression	[221]
		Random Forest	[242]
	Code readability prediction	Logistic Regression, Support Vector Machines, Naive Bayes, k-Nearest Neighbor and Decision Trees	[315]
		multilayer perceptron, Bayesian classifier, a Logistic Regression	[41]
		Logistic Regression, Support Vector Machines, Naive Bayes, k-Nearest Neighbor and Decision Tree	[315]
	Query quality prediction	logistic regression	[366]
		Random Forest	[223]
		classification tree	[110]
Vulnerability detection	Vulnerability detection	LSTM	[319]
		random forest(RF), KNN, Decision Tree(DT), Naive Bayes(NB), SVM, MLP, LR	[54]
		Random forest, deepneural network, k-nearest neighbor, Logistic regression and Support Vector Machine	[171]
		CNN	[113]
		SVM	[424]
		LSTM	[65]
		Logistic Regression (LR), Multi-Layer Perceptron (MLP) models , k-means cluster	[326]
		Random Forest, PCA	[357]
		logistic regression	[332]

		Decision Trees, k-Nearest Neighbor, Naive Bayes, Random Forest and support vector machine (SVM)	[323]
		logistic regression technique	[333]
		Balanced Random Forest (BRF), the weighted-SVM (w-SVM), and the weightedLogistic Regression (w-LR)	[161]
		C4.5/J48,Random Tree,and Random Forest classifiers, Naive Bayes (NB), K-Nearest Neighbor(KNN), and Logistic Regression(LR), Multi-Layer Perceptron(MLP),and Support Vector Machine (SVM)	[220]
		RF,GaussianNaive Bayes,k-nearest neighbor(k-NN), SVM,gradient boosting, andAdaBoost, and a logistic regression	[58]
		Multilayer Perceptron (MLP)	[436]
Perfoemance prediction	Performance prediction	Random Sampling Approach, Clustering (K-Means) Approach, PCA Approach, WRAPPER Approach, Logistic Regression Model	[214]
		Multivariate Adaptive Regression Splines (MARS), Classification and Regression Trees(cart), Genetic Programming (GP), Kriging	[384]
		CART	[105]
		Logistic regression, Naive Bayes, KNN, KS, MLP, Radial basis function, SVM, C4.5, SimpleCART, Logistic Model Tree, Random Forest	[48]
	Configuration prediction	multiple linear regression	[150]
		RFHOC, Hyperopt3 and SMAC4	[22]
		Deep feedforward neural network	[107]
		Classification trees	[236]
		Linear regression	[336]
		CART	[249]
		LDA	[367]
Code smell detection	Code smell detection	Convolutional Neural Network	[198]
		Random Forest, J48, Logistic Regression, Decision Table and Naive Bayes	[272]
		CNN, Random forest, J48, SVM, Sequential minimal optimization, Navie bayes	[78]
		J48, Random forest, Navie bayes,SVM and JRIP	[283]
		J48, JRip, Random Forest, Navie bayes, SMO, LibSVM	[90]
		Naive Bayes	[282]
		J48, JRip, Random Forest, Navie bayes, SMO, LibSVM	[72]
		n-gram	[11]
		ADTree, Decision Table Majority, Logistic Regression, Multilayer Perceptron, Multinomial Regression, Support Vector Machine and Naive Bayes	[273]
		CNN	[195]
		Adopted Support Vector Machines, Logistic Regression, ADTree, Decision Table Majority, Logistic Regression, Multilayer Perceptron, Naive Bayes and Simple Logistic Regression	[46]
		CART, C5.0 Decision Trees, Random Forest	[266]
Traceability detection	Traceability Links Recovery	Latent Dirichlet Allocation, Genetic Algorithms	[276]
		ADTree, Bagging, FLR (Fuzzy Lattice Reasoning) classifier, Naive Bayes, LogitBoost, ZeroR	[80]
		Random forest	[224]
		Decision Tree	[391]
		Hierarchical Bayesian Networks	[235]
	Link classification	Support Vector Machines (SVM) , Random Forest, Multinomial Naive Bayes (MNB), Multi-Layer Perceptron (MLP), Multinomial Logistic Regression (MLR)	[122]

	Bug fixing traceability	Learning from Positive and Unlabeled Examples (LPU) and Support Vector Machine (SVM)	[361]
	Traceability enhancement	Naive Bayes, J48 Decision Tree, and Random Forrest.	[308]
Code clone detection	Code clone detection	KMeans clustering algorithm	[412]
		Random forest	[359]
		Siamese architecture neural network	[320]
		Siamese Network	[393]
		Deep fusion learning model	[87]
		Siamese architecture with LSTM	[285]
		Bayesian network	[381]
		Siamese neural network	[247]
Issue detection	Misuse issue detection	extended BM25F algorithm, Naive Bayes model	[368]
	Real-time issue detection	XGBoost	[441]
	Energy issue detection	Dbscan clustering algorithm	[186]
	Issue life prediction	Random Forest	[158]
	Issue report segment detection	Support Vector Machine (SVM), Naive Bayes (NB), Single Layer Perceptron (SLP), K-Nearest Neighbor (KNN) and Random Forest (RF)	[244]
	Issue prediction	Multinomial Naïve Bayes (MNB), Logistic Regression(LR), Support Vector Classifier (SVC), Decision Tree Classifier(DTC), MLP Classifier (MLPC), Random Forest Classifier (RFC), Gradient Boosting Classifier(GBC)	[307]
	Issue-related risk prediction	sparse logistic regression	[61]
Self-Admitted Technical Debt detection	SATD detection	Random Forest	[408]
		CNN, RNN	[425]
		SATD, NBM, SVM, kNN, bestSub and NLP	[129]
		Bi-lstm	[382]
		Random Forest, XGBoost	[343]
Malware detection	Malware detection	Logistic Regression and Support Vector Machine	[49]
		SVM, Dtree, RF, KNN, siamese-network based learning method, MultiLayer Perceptron	[20]
		SVM, RandomForest ensemble decision-trees algorithm, RIPPER rule-learning algorithm and the tree-based C4.5 algorithm	[12]
		Multiple Kernel Learning, SVM	[254]
Log analysis	Log analysis	basic Linear, the Radial Basis Function and the Multilayer Perceptron learner, SVM	[316]
		Random forest	[182]
		R-CNN	[29]
		Random forest	[149]

Table 14. Relevant studies in software management.

Research topic	**Family**	**Predictive model**	**Ref.**

Developer behavior analysis	Developer discussion analysis	Naive Bayes	[19]
		Support Vector Machines, Logistic Regression	[312]
		Logistic Regression, Naive Bayes and Support Vector Machines	[389]
	Stackholder prediction	Gaussian Process classifier	[189]
	Crowd worker assignment	Random forest	[415]
	Task dificulty assessment	Naive Bayes, J48 and Support Vector Model	[91]
	Long-Time Contributor Prediction	Naive Bayes, Support Vector Machine, Decision Tree, K-Nearest Neighbor and Random Forest	[23]
		Logistic regression	[443]
	Developer turnove prediction	Naive Bayes, SVM, Decision tree, kNN and Random forest	[24]
	Intention mining	CNN	[132]
	Work-item notification prediction	Bayesian method, decision tree learners, SVM	[240]
	Developer response prediction	Random forest	[344]
	Developer sentiment analysis	RNN	[191]
		Random Forest, Decision Tree, Support Vector Machine, Multilayer Perceptron, Adaboost, Naive-Bayes and Logistic Regression	[136]
		LinearSVM	[190]
		LSTM	[59]
		logit regression model	[75]
		Naive Bayes, K-Nearest Neighbor, C4.5-like trees, SVM, Multi-layer Perceptron for neural network and Random Forest	[97]
		LSTM	[60]
Software repository mining	Stack Overflow mining	CNN	[401]
		Hidden Markov Model	[101]
		Random Forest Classifier and XGBoost	[13]
		Random forest	[434]
		Conditional Random Field	[8]
		BiLSTM + attention	[284]
		RF and SVM	[31]
		XGBoost, CART, Bayesian Ridge, Ridge, Lasso, GNB	[419]
	Github mining	Random Forest, SVM, K-means	[233]
		SVM, Random Forest, Logistic Regression, Naive Bayes and k-Nearest Neighbors	[298]
		Random Forest Classifier	[243]
	App store mining	SVM	[257]
		Support Vector Machines and Naive Bayes	[139]
	Other	Random Forest, Decision Tree, Support Vector Machine based on Linear Kernel, Extreme Gradient Boosting, CNN + Bi-LSTM networks, CNN + fully connected output layer, Bi-LSTM	[255]
Software classification	Software classification	Support Vector Machines, Naivee Bayes, Decision Tree, RIPPER and IBK	[194]
		K-means	[134]
		Decision Tree, Naive Bayes, SVM	[219]
Software release managenent	Release management	Decision Tree, SVM and Random Forest models	[258]

	Software release notes	Decision tree, Simple logistics and Support vector machine	[4]
License prediction	Software license exception detection	Decision tree, Naive Bayes, Random Forest, Support Vector Machine	[373]
	Changed code license prediction	Conditional random field learning algorithm	[201]
Energy efficiency	Workload prediction	Expectation Maximization Algorithm	[153]
	Software energy consumption	Naive Bayes, J48, Sequential Minimal Optimization, Logistic Regression, Random Forest, k-Nearest Neighbor, ZeroR and MultiLayer Perceptron	[218]
Behavior detection	Missing behavior prediction	Decision tree	[140]
	Fake reviews detection	Naive Bayes, Random Forest, Decision Tree, Support Vector Machine, Linear support vector classification, Multilayer perceptron	[216]