Machine Learning Applied to Software Testing: A Systematic Mapping Study

Vinicius H. S. Durelli, Rafael S. Durelli, Simone S. Borges, Andre T. Endo, Marcelo M. Eler, Diego Dias, and Marcelo P. Guimarães

***Abstract*—Context: Software testing involves probing into the behavior of software systems to uncover faults. Most testing activities are complex and costly, so a practical strategy that has been adopted to circumvent these issues is to automate software testing. There has been a growing interest in applying machine learning (ML) to automate various software engineering activities, including testing-related ones.**

Objective: We set out to review the state-of-the-art of how ML has been explored to automate and streamline software testing and provide an overview of the research at the intersection of these two fields by conducting a systematic mapping study.

Method: We selected 48 primary studies. These selected studies were then categorized according study type, testing activity, and ML algorithm employed to automate the testing activity.

Results: The results highlight the most widely used ML algorithms and identify several avenues for future research. We found that ML algorithms have been used mainly for test case generation, refinement, and evaluation. Also, ML has been used to evaluate test oracle construction and to predict the cost of testing-related activities.

Conclusions: The results of our study outline the ML algorithms that are most commonly used to automate software testing activities, helping researchers to understand the current state of research concerning ML applied to software testing. We also found that there is a need for better empirical studies examining how ML algorithms have been used to automate software testing activities.

***Index Terms*—Software Testing, Machine Learning, Systematic Mapping Study**

I. Introduction

MOST early software applications belonged to the scientific computing and data processing domains [1]. Over the past few decades, however, there has been a substantial growth in the software industry, which was primarily driven by advances in technology. Consequently, software has become increasingly important in modern society. As software becomes more pervasive in everyday life, software engineers must meet stringent requirements to obtain reliable software. To keep up with all these advances, software engineering has come a long way since its inception. Yet, a number of software projects still fail to meet expectations due to a combination of factors as, for instance, cost overruns and poor quality. Evidence suggests that one of the factors that contribute the most to budget overruns is fault-detection and fault-correction: as pointed out by Westland [2], uncorrected faults become increasingly more expensive as software projects evolve. To mitigating such overheads, there has been a growing interest in software testing, which is the primary method to evaluate software under development [3].

Software testing plays a pivotal role in both achieving and evaluating the quality of software. Despite all the advances in software development methodologies and programming languages, software testing remains necessary. Basically, testing is a process whose purpose is to make sure that the software artifacts under test do what they were designed to do and also that they do not do anything unintended, thus raising the quality of these artifacts [4]. Nevertheless, testing is costly, resource-consuming, and notoriously complex: studies indicate that testing accounts for more than 50% of the total costs of software development [5]. Moreover, as any human-driven activity, testing is error-prone and creating reliable software systems is still an open problem. In hopes of coping with this problem, researchers and practitioners have been investigating more effective ways of testing software.

A practical strategy for facing some of the aforementioned issues is to automate software testing. Thus, a lot of effort has been put into automating testing activities. Artificial intelligence (AI) techniques have been successfully used to reduce the effort of carrying out many software engineering activities [6–8]. In particular, machine learning[1] (ML) [9], which is a research field at the intersection of AI, computer science, and statistics, has been applied to automate various software engineering activities [10]. It turns out that some software testing issues lend themselves to being formulated as learning problems and tackled by learning algorithms, so there has been a growing interest in capitalizing on ML to automate and streamline software testing. In addition, software systems have become increasingly complex, so some conventional testing techniques may not scale well to the complexity of these modern software systems. This ever-increasing complexity of modern software systems has rendered ML-based techniques attractive.

The remainder of this paper is organized as follows. Section II provides background on software testing and ML. Section III describes the rationale behind our research. Section IV details the mapping study we carried out. Section V discusses the results and their implications. Section VI discusses future research in the area. Section VII outlines the threats to the

Vinicius Durelli and Diego Dias are with the Department of Computer Science, Federal University of São João del Rei, Brazil, (e-mails: durelli@ufsj.edu.br; diegodias@ufsj.edu.br)

Rafael Durelli is with the Department of Computer Science, Federal University of Lavras, Brazil, (e-mail: rafael.durelli@dcc.ufla.br)

Marcelo Eler is with the Department of Mathematics and Computer Science, University of São Paulo, Brazil, (e-mail: marceloeler@usp.br)

Simone Borges and Andre Endo are with the Department of Computer Science, Federal University of Technology, Paraná, Brazil, (e-mails: simoneborges@utfpr.edu.br; andreendo@utfpr.edu.br)

Marcelo Guimarães is with the Department of Computer Science, Federal University of São Paulo, Brazil, (e-mail: marcelodepaiva@gmail.com)

[1]Machine learning is also known as predictive analytics or statistical learning.

validity of this mapping study. Section VIII presents concluding remarks.

II. Background

This section covers background on software testing and ML. The discussion is divided into two parts: the first covers the purpose of software testing, giving special emphasis to elucidating the most fundamental concepts; the second part lays out the essential background on ML.

A. Software Testing

Software testing is a quality assurance activity that consists in evaluating the system under test (SUT) by observing its execution with the aim of revealing failures [4]. A failure is detected when the SUT external behavior is different from what is expected of the SUT according to its requirements or some other description of the expected behavior [3]. Since this activity requires the execution of the SUT, it is often referred to as dynamic analysis. In contrast, there are quality assurance activities that do not require the execution of the SUT [5].

An important element of the testing activity is the test case. Essentially, a test case specifies in which conditions the SUT must be executed in hopes of finding a failure. When a test case reveals a failure it is considered successful (or effective). A test case embodies the input values needed to execute the SUT [3]. Therefore, test case inputs vary in nature, ranging from user inputs to method calls with the test case values as parameters. To evaluate the results of test cases, testers must know what output the SUT would produce for those test cases. The element that verifies the correctness of the outputs produced by the SUT is referred to as oracle. Usually, testers play the role of oracle. However, it is worth emphasizing that an oracle can be a specification or even another program.

As stated by Ammann and Offutt [3], regardless of how thoroughly planned and carried out, the main limitation of testing activities is that they are able to show only the existence of failures, not the lack thereof. Assuring that an SUT will not fail in the future requires exhaustive testing, which means the SUT has to be run against all possible inputs in all possible scenarios. Performing exhaustive testing, however, is usually impossible or impractical due to the large size of the input domain and the large amount of combinations of scenarios a SUT can be executed [11]. As a result, testers have to come up with some standard of test adequacy that allows them to decide when the SUT has been tested thoroughly enough. This has prompted the development of test adequacy criteria. Testing criteria are discussed in the next subsection.

1) Testing techniques and criteria: As alternatives for exhaustive testing, testing techniques have been proposed to help developers and testers to create a reduced and yet effective test suite [12]. Each testing technique has specific criteria to cover a particular aspect of the program and each criterion defines different test requirements that should be met by a test suite. Test requirements can be generated from different parts of the software, e.g., specification and implementation. In this context, the SUT can be instrumented so that it reports on the execution of a test suite to measure how well the test suite satisfies the test requirements [5].

