applied sciences

MDPI

Article

Exploring the Intersection between Software Maintenance and Machine Learning—A Systematic Mapping Study

Oscar Ancán Bastías *, Jaime Díaz and Julio López Fenner

Department of Computer Science and Informatics, Universidad de La Frontera, Temuco 4811230, Chile
* Correspondence: oscar.ancan@ufrontera.cl

Abstract: While some areas of software engineering knowledge present great advances with respect to the automation of processes, tools, and practices, areas such as software maintenance have scarcely been addressed by either industry or academia, thus delegating the solution of technical tasks or human capital to manual or semiautomatic forms. In this context, machine learning (ML) techniques play an important role when it comes to improving maintenance processes and automation practices that can accelerate delegated but highly critical stages when the software launches. The aim of this article is to gain a global understanding of the state of ML-based software maintenance by using the compilation, classification, and analysis of a set of studies related to the topic. The study was conducted by applying a systematic mapping study protocol, which was characterized by the use of a set of stages that strengthen its replicability. The review identified a total of 3776 research articles that were subjected to four filtering stages, ultimately selecting 81 articles that were analyzed thematically. The results reveal an abundance of proposals that use neural networks applied to preventive maintenance and case studies that incorporate ML in subjects of maintenance management and management of the people who carry out these tasks. In the same way, a significant number of studies lack the minimum characteristics of replicability.

Keywords: software maintenance; machine learning; systematic mapping

Citation: Ancán Bastías, O.; Díaz, J.; López Fenner, J. Exploring the Intersection between Software Maintenance and Machine Learning—A Systematic Mapping Study. *Appl. Sci.* **2023**, *13*, 1710. https://doi.org/10.3390/app13031710

Academic Editors: José Carlos Bregieiro Ribeiro and Rolando Miragaia

Received: 28 November 2022
Revised: 10 January 2023
Accepted: 11 January 2023
Published: 29 January 2023

1. Introduction

Traditionally, industry and academia have relegated software maintenance to the background, assigning greater relevance to the development stage [1]. In [2], it was reported that a significant number of institutions that work and depend on software operation and maintenance disregard the implications of this stage in the environment in which the software is useful. This hegemonic nature of the software development stage has directly affected the training of software engineers who are exclusively dedicated to software maintenance, indicating difficulties in finding software engineers specialized in software maintenance [3,4]. This is reflected in the paucity of evidence of education initiatives focused on software maintenance, which have achieved only relative success, as described in [5].

From the point of view of the costs associated with the maintenance stage, these are substantially higher than the costs involved in the development stage [6]. The software industry has been aware of this concern for a long time [7]. Previous studies have suggested that the costs of the maintenance stage are close to 40% to 80% [8] and can even bankrupt a software project if the management of the technical debt is not adequate by generating interest that consumes the budget to be assigned to new features [9].

In recent years, the ML boom has promoted the advance of software production. Nevertheless, the attempts to incorporate it to improve maintainability, which is considered a critical quality attribute for far-reaching projects, have yielded inconclusive results, as indicated in [10]. One of the issues detected, for example, points to each study proposing a new model from zero, without expanding on or working from the existing models. They

have no independent validation; they are only validated by their authors and are impossible to replicate because their datasets are private or inaccessible. Additionally, the increase in the complexities of software has taken a step toward increasing the data to be analyzed, but the metrics associated with maintainability have not kept pace [11].

In the extent to which it has been explored in the last 10 years, there has been no effort to review the trends of the intersection between ML techniques and software maintenance. Therefore, our study will make it easier for researchers and industry professionals to make evidence-based decisions by considering potential gaps or areas with abundant studies that can be consolidated to generate certainties based on research.

Given this context, the objectives of our study are: (1) to identify the extent to which the trends in ML are present in studies related to software maintenance and to (2) characterize the intersection between ML and software maintenance, as well as its context. This article presents a systematic mapping study (SMS) that was developed under the protocol proposed by Petersen [12,13]. The review and analysis of hundreds of articles enabled us to establish the main study topics within the discipline, which are the areas that have not been addressed and what we should work on in the future.

Our study is organized as follows: Section 2 presents the incorporation of ML into software maintenance; Section 3 presents the systematic mapping method used; Section 4 describes the main results of the study; Section 5 provides the answers to the proposed research questions, and, finally, the conclusions of our study are offered in Section 7.

2. Related Work

Within the review of the evaluated articles, 15 secondary studies that analyzed topics similar to that of the proposed research were identified. To identify relevant research, we selected two of the four sets of basic research questions (RQs, detailed in Section 3.1): those related to Maintenance (a) and Machine Learning (b). It should be noted that this selection was the result of the same research criteria as those detailed below.

To classify them, we chose to identify the type of "Maintenance Issue" proposed by the Software Engineering Body of Knowledge [1] proposal. The subsection "Maintenance Issue" comprised four topics: Technical Issue; Management Issues; Cost Estimation; Maintenance Measurement. This same differentiation was used to disclose the related secondary studies.

It can be seen from the data in Table 1 that 27% of the secondary studies referred to a particular study interest for both topics. The rest of the studies individually analyzed the state of the art, but managed to contribute in their respective lines.

Table 1. RQs tackled in related work.

Ref.	Maintenance Issue	Maintenance (a)	ML (b)
[14]	Technical Issue		X
[15]	Management Issues	X	
[16]	Cost Estimation	X	
[17]	Management Issues	X	
[18]	Management Issues		X
[19]	Technical Issue	X	X
[20]	Maintenance Measurement	X	X
[21]	Cost Estimation		X
[22]	Technical Issue	X	X
[23]	Technical Issue	X	
[24]	Technical Issue	X	
[25]	Cost Estimation	X	
[26]	Technical Issue		X
[10]	Technical Issue	X	X

Since 2011, there have been some initiatives with similar study themes. We highlighted the following categorized initiatives.

Technical issues This is the most commonly mentioned topic in the secondary literature. It represents 47% of the total number of secondary items identified.

The work of Caram et al. [19] analyzed methods and techniques based on machine learning for detecting "code smells" within a software development environment. The most commonly used techniques were genetic algorithms (GAs), which were used by 22.22% of the articles. Regarding the smells addressed by each technique, there was a high level of redundancy in that a wide range of algorithms covered the smells. Regarding performance, the best average was provided by the decision tree, followed by the random forest. None of the machine learning techniques handled 5 of the 25 analyzed smells. Most of them focused on several code smells, and in general, there was no outperforming technique, except for a few specific smells.

Azeem et al. [22] investigated the same phenomenon as that described above. While the research community studied the methodologies when defining heuristic-based code smell detectors, there is still a lack of knowledge on how machine learning approaches have been adopted and how the use of ML to detect code smells can be improved.

The authors studied them from four different perspectives: (i) the code smells considered, (ii) the setup of machine learning approaches, (iii) the design of the evaluation strategies, and (iv) a meta-analysis of the performance achieved by the models proposed so far. As a result, the analyses performed showed that God Class, Long Method, Functional Decomposition, and Spaghetti Code had been heavily considered in the literature. The analyses also revealed several issues and challenges that the research community should focus on in the future.

Alsolai et al. [10] analyzed machine learning techniques for software maintenance prediction. This review identified and investigated several research questions to comprehensively summarize, analyze, and discuss various viewpoints concerning software maintainability measurements, metrics, datasets, evaluation measures, individual models, and ensemble models. The 56 selected studies (evaluated up to 2018) indicated relatively little activity in software maintainability prediction compared with other software quality attributes. Most studies focused on regression problems and performed k-fold cross-validation. Individual prediction models were employed in most studies, while ensemble models were relatively rare. In conclusion, ensemble models demonstrated increased accuracy in prediction in comparison with individual models and were shown to be valuable models in predicting software maintainability. However, their application is relatively rare, and there is a need to apply these and other models to a wide variety of datasets to improve the accuracy and consistency of the results.

Management Issues (27% of the total). Jayatilleke et al. [17] discussed contextual software requirements as an essential part of the development of a software initiative, giving way to inevitable changes due to changes by customers and business rules. Given the importance of the topic, the paper discussed requirement management in several aspects. The research questions were related to (i) causes of requirement changes, (ii) processes for change management, (iii) techniques for change management, and (iv) business decisions about the changes themselves.

Alsolai and Roper [18] analyzed feature selection techniques in a software quality context; they sought to predict software quality by using feature selection techniques. The authors stated that feature selection techniques are essential for improving machine learning models, given their predictive capabilities. Regarding the results, studies of feature selection methods were those with the highest results (60% of the total), and RELIEF was the most commonly used technique. Most studies focused on software error prediction (classification problem) by using the area under the curve (AUC) as the primary evaluation metric. In general terms, the application of this quality prediction technique is still limited.

Lee and Seo [24] reviewed bug-reporting processes that aimed to improve software management performance. Software engineers have persistently highlighted the need to automate these processes. However, the accuracy of the existing methods could be more satisfactory. The results of this study indicated that research in the field of bug

deduplication still needs to be improved and, therefore, requires multiple studies that integrate clustering and natural language processing.

Cost Estimation (20% of the total). Malhotra [16] studied the early prediction of changes based on metrics. This prediction allows one to obtain quality software, maintainability, and a lower overall cost. The critical findings of the review were: (i) less use of method-level metrics, machine learning methods, and commercial datasets; (ii) inappropriate use of performance measures and statistical tests; (iii) lack of use of feature reduction techniques; (iv) lack of risk indicators used to identify change-prone classes; (v) inappropriate use of validation methods.

In recent years, it has been identified that technical debt is one of the aspects that most affects software maintainability. Technical debt is an economic metaphor that was initially used by Cunningham in 1992 [27], and it arises when an organization develops poor-quality or immature software devices to reach the market as quickly as possible [28]. Generally, technical debt accumulates throughout the development phase [9], which considerably degrades the software, directly affecting its maintainability [29].

Alfayez et al. [25] reviewed technical debt prioritization. Technical debt (TD) prioritization is essential for allocating such resources in the best way to determine which TD items need to be repaired first and which items are to be delayed until later releases. The search strategy that was employed identified a quasi-gold-standard set that automatically established a search string for retrieving papers from select research databases. The application of selection criteria identified 24 TD prioritization approaches. The analysis of the identified approaches revealed a need for more that account for cost, value, and resource constraints and a lack of industry evaluation.

Maintenance Measurement (7% of the total). Carvalho et al. [20] studied machine learning methods for predictive maintenance through a systematic literature review. This paper aimed to present a review of machine learning methods applied to predictive maintenance by exploring the performance of the current state-of-the-art machine learning techniques. This review focused on two scientific databases and provided a valuable foundation for machine learning techniques and their main results, challenges, and opportunities. It also supports new research work in the predictive maintenance field. As a result, the authors identified that each proposed approach addressed specific equipment, making it more challenging to compare it to other techniques.

In addition, predictive maintenance has emerged as a new tool for dealing with maintenance events. In addition, some of the works assessed in this review employed standard ML methods without parameter tuning. This scenario may be because PdM is a recent topic and is beginning to be explored by industrial experts [30].

In conclusion, the studies that have a closer relationship to our current topic (detailed in Section 3.1) are those that study the analysis of code smells in software development by employing machine learning techniques (see Table 1). Our SMS iteration shares a thematic context and interest with researchers working on software maintenance. This area needs to be sufficiently explored and may become a trend given the current technological advances.

3. Method

Systematic mapping studies (SMSs) in software engineering consider a series of stages framed within the protocol defined by Petersen [12,13]. The application of this protocol involves the categorization and analysis of results related to a subject or area of interest, determining the scope of the study, and classifying the knowledge gained. To conduct an SMS, the stages described in Figure 1 must be sequentially applied. In this way, as advancements are made throughout the stages of the process, concrete results that will serve as the direct input of the following stage are obtained.

Considering this process, the executed stages are described next.

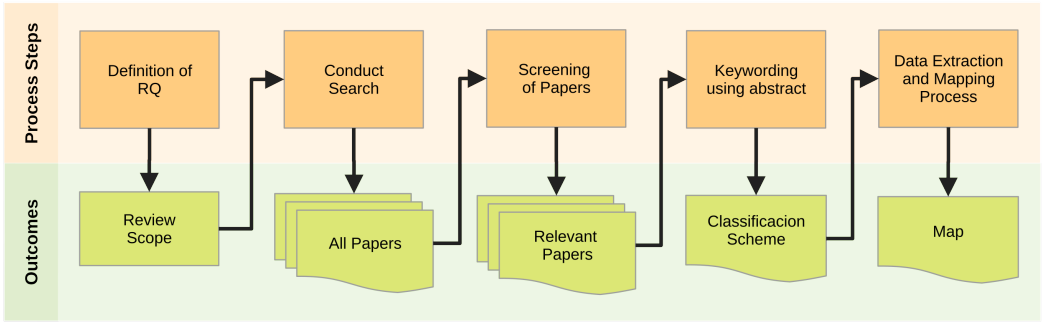

Figure 1. Stages of the systematic mapping process.

3.1. Research Question

Given the general objective of this study, the scopes of analysis were established (see Figure 2) and the research questions were organized as follows: (a) software maintenance; (b) machine learning; (c) research context and approach; finally, the questions that address the (d) intersection of topics (a), (b), and (c).

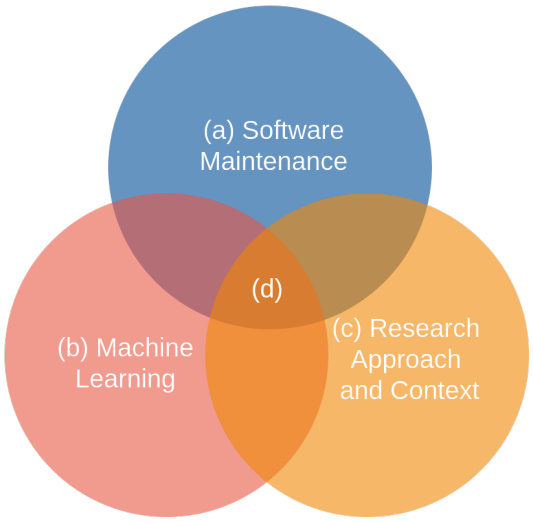

Figure 2. Study areas.

3.1.1. Research Questions on Software Maintenance

According to Table 2, question RQ-M1 investigates the type of maintenance described in the articles, considering four maintenance categories: corrective, perfective, preventive, and adaptive. Question RQ-M2 analyzes the software metrics used in the articles selected. Questions RQ-M3 and RQ-M4 explore the principal maintenance problems addressed in articles and if they used specific maintenance techniques to approach these problems. In the same vein, question RQ-M5 investigates the studies that incorporated a tool of their own for maintenance processes, such as static analyzers, style analyzers, and others.

Table 2. Research questions on maintenance.

ID	Research Question
RQ-M1	What type of maintenance was considered?
RQ-M2	What metrics were adopted?
RQ-M3	What type of maintenance problem was addressed?
RQ-M4	What types of maintenance techniques were mentioned?
RQ-M5	What maintenance tools were most commonly used?

3.1.2. Research Questions on ML

Table 3 lists five questions that were used to analyze the studies that applied ML to software maintenance. Questions RQ-M1 and RQ-M2 address the algorithms and models used, in addition to their main characterizations. Question RQ-ML3 raises questions related to the characteristics of the datasets used, mainly considering availability and type. Finally, questions RQ-M4 and RQ-M5 analyze the hardware and software configurations and the various options that promote the replicability of the studies.

Table 3. Research questions about **machine learning**.

ID	Research Question
RQ-ML1	What types of ML algorithms were used in the articles?
RQ-ML2	What were the characteristics of the ML models considered?
RQ-ML3	What were the main characteristics of the datasets used?
RQ-ML4	What was the software configuration?
RQ-ML5	What was the hardware configuration?

3.1.3. Questions of Context and Approach

Table 4 shows three questions related to the context and approach of the study. Question RQ-C1 examines the types of studies existing on the topic. Three types are considered: (a) academic, (b) industry, or (c) hybrids. Question RQ-C2 investigates the contributions made by each of the studies. These contributions were categorized as the model, strategy, tool, recommendations, and relative findings. With respect to the types of research methods used in the studies, question RQ-C3 considers the following classification categories: survey, experiment, case study, experience report, proof of concept, theoretical, lesson learned, proposal, and evaluation.

Table 4. Research questions on context.

ID	Research Question
RQ-C1	What type of research was conducted?
RQ-C2	What were the main contributions of the studies?
RQ-C3	What type of research method was adopted?

3.1.4. Questions on Topic Intersection

The questions in Table 5 are intertwined with the dimensions of software maintenance, ML, and context, defined as (d) in Figure 2. The aim of these questions is to find gaps or areas with an excess of studies that can serve as a guide for future studies or sectors of technological development. In this light, question RQ-I1 addresses the intersections among the types of maintenance, the type of study, and the ML algorithms used. On the other hand, question RQ-I2 studies the cross between the type of contribution in a particular year while considering the type of algorithm.

Table 5. Research questions on the intersection of themes.

ID	Research Question
RQ-I1	What is the distribution of studies with respect to the type of maintenance, type of research, and algorithms used?
RQ-I2	What are the main contributions, considering years and types of machine learning algorithms?

3.2. Search Execution

From the research questions posed in Section 3.1, a search string was established to define the topic domain, related terms, and the respective partial string search that took the

step to the final search string. The rationale for the construction of the search string was derived from the guidelines of the PICOC criteria [31,32]. Details on the construction of the search string are given in Table 6.

Table 6. Search string structure.

ID	Domain	Related Term	Search Strings
C1	Software Maintenance	maintenance	"software maintenance" OR maintenance
		software maintenance	
C2	Machine Learning	machine learning	"machine learning" OR "artificial intelligence" OR "deep learning"
		artificial intelligence	
		deep learning	
C3	Approach	methodology	methodology OR technique OR method OR approach ORtool OR framework
		technique	
		method	
		approach	
		tool	
		framework	

From the union of C1, C2, and C3, the following result was obtained.

Search String:

*("software maintenance" **OR** maintenance) **AND** ("machine learning" **OR** "artificial intelligence" **OR** "deep learning") **AND** (methodology **OR** technique **OR** Method OR approach **OR** tool **OR** framework)*

All similar terms were related by using the OR operator, whereas the domains were linked with the AND operator. It should be noted that the search string was susceptible to slight changes depending on the search engine on which it was executed. However, this did not affect the study results.

3.3. Sources of Analysis

Referring to the selection of data sources, search engines were included with full access from the study site. The sources in which the search was applied were *ACM*, *IEEE Xplore*, *ScienceDirect*, *Scopus*, *Springer*, *Wiley*, and *WOS*. Figure 3 provides a summary of these results.

3.4. Article Selection Process

To obtain the selected articles, inclusion/exclusion criteria were used on the results delivered by each of the search engines (see Figure 3):

- Inclusion: Proposals, strategies, tools, experiments, methodologies, devices, and/or models that supported the incorporation of ML in software maintenance.
- Exclusion: Studies that were not related to software engineering, programming, or technology applied in a traditional environment. Literature reviews.
- Time range: Between 2011 and 2021.

To select the research works, we first used the inclusion criteria to analyze them by title, abstract, and keywords, thereby obtaining the greatest number of works that made

significant contributions to the study area. In addition, a filter for the time range and prompt elimination of repeated results were applied.

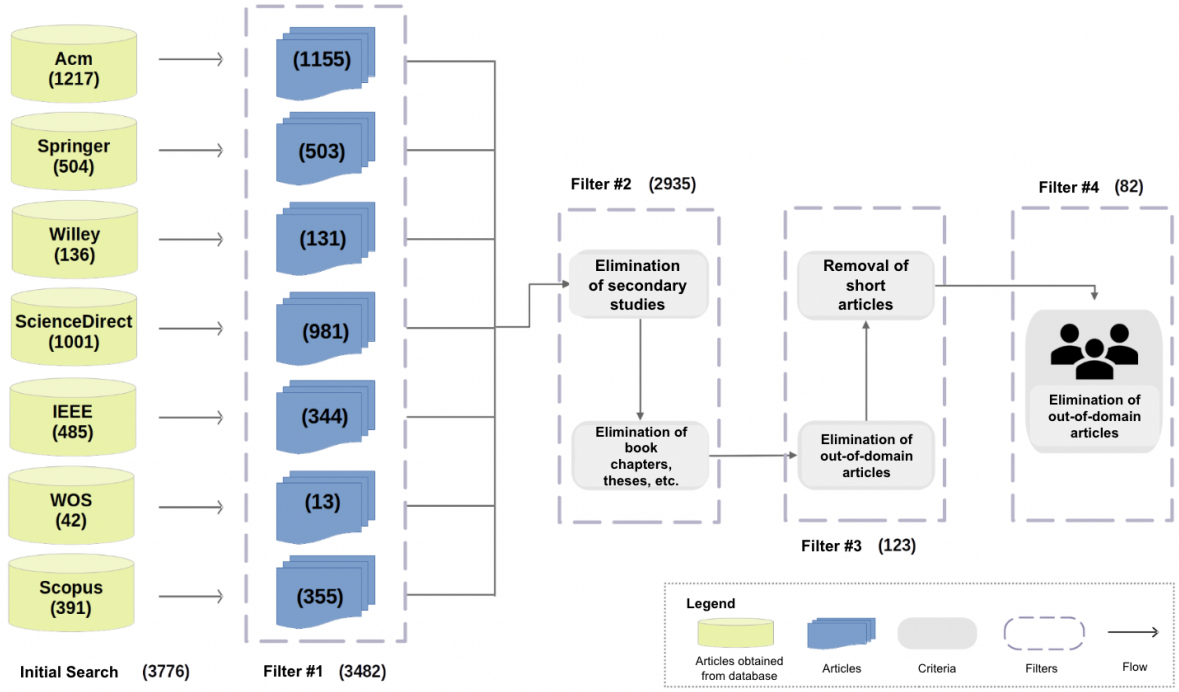

Figure 3. Article filtering stages.

The following iteration used the exclusion criterion on the abstracts of the articles. Brief articles (fewer than three pages), theses, technical reports, and tutorials were eliminated. Only articles that stayed within the domain of the incorporation of ML into software maintenance were selected. Finally, a complete analysis of the candidate documents was performed, and they were put to individual blind voting by each of the authors. Agreement among the researchers was validated by using Fleiss' Kappa index, which was proposed by Gwet [33], and this had a reliability of 78.7%.

The previously described criteria gave rise to filters for sequentially and incrementally grouping and applying the criteria. Table 7 presents the details of each of the filters.

Table 7. Details of the filters applied.

Stage	Filter Element	Detail
Initial Search	Metadata	- All articles not between 2011 and 2021. - Language other than English
Filter 1	Title	- Removal of repeated items
Filter 2	Type of article	- Elimination of book chapters, theses, technical reports, tutorials, books. - Elimination of secondary studies
Filter 3	Title and abstract	- Articles that were not within the spectrum of the software and ML maintenance domain - Elimination of short articles.
Filter 4	Title and abstract	- Cross-review of selected articles that were not within the spectrum of the maintenance domain of software and ML

3.5. Quality Assessment

In order to assess the quality of the primary studies and identify articles that would add value to this research, we established criteria to ensure a minimum of quality. We evaluated each article by using a checklist with options of Y (value = 1) and N (value = 0). Based on the above, we defined four quality criteria that were applied by the reviewers in the various filters:

- **QC1**: The primary article states a clear research objective, and there is an adequate description of the context.
- **QC2**: The primary article indicates what type of research is applied to achieve the research objective.
- **QC3**: The primary article uses a clear and adequate method with the study's objective.
- **QC4**: The primary article provides results related to the incorporation of machine learning in practices associated with software maintenance.

It is possible to review the details of applying these criteria in the supplementary files of this research in Section 3.7.

3.6. Data Extraction

For the data extraction process to be rigorous and to facilitate the management of the extracted data, a classification scheme was designed, which is shown in Figure 4. This scheme addresses the four previously described dimensions (software maintenance, machine learning, research context and approach, and topic intersections), each of which addresses its corresponding research question.

All of the categories related to the area of maintenance emerged from the topic classification proposed in the Software Engineering Body of Knowledge [1]. This classification considers the type, related problems, tools, processes, and techniques.

3.7. Study Support Files

In support of our study, we provide the complete dataset [34] used to run the analyses, for the purpose of replication and validation of our study.

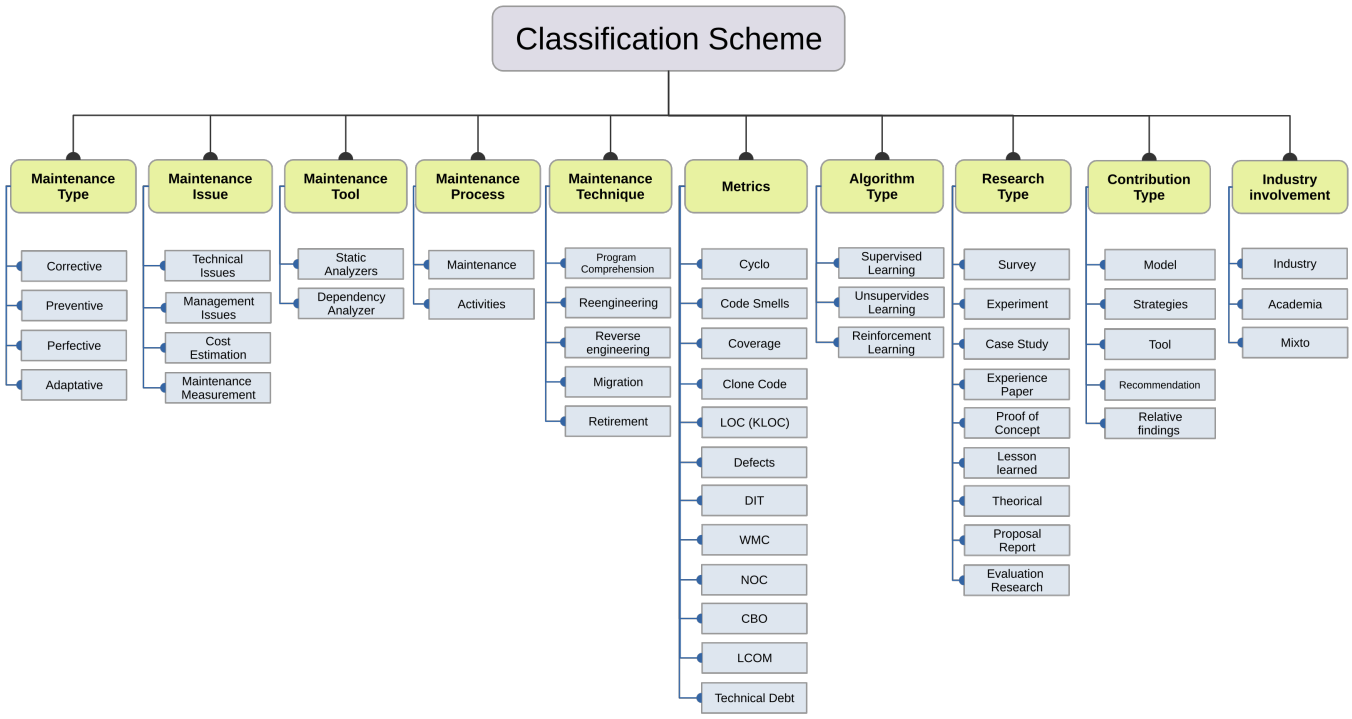

Figure 4. Article classification scheme.

4. Results

This section presents the answers to the research questions posed in Section 3.1 on software maintenance (Section 3.1.1), the use of machine learning (Section 3.1.2), research context (Section 3.1.3), and, finally, topic intersections (Section 3.1.4).

4.1. Answers to the Questions of Context and Research Approach

4.1.1. Publication Years

Figure 5 presents the range of years included in this study. As of 2018, an increase was noted in the number of publications in related conferences. This trend was considerably reduced in 2021, where results similar to those of 2015 were obtained. Of the total of the studies, 79% corresponded to conference articles, whereas 21% corresponded to JCR journal articles. In our opinion, these data could indicate that, despite the use of ML in software maintenance, research in this area is still at an early stage.

4.1.2. Data Sources

Considering the seven data sources (ACM, IEEE, Science Direct, Scopus, Springer, Wiley, and WOS), the initial search yielded a total of 3776 articles, where the sources that provided the greatest numbers of articles came from ACM and Science Direct, with Springer and IEEE in second place at 13%. Then, the application of filters 1 and 2 generated an approximate reduction of 22% of the articles, with the articles from Springer (0), Wiley (126), and WOS (11) substantially decreasing.

In addition, the application of filters 3 and 4 and their respective criteria generated a 97% reduction with respect to the total obtained with the initial search, ultimately ending in 81 articles. It must be mentioned that after applying filter 4, only articles from ACM (31), Science Direct (4), IEEE (26), and Scopus (20) remained (see Figures 3 and 6).

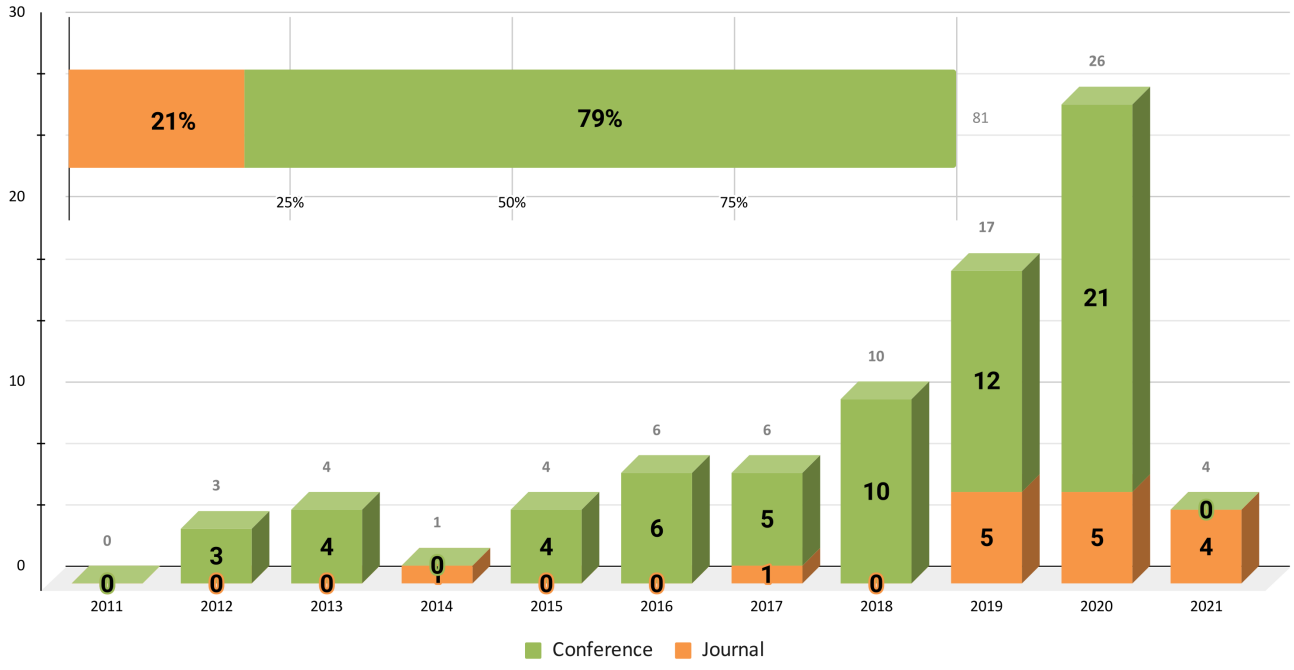

Figure 5. Evolution of the subject.

Concerning the quartiles of scientific journals, 62.5% belonged to quartile 2, while 25% belonged to quartile 1, and they were mainly distributed in SCIE-type editions. The rest of the articles were equally distributed between quartiles 3 and 4 (see Figure 7a,b).

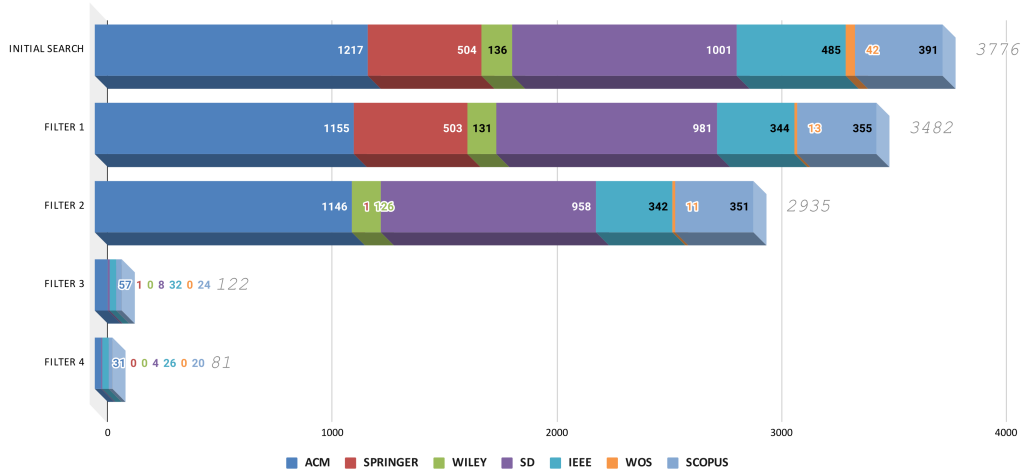

Figure 6. Distribution of results by data source.

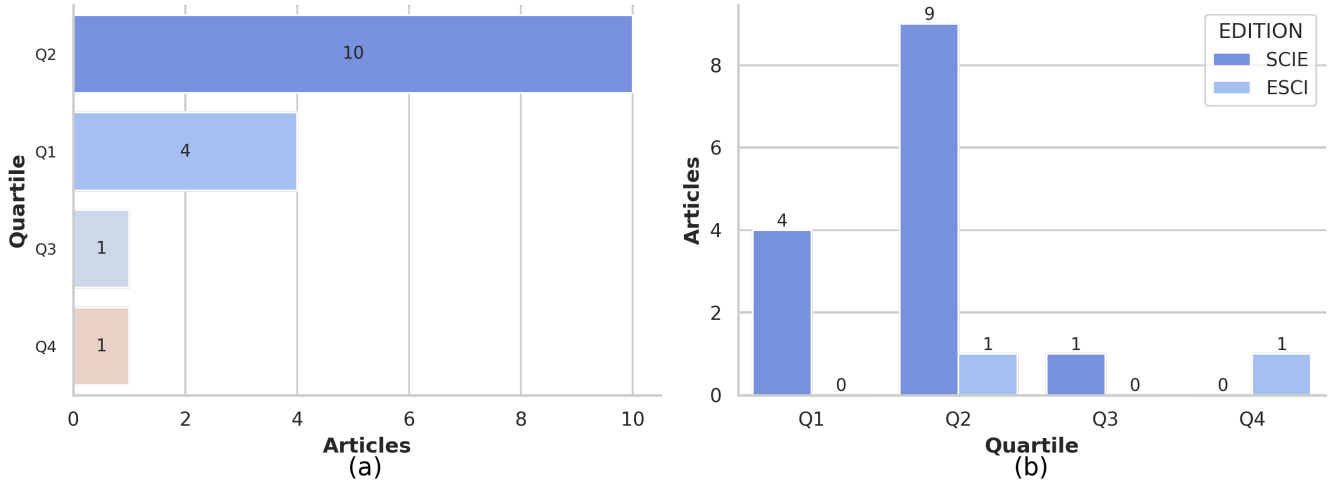

Figure 7. (**a**) Journal quartiles; (**b**) quartiles and types of editions.

In addition, the countries with the most conferences were the USA, China, and India (between 10% and 12% each). In the second place, it was possible to find conferences in Canada, Brazil, and South Korea, which were equally distributed (7.7%), followed by Europe, with Portugal, France, and Spain (4.6%) (see Figure A1a). The conferences with the most articles studied corresponded to ASE (IEEE International Conference on Automated Software Engineering) and ICPC (International Conference on Program Comprehension), with four articles each (see Figures 8 and A1b).

RQ-C1: What type of study was conducted?

Table 8 is quite revealing in different ways. It is surprising that 95% of the articles had no participation from the industry of any kind. These articles mainly delineated the use of the datasets and/or corpora that are commonly used in academia, experimented with excessively limited projects, or analyzed open-source projects. Less than 4% of the articles included both elements of academia applied to industry or vice versa.

RQ-C2: What were the main contributions of the studies?

Figure 9b presents the results of the analyses of the data extraction of the main contributions of the studies. It is noted in this figure that a considerable number of articles proposed support or prediction strategies for maintenance processes by using ML.

On the other hand, a large number of articles (18) suggested relative findings that were only reproducible under very specific conditions. This, added to the few details provided by most of the articles regarding their software and hardware configurations, raises serious

difficulties in terms of the replicability of the studies (see the results of questions RQ-ML4 and RQ-ML5).

A small number of articles (3) proposed recommendations or guidelines for carrying out certain ML-supported maintenance tasks.

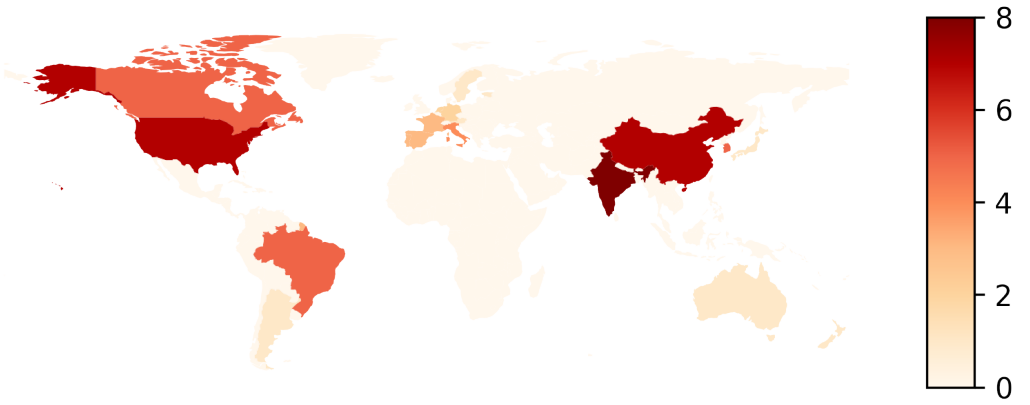

Figure 8. Conferences by country.

Table 8. Industry participation.

Industry Participation	Total	Percent
Academy	77	95.1%
Industry	1	1.2%
Both	3	3.7%

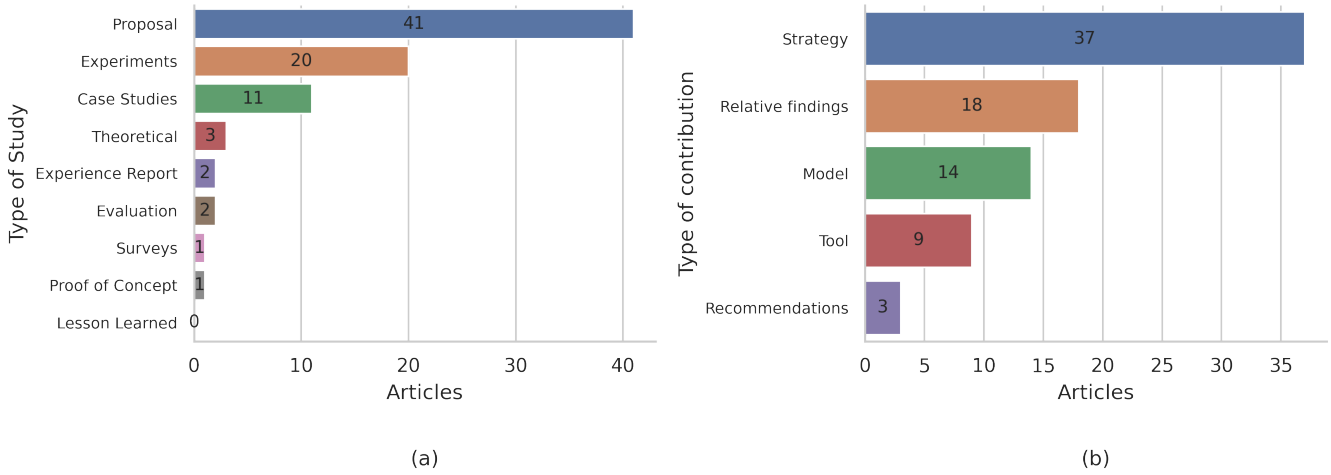

Figure 9. (**a**) Types of studies; (**b**) types of contributions.

RQ-C3: What type of research method was adopted?

It should be emphasized that the ratings of the methods used (proposal, experiment, case studies, theoretical, experience report, evaluation, survey, proof of concept, and lesson learned) came from the authors' own statements. From the data in Figure 9a, it was observed that 41 articles were proposals in which ML was the center of an innovation applied to some software maintenance activity.

In this vein, 20 articles described experiments that incorporated ML. However, these experiments mainly yielded relative results or results that required arduous processes of validation and reproducibility. In the articles that declared their experiments as central research methods, there were characteristics inherent to the proofs of concept, rather than formally defined experiments. On the other hand, only one article explicitly stated

that it was a proof of concept, and there were unclear areas that were confused with experience reports.

4.1.3. Questions about Maintenance

RQ-M1: What type of maintenance was considered?

From the classification diagram and its respective results, how the studies were distributed according to the type of maintenance performed was examined. Software maintenance is a set of activities that begin just prior to delivery that are necessary to support a software system [1]. Typically, software maintenance includes four types of maintenance [1,35]:

- Adaptive maintenance: The adjustment of the source code due to changes in the hardware or changes in software libraries.
- Corrective maintenance: This includes the correction of various types of errors. These errors are usually severe and unexpectedly generated by a flow that was not tested in depth in the development stage.
- Perfective maintenance: This corresponds to the inclusion of a new feature in the system that considerably alters the source code, affecting other components.
- Preventive maintenance: This corresponds to a series of actions that seek to avoid a possible failure. It is common to refer to this type of maintenance as code refactoring [36,37]. However, in development techniques such as test-driven development (TDD), refactoring acts as part of the development process of a project that has not yet gone into production, as mentioned in [38].

Figure 10 presents the results obtained after the classification and later count, where it is possible to identify a majority group of studies that focused on corrective and preventive maintenance, as well as a considerably smaller group that addressed adaptive and perfective maintenance.

More than half of the articles considered preventive maintenance by using ML-based prediction tools. In the same vein, 40% of the studies used maintenance strategies derived from evident errors that had to be corrected. Only five studies addressed adaptive maintenance (2%). This type of maintenance is mainly tasked with incorporating adjustments in the software due to changes in hardware or external libraries that require an update or have been deprecated. The few studies that reported perfective maintenance were noteworthy (4%). This type of maintenance is derived from the inclusion of new features or requirements in the software and is considered the most expensive in the industry.

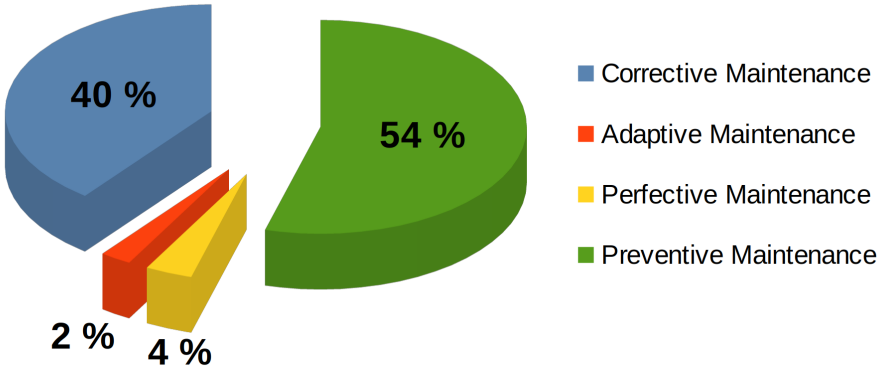

Figure 10. Types of maintenance.

RQ-M2: What metrics were adopted?

Eleven software metrics were identified in the selected articles. These metrics were inclusive and were commonly combined with others. Due to this, a total of 89 uses were counted. Most of the metrics identified came from Chidamber and Kemerer [39] for

object-oriented design, as well as traditional ones, such as cyclomatic complexity and code lines; others were related to test coverage, while those related to software evolution and maintenance were code smells, defects, and code clones (see Table 9).

In the graph in Figure 11, there is a predominance of code smells as the metric linked to software maintenance, with 26 selected articles. Other metrics typically related to software maintenance, such as defects and code clones, were present in nine and five articles, respectively.

The broad use of code lines (KLOCs) in combination with other metrics, such as cyclomatic complexity, in studies on maintenance also stood out. With respect to the object-oriented design metrics, these varied between four and eight articles.

Table 9. Details of identified metrics.

Name	Acronym	Details
Cyclomatic complexity	Cyclo	P65, P67
Smells in code	Code smells	P04, P06, P11, P14, P15, P17, P20, P21, P24, P25, P29, P30, P32, P40, P43, P46, P47, P48, P56, P60, P61, P62, P65, P67, P68, P77
Test Coverage	Coverage	P20
Duplicate code	CC	P01, P03, P13, P18, P30
Lines of code	KLOC	P35, P38, P47, P54, P61, P62, P63, P54, P67, P68, P70, P75, P76, P78
Code error/defects	Defects	P02, P09, P10, P12, P16, P19, P22, P26, P27
Deep inheritance	DIT	P33, P34, P35, P38, P47, P61, P67
Weighted methods per class	WMC	P35, P38, P54, P61, P63, P65, P67
Number of children	NOC	P20, P35, P38, P47, P54, P61, P63, P67
Coupling between objects	CBO	P35, P54, P61, P67
Lack of cohesion	LCOM	P35, P38, P54, P61, P62, P67
Unspecified	–	P05, P07, P08, P23, P28, P31, P36, P37, P39, P41, P42, P44, P45, P49, P50, P51, P52, P53, P55, P57, P58, P59, P64, P66, P69, P71, P72, P73, P74, P79, P80, P81

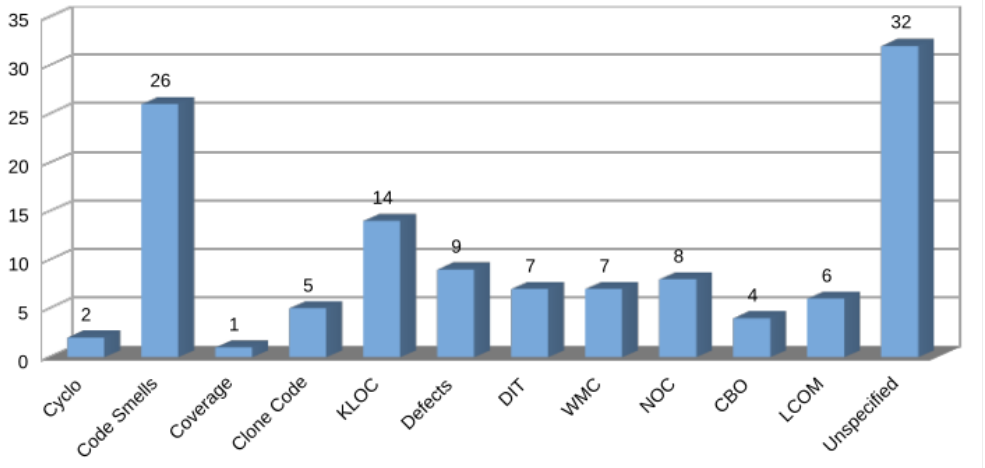

Figure 11. Types of metrics used in studies.

RQ-M3: What kinds of maintenance problems were studied?

Given the classification provided by Swebok in 2014 [1], the selected articles mainly addressed technical problems (78%) related to maintenance, corrections, correction predictions, impact analyses, and others. Conversely, the management of maintenance issues was relegated to a distant second place in relevance with 2.5%. It should be noted that few studies considered other topics related to the costs and incorporation of metrics (see Table 10).

Table 10. Details of studies and maintenance issues studied.

Maintenance Issue	Articles	Percentage
Technical problems	78	96.3%
Management problems	2	2.5%
Maintenance cost estimation	1	1.2%
Software maintenance measurement	0	0.0%
Total	81	100%

RQ-M4: What type of maintenance technique was studied?

The classification diagram proposed by Swebok for maintenance techniques suggests four general types of techniques: those related to understanding the software, re-engineering, reverse engineering, migration, and, finally, removal. Of the 81 articles selected, around 73% used techniques and support strategies to understand the software while considering improvements in style, good practices, and other aspects (see Figure 12). Additionally, 27% of the reviewed articles addressed re-engineering as a central strategy for performing some type of preferably preventive maintenance. It should be noted that no studies that addressed reverse engineering, migration, or removal were identified.

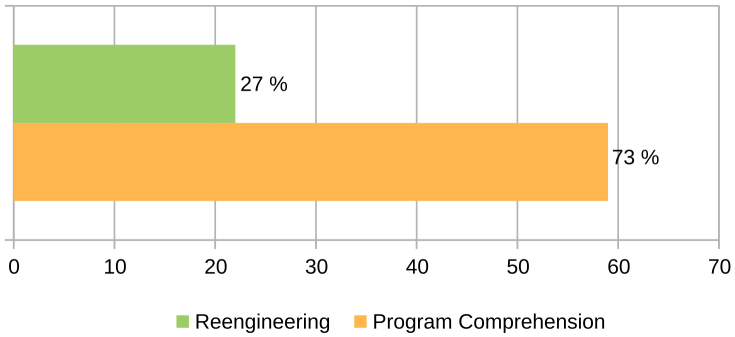

Figure 12. Types of maintenance techniques used in the studies.

RQ-M5: What are the most commonly used maintenance tools?

The tools reported included code-source static analysis and dependency analysis. Of the selected studies, 14.8% (12) mentioned the use of static analysis, whereas the use of dependency analysis (1.2%) and program slicers (1.2%) was equally distributed. On the other hand, a significant 82.7% (67) of the articles did not report the use of any maintenance tools despite addressing mainly technical maintenance issues (Figure 10).

4.1.4. Questions about ML

RQ-ML1L What types of machine learning algorithms were used in the articles?

It must be clarified that a large number of articles used more than one ML algorithm in their procedures, which is why the numbers presented on the map in Figure 13 exceed 81 articles. In relation to the type of algorithm used, there was a predominance of supervised algorithms.

A total of 31 articles indicated that their studies used multilayer perceptron neural networks, convolutional neural networks, and recurrent neural networks. In the same way,

21 articles incorporated an SVM that was applied to software maintenance. The articles that incorporated naive Bayes (18), decision trees (18), and random forests (17) presented a similar distribution. Only three articles used algorithms inspired by natural processes. Traditionally, such algorithms have been classified as reinforcement learning algorithms.

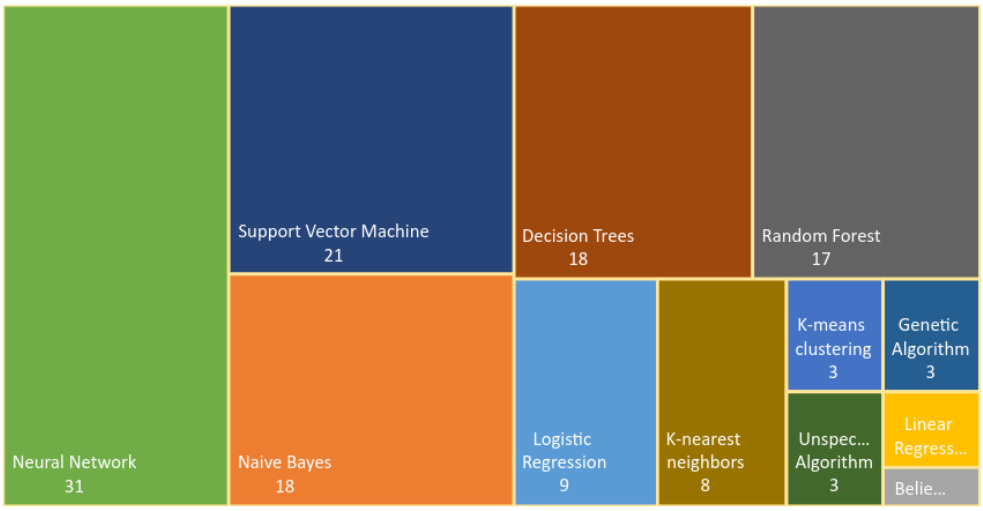

Figure 13. Distribution of the ML algorithms used.

RQ-ML2: What characteristics did the ML models included present?

With regard to the incorporation of previously trained models, the results showed an equal distribution to that of those who did not provide details of any type of model (33%), who did not include a previously trained model (31%), and who incorporated previously trained models in their study (35.8%), but without naming the models specifically (see Figure 14b).

However, a surprising 5.4% of those that included previously trained models made their models available for study in a repository or on a website (see Figure14a).

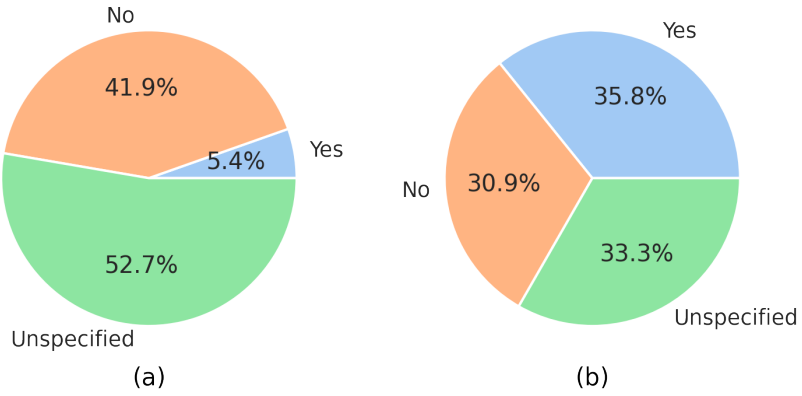

Figure 14. (**a**) Availability of the models used; (**b**) previously trained models.

RQ-ML3: What were the main characteristics of the datasets used?

Concerning the most common types of datasets (images, text, numeric, audio, time, series, categorical), 61% of the articles pointed out that their datasets were mainly text-based, whereas only 2% of the articles reported exclusively numerical datasets. Additionally, 22% of the articles did not detail the types of datasets used.

Only 62% of the datasets in the selected articles were available or had their origin described, with the PROMISE (More information in http://promise.site.uottawa.ca/SERepository/ (accessed on 3 October 2022)) dataset being the most commonly used. In addition,

22% mentioned but did not provide their datasets. The rest of the articles (16%) offered no indications of the datasets used.

RQ-ML4 and RQ-ML5: What was the configuration of the software and hardware?

For the case of RQ-ML4, only 40% of the articles gave details of the software used (see Figure 15). Then, of that percentage, 36% of the articles included details about the library or ML-related tool, where the most frequently used was Weka, while the least frequently used were Keras and TensorFlow.

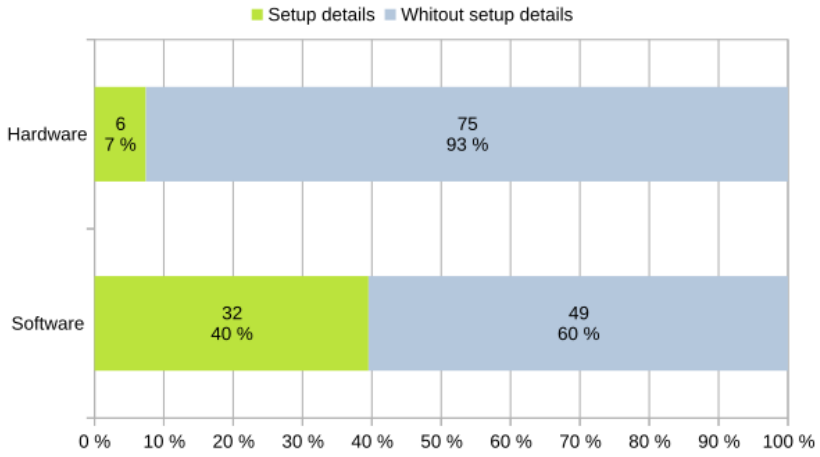

Figure 15. Distribution of articles indicating software and/or hardware details.

A total of 27% indicated the tool or maintenance library used for the study, emphasizing tools such as PMD and iPlasma. The articles that detailed an IDE, programming language, library, or development tool and the operating system used were equally distributed (Figure 16a). The results of this analysis are provided in Table A3 of Appendix B.

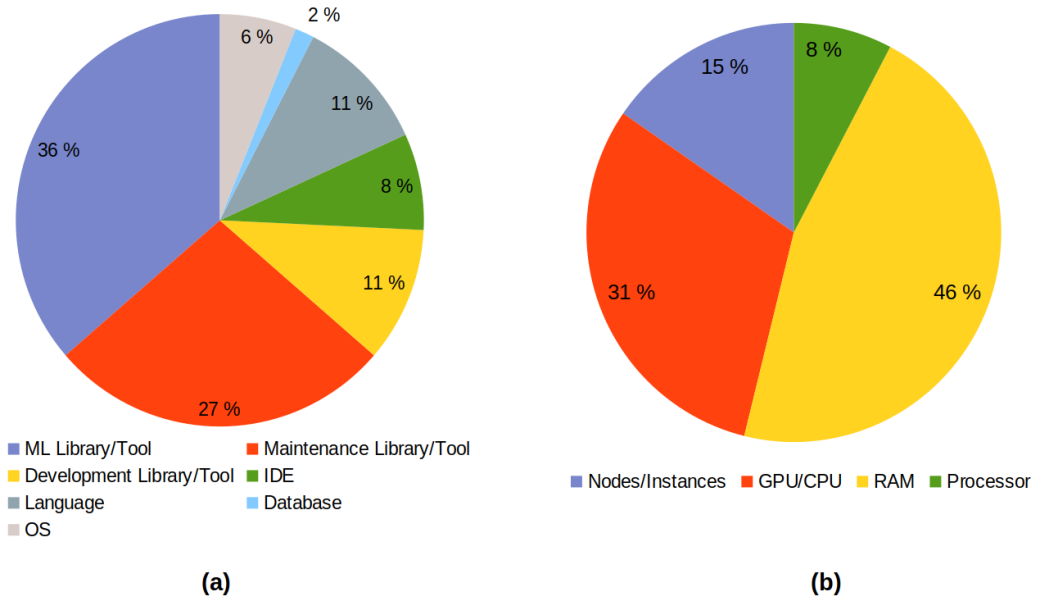

Figure 16. (**a**) Types of detailed software configurations; (**b**) types of hardware configurations identified.

In terms of question RQ-ML5, which looks at the details of the hardware, the situation was quite critical—only 7% of the articles gave indications of their configurations (see Figure 15). Considering this percentage, 46% detailed aspects such as the required RAM, 31% detailed characteristics of the GPU and/or CPU, 1% provided the number of nodes or

instances used to process the data, and, finally, a limited 8% gave basic information about the type of processor used to reach the results.

5. Discussion

In this section, we will discuss the key findings obtained from the analysis of the answers to the research questions related to maintenance, ML, and context.

5.1. Maintenance

From the evidence compiled to answer question RQ-M1, it may be stated that more than half of the reviewed articles (54%) addressed preventive maintenance by taking refactoring as the central topic. The boom on this topic began after the release of the text by Martin Fowler et al. [40], with the suggestion of 22 code smells and a series of strategies for refactoring. However, the code smells and code refactoring strategies lacked the operationalization, systematization, and metrics for putting their applications into practice, giving way to a series of studies that addressed these topics [41,42].

This was consistent with the evidence regarding the benefits of refactoring: improvements in the understanding of the code [43] and the considerable improvements by correcting code smells [44]. In spite of such benefits, there are also questions regarding code refactoring, such as the accidental introduction of bugs when refactoring [45], problems when performing a merge in code repositories [46], and questions regarding which smells would not be relevant for refactoring code if the effort when conducting perfective maintenance is similar in projects with and without the presence of code smells [47].

It is worth noting that a large part of the findings described above arose mainly from the mining and analysis of open code projects, which is why these results are not fully transferable and/or generalizable to private projects. It is surprising that only 4% of the articles addressed perfective maintenance, which traditionally consumes the greatest amount of effort and budget in software projects [48].

There is, therefore, an emphasis on anticipating or predicting where software can fail to execute preventive maintenance automatically or semiautomatically (something that might fail), but there is little evidence about software in operation to which new features must be added. In the same vein, suggestions such as those of Lehman [49] remain immutable, with limited progress in reducing the degradation of the software despite the ML boom.

In relation to the types of metrics used (RQ-M2), the predominance of metrics that have traditionally been associated with preventive code maintenance, such code smells, predominated. Although this description arose at the end of the 1990s through a text by Martin Fowler [40], the industry knew the impact of code smells long before that [50]. The data showed the massive use of this type of metric thanks to the numerous advances that linked code smells with effort [51,52], which has reinforced their predominance.

Additionally, the increase in the complexity of the software has taken a step toward increasing the data to be analyzed; however, the metrics associated with maintainability have not kept pace. Recent studies demonstrated that, although code metrics can help investigate maintenance efforts, the results of the studies are contradictory [53]. The predominance of code lines as a metric associated with maintenance is surprising given the current advance in the discipline. However, it is possible to hypothesize that its predominance is based on its being an essential input in other software metrics that use KLOC, such as the flows proposed in [42].

Other aspects of the central elements of these results are related to design and object-oriented software in general. In this context, the traditional way to measure this type of software is to use CK metrics [39]. The results reported a cross-sectional use of these metrics and their relevance in the industry today despite the constant evolutions in programming paradigms.

With respect to the predominance of the maintenance issues considered (RQ-M3), the technical problems in maintenance were described by most of the articles, with the problems of the management associated with software maintenance and the management

of human capital being relegated to second place; these were characterized by a constant fear of the obsolescence generated by the type of technology associated with inherited systems and overloads of tasks with a certain role. These factors are common causes of stress in software developers [54,55].

The excess of studies that addressed technical problems could be explained by the boom in tools focused on corrections and suggestions [14,22,56]. However, after decisions on the use, there is a total lack of management processes that support the maintenance process. Therefore, instead of considering areas with errors, a systemic approach that oversees the entirety of the interactions in a maintenance process (including people) rather than isolated problems is required.

The possible interpretations of the results obtained from question RQ-M4 might consider that in recent years, the education of software engineers has benefited from a wave of good practices, guidelines, and technical support, which have improved code understandability and whose main focus is prevention [43,57]. However, elements such as the relation between time pressure and technical debt are scarcely addressed at the training level.

Traditional [58] and recent reports [59] have highlighted the impacts of these factors on the quality of code, regardless of the "good intentions" when coding. In the same way, the predominance of techniques such as re-engineering accounts for a number of software packages, which, as it is no longer possible to repair them given their level of degradation, are re-designed in order to install new software from zero. This raises the following question: Why is there no propagation of software removal techniques considering the impact of removing or replacing software that is currently in use?

On the other hand, few studies specified the use of maintenance support tools (RQ-M5), with the main tool being static code analyzers. The handling of thresholds that activate metrics for this type of software was mentioned as a problem in previous publications [60], indicating code as defective when it is not and vice versa. Software has benefitted from such tools, but their use with factory configurations or by default should be avoided.

5.2. Machine Learning

The results obtained from question RQ-ML1 establish that supervised ML algorithms are hegemonic. A considerable number of articles incorporated multilayer neural networks and SVM algorithms. This was mainly substantiated by the upsurge in libraries supported by large software companies, such as Tensorflow, Keras, and others, in addition to their start-up in a wide-reaching and versatile language, such as Python, which facilitates the mining of code repositories.

The lack of studies that reported experiences with algorithms that used reinforcement learning underscored the lack of environments and libraries that directly support such algorithms. In many cases, the existing algorithms do not work in specific situations, and their adjustment can be complex or even require ad hoc development.

With respect to the use of previously trained models (RQ-ML2), the poor availability of the models used is surprising. It could be argued that the reasons for this finding are supported by intellectual property or the licensing of certain knowledge. In the same way, the main dataset used was PROMISE. This dataset has characteristics that make it suitable for use because, for example, the structure, documentation, and nature of the data that come from industrial projects make this dataset suitable for laboratory situations.

In relation to the characteristics of the hardware and software (RQ-ML4, RQ-ML5), the few details of the configurations of both the hardware and software stand out. A total of 93% of the articles did not provide the details of their hardware configurations, whereas 60% did not do so with respect to the software. There was also no repository or anything similar provided so that the studies could be replicated or the configurations could be investigated. These results are supported by the criticisms of reproducibility and replicability in deep learning (DL) and ML proposed in [61], which described innumerable articles that did not make the evidence that could ensure these characteristics clear. The replicability of

studies that obtain results from various sources is fundamental in software engineering. Accordingly, including details related to the hardware and software characteristics used to reach results in articles facilitates the replicability of thereof.

5.3. Orthogonal Questions

RQ-I1: What was the distribution of studies in terms of maintenance type, study type, and the algorithms used?

There are very particular phenomena that can be concluded upon after our work (see Figure 17). First, 63.57% of the studies worked with a preventive approach (with respect to the type of maintenance), and the second approach was corrective maintenance (29.9%). The rest of the alternatives had rates that were very far below the average.

If we observe the types of ML algorithms that were used, they highlighted, in the first place, those related to artificial neural networks (ANNs), followed by support vector machines (SVMs) and random forests. This behavior is noted regardless of the type of software maintenance studied. With respect to the research mechanism, almost half of the studies opted for the proposal of new mechanics, but these, however, were not necessarily accompanied by robust methodological or evaluative processes to achieve their replication.

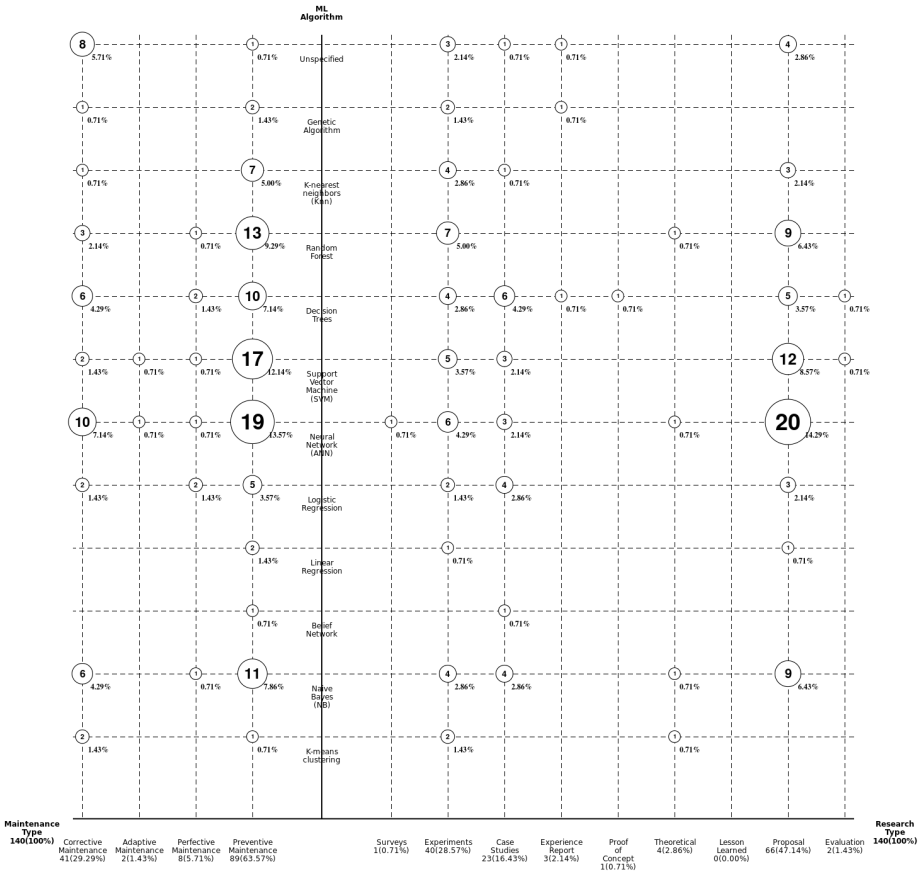

Figure 17. Distribution of studies by type of maintenance, research, and ML algorithm.

RQ-I2: What was the distribution of studies in terms of software metrics, contributions of the articles, and types of ML algorithms?

Of all the 11 software metrics in the evaluated articles, a predominance of code smells as metrics linked to software maintenance was noted in 27 articles. Additionally, this option concentrated on the use of the supervised learning type of ML algorithm, capturing almost the entirety of this metric. Second, the KLOC metric was noted; again, it was measured predominantly through supervised learning. In total, the supervised learning approach stood out in 85.6% of the total results, regardless of the metric evaluated.

With respect to the contributions, half of the articles suggested support or prediction strategies for maintenance processes using ML. However, 28.93% suggested relative findings. These are reproducible under certain conditions but not necessarily replicable (see Figure 18).

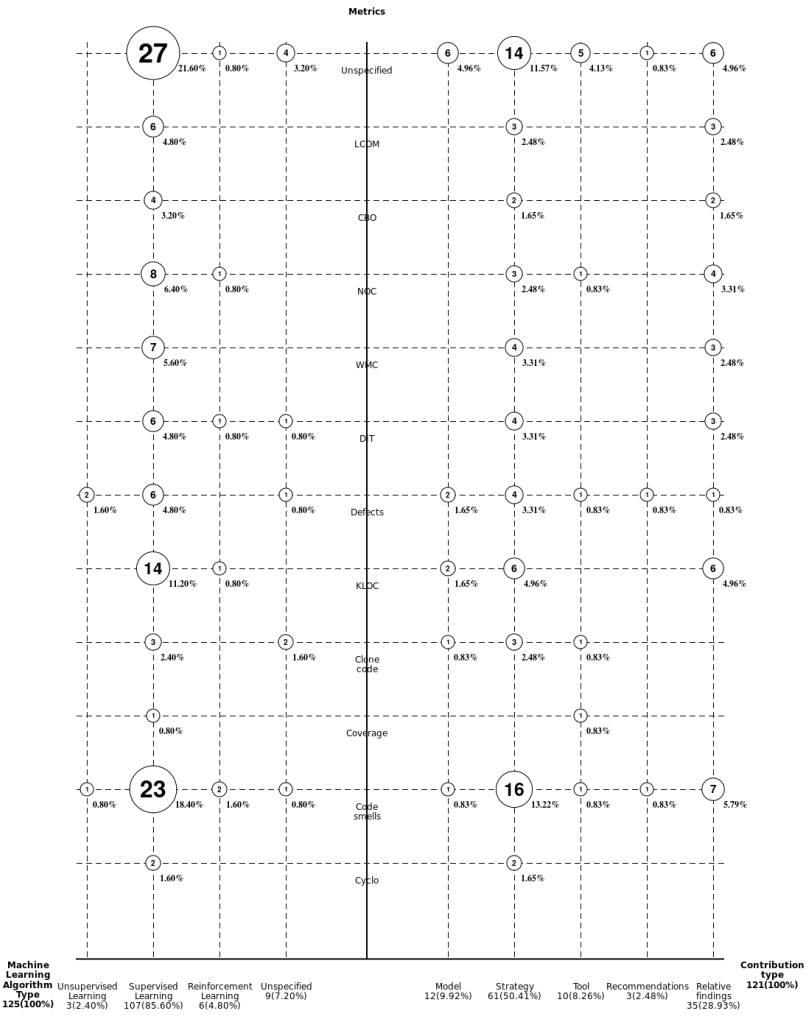

Figure 18. Distribution of studies by type of ML algorithm, software metrics, and type of contribution.

6. Threats to Validity

The following aspects were considered with their respective mitigation plans:

- Study search: To mitigate this threat, the predefined search string was used in the main electronic databases. Before conducting a real search on each site, we performed a pilot search in all of the selected databases to verify the accuracy of our search string.
- Study selection bias: The studies were selected by applying explicitly defined inclusion and exclusion criteria. To avoid possible biases, validation was also performed by cross-checking all of the studies selected. The Kappa coefficient value was 0.78, which indicated that there was a reasonable level according to the authors, and the risk of finding differences in the relevance of the studies was significantly reduced.
- Data extraction bias: To collect consistent data and avoid data extraction biases, a summary was made of the results of the data found. First of all, one of the authors built the result distribution tool. Next, two authors distributed the number of studies equally and collected the data according to the data extraction form. The same two authors discussed their results and shared them regularly to avoid data extraction bias.

Regarding validity:

1. External validity: The threats to external validity are restrictions that limit the ability to generalize the results. The inherent threat related to external validity is if the primary studies reported machine learning initiatives applied to software maintenance. We mitigated this threat by choosing peer-reviewed studies and excluding gray literature.
2. Validity of the conclusions: This corresponds to questions that affect the ability to draw the correct conclusions. Although we applied the best practices from benchmark studies [13,32,62,63] in the different stages, it was impossible to identify all of the existing relevant primary studies. Therefore, we managed this threat to validity by discussing our results in sessions with industry professionals dedicated to software maintenance.
3. Validity of constructs: This is related to the generalization of the results to the concept or theory that underlies the study. The main threat is the subjectivity in our results. To mitigate this threat, the three researchers carried out the main steps of the systematic mapping study independently. Later, they discussed their results to reach a consensus.

7. Conclusions

It is interesting to analyze each of the selected articles at this level of detail to help us visualize future lines of research. As a final reflection, we emphasize two main points.

First, a large number of articles (28.93%) suggested "relative" findings; that is to say, their methodological processes were conditional upon particular scenarios. The articles that presented these scenarios are difficult to replicate in other situations.

From our perspective, replicability is one of the basic characteristics when consolidating scientific advances. When a study is published, there is a series of standards that guarantee that anyone can repeat the research under the conditions presented and obtain the same result; thus, this can result in the soundness of the study or the detection of any errors. Can we talk about new knowledge when the verification of the scientific method is disqualified?

Second, software maintenance combined with ML techniques was the topic of study. Although it is true that there has been a growing trend since 2011, especially in conferences, this is not reflected in more formal initiatives, such as in scientific journals. This may be for different reasons, such as that replicability is a critical aspect that is not permitted in more rigorous instances, or that the use of ML techniques is focused on other more popular initiatives.

However, for the authors, it is important to carry out such exercises, especially given the potential for and space of new studies that can positively affect how we build and maintain software.

Author Contributions: Conceptualization, O.A.B.; methodology, O.A.B. and J.D.; validation, O.A.B. and J.D.; formal analysis, O.A.B. and J.D.; investigation, O.A.B.; resources, O.A.B. and J.D.; data curation, O.A.B.; writing—original draft preparation, O.A.B.; writing—review and editing, O.A.B., J.D. and J.L.F.; visualization, O.A.B.; supervision, O.A.B. and J.D.; project administration, O.A.B., J.D. and J.L.F.; funding acquisition, O.A.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Universidad de La Frontera, Vicerrectoría de Investigación y Postgrado. Oscar Ancán Bastías is grateful to the research project DIUFRO DI21-0052.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: Special thanks go to Jonathan Jara and Mauricio Campos for their helpful technical support in this work.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Primary Studies

Tables A1 and A2 show lists of the articles selected for review.

Table A1. Selected articles—part 1.

ID	Article	Ref.
P1	A Defect Estimator for Source Code: Linking Defect Reports with Programming Constructs Usage Metrics	[64]
P2	A Hybrid Bug Triage Algorithm for Developer Recommendation	[65]
P3	A Method for Identifying and Recommending Reconstructed Clones	[66]
P4	A Model to Detect Readability Improvements in Incremental Changes	[67]
P5	A Neural-Network Based Code Summarization Approach by Using Source Code and Its Call Dependencies	[68]
P6	A Preliminary Study on the Adequacy of Static Analysis Warnings with Respect to Code Smell Prediction	[69]
P7	A Search-Based Training Algorithm for Cost-Aware Defect Prediction	[70]
P8	Achieving Guidance in Applied Machine Learning through Software Engineering Techniques	[71]
P9	Active Semi-Supervised Defect Categorization	[72]
P10	An Exploratory Study on the Relationship between Changes and Refactoring	[73]
P11	Applying Machine Learning to Customized Smell Detection: A Multi-Project Study	[74]
P12	Bug Localization with Semantic and Structural Features Using Convolutional Neural Network	[75]
P13	CloneCognition: Machine Learning Based Code Clone Validation Tool	[76]
P14	Combining Spreadsheet Smells for Improved Fault Prediction	[77]
P15	Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection	[78]
P16	DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network	[79]
P17	Detecting Bad Smells with Machine Learning Algorithms: An Empirical Study	[80]
P18	Functional Code Clone Detection with Syntax and Semantics Fusion Learning	[81]
P19	Improving Bug Detection via Context-Based Code Representation Learning	[82]
P20	InSet: A Tool to Identify Architecture Smells Using Machine Learning	[83]
P21	Machine Learning Techniques for Code Smells Detection	[56]
P22	Predicting Software Defects with Explainable Machine Learning	[84]
P23	Prediction of Web Service Anti-Patterns Using Aggregate Software Metrics and Machine Learning	[85]
P24	Report on Evaluation Experiments Using Different Machine Learning Techniques for Defect Prediction	[86]
P25	Software Analytics in Practice: A Defect Prediction Model Using Code Smells	[87]
P26	Software Defect Prediction Based on Manifold Learning in Subspace Selection	[88]
P27	Towards an Emerging Theory for the Diagnosis of Faulty Functions in Function-Call Traces	[89]
P28	Towards Better Technical Debt Detection with NLP and Machine Learning Methods	[90]
P29	Understanding and Detecting Harmful Code	[91]
P30	What is the Vocabulary of Flaky Tests?	[92]
P31	A Machine Learning Approach to Classify Security Patches into Vulnerability Types	[93]
P32	A Model Based on Program Slice and Deep Learning for Software Defect Prediction	[94]
P33	A Novel Four-Way Approach Designed With Ensemble Feature Selection for Code Smell Detection	[95]
P34	A self-adaptation framework for dealing with the complexities of software changes	[96]
P35	A Study on Software Defect Prediction using Feature Extraction Techniques	[97]
P36	Aging Related Bug Prediction using Extreme Learning Machines	[98]
P37	Aging-Related Bugs Prediction Via Convolutional Neural Network	[99]
P38	An Empirical Framework for Code Smell Prediction using Extreme Learning Machine	[100]
P39	Application of LSSVM and SMOTE on Seven Open Source Projects for Predicting Refactoring at Class Level	[101]
P40	Assessment of Code Smell for Predicting Class Change Proneness Using Machine Learning	[102]
P41	Automated Identification of On-hold Self-admitted Technical Debt	[103]
P42	Automatic Clone Recommendation for Refactoring Based on the Present and the Past	[104]
P43	Comparison of Machine Learning Methods for Code Smell Detection Using Reduced Features	[105]
P44	Convolutional Neural Networks-Based Locating Relevant Buggy Code Files for Bug Reports	[106]
P45	Detect functionally equivalent code fragments via k-nearest neighbour algorithm	[107]

Table A2. Selected articles—part 2.

ID	Article	Ref.
P46	Detecting code smells using machine learning techniques: Are we there yet?	[108]
P47	Experience report: Evaluating the effectiveness of decision trees for detecting code smells	[109]
P48	Identification of Code Smell Using Machine Learning	[110]
P49	Improving Bug Detection and Fixing via Code Representation Learning	[111]
P50	Machine Learning based Static Code Analysis for Software Quality Assurance	[112]
P51	SMURF: A SVM-based Incremental Anti-pattern Detection Approach	[113]
P52	Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities	[114]
P53	Support vector machines for anti-pattern detection	[115]
P54	The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring	[116]
P55	A comparative study of supervised learning algorithms for re-opened bug prediction	[117]
P56	A hybrid approach to detect code smells using deep learning	[118]
P57	A neural network approach for optimal software testing and maintenance	[119]
P58	Automatic Localization of Bugs to Faulty Components in Large Scale Software Systems	[120]
P59	Automatic traceability maintenance via machine learning classification	[121]
P60	Code smell detection: Towards a machine learning-based approach	[122]
P61	Deep Learning Approach for Software Maintainability Metrics Prediction	[123]
P62	Deep Learning Based Code Smell Detection	[124]
P63	Deep learning based feature envy detection	[125]
P64	Deep learning code fragments for code clone detection	[126]
P65	Detecting Code Smells using Deep Learning	[127]
P66	Duplicate Bug Report Detection and Classification System Based on Deep Learning Technique	[128]
P67	Finding bad code smells with neural network models	[129]
P68	Improving machine learning-based code smell detection via hyper-parameter optimization	[130]
P69	Improving statistical approach for memory leak detection using machine learning	[131]
P70	Predicting Software Maintenance effort using neural networks	[132]
P71	Semantic clone detection using machine learning	[133]
P72	Software Defects Prediction using Machine Learning Algorithms	[134]
P73	Towards anticipation of architectural smells using link prediction techniques	[135]
P74	Using software metrics for predicting vulnerable classes and methods in Java projects	[136]
P75	A machine learning based framework for code clone validation	[137]
P76	A Novel Machine Learning Approach for Bug Prediction	[138]
P77	Code smell detection by deep direct-learning and transfer-learning	[139]
P78	Deep learning based software defect prediction	[140]
P79	An empirical investigation into learning bug-fixing patches in the wild via neural machine translation	[141]
P80	On learning meaningful code changes via neural machine translation	[142]
P81	Learning how to mutate source code from bug-fixes	[143]

Appendix B. Software and Hardware Details

Table A3 shows the articles that included software and hardware details.

Table A3. Articles indicating software and hardware details.

ID	Software Setup							Hardware Setup			
	ML Library/Tool	**M.Lib./Tool**	**D.Lib./Tool**	**IDE**	**Language**	**Database**	**OS**	**Nodes/Instances**	**GPU/CPU**	**RAM**	**Processor**
P37	x				x				x	x	
P40		x									
P42	x										
P44	x				x						
P46	x										
P47	x	x									
P48				x							
P52	x		x		x		x	x		x	x
P54	x	x				x	x	x	x	x	
P55	x										
P56		x			x						
P59	x										
P60		x									
P62	x	x	x	x	x						
P63	x	x	x	x							
P64	x	x	x	x							
P65	x	x									
P66	x						x		x	x	
P67	x										
P68		x	x								
P69	x	x									
P70		x									
P71			x				x			x	
P72	x										

Table A3. *Cont.*

ID	Software Setup							Hardware Setup			
	ML Library/Tool	**M.Lib./Tool**	**D.Lib./Tool**	**IDE**	**Language**	**Database**	**OS**	**Nodes/Instances**	**GPU/CPU**	**RAM**	**Processor**
P73	x										
P74	x	x									
P75	x	x	x		x						
P76				x							
P77	x	x			x				x	x	
P79	x	x									
P80	x	x									
P81	x	x									

Appendix C. Journals and Conference Details

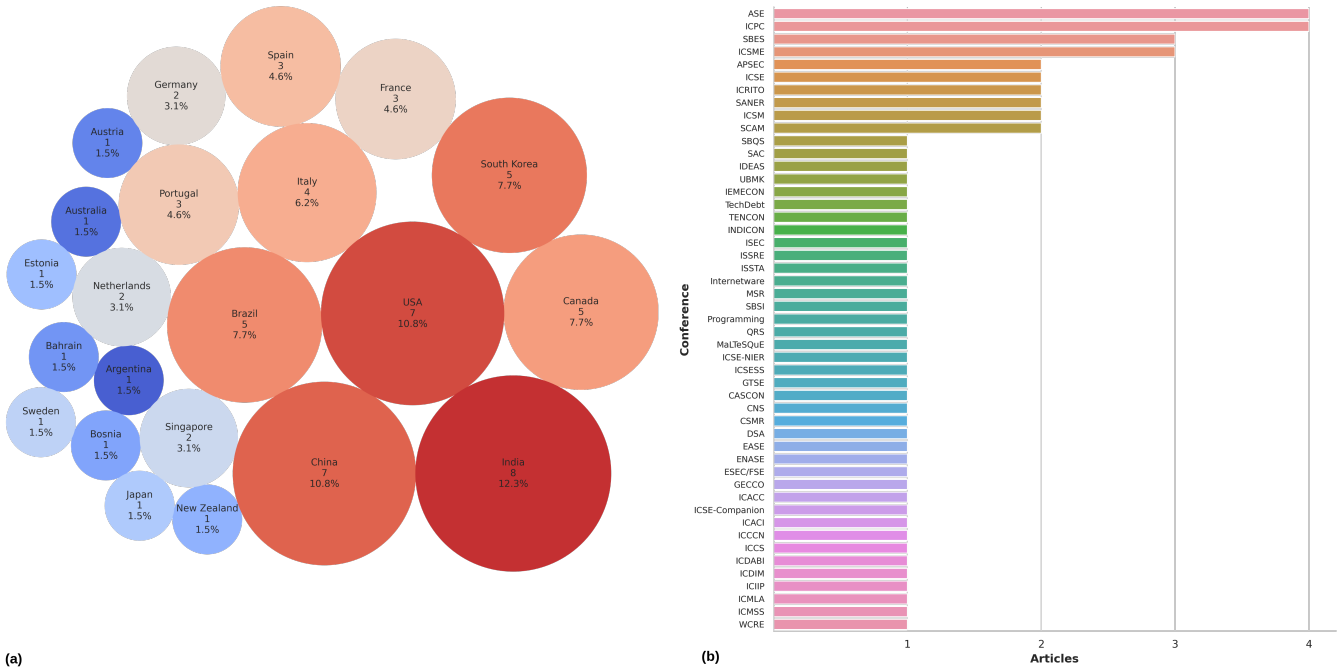

Figure A1. (**a**) Conferences by country and (**b**) conference details.

References

1. Bourque, P.; Fairley, R.E. Guide to the Software Engineering Body of Knowledge (SWEBOK) version 3. In *Technical Report*; IEEE Computer Society: Washington, DC, USA, 2014.
2. Van Vliet, H.; Van Vliet, H.; Van Vliet, J. *Software Engineering: Principles and Practice*; John Wiley & Sons: Hoboken, NJ, USA, 2008; Volume 13.
3. Erlikh, L. Leveraging legacy system dollars for e-business. *IT Prof.* **2000**, *2*, 17–23. [CrossRef]
4. Breaux, T.; Moritz, J. The 2021 Software Developer Shortage is Coming. *Commun. ACM* **2021**, *64*, 39–41. [CrossRef]
5. Gallagher, K.; Fioravanti, M.; Kozaitis, S. Teaching Software Maintenance. In Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 29 September–4 October 2019.
6. Börstler, J.; Paech, B. The role of method chains and comments in software readability and comprehension—An experiment. *IEEE Trans. Softw. Eng.* **2016**, *42*, 886–898. [CrossRef]
7. Kafura, D.; Reddy, G.R. The use of software complexity metrics in software maintenance. *IEEE Trans. Softw. Eng.* **1987**, *SE-13*, 335–343. [CrossRef]
8. Glass, R.L. Frequently forgotten fundamental facts about software engineering. *IEEE Softw.* **2001**, *18*, 112. [CrossRef]
9. Ampatzoglou, A.; Ampatzoglou, A.; Chatzigeorgiou, A.; Avgeriou, P. The financial aspect of managing technical debt: A systematic literature review. *Inf. Softw. Technol.* **2015**, *64*, 52–73. [CrossRef]
10. Alsolai, H.; Roper, M. A systematic literature review of machine learning techniques for software maintainability prediction. *Inf. Softw. Technol.* **2020**, *119*, 106214. [CrossRef]
11. Lenarduzzi, V.; Sillitti, A.; Taibi, D. Analyzing Forty Years of Software Maintenance Models. In Proceedings of the 39th International Conference on Software Engineering Companion, Buenos Aires, Argentina, 20–28 May 2017. [CrossRef]
12. Petersen, K.; Feldt, R.; Mujtaba, S.; Mattsson, M. Systematic mapping studies in software engineering. In Proceedings of the Evaluation and Assessment in Software Engineering (EASE), Bari, Italy, 26–27 June 2008; Volume 8, pp. 68–77.
13. Petersen, K.; Vakkalanka, S.; Kuzniarz, L. Guidelines for conducting systematic mapping studies in software engineering: An update. *Inf. Softw. Technol.* **2015**, *64*, 1–18. [CrossRef]
14. Heckman, S.; Williams, L. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Inf. Softw. Technol.* **2011**, *53*, 363–387. [CrossRef]
15. Penzenstadler, B.; Raturi, A.; Richardson, D.; Calero, C.; Femmer, H.; Franch, X. Systematic mapping study on software engineering for sustainability (SE4S). In Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, London, United Kingdom, 13–14 May 2014; pp. 1–14.
16. Malhotra, R.; Bansal, A. Predicting change using software metrics: A review. In Proceedings of the 2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)(Trends and Future Directions), Noida, India, 2–4 September 2015; pp. 1–6.

17. Jayatilleke, S.; Lai, R. A systematic review of requirements change management. *Inf. Softw. Technol.* **2018**, *93*, 163–185. [CrossRef]
18. Alsolai, H.; Roper, M. A systematic review of feature selection techniques in software quality prediction. In Proceedings of the 2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA), Ras Al Khaimah, United Arab Emirates, 19–21 November 2019; pp. 1–5.
19. Caram, F.L.; Rodrigues, B.R.D.O.; Campanelli, A.S.; Parreiras, F.S. Machine learning techniques for code smells detection: A systematic mapping study. *Int. J. Softw. Eng. Knowl. Eng.* **2019**, *29*, 285–316. [CrossRef]
20. Carvalho, T.P.; Soares, F.A.; Vita, R.; Francisco, R.d.P.; Basto, J.P.; Alcalá, S.G. A systematic literature review of machine learning methods applied to predictive maintenance. *Comput. Ind. Eng.* **2019**, *137*, 106024. [CrossRef]
21. Pizzoleto, A.V.; Ferrari, F.C.; Offutt, J.; Fernandes, L.; Ribeiro, M. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *J. Syst. Softw.* **2019**, *157*, 110388. [CrossRef]
22. Azeem, M.I.; Palomba, F.; Shi, L.; Wang, Q. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Inf. Softw. Technol.* **2019**, *108*, 115–138. [CrossRef]
23. Sabir, F.; Palma, F.; Rasool, G.; Guéhéneuc, Y.G.; Moha, N. A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. *Softw. Pract. Exp.* **2019**, *49*, 3–39. [CrossRef]
24. Lee, D.G.; Seo, Y.S. Systematic review of bug report processing techniques to improve software management performance. *J. Inf. Process. Syst.* **2019**, *15*, 967–985.
25. Alfayez, R.; Alwehaibi, W.; Winn, R.; Venson, E.; Boehm, B. A systematic literature review of technical debt prioritization. In Proceedings of the 3rd International Conference on Technical Debt, Seoul, Republic of Korea, 28–30 June 2020; pp. 1–10.
26. Li, N.; Shepperd, M.; Guo, Y. A systematic review of unsupervised learning techniques for software defect prediction. *Inf. Softw. Technol.* **2020**, *122*, 106287. [CrossRef]
27. Cunningham, W. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* **1992**, *4*, 29–30. [CrossRef]
28. Kruchten, P.; Nord, R.; Ozkaya, I. *Managing Technical Debt: Reducing Friction in Software Development*; Addison-Wesley Professional: Boston, MA, USA, 2019.
29. Zazworka, N.; Shaw, M.A.; Shull, F.; Seaman, C. Investigating the Impact of Design Debt on Software Quality. In Proceedings of the 2nd Workshop on Managing Technical Debt, Waikiki, HI, USA, 23 May 2011; pp. 17–23. [CrossRef]
30. Nacchia, M.; Fruggiero, F.; Lambiase, A.; Bruton, K. A Systematic Mapping of the Advancing Use of Machine Learning Techniques for Predictive Maintenance in the Manufacturing Sector. *Appl. Sci.* **2021**, *11*, 2546. [CrossRef]
31. Petticrew, M.; Roberts, H. *Systematic Reviews in the Social Sciences: A Practical Guide*; John Wiley and Sons: Hoboken, NJ, USA, 2006.
32. Kitchenham, B.; Charters, S. Guidelines for performing Systematic Literature Reviews in Software Engineering. In *Technical Report*; School of Computer Science and Mathematics Keele University Keele: Staffs, UK, 2007.
33. Gwet, K.; others. Inter-rater reliability: Dependency on trait prevalence and marginal homogeneity. *Stat. Methods Inter-Rater Reliab. Assess. Ser.* **2002**, *2*, 9.
34. Ancán Bastías, O.; Díaz Arancibia, J.; Lopez Fenner, J. Exploring the Intersection Between Software Maintenance and Machine Learning—A Systematic Mapping Study. 2022. Available online: https://zenodo.org/record/7449154#.Y8SfeHZBxPY (accessed on 5 December 2022).
35. *ISO/IEC/IEEE. ISO/IEC/IEEE 14764:2022(en)*; Software Engineering—Software Life Cycle Processes—Maintenance. ISO: Geneva, Switzerland, 2022.
36. Opdyke, W.F. *Refactoring Object-Oriented Frameworks*; University of Illinois at Urbana-Champaign: Champaign, IL, USA, 1992.
37. Tripathy, P.; Naik, K. *Software Evolution and Maintenance: A Practitioner's Approach*; John Wiley & Sons: Hoboken, NJ, USA, 2014.
38. Beck, K. *Test-Driven Development: By Example*; Addison-Wesley Professional: Boston, MA, USA, 2003.
39. Chidamber, S.; Kemerer, C. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* **1994**, *20*, 476–493. [CrossRef]
40. Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D. *Refactoring: Improving the Design of Existing Code*, 1st ed.; Addison-Wesley Professional: Boston, MA, USA, 1999.
41. Mantyla, M.; Vanhanen, J.; Lassenius, C. A taxonomy and an initial empirical study of bad smells in code. In Proceedings of the International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings, Amsterdam, Netherlands, 22–26 September 2003; pp. 381–384. [CrossRef]
42. Lanza, M.; Marinescu, R. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*; Springer Science & Business Media: Berlin, Germany, 2006.
43. Fakhoury, S.; Roy, D.; Hassan, A.; Arnaoudova, V. Improving Source Code Readability: Theory and Practice. In Proceedings of the 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), Montreal, QC, Canada, 25–26 May 2019; pp. 2–12.
44. Cedrim, D.; Garcia, A.; Mongiovi, M.; Gheyi, R.; Sousa, L.; de Mello, R.; Fonseca, B.; Ribeiro, M.; Chávez, A. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In Proceedings of the 2017 11th Joint Meeting on foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 465–475.
45. Da Costa, D.A.; McIntosh, S.; Shang, W.; Kulesza, U.; Coelho, R.; Hassan, A.E. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Trans. Softw. Eng.* **2016**, *43*, 641–657. [CrossRef]
46. Mahmoudi, M.; Nadi, S.; Tsantalis, N. Are refactorings to blame? An empirical study of refactorings in merge conflicts. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019; pp. 151–162.

47. Ancán, O.; Cares, C. Are Relevant the Code Smells on Maintainability Effort? A Laboratory Experiment. In Proceedings of the 2018 IEEE International Conference on Automation/XXIII Congress of the Chilean Association of Automatic Control (ICA-ACCA), Concepcion, Chile, 17–19 October 2018; pp. 1–6.
48. Davidsen, M.K.; Krogstie, J. A longitudinal study of development and maintenance. *Inf. Softw. Technol.* **2010**, *52*, 707–719. [CrossRef]
49. Lehman, M.; Ramil, J.; Wernick, P.; Perry, D.; Turski, W. Metrics and laws of software evolution-the nineties view. In Proceedings of the Proceedings Fourth International Software Metrics Symposium, Albuquerque, NM, USA, 5–7 November 1997; pp. 20–32. [CrossRef]
50. Canfield, R.V. Cost Optimization of Periodic Preventive Maintenance. *IEEE Trans. Reliab.* **1986**, *35*, 78–81. [CrossRef]
51. Yamashita, A. How Good Are Code Smells for Evaluating Software Maintainability? Results from a Comparative Case Study. In Proceedings of the 2013 IEEE International Conference on Software Maintenance, Eindhoven, Netherlands, 22–28 September 2013; pp. 566–571. [CrossRef]
52. Yamashita, A.; Moonen, L. Do code smells reflect important maintainability aspects? In Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM), Trento, Italy, 23–28 September 2012; pp. 306–315. [CrossRef]
53. Chowdhury, S.; Holmes, R.; Zaidman, A.; Kazman, R. Revisiting the debate: Are code metrics useful for measuring maintenance effort? *Empir. Softw. Eng.* **2022**, *27*, 158. [CrossRef]
54. Rajeswari, K.; Anantharaman, R. Development of an instrument to measure stress among software professionals: Factor analytic study. In Proceedings of the 2003 SIGMIS Conference on Computer Personnel Research: Freedom in Philadelphia–Leveraging Differences and Diversity in the IT Workforce, Pennsylvania, Philadelphia, 10–12 April 2003; pp. 34–43.
55. Amin, A.; Basri, S.; Hassan, M.F.; Rehman, M. Software engineering occupational stress and knowledge sharing in the context of global software development. In Proceedings of the IEEE 2011 National Postgraduate Conference, Perak, Malaysia, 19–20 September 2011; pp. 1–4.
56. Luiz, F.C.; de Oliveira Rodrigues, B.R.; Parreiras, F.S. Machine learning techniques for code smells detection: An empirical experiment on a highly imbalanced setup. In Proceedings of the XV Brazilian Symposium on Information Systems, Aracaju, Brazil, 20–24 May 2019; pp. 1–8.
57. Izu, C.; Schulte, C.; Aggarwal, A.; Cutts, Q.; Duran, R.; Gutica, M.; Heinemann, B.; Kraemer, E.; Lonati, V.; Mirolo, C.; et al. Fostering Program Comprehension in Novice Programmers—Learning Activities and Learning Trajectories. In Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education, Aberdeen, UK, 15–17 July 2019; pp. 27–52. [CrossRef]
58. Suryanarayana, G.; Samarthyam, G.; Sharma, T. Chapter 1—Technical Debt. In *Refactoring for Software Design Smells*; Suryanarayana, G., Samarthyam, G., Sharma, T., Eds.; Morgan Kaufmann: Burlington, MA, USA, 2015; pp. 1–7. [CrossRef]
59. Verdecchia, R.; Kruchten, P.; Lago, P. Architectural Technical Debt: A Grounded Theory. In *Proceedings of the Software Architecture*; Lecture Notes in Computer Science; Jansen, A., Malavolta, I., Muccini, H., Ozkaya, I., Zimmermann, O., Eds.; Springer International Publishing: Cham, Switzerland, 2020; pp. 202–219. [CrossRef]
60. Fontana, F.A.; Ferme, V.; Zanoni, M.; Yamashita, A. Automatic Metric Thresholds Derivation for Code Smell Detection. In Proceedings of the 2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics, Florence, Italy, 17 May 2015; pp. 44–53. [CrossRef]
61. Liu, C.; Gao, C.; Xia, X.; Lo, D.; Grundy, J.; Yang, X. On the Reproducibility and Replicability of Deep Learning in Software Engineering. *ACM Trans. Softw. Eng. Methodol.* **2021**, *31*, 15:1–15:46. [CrossRef]
62. Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; Wesslén, A. *Experimentation in Software Engineering*, 1 ed.; Springer Science & Business Media: Berlin, Germany, 2012.
63. Kitchenham, B.; Pearl Brereton, O.; Budgen, D.; Turner, M.; Bailey, J.; Linkman, S. Systematic literature reviews in software engineering—A systematic literature review. *Inf. Softw. Technol.* **2009**, *51*, 7–15. [CrossRef]
64. Kapur, R.; Sodhi, B. A defect estimator for source code: Linking defect reports with programming constructs usage metrics. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2020**, *29*, 1–35. [CrossRef]
65. Zhang, T.; Lee, B. A hybrid bug triage algorithm for developer recommendation. In Proceedings of the 28th annual ACM symposium on applied computing, Coimbra, Portugal, 18–22 March 2013; pp. 1088–1094.
66. Rongrong, S.; Liping, Z.; Fengrong, Z. A Method for Identifying and Recommending Reconstructed Clones. In Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences, Wuhan, China, 12–14 January 2019; pp. 39–44.
67. Roy, D.; Fakhoury, S.; Lee, J.; Arnaoudova, V. A model to detect readability improvements in incremental changes. In Proceedings of the 28th International Conference on Program Comprehension, Seoul, Republic of Korea, 13–15 July 2020; pp. 25–36.
68. Liu, B.; Wang, T.; Zhang, X.; Fan, Q.; Yin, G.; Deng, J. A neural-network based code summarization approach by using source code and its call dependencies. In Proceedings of the 11th Asia-Pacific Symposium on Internetware, Fukuoka, Japan, 28–29 October 2019; pp. 1–10.
69. Lujan, S.; Pecorelli, F.; Palomba, F.; De Lucia, A.; Lenarduzzi, V. A preliminary study on the adequacy of static analysis warnings with respect to code smell prediction. In Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation, Virtual, 13 November 2020; pp. 1–6.

70. Panichella, A.; Alexandru, C.V.; Panichella, S.; Bacchelli, A.; Gall, H.C. A search-based training algorithm for cost-aware defect prediction. In Proceedings of the Genetic and Evolutionary Computation Conference 2016, Colorado, Denver, USA, 20–24 July 2016; pp. 1077–1084.
71. Reimann, L.; Kniesel-Wünsche, G. Achieving guidance in applied machine learning through software engineering techniques. In Proceedings of the Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming, Porto, Portugal, 23–26 March 2020; pp. 7–12.
72. Thung, F.; Le, X.B.D.; Lo, D. Active semi-supervised defect categorization. In Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, Florence, Italy, 18–19 May 2015; pp. 60–70.
73. Palomba, F.; Zaidman, A.; Oliveto, R.; De Lucia, A. An exploratory study on the relationship between changes and refactoring. In Proceedings of the 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), Buenos Aires, Argentina, 22–23 May 2017; pp. 176–185.
74. Oliveira, D.; Assunção, W.K.; Souza, L.; Oizumi, W.; Garcia, A.; Fonseca, B. Applying Machine Learning to Customized Smell Detection: A Multi-Project Study. In Proceedings of the 34th Brazilian Symposium on Software Engineering, Natal, Brazil, 21–23 October 2020; pp. 233–242.
75. Xiao, Y.; Keung, J.; Mi, Q.; Bennin, K.E. Bug localization with semantic and structural features using convolutional neural network and cascade forest. In Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018, Christchurch, New Zealand, 28–29 June 2018; pp. 101–111.
76. Mostaeen, G.; Svajlenko, J.; Roy, B.; Roy, C.K.; Schneider, K.A. Clonecognition: Machine learning based code clone validation tool. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; pp. 1105–1109.
77. Koch, P.; Schekotihin, K.; Jannach, D.; Hofer, B.; Wotawa, F.; Schmitz, T. Combining spreadsheet smells for improved fault prediction. In Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, Gothenburg, Sweden, 27 May–3 June 2018; pp. 25–28.
78. Pecorelli, F.; Palomba, F.; Di Nucci, D.; De Lucia, A. Comparing heuristic and machine learning approaches for metric-based code smell detection. In Proceedings of the 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), Montreal, QC, Canada, 25–26 May 2019; pp. 93–104.
79. Cheng, X.; Wang, H.; Hua, J.; Xu, G.; Sui, Y. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2021**, *30*, 1–33. [CrossRef]
80. Cruz, D.; Santana, A.; Figueiredo, E. Detecting bad smells with machine learning algorithms: An empirical study. In Proceedings of the 3rd International Conference on Technical Debt, Seoul, Republic of Korea, 28–30 June 2020; pp. 31–40.
81. Fang, C.; Liu, Z.; Shi, Y.; Huang, J.; Shi, Q. Functional code clone detection with syntax and semantics fusion learning. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, 18–22 July 2020; pp. 516–527.
82. Li, Y.; Wang, S.; Nguyen, T.N.; Van Nguyen, S. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. Acm Program. Lang.* **2019**, *3*, 1–30. [CrossRef]
83. Cunha, W.S.; Armijo, G.A.; de Camargo, V.V. InSet: A Tool to Identify Architecture Smells Using Machine Learning. In Proceedings of the 34th Brazilian Symposium on Software Engineering, Natal, Brazil, 21–23 October 2020; pp. 760–765.
84. Santos, G.; Figueiredo, E.; Veloso, A.; Viggiato, M.; Ziviani, N. Predicting software defects with explainable machine learning. In Proceedings of the 19th Brazilian Symposium on Software Quality, São Luís, Brazil, 1–4 December 2020; pp. 1–10.
85. Tummalapalli, S.; Kumar, L.; Murthy, N.B. Prediction of web service anti-patterns using aggregate software metrics and machine learning techniques. In Proceedings of the 13th Innovations in Software Engineering Conference on Formerly known as India Software Engineering Conference, Jabalpur, India, 27–29 February 2020; pp. 1–11.
86. Grigoriou, M.; Kontogiannis, K.; Giammaria, A.; Brealey, C. Report on evaluation experiments using different machine learning techniques for defect prediction. In Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering, Toronto, ON, Canada, 10–13 November 2020; pp. 123–132.
87. Soltanifar, B.; Akbarinasaji, S.; Caglayan, B.; Bener, A.B.; Filiz, A.; Kramer, B.M. Software analytics in practice: A defect prediction model using code smells. In Proceedings of the 20th International Database Engineering & Applications Symposium, Montreal, QC, Canada, 11–13 July 2016; pp. 148–155.
88. Gao, Y.; Yang, C. Software defect prediction based on manifold learning in subspace selection. In Proceedings of the 2016 International Conference on Intelligent Information Processing, Wuhan, China, 23–25 December 2016; pp. 1–6.
89. Murtaza, S.S.; Hamou-Lhadj, A.; Madhavji, N.H.; Gittens, M. Towards an emerging theory for the diagnosis of faulty functions in function-call traces. In Proceedings of the Fourth SEMAT Workshop on General Theory of Software Engineering, Florence, Italy, 16–24 May 2015; pp. 59–68.
90. Rantala, L. Towards better technical debt detection with NLP and machine learning methods. In Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Seoul, Republic of Korea, 5–11 October 2020; pp. 242–245.
91. Lima, R.; Souza, J.; Fonseca, B.; Teixeira, L.; Gheyi, R.; Ribeiro, M.; Garcia, A.; de Mello, R. Understanding and Detecting Harmful Code. In Proceedings of the 34th Brazilian Symposium on Software Engineering, Natal, Brazil, 21–23 October 2020; pp. 223–232.

92. Pinto, G.; Miranda, B.; Dissanayake, S.; d'Amorim, M.; Treude, C.; Bertolino, A. What is the vocabulary of flaky tests? In Proceedings of the 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29–30 June 2020; pp. 492–502.
93. Wang, X.; Wang, S.; Sun, K.; Batcheller, A.; Jajodia, S. A machine learning approach to classify security patches into vulnerability types. In Proceedings of the 2020 IEEE Conference on Communications and Network Security (CNS), Avignon, France, 29 June–1 July 2020; pp. 1–9.
94. Tian, J.; Tian, Y. A model based on program slice and deep learning for software defect prediction. In Proceedings of the IEEE 2020 29th International Conference on Computer Communications and Networks (ICCCN), Honolulu, HI, USA, 3–6 August 2020; pp. 1–6.
95. Kaur, I.; Kaur, A. A novel four-way approach designed with ensemble feature selection for code smell detection. *IEEE Access* **2021**, *9*, 8695–8707. [CrossRef]
96. Wan, J.; Li, Q.; Wang, L.; He, L.; Li, Y. A self-adaptation framework for dealing with the complexities of software changes. In Proceedings of the 2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS), Beijing, China, 24–26 November 2017; pp. 521–524.
97. Malhotra, R.; Khan, K. A study on software defect prediction using feature extraction techniques. In Proceedings of the IEEE 2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO), Noida, India, 4–5 June 2020; pp. 1139–1144.
98. Kumar, L.; Sureka, A. Aging related bug prediction using extreme learning machines. In Proceedings of the IEEE 2017 14th IEEE India Council International Conference (INDICON), Roorkee, India, 15–17 December 2017; pp. 1–6.
99. Liu, Q.; Xiang, J.; Xu, B.; Zhao, D.; Hu, W.; Wang, J. Aging-Related Bugs Prediction Via Convolutional Neural Network. In Proceedings of the IEEE 2020 7th International Conference on Dependable Systems and Their Applications (DSA), Xi'an, China, 28–29 November 2020; pp. 90–98.
100. Gupta, H.; Kumar, L.; Neti, L.B.M. An empirical framework for code smell prediction using extreme learning machine. In Proceedings of the IEEE 2019 9th Annual Information Technology, Electromechanical Engineering and Microelectronics Conference (IEMECON), Jaipur, India, 13–15 March 2019; pp. 189–195.
101. Kumar, L.; Sureka, A. Application of LSSVM and SMOTE on seven open source projects for predicting refactoring at class level. In Proceedings of the IEEE 2017 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, China, 4–8 December 2017; pp. 90–99.
102. Pritam, N.; Khari, M.; Kumar, R.; Jha, S.; Priyadarshini, I.; Abdel-Basset, M.; Long, H.V.; et al. Assessment of code smell for predicting class change proneness using machine learning. *IEEE Access* **2019**, *7*, 37414–37425. [CrossRef]
103. Maipradit, R.; Lin, B.; Nagy, C.; Bavota, G.; Lanza, M.; Hata, H.; Matsumoto, K. Automated identification of on-hold self-admitted technical debt. In Proceedings of the 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), Adelaide, SA, Australia, 28 September–2 October 2020; pp. 54–64.
104. Yue, R.; Gao, Z.; Meng, N.; Xiong, Y.; Wang, X.; Morgenthaler, J.D. Automatic clone recommendation for refactoring based on the present and the past. In Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 23–29 September 2018; pp. 115–126.
105. Karađuzović-Hadžiabdić, K.; Spahić, R. Comparison of machine learning methods for code smell detection using reduced features. In Proceedings of the IEEE 2018 3rd International Conference on Computer Science and Engineering (UBMK), Sarajevo, Bosnia and Herzegovina, 20–23 September 2018; pp. 670–672.
106. Liu, G.; Lu, Y.; Shi, K.; Chang, J.; Wei, X. Convolutional neural networks-based locating relevant buggy code files for bug reports affected by data imbalance. *IEEE Access* **2019**, *7*, 131304–131316. [CrossRef]
107. Kong, D.; Su, X.; Wu, S.; Wang, T.; Ma, P. Detect functionally equivalent code fragments via k-nearest neighbor algorithm. In Proceedings of the 2012 IEEE Fifth International Conference on Advanced Computational Intelligence (ICACI), Nanjing, China, 18–20 October 2012; pp. 94–98.
108. Di Nucci, D.; Palomba, F.; Tamburri, D.A.; Serebrenik, A.; De Lucia, A. Detecting code smells using machine learning techniques: Are we there yet? In Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (Saner), Campobasso, Italy, 20–23 March 2018; pp. 612–621.
109. Amorim, L.; Costa, E.; Antunes, N.; Fonseca, B.; Ribeiro, M. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In Proceedings of the 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), Gaithersbury, MD, USA, 2–5 November 2015; pp. 261–269.
110. Jesudoss, A.; Maneesha, S.; et al. Identification of code smell using machine learning. In Proceedings of the 2019 International Conference on Intelligent Computing and Control Systems (ICCS), Madurai, India, 15–17 May 2019; pp. 54–58.
111. Li, Y. Improving bug detection and fixing via code representation learning. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings, Seoul, Republic of Korea, 27 June–19 July 2020; pp. 137–139.
112. Sultanow, E.; Ullrich, A.; Konopik, S.; Vladova, G. Machine learning based static code analysis for software quality assurance. In Proceedings of the IEEE 2018 Thirteenth International Conference on Digital Information Management (ICDIM), Berlin, Germany, 24–26 September 2018; pp. 156–161.

113. Maiga, A.; Ali, N.; Bhattacharya, N.; Sabane, A.; Guéhéneuc, Y.G.; Aimeur, E. Smurf: A svm-based incremental anti-pattern detection approach. In Proceedings of the IEEE 2012 19th Working Conference on Reverse Engineering, Kingston, ON, Canada, 15–18 October 2012; pp. 466–475.
114. White, M.; Tufano, M.; Martinez, M.; Monperrus, M.; Poshyvanyk, D. Sorting and transforming program repair ingredients via deep learning code similarities. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER) , Hangzhou, China, 24–27 February 2019; pp. 479–490.
115. Maiga, A.; Ali, N.; Bhattacharya, N.; Sabané, A.; Guéhéneuc, Y.G.; Antoniol, G.; Aïmeur, E. Support vector machines for anti-pattern detection. In Proceedings of the 2012 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, 3–7 September 2012; pp. 278–281.
116. Aniche, M.; Maziero, E.; Durelli, R.; Durelli, V. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Trans. Softw. Eng.* **2020**, *48*, 1432–1450. [CrossRef]
117. Xia, X.; Lo, D.; Wang, X.; Yang, X.; Li, S.; Sun, J. A comparative study of supervised learning algorithms for re-opened bug prediction. In Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering, Washington, DC, USA, 5–8 March 2013; pp. 331–334.
118. Hadj-Kacem, M.; Bouassida, N. A Hybrid Approach To Detect Code Smells using Deep Learning. In Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering, Setubal, Portugal, 23–24 March 2018; pp. 137–146.
119. Zaryabi, A.; Ben Hamza, A. A neural network approach for optimal software testing and maintenance. *Neural Comput. Appl.* **2014**, *24*, 453–461. [CrossRef]
120. Jonsson, L.; Broman, D.; Magnusson, M.; Sandahl, K.; Villani, M.; Eldh, S. Automatic localization of bugs to faulty components in large scale software systems using bayesian classification. In Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), Vienna, Austria, 1–3 August 2016; pp. 423–430.
121. Mills, C.; Escobar-Avila, J.; Haiduc, S. Automatic traceability maintenance via machine learning classification. In Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 23–29 September 2018; pp. 369–380.
122. Fontana, F.A.; Zanoni, M.; Marino, A.; Mäntylä, M.V. Code smell detection: Towards a machine learning-based approach. In Proceedings of the 2013 IEEE international conference on software maintenance, Eindhoven, The Netherlands, 22–28 September 2013; pp. 396–399.
123. Jha, S.; Kumar, R.; Abdel-Basset, M.; Priyadarshini, I.; Sharma, R.; Long, H.V.; et al. Deep learning approach for software maintainability metrics prediction. *IEEE Access* **2019**, *7*, 61840–61855. [CrossRef]
124. Liu, H.; Jin, J.; Xu, Z.; Zou, Y.; Bu, Y.; Zhang, L. Deep learning based code smell detection. *IEEE Trans. Softw. Eng.* **2019**, *47*, 1811–1837. [CrossRef]
125. Liu, H.; Xu, Z.; Zou, Y. Deep learning based feature envy detection. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 385–396.
126. White, M.; Tufano, M.; Vendome, C.; Poshyvanyk, D. Deep learning code fragments for code clone detection. In Proceedings of the 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, 3–7 September 2016; pp. 87–98.
127. Das, A.K.; Yadav, S.; Dhal, S. Detecting code smells using deep learning. In Proceedings of the IEEE Region 10 Annual International Conference, Proceedings/TENCON (TENCON), Kochi, India, 17–20 October 2019; pp. 2081–2086.
128. Kukkar, A.; Mohana, R.; Kumar, Y.; Nayyar, A.; Bilal, M.; Kwak, K.S. Duplicate bug report detection and classification system based on deep learning technique. *IEEE Access* **2020**, *8*, 200749–200763. [CrossRef]
129. Kim, D.K. Finding bad code smells with neural network models. *Int. J. Electr. Comput. Eng.* **2017**, *7*, 3613. [CrossRef]
130. Shen, L.; Liu, W.; Chen, X.; Gu, Q.; Liu, X. Improving machine learning-based code smell detection via hyper-parameter optimization. In Proceedings of the 2020 27th Asia-Pacific Software Engineering Conference (APSEC), Singapore, 1–4 December 2020; pp. 276–285.
131. Sor, V.; Oü, P.; Treier, T.; Srirama, S.N. Improving statistical approach for memory leak detection using machine learning. In Proceedings of the 2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, 22–28 September 2013; pp. 544–547.
132. Jindal, R.; Malhotra, R.; Jain, A. Predicting Software Maintenance effort using neural networks. In Proceedings of the 2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)(Trends and Future Directions), Noida, India, 2–4 September 2015; pp. 1–6.
133. Sheneamer, A.; Kalita, J. Semantic clone detection using machine learning. In Proceedings of the 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA), Anaheim, CA, USA, 18–20 December 2016; pp. 1024–1028.
134. Assim, M.; Obeidat, Q.; Hammad, M. Software defects prediction using machine learning algorithms. In Proceedings of the 2020 International Conference on Data Analytics for Business and Industry: Way Towards a Sustainable Economy (ICDABI), Sakheer, Bahrain, 26–27 October 2020; pp. 1–6.
135. Díaz-Pace, J.A.; Tommasel, A.; Godoy, D. Towards anticipation of architectural smells using link prediction techniques. In Proceedings of the 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), Madrid, Spain, 23–24 September 2018; pp. 62–71.

136. Sultana, K.Z.; Anu, V.; Chong, T.Y. Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach. *J. Softw. Evol. Process.* **2021**, *33*, e2303. [CrossRef]
137. Mostaeen, G.; Roy, B.; Son, L.H.; Roy, C.K.; Schneider, K.; Svajlenko, J. A machine learning based framework for code clone validation. *J. Syst. Softw.* **2020**, *169*, 110686. [CrossRef]
138. Puranik, S.; Deshpande, P.; Chandrasekaran, K. A novel machine learning approach for bug prediction. *Procedia Comput. Sci.* **2016**, *93*, 924–930. [CrossRef]
139. Sharma, T.; Efstathiou, V.; Louridas, P.; Spinellis, D. Code smell detection by deep direct-learning and transfer-learning. *J. Syst. Softw.* **2021**, *176*, 110936. [CrossRef]
140. Qiao, L.; Li, X.; Umer, Q.; Guo, P. Deep learning based software defect prediction. *Neurocomputing* **2020**, *385*, 100–110. [CrossRef]
141. Tufano, M.; Watson, C.; Bavota, G.; Di Penta, M.; White, M.; Poshyvanyk, D. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 832–837.
142. Tufano, M.; Pantiuchina, J.; Watson, C.; Bavota, G.; Poshyvanyk, D. On learning meaningful code changes via neural machine translation. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 25–36.
143. Tufano, M.; Watson, C.; Bavota, G.; Di Penta, M.; White, M.; Poshyvanyk, D. Learning how to mutate source code from bug-fixes. In Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 29 September–4 October 2019; pp. 301–312.