Functional and structural testing are two of the most commonly used testing techniques. The functional testing technique is also known as black-box testing because it only uses the SUT specification to generate test cases. In this technique, the internal structure of the SUT is not taken into account. The two most popular functional criteria are equivalence partitioning and boundary-value analysis. Structural testing (also known as white-box testing), on the other hand, creates test cases based on the SUT implementation. Its purpose is to make sure that all structures (e.g., paths, instructions, and branches) of the SUT are exercised during execution of the test suite. Basically, structural testing criteria are usually classified as control-flow and data-flow. Control-flow criteria specify test requirements based on the execution flow of the SUT [13]. Two widely used goals related to control-flow criteria are executing all instructions or exercising all branches at least once. Data-flow criteria are based on the assumption that testers should focus on the flows of data values, i.e., variable uses and definitions [14]. One common goal of this type of criteria is to execute every definition of a data value (i.e., variable) and its associated uses at least once. This criteria is known as the all-uses criterion and takes into account only the def-use pairs that have some path from the definition to the use in which the considered variable is not redefined. This special path is called def-clear path.

Mutation testing is a less widely used technique that has been mostly used in academic settings. This technique is centered around the idea of changing the SUT in such a way that the changes made to the SUT mimic mistakes that a competent programmer would make. The elements that describe how the SUT should be changed are referred to as mutation operators, and the resulting different versions of the SUT are called mutants. Then, after mutant generation, testers have to come up with test cases for uncovering the seeded faults. When a test case causes a mutant to behave differently from the original SUT, the test case is said to kill the mutant. Mutation testing assumes that a well designed test suite can kill all mutants. Therefore, testers have to improve the test suite until it is able to kill all mutants that are not equivalent to the original SUT.

The goal when applying mutation testing is to obtain a *mutation score* of 100%: the mutation score is the percentage of non-equivalent mutants that have been killed [3]. A score of 100% indicates that the test suite is able to detect all the faults represented by the mutants. Usually, achieving a mutation score of 100% is impractical, so a threshold value can be established; representing the minimum value for the mutation score.

2) Testing phases: Software testing activities can be carried out during the whole software life-cycle to assure that failures are discovered the earlier as possible in the software development process. Testing phases, also known as test levels, are commonly used throughout software development projects: acceptance testing, system testing, integration testing, module testing, and unit testing [3]. Acceptance testing activities evaluate the SUT with respect to requirements and business processes. System testing asses the architectural design. It is worth mentioning that, in many companies, there is no

difference between system and acceptance testing.

Integration testing is carried out in the hopes of finding failures that arise from module integration. During integration testing, test cases are designed to assess whether the interfaces between modules communicate properly. Thus, integration testing assumes that modules work as expected. Module testing is carried out to evaluate modules in isolation, test cases are designed to asses how the units within the module under test interact with each other as well as their associated data structures. Unit testing has to do with exercising the smallest unit of the SUT in isolation. In object oriented programs, for instance, the smallest unit is usually either a class or a method.

Regression testing is performed throughout the life-cycle of a system, thus rather than been considered a phase or level, it can be considered a sub-phase of the aforementioned testing phases. Regression testing has to do with re-running the existing test cases whenever an element of the system is changed to ensure that the elements that were previously developed and tested still perform correctly. More specifically, regression testing is performed with the intention of checking whether recent changes have not introduced unintended consequences elsewhere in the system [3].

Since executing all test cases whenever the system is changed is costly and time-consuming, many research efforts have been investigating ways of selecting only the most effective subset of the test suite, i.e., the subset that is more likely to reveal failures. In this context, two techniques are usually applied to select test cases: test prioritization and test minimization. Test case prioritization sorts the test suite in a way that the test cases with higher priority are executed before the test cases that have a lower priority. Priorities are assigned according to different criteria, including the probability of revealing failures or the business value of the features exercised. Basically, test case minimization removes redundant test cases from the test suite, so that regression testing activities become less time-consuming and costly.

3) Test automation: Executing test cases manually is costly, time-consuming, and error-prone. Therefore, many testing frameworks and tools have been developed along the years with the intent of supporting the automated execution of test cases at different levels. Testing frameworks that support unit testing have been widely used specially because of the popularization of agile methodologies and test-focused strategies. More recently, many record-and-play or even script based frameworks and tools to perform graphical user interface (GUI) testing have become more popular among developers.

Even though automating the execution of test cases represented a significant improvement in the field, software testing activities tend to become more difficult and costly as systems become increasingly more complex. The classic answer of software engineers to reduce cost and complexity is automation. Hence, in the last years, many efforts have been carried out to come up with automated approaches for generating test inputs and stimuli to meet different test goals (e.g., branch coverage). Three different techniques to generate test cases automatically stand out in this scenario: symbolic execution, search-based, and random approaches [15].

B. Machine Learning

Essentially, problem solving using computers revolves around coming up with algorithms, which are sequences of instructions that when carried out turn the input (or set of inputs) into an output (or set of outputs). For instance, a number of algorithms for sorting have been proposed over the years. As input, these algorithms take a set of elements (e.g., numbers) and the output is an ordered list (e.g., list of numbers in ascending or descending order).

Many problems, however, do not lend themselves well to being solved by traditional algorithms. An example of problem that is hard to solve through traditional algorithms is predicting whether a test case is effective. Depending on the SUT, we know what the input is like: for instance, for a program that implements a sorting algorithm, it is a list of elements (e.g., numbers). We also know what the output should be: an ordered list of elements. Nevertheless, we do not know what list of elements is most likely to uncover faults: that is, what inputs will exercise different parts of the program's code.

There are many problems for which there is no algorithm. In effect, trying to solve these problems through traditional algorithms has led to limited success. However, in recent years, a vast amount of data concerning such problems has become available. This rise in data availability has prompted researchers and practitioners to look at solutions that involve learning from data: machine learning (ML) algorithms.

Apart from the explosion of data being captured and stored, the recent widespread adoption of ML algorithms has been largely fueled by two contributing factors: *(i)* the exponential growth of compute power, which has made it possible for computers to tackle ever-more-complex problems using ML, and *(ii)* the increasing availability of powerful ML tools [16, 17]. Due to these advances, researchers and practitioners have applied ML algorithms to an ever-expanding range of domains. Some of the domains in which ML algorithms have been used to solve problems are: weather prediction, Web search engines, natural language processing, speech recognition, computer vision, and robotics [18–20]. It is worth noting, however, that ML is not new. As pointed out by Louridas and Ebert [21], ML has been around since the 1970s, when the first ML algorithms emerged.

Let us go back to the problem of predicting the effectiveness of test cases. When facing problems of this nature, data comes into play when we need to know what an effective test case looks like. Although we might not know how to come up with an effective test case, we make an assumption that some effective test cases will be present in the collected data (e.g., set of inputs for a program whose run-time behavior was also recorded). If a ML algorithm is able to learn from the available test case data, and assuming that the program under test did not deviate much from the version used during data collection, it is possible to make predictions based on the results of the algorithm. Although the ML algorithm may not be able to identify the whole test case evaluation process, it can still detect some hidden structures and patterns in the data. In this context, the result of the algorithm is an approximation (i.e., a model). In a broad sense, ML algorithms process the available

data to build models. The resulting models embody patterns that allows us to make inferences and better characterize problems as predicting the effectiveness of test cases.

At its core, ML is simply a set of algorithms for designing models and understanding data [19, 20]. Therefore, as stated by Mohri et al. [18], ML algorithms are data-driven methods that combine computer science concepts with ideas from statistics, probability, and optimization. As emphasized by Shalev-Shwartz and Ben-David [22], the main difference in comparison with traditional statistics and other fields is that in computer science ML is centered around learning by computers, so algorithmic considerations are key.

A number of ML algorithms have been devised over the years. Essentially, these ML algorithms differ in terms of the models they use or yield. These algorithms can be broadly classified as supervised or unsupervised (a more in-depth explanation of these two categories of learning types is presented in Subsection V-E).

Software has been playing an increasingly important role in modern society. Therefore, ensuring software quality is vital. Although many factors impact the development of reliable software, testing is the primary approach for assessing and improving software quality [3]. However, despite decades of research, testing remains challenging. Recently, a strategy that has been adopted to circumvent some of the open issues is applying ML algorithms to automate software testing. We set out to provide an overview of the literature on how researchers have harnessed ML algorithms to automate software testing. We detail the rationale behind our research in the next section.

III. Problem Statement and Justification

Although applying ML to tackle software testing problems is a relatively new and emerging research trend, a number of studies have been published in the last two decades [23–28, 82, 83, 85, 86, 89, 91]. Different ML algorithms have been adapted and used to automate software testing, however, it is not clear how research in this area has evolved in terms of what has already been investigated. Despite the inherent value of examining the nature and scope of the literature in the area, few studies have attempted to provide a general overview of how ML algorithms have contributed to efforts to automate software testing activities. Noorian et al. [29], for instance, proposed a framework that can be used to classify research at the intersection of ML and software testing. Nevertheless, their classification framework is not based on a systematic review of the literature, which to some extent undermines the scope and validity of such framework.

Drawing from his personal experience, Briand [26], gives an account of the state of the art in ML applied to software testing by describing a number of applications the author was involved with over the years as well as a brief overview of other related research. Furthermore, owing to his assumption that ML has the potential to help testers cope with some long-standing software testing problems, Briand argues that more research should be performed towards synthesizing the knowledge at the intersection of these research areas. Although evidence suggests that software testing is the subject for which a substantial number of systematic literature reviews have been carried out [30], to the best of our knowledge, there are no up-to-date, comprehensive systematic reviews or systematic mappings providing an overview of published research that combines these two particular research areas.

In order to fill in such a gap, we carried out a systematic mapping study covering the existing research at the intersection of software testing with ML. According to Kitchenham et al. [31], systematic mapping is a research methodology whose goal is to survey the literature to synthesize a comprehensive overview of a given topic, identifying research gaps, and providing insight into future research directions. Using this methodology, we set out to survey the target literature to gain an overview of the state of the art in ML applied to software testing. The overarching motivation is to provide researchers and practitioners with a better understanding of which ML algorithms have already been tuned and applied to cope with software testing problems. Moreover, we investigated what research techniques are the most used in this field as well as the most prolific researchers. Given that our focus is on answering broad questions instead of analyzing particular facets of this research area, we decided to conduct a systematic mapping rather than a form of secondary study that requires a more in-depth analysis (i.e., systematic literature review).

This study provides up-to-date information on the research at the intersection of ML and software testing: outlining the most investigated topics, the strength of evidence for, and benefits and limitations of ML algorithms. We believe that the results of this systematic mapping will enable researchers to devise more effective ML-based testing approaches since these research efforts can capitalize on the best available knowledge. In addition, given that ML is not a panacea for all software testing issues, we conjecture that this study is an important step to make headway in applying ML to software testing. Essentially, the results of this study have the potential to enable practitioners and researchers to make informed decisions about which ML algorithms are best suited to their context: as stated by Kitchenham et al. [32], secondary studies as ours can be used as a starting point for further research. Another contribution of our study is the identification of research gaps, paving the way for future research in this area.

IV. Mapping Study Process

This section describes the process we followed throughout the conduction of this systematic mapping study, which was based on the guidelines for conducting secondary studies proposed by Kitchenham et al. [31] and Petersen et al. [30]. We designed this mapping study to be as inclusive as possible, so we did not use any sort of quality assessment to filter primary studies. The next subsections describe how we followed the guidelines to answer the research questions posed by this mapping study.

A. Research questions

We set out to devise research questions (RQs) that emphasize the classification of the literature in a way that is interesting to researchers and practitioners and also gives them insights

into how ML has been used to automate software testing. The scope and goal of our study can be formulated using the Goal-Question-Metric approach [33] as follows.

> **Analyze** the state of the art in ML applied to software testing
> **for the purpose of** exploration and analysis
> **with respect to** the intensity of the research in the area, trends, advantages and drawbacks of using ML to automate software testing, hindrances to using ML to automate software testing, what extent the application of ML to automate software testing has been empirically evaluated, the most-active researchers in the area
> **from the point of view of** researchers and practitioners
> **in the context of** software testing.

As pointed out by Kitchenham et al. [31], RQs must embody the goal of secondary studies. Accordingly, the goal of our study can be broken down into eight main RQs and a subquestion:

- **RQ_1:** What is the intensity of the research on ML applied to software testing?
- **RQ_2:** What types of ML algorithms have been used to cope with software testing issues?
- **RQ_3:** Which software testing activities are automated by ML algorithms?
- **RQ_4:** What trends can be observed among research studies discussing the application of ML to support software testing activities?
- **RQ_5:** What are the drawbacks and advantages of the algorithms when applied to software testing?
- **RQ_6:** What problems have been observed by researchers when applying ML algorithms to support software testing activities?
- **RQ_7:** To what extent have these ML-based approaches been evaluated empirically?
  - **$RQ_{7.1}$:** Which empirical research methods do researchers use to evaluate ML algorithms when applied to software testing?
- **RQ_8:** Which individuals are most active in this research area?

B. Search process

Although ML algorithms have been around for more than forty years, we believe that mainly over the last three decades there has been a surge of interest in applying these algorithms to solve practical problems outside the realm of AI. As mentioned, some factors that have influenced this burgeoning interest for ML are *(i)* the plummeting cost of computational power, *(ii)* development of robust and efficient algorithms that can deal with more diverse sources and types of data, and *(iii)* a wide variety of tools that can be used to support and speed up the development of ML-based applications. Based on this, at first, we chose to consider only primary studies that were published over the last few decades: from 1980 to August 2017. Afterwards, given that our results account for neither a substantial portion of 2017 nor 2018, we decided to update our systematic mapping study on grounds of expanding the collected evidence and providing up-to-date information on the research at the intersection of ML and software testing. Essentially, updating our systematic mapping study involved re-running the original searches (using the same inclusion and exclusion criteria). We filtered the updated searches by publication year: we looked for primary studies that were published from June 2017 to August 2018. The purpose of the small overlap with the first search is to allow for time lags in the indexing of studies.

We used automated searching as the main search strategy. In hopes of finding as many relevant primary studies as possible and properly answering our RQs, we examined the four digital libraries that together cover most of the literature on software engineering and a general indexing system. More precisely, we searched IEEE Digital Library[2] and ACM Digital Library[3] because these digital libraries include prime international journals and a wealth of important computing-related conferences and workshops. In addition, we searched SpringerLink[4] and ScienceDirect[5] because these two digital libraries also index a number of recognized international journals on related topics. To reduce the need to search many publisher-specific sources, we decided to take Web of Science[6] into account as well. Web of Science is a general indexing service that index papers published by many digital libraries such as ACM, Elsevier, IEEE, Springer, and Wiley. To broaden the scope of our study, during the re-run of the searches, we also searched the Society for Industrial and Applied Mathematics (SIAM)[7] and the Proceedings of the VLDB (Very Large Data Bases) Endowment (PVLDB).[8] Specifically, we searched SIAM website looking for studies that were published in in the proceedings of the SIAM International Conference on Data Mining (SDM).

When conducting automated searches in digital libraries, search keywords are vital to obtain good results, and so they have to be chosen carefully. However, given that terminology is not well established in software engineering (and most of its subareas) [31], and due to the interdisciplinarity of the subject area, we conjectured that it would be difficult to identify a reliable set of keywords to use in our search string. Thus, we derived the keywords for our search string from the RQs and based on the keywords used in the set of known papers. This set of known papers was selected through the construction of a quasi-gold standard as proposed by Zhang et al. [34]. The quasi-gold standard is created by manually searching a set of journals and conference proceedings for a given period: the quasi-gold standard results in a set of studies that are venue- and period-specific [34]. This set of papers is then used to evaluate the completeness of subsequent automated searches. The quasi-gold standard used in this mapping study is presented in Subsection IV-B1.

[2]http://ieeexplore.ieee.org/Xplore/home.jsp
[3]http://dl.acm.org
[4]http://link.springer.com
[5]http://www.sciencedirect.com
[6]https://apps.webofknowledge.com/
[7]https://archive.siam.org/meetings/sdm18/
[8]https://www.vldb.org/pvldb/

We experimented with different combinations of keywords by linking them using Boolean operators (i.e., AND and OR). Basically, the search string used in our study is twofold: the first part contains all keywords related to software testing and the second part is comprised of ML-related keywords. The two parts are linked using the Boolean operator AND. The following combination of keywords was considered the most appropriate for our study:

> (Test OR Software Testing OR Test Automation OR Test Oracle OR Metamorphic Test OR Test Data Generation OR Mutation Analysis OR Mutation Testing OR Test Generation)
>
> AND
>
> (Machine Learning OR Support Vector Machines OR Decision Trees OR Learning Based OR Active Learning OR Learning Automata OR Artificial Neural Networks OR Q-Learning OR Classification OR Grammar Induction)

We also complemented the final set of primary studies by carrying out backward snowballing for the primary studies included after the initial search (this ancillary method is briefly described in Subsection IV-B2). Due to time restrictions, we did not perform backward snowballing for the primary studies we selected during the re-runs of the searches: that is, we did not perform backward snowballing for the most recent primary studies, which were published from September 2017 onwards.

1) Quasi-gold standard: As proposed by Zhang et al. [34], we started the creation of the quasi-gold standard by identifying relevant publication venues for manual and automated search. Given that the subject area is at the intersection of two research areas, we had to include publication venues from both research areas. Next, we also selected libraries (databases) for automated search. Most target venues are indexed by the five digital libraries we chose (see Subsection IV-B), so the only publication venues we had to search manually were *Software Testing, Verification and Reliability* and the *International Journal of Intelligent Systems*, which are indexed by Wiley Online Library. Three reviewers created lists of potentially relevant publication venues in an independent fashion. These lists were then merged to form the list shown in Table I, which shows all publications venues we emphasized during the creation of the quasi-gold standard as well as the digital libraries that index these venues. As can be seen in Table I, most publication venues are software engineering and software testing related. Given that manual and automated search are time-consuming, we decided to narrow down the list of nominated venues to include only the most relevant venues.

During the creation of the quasi-gold standard, we screened all papers in the selected venues: more specifically, we read the titles and abstracts of the returned papers, applying the inclusion and exclusion criteria defined in Subsection IV-C. After searching the venues listed in Table I, we selected 21 primary studies. In addition, at this stage, two researchers were approached to list studies that could also be included in the quasi-gold standard: seven papers were nominated. Initially, the first author checked the titles and abstracts of the suggested studies. After applying the inclusion and exclusion criteria to the nominated studies, six were included. After checking the full texts of the 27 papers, 24 remained in the quasi-gold standard. It is worth mentioning that none of the six suggested studies were published in venues that are listed in Table I.

TABLE I
PUBLICATION VENUES INVESTIGATED DURING THE CREATION OF THE QUASI-GOLD STANDARD.

#	Publication Venue	Indexed By
	Software Engineering and Software Testing Related Journals	
1	Information and Software Technology	ScienceDirect
2	Journal of Systems and Software	ScienceDirect
3	Software Testing, Verification and Reliability	Wiley
	Software Engineering and Software Testing Related Conferences, Workshops, and Symposia	
4	Empirical Software Engineering and Measurement (ESEM)	IEEE/ACM
5	IEEE International Conference on Software Testing, Verification and Validation (ICST)[9]	IEEE
6	IEEE/ACM International Automation of Software Test (AST)	IEEE/ACM
7	IEEE/ACM International Conference on Automated Software Engineering (ASE)	IEEE/ACM
8	International Conference on Quality Software	IEEE
9	International Conference on Software Engineering (ICSE)	IEEE
10	International Conference on Testing Software and Systems (ICTSS)	Springer (LNCS)
11	IEEE International Symposium on Software Reliability Engineering (ISSRE)	IEEE
	ML Related Journals	
12	Engineering Applications of Artificial Intelligence	ScienceDirect
13	International Journal of Intelligent Systems	Wiley
14	Machine Learning	Springer
	ML Related Conferences, Workshops, and Symposia	
15	IEEE International Conference on Tools with Artificial Intelligence (ICTAI)	IEEE
16	Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)	IEEE/ACM

2) Backward snowballing: As mentioned, in hopes of avoiding missing potentially relevant studies, we applied backward snowballing to identify more papers that can be included in our study. One of the advantages of snowballing is that by checking the references of a set of relevant papers it is possible to find important papers even when they use different terminology; as long as the authors in the area refer to each other [35]. Essentially, snowballing is an iterative process whose initial input is a set of relevant studies, which is comprised of a subset or all of the studies selected over the course of the study selection phase. In each of the subsequent iterations, the papers referenced in the last analyzed studies are checked. The process ends when no new studies are selected. Throughout the backward snowballing process, we followed the guidelines provided by Wohlin [35].

C. Primary study selection process

This section defines the inclusion and exclusion criteria that were used throughout the conduction of this secondary study. The following criteria were used as inclusion criteria:

- I_1**:** Our initial selection relies on the filtering provided by the peer-review process, so all selected studies must have undergone peer review. Only studies published in scholarly venues such as journals, conference proceedings, and workshop proceedings were taken into account.
- I_2**:** Studies that report on ML algorithms applied to software testing.

Studies that fall into at least one of the following categories were not eligible to be selected:

- E_1**:** Studies on *(i)* approaches to testing ML algorithms, *(ii)* fault prediction techniques, *(iii)* debugging approaches, *(iv)* any sort of hardware testing approach, and *(v)* approaches based on evolutionary computation (e.g., genetic algorithms and evolutionary programming).
- E_2**:** The study describes the application of a ML algorithm, but the algorithm is not applied to automate a testing-related activity or problem.
- E_3**:** Gray literature (e.g., technical reports, working papers, and presentations) or studies published in the form of abstract or panel discussion.
- E_4**:** Often, research efforts are published at various stages of their evolution. In the context of this mapping study, duplicate versions of studies should be excluded. Only the most comprehensive or recent version of each study should be included.
- E_5**:** Peer-reviewed studies that are not published in journals, conference proceedings, or workshop proceedings (e.g., PhD thesis and patents);
- E_6**:** Studies that are not written in English.

These inclusion and exclusion criteria were applied as described in Subsection IV-C1. As discussed in Subsection IV-C1, during the application of these criteria, we went over several parts of the returned papers as, for instance, title, abstract, and keywords. Additionally, we carried out a pilot study to resolve disagreements and misunderstandings concerning these criteria.

1) Selection process: The inclusion and exclusion criteria were applied in three stages. First, papers were filtered based on title, keywords, and venue. This first step is aimed at excluding papers that are clearly irrelevant. Thus, criteria I_1, E_3, E_5, and E_6 were applied first. We realized that often a more in-depth analysis is needed to determine whether the ML-based approach described in a paper is applied to software testing, hence, the criteria I_2 and E_2 were not applied during the first round. Similarly, E_1 was not used in the first round because applying this criterion requires a more thorough examination of the papers: usually, abstract and keywords are not enough to pin down the content of a paper. During the second round, two reviewers read the abstracts of the papers selected in the first round. Throughout this round, the reviewers applied criteria I_2, E_1, and E_2. The resulting set of candidate papers was examined by two reviewers and disagreements concerning whether any borderline paper is eligible or not were resolved by discussion and, when needed, settled by a third reviewer. In the final round, the two reviewers independently filtered the candidate papers by reading them in their entirety. Criteria I_2, E_1, E_2, and E_4 were applied to select the final set of primary studies. Disagreements on selection results were discussed and addressed by two reviewers. When needed, a third reviewer was consulted.

D. Data extraction

To answer the RQs described in Subsection IV-A, we extracted from each primary study the information shown in the data extraction form presented in Appendix B. The data extraction form includes fields designed to gather general publication information, such as title and year of publication, as well as fields that were framed to reflect the RQs.

It is worth noting that, before carrying out our systematic study, we discussed the definitions of these fields, which we refer to as data items (DIs), to clarify their meanings to all data extractors. Furthermore, to make sure that all data extractors had a clear understanding of the DIs, we pilot-tested the data extraction form using the quasi-gold standard. During the pilot, we aimed at resolving disagreements and misconceptions about the DIs.

During the conduction of the original systematic mapping, two data extractors performed the data extraction on the resulting set of selected studies independently. Having extracted the information from all selected studies, the two data extractors checked all data to make sure that the extracted information is valid and clear for further analysis. The extracted data were kept in a spreadsheet. As mentioned, with the purpose of incorporating new evidence published since the original searches were completed, we repeated the extraction method for the papers returned from the re-runs of the searches. During the update, three data extractors performed data extraction on the set of selected studies, updating the original spreadsheet accordingly.

E. Data synthesis

The purpose of data synthesis is to summarize the extracted data in meaningful ways in hopes of answering the RQs defined in Subsection IV-A. More specifically, descriptive statistics and frequency analysis are used to answer the RQs. We devised classification schemes by means of keywording relevant topics addressed by some of the RQs. The resulting classifications were devised and refined as the mapping process advanced. Several facets were defined for classification purposes. For instance, to answer RQ_7 we classified the primary studies according to the nature of the research reported in them.

V. Study results

We carried out this mapping study according to the procedure described in Section IV. During the first literature search, 38 papers met the inclusion criteria. Upon updating the searches (i.e., re-running them as per the original systematic mapping protocol), 10 new papers met the inclusion criteria. So, in total, we selected 48 primary studies. A brief summary of each study is provided in Appendix C.

A. Mapping primary studies according to publication type

The primary studies were published as conference paper (21 studies), symposium paper (five studies), journal paper (11 studies), workshop paper (10 studies), or book chapter (one study). The distribution of the selected studies according to publication type is shown in Figure 1. Conferences are the most common venue in which research at the intersection of ML and software testing has been published: 21 studies were published as conference papers, which accounts for approximately 44% of the selected studies. The least common publication type is book chapter: only one study (around 2%).

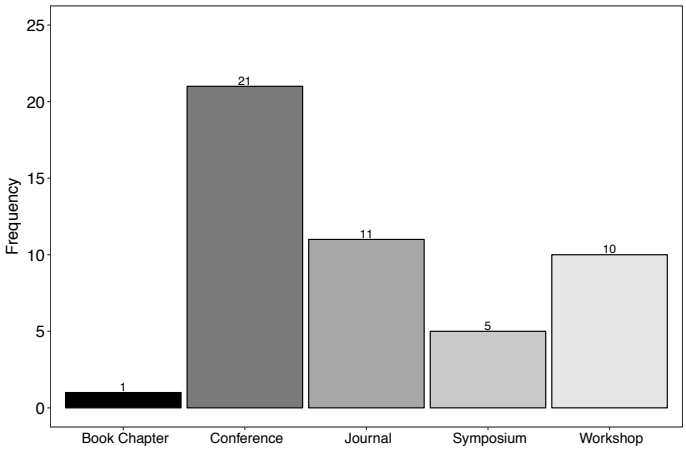

Fig. 1. Distribution of selected studies according to publication type.

Appendix D lists all venues in which the selected studies were published, presenting their types, number of selected studies that appear in each of them, and the corresponding percentage of studies published in the venue considering the total number of selected studies. The selected studies were published in 36 different venues. Most primary studies were published in the journal Software Testing, Verification and Reliability (four primary studies) and in the International Workshop in Automation of Software Testing (AST) (four primary studies). Other popular venues are the following: International Symposium on Software Testing and Analysis (ISSTA) (three primary studies), International Conference on Software Testing, Verification and Validation (ICST), International Conference on Quality Software, and International Conference on Software Maintenance (two primary studies each).

B. Mapping primary studies according to publication year

Figure 2 shows the distribution of the selected studies over the time period from 1995 to 2018. According to our results, ML algorithms have been used to automate software testing since 1995. Since then, there has been at least one study related to ML and software testing each year. The results seem to suggest that since 2010 there has been a surge of interest in applying ML algorithms to automate software testing activities. In particular, this renewed interest in ML-related approaches to software testing was more pronounced in 2011, 2013, 2016, 2017, and 2018. Our results would seem to suggest that ML-based approaches to software testing have been receiving increased attention recently.

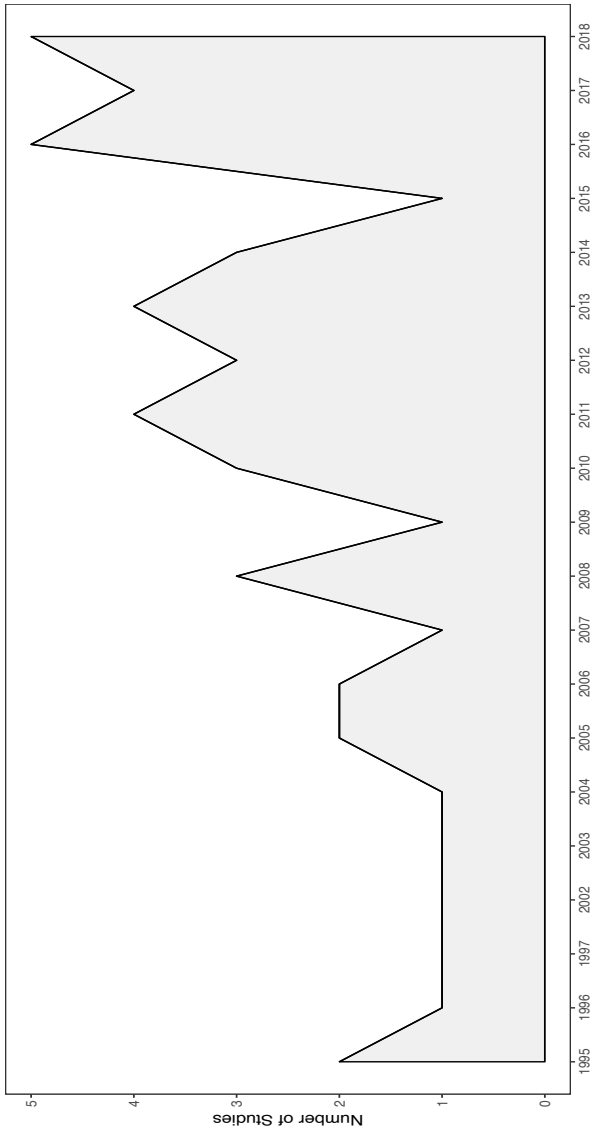

Fig. 2. Distribution of selected studies over time period.

C. Mapping primary studies according to research facet

We categorized the primary studies according to the nature of the research reported in each study. We used the classification scheme proposed by Wieringa et al. [36], which is straightforward to interpret and has been used in a number of secondary studies similar to ours. According to Wieringa et al., studies can be classified into the following classes:

- **Solution proposal:** these studies propose a novel solution to a problem. The applicability and potential benefits of the solution are borne out only by an example, proof-of-concept implementation, or sound argument. Therefore, studies that fall into this category do not present a full-blown validation of the proposed solution.
- **Validation research:** studies in this category provide preliminary empirical evidence to substantiate claims concerning the solutions or implementations thereof. Examples of research methods used to gather evidence are quasi-experiments, case studies, and prototyping. In

the context of our study, this category groups all studies that used "toy programs" to evaluate the solutions or tools. In addition, studies whose research setup is relatively less rigorous are classified as validation research.

- **Evaluation research:** studies that fall into this category go further than validation research studies by using sound research methods to evaluate novel solutions or tools in practice. Thus, these studies provide sounder evidence. It is worth noting that both validation and evaluation research can be categorized as empirical research. However, the main difference lies in the soundness and rigor of these empirical studies. Studies that employ thorough, methodologically sound research and formal methods as hypothesis testing and experiments on real-world programs are categorized as evaluation research.
- **Philosophical papers:** as described by Wieringa et al., these studies present a new way of looking at the current research in the area by describing, for instance, new taxonomies or conceptual frameworks.
- **Opinion papers:** primary studies in this category report the author's opinion about some aspects of the research area.
- **Personal experience papers:** this type of study describes the author's personal experience drawn from the participation in one or more project in academic or industrial setting. Usually, these papers contain a list of lessons learned by the author and the experience is reported without taking research methods into account. Hence, the evidence in these papers is often anecdotal in nature.

As shown in Table II, most primary studies propose novel ways to capitalize on ML to improve how software testing activities are performed. 12 primary studies were classified as solution proposals. As mentioned, studies that fall into this category present little or no evidence to back up their claims. Therefore, approximately 25% of the papers do not provide empirically grounded evidence to back up their claims. From these data we can assume that a considerable amount of the existing research at the intersection of ML and software testing is relatively weak in terms of scientific evidence.

It is worth noting that primary studies can span more than one category. For instance, papers that propose a novel approach and present a validation of the approach are quite common: as listed in the third row of Table II, 19 studies fall into the solution proposal and validation research categories. During the classification of the selected studies, we analyzed the potential contribution of the selected papers according to the following dimensions: soundness and rigor of their experimental setup and the quality of the evidence they report. Studies whose research setups were considered relatively less rigorous were classified as validation research.

Evaluation research studies provide higher-quality evidence concerning the applicability of the solutions they describe. As shown in Table II, 13 primary studies present evidence from rigorous experiments to back up their proposed solutions. these primary studies presenting solutions that are borne out by evidence were classified as solution proposal and evaluation research.

Two primary studies were classified as solution proposal and philosophical: PS17 [60] and PS19 [62]. In PS17, Noorian et al. outline a taxonomy in the form of a conceptual framework whose purpose is to help researchers and practitioners classify research efforts at the intersection of ML and software testing. Apart from the authors' experience, the resulting taxonomy draws from a small subset of the existing literature in the area. Given that this primary study presents a novel way of classifying the literature by describing a taxonomy, it was classified as solution proposal and philosophical.

TABLE II
STUDIES CLASSIFIED ACCORDING TO RESEARCH TYPE.

Research Type	Number of Studies
Solution proposal	12
Solution proposal and evaluation research	13
Solution proposal and validation research	19
Solution proposal and philosophical	2
Evaluation research	1
Philosophical	1
Opinion and personal experience	1

Traditionally, software engineering activities are carried out in value-neutral settings in which artifacts are regarded as equally important. The primary study PS19 [62] was classified as philosophical because the author advocates a shift from a value-neutral mindset to a value-based one. Based on this assumption, the author proposes a framework in which ML is used to generate value-based test data. However, no empirical evidence to support this value-based test data generation framework is provided. Therefore, since the paper contributes novel ideas instead of empirical evidence, it was also classified as solution proposal.

Only one primary study was classified as being exclusively evaluation research, PS33 [76]. In this study, Agarwal et al. report a set of experiments designed to probe into the utility of info-fuzzy networks (IFNs) and artificial neural networks (ANNs) as automated oracles in software testing, investigating the advantages and drawbacks of each approach to oracle generation.

Only one primary study was classified as philosophical: PS27 [70]. In PS27, Namin and Sridharan make a case for using Bayesian reasoning algorithms to automate software testing in a reliable and efficient fashion. Namin and Sridharan briefly explain why they believe that Bayesian reasoning algorithms are applicable to software testing, discuss some challenges that need to be properly addressed before Bayesian reasoning algorithms can be widely adopted, and outline potential solutions to these challenges.

In PS16 [59], Briand briefly summarizes his assessment of the state of the art in ML applied to software testing by mainly describing research efforts in which he was involved with over the years. While advocating that more research needs to be done in the area, Briand also gives an account of his personal experience by describing lessons learned. Therefore, PS16 was classified as opinion and personal experience.

Most primary studies describing solution proposals usually provide some sort of evidence to support their solution. We

surmise that this is the case because there has been a growth in the use of experimental methods in software engineering. According to results of our mapping study, there seems to be a growing commitment to empiricism in recent years. This commitment is embodied in improved research designs that are able to better support the claims and conclusions presented in more recent studies. Considering the set of primary studies, the first primary study with methodologically sound experimental design (i.e., evaluation research) was published in 2011. Up to that point, all published empirical evaluations were classified as validation research. The results would seem to suggest that there is room for improvement in the sense that the proposed solutions should not just be suggested and published, as is the case in most of the primary studies. We argue that more empirical evidence is needed to advance the applications of ML algorithms in software testing. It is also essential to evaluate new solutions against existing ones.

As shown in Table III most primary studies claim to have employed some empirical strategy to evaluate their proposed solutions. However, when reading the selected studies we often found that sections that were termed *"experiment"* or *"case study"* could not be strictly considered as such, as the descriptions presented in these sections lack the necessary rigor and data collection methods to be considered either an experiment or case study. Thus, the classification presented in Table II gives a more accurate overview regarding the extent to which empirical methods have been used to evaluate the application of ML algorithms to automate software testing. Moreover, we argue that this also indicates that a great deal of the efforts to evaluate and validate research in the area are somewhat poorly thought through and ill-described.

TABLE III
EMPIRICAL STRATEGIES CARRIED OUT TO EVALUATE THE SOLUTIONS DESCRIBED IN THE PRIMARY STUDIES.

Evaluation Method	Primary Studies	Total
Case study	PS3, PS7, PS10, PS12, PS18, PS24, PS28, PS31, PS37, PS44, and PS45	11
Experiment	PS1, PS2, PS4, PS5, PS6, PS8, PS9, PS11, PS14, PS15, PS20, PS21, PS22, PS23, PS25, PS26, PS29, PS30, PS33, PS34, PS35, PS36, PS38, PS39, PS40, PS41, PS42, PS43, PS46, PS47, and PS48	31
Not applicable	PS16, PS17, and PS27	3
None	PS13, PS19, and PS32	3
		48

D. Mapping primary studies according to testing facet

The classification scheme herein proposed was created iteratively as we performed data extraction and aggregation. The classification scheme was revised as we gathered more information about the current research in the area. During the first time the systematic mapping was conducted, two reviewers worked together to create the classification scheme. During the update, three reviewers double-checked and refined the classification scheme. Over the course of the following subsections we detail the categories in Table IV that contain more than two primary studies.

TABLE IV
SOFTWARE TESTING FACETS SUPPORTED BY ML IN THE PRIMARY STUDIES.

Software Testing Facet	Primary Studies	Total
Compatibility Testing	PS32	1
Conformance Testing	PS22	1
Detection of Metamorphic Relations	PS11 and PS41	2
Mutation Testing Automation	PS1, PS7, and PS35	3
Test Case Design	PS4, PS15, PS19, PS21, PS26, PS28, PS43, PS44, PS46, and PS48	10
Test Case Evaluation	PS1, PS2, PS8, PS34, PS36, PS38, PS39, and PS42	8
Test Case Prioritization	PS9, PS25, PS37, and PS40	4
Test Case Refinement	PS9, PS12, PS13, PS15, and PS23	5
Test Oracle Construction	PS5, PS6, PS20, PS22, PS24, PS29, PS30, PS31, PS33, and PS34	10
Testing Cost Estimation	PS10, PS14, and PS18	3
Test Harness	PS45	1
Testing Technique Selection	PS3	1

1) Test case design: A key aspect of software testing is designing test inputs, i.e., test cases. The goal when devising such a set of test inputs is to uncover as many failures as possible. However, as mentioned, exhaustive testing is oftentimes not feasible in practice. As a result, testers have to rely on some standard of test adequacy in order to decide when the SUT has been tested thoroughly enough. This has motivated the development of test adequacy criteria. After settling on a test adequacy criterion, testers have to decide on how to go about devising a test set that satisfies that criterion. To satisfy a given test adequacy criterion testers have to design test cases that exercise certain features of the SUT as, for instance, the source code [3]. Since this is a taxing task to be done manually, testers usually turn to automatic test data generation.

According to the results of our mapping study, in recent years, there has been a growing interest in applying ML to automate test case generation (i.e., test case design). As shown in Table IV, this is one of the most investigated topics in the area with 10 primary studies. Four of the 10 studies on this topic were published from 2017 to August 2018.

Bergadano and Gunetti [71] (PS28) devised a test case generation approach that is based on the inductive learning of programs from finite sets of input and output examples. Given a program P and a set of alternative programs P', the proposed approach yields test cases that are adequate in the sense that they are able to distinguish P from all programs in P'. As Bergadano and Gunetti emphasize, although the approach is

similar to fault-based approaches, the programs in P' are not restricted to being simple mutations of P.

In PS21 [64], Choi et al. tackled the problem of automatically generating a test suite for Android applications for which there is no existing model of the GUI. The proposed approach uses ML to learn a model of the application during testing. The learned model is then used to generate inputs that visit states of the application that have not been explored. When the application is executed using the generated inputs, the execution is observed in order to refine the model. An important feature of the approach is that it avoids restarting the application under test, which in many cases is computationally costly. Choi et al. carried out an experiment to compare how their approach compares to random testing and $\mathcal{L}^*$-based testing. The results of this experiment seem to indicate that their approach can achieve better coverage.

PS26 [69] reports on a test case design approach for web-applications. More specifically, in PS26, Sant et al. apply a ML approach to turn user session data into models of web applications. The resulting model is then randomly traversed to generate test data. In PS48 [91] a test case design approach for mobile applications is present. In this recent study, Rosenfeld et al. [91] describe an approach that leverages ML algorithms to analyze GUI elements of Android apps. After analyzing these elements, the proposed approach generates functional test cases.

Some primary studies in this category evaluate the proposed approaches using only one medium-sized program or several toy programs (e.g., PS19 and PS28). Due to the simplicity of such programs, it is unlikely that they expose the limitations of these test-data generation approaches. Hence, evaluating test generation approaches using toy programs provides limited utility. Consequently, the evidence presented in these primary studies is insufficient to draw conclusions on the effectiveness of these ML-based test-data generation. Several primary studies, however, provide a stronger case for applying ML algorithms to automate test-data generation (e.g., PS15 [58] and PS21).

2) The oracle problem: As mentioned, software testing involves exploring the behavior of the program under test so as to uncover faults. In this context, when the program is run with a certain input, it is vital to tell apart the correct from the potentially incorrect behaviors. This conundrum is referred to as the test oracle problem [37]. Without a test oracle, testers have to use domain specific information to ascertain whether the observed behavior is correct, which is in many cases impractical due to the complexity and size of present-day software systems. To make matters worse, sometimes software systems lack the documentation needed to determine the correctness of the observed behavior. To overcome these problems, researchers have sought techniques for oracle automation. However, it is worth mentioning that considering the software testing literature as a whole, test oracle automation has received significantly less attention compared to many other aspects of test automation (e.g., automated test input generation) [37]. By contrast, a significant amount of the research at the intersection of software testing with ML has been concerned with automating test oracles. In fact, 10 of the 48 selected studies report on approaches that leverage ML algorithms to construct test oracles. We believe that this is the case because ML presents novel tools to predict outcomes and, in the case of software testing, this constitutes a powerful tool for implementing test oracles.

Wang et al. [48] (PS5) examined how ML algorithms can be used to automatically generate test oracles for reactive programs without relying on explicit specifications. Essentially, their approach turns test traces into feature vectors, which are used to train the ML algorithm. The model yielded by the algorithm then acts as a test oracle.

The oracle problem appears in different contexts. Chan et al. [49] in PS6 tackled this problem in the context of mesh simplification programs. Mesh simplification programs yield three-dimensional polygonal models that are similar to the original, albeit simpler in the sense that they have fewer polygons. That is, these programs produce different graphics despite operating on the same input (i.e., the original polygonal model). As noted by Chan et al., this results in a test oracle problem. Chan et al. developed an approach that trains a classifier using a reference model of the SUT. This supervised ML approach groups test cases into two categories: passed and failure-causing. To improve the accuracy of its predictions, the approach also pipes test cases classified as passed by the ML algorithm to an analytical metamorphic testing (MT) module. Their results show that this can significantly improve the effectiveness of the proposed approach.

Jin et al. [72] (PS29) investigated how artificial neural networks (ANNs) can be used to ease the test oracle problem. Similarly, in PS30 [73], Vineeta et al. outline two ML approaches toward implementing test oracles. Specifically, the first approach builds on ANNs and the second one builds on decision trees to predict the expected outputs of the SUT. The applicability of these approaches was examined through an example using a toy program. In PS20 [63], Vanmali et al. also looked into how ANNs can be used to create a test oracle for a credit approval application.

As mentioned, PS33 [76] is the only primary study concerned exclusively with evaluating the effectiveness of two different approaches that have been used to implement test oracles, i.e., IFNs and ANNs. According to the results of this study, IFNs significantly outperform ANNs in terms of computation time while achieving almost the same fault-defection effectiveness. This comparative study also provides another insight into the characteristics of these two approaches: the experiment results indicate that the performance of the oracles are highly dependent on the amount of available test data.

3) Test case evaluation: When carrying out testing efforts, testers need to be able to assess the quality of a given test suite. However, evaluating the quality of test suites is complex because it is hard to formalize and measure what characteristics of the test cases influence quality. In the absence of precise quality indicators for test suites, the coverage of a test suite is usually used as proxy for its fault detection effectiveness.

An adequate test suite is one that implies that the SUT is free of errors if it runs correctly. However, there is no trustworthy model through which adequacy can be properly measured; hence adequacy is often quantified using proxy measures of code behavior as, for instance, branch coverage and mutation

coverage. However, as noted by Fraser and Walkinshaw [51], these program-based adequacy metrics can be impractical and may be misleading even when they are satisfied. One alternative approach that has the potential to overcome the shortcomings of program-based adequacy metrics, is the idea of behavioral coverage, which is essentially concerned with inferring a model from a system by observing its behavior (i.e., outputs) during the execution of a test suite. If one can show that the model is accurate, it follows that the test suite can be considered adequate. This approach is appealing because it eliminates the need to use proxy source-code approximations. Despite the potential of this approach, its adoption has been hindered by the complexity of inferring models. To deal with this complexity, Fraser and Walkinshaw [51] (PS8) employed ML algorithms to infer models from observed inputs and outputs. More specifically, Fraser and Walkinshaw came up with an ML-based approach to cope with the adequacy problem, the resulting approach evaluates the extend to which a test suite covers the observable program behavior.

This category also comprises a research effort whose purpose is to predict the feasibility of test cases: PS2 [45]. In the context of GUIs, test cases take the form of sequences of events that are executed in hopes of detecting faults in the application. However, test cases might be rendered infeasible if one or more events in the sequence are disabled or inaccessible. This type of test case terminates prematurely and end up wasting resources. To prune away infeasible test cases from test suites, Gove and Faytong [45] propose two approaches that capitalize on two ML algorithms: support vector machines (SVMs) and grammar induction. These two approaches to identifying infeasible test cases differ mainly in terms of their results. SVMs are a highly effective classifier, but the models produced by this algorithm, albeit accurate, is not easily interpretable by humans. In contrast to SVMs, grammar induction yields human-readable results, which allow for interpretation by the tester. Nonetheless, grammar induction is notably computationally expensive. In a more recent study, Felbinger et al. [85] (PS42) outline an approach for test evaluation that is based on inferring a model from the test suite and using the similarity between the inferred model and the SUT as a measure of test suite adequacy.

4) Test case prioritization and refinement: Previous research has proposed two main approaches to streamline regression testing: test case prioritization[10] and test case refinement.[11] Since these approaches are closely related in purpose, the primary studies that have employed ML to cope with the issue of speeding up regression testing using either test case prioritization or refinement are discussed in this subsection.

a) Test case prioritization: The time taken by regression testing is usually dictated by the size of the test suite. As regression test suites grow, they become computationally demanding to run. A large test suite might take weeks to run [38]. In such cases, testers are pressed to come up with ways to improve the effectiveness of the testing effort. In a limited-resource setting, test case prioritization can be used to mitigate some of the cost associated with regression testing. Essentially, test case prioritization involves arranging the execution of test cases in a particular order so as to optimize the rate of fault detection. The basic idea is centered around the hypothesis that most faults can be detected as early as possible by prioritizing the most relevant (i.e., higher priority) test cases.

Lenz et al. [52] (PS9) present a ML-based approach to link test results (i.e., structural coverage information and mutation score) from the application of different testing criteria. The proposed approach then groups the test results into similar functional clusters. Afterwards, information related to the existing test cases and the clusters generated in the previous step are used as a training set for a ML algorithm, which yields classifiers according to the tester's goals. As stated by Lenz et al., different classifiers can be obtained and employed to different purposes, including prioritization and refinement of test cases.

In PS25 [68], Tonella et al. reformulated the test case prioritization problem as a ML problem. Their proposed solution uses case-based reasoning (CBR) to learn an effective way to order the test cases. While sorting through the test cases, the proposed solution takes into account priority information from the user: the solution prompts the tester with pairs of test cases, and asks it to select the most important ones. Additionally, the tester input is integrated with additional information (e.g., structural coverage information) to generate an ordering of test cases. To evaluate their solution, Tonella et al. carried out an experiment using the program `space`, which contains 9,564 lines of code and 136 functions. The results of this experiment would seem to indicate that prioritization using CBR outperforms coverage-based prioritization approaches. In a more recent study, Spieker et al. [83] (PS40) introduce Retecs, which is an approach for automatically learning test case selection and prioritization. The proposed approach employs reinforcement learning to select and prioritize test cases according to their duration, previous last execution and failure history. According to Spieker et al., in comparison to similar approaches, their approach offers a more lightweight learning method that uses only one source of data, namely test case failure history.

b) Test case refinement: During the life cycle of software systems, the existing test suites need to be refined so as to better reflect changing test requirements. Often, to cope with new or changed requirements, new test cases are included to test suites. As a result, the size of test suites grows, increasing the cost of running them on the SUT (i.e., regression testing). To keep the expense of regression testing in check, sometimes the amount of test cases needs to be reduced. Given that changes to test suites must be carried out in a sensible and planned fashion, testers usually employ test case refinement algorithms to help them select an effective subset of test cases, thereby reducing testing cost. These algorithms compute an optimal subset of test cases by removing ineffective, redundant, and obsolete test cases from test suites.

Our results indicate that a considerable amount of research has been carried out to provide methodological and tool support to help testers understand the shortcomings and potential redundancies of test suites and thus being able to refine them in a cost effective fashion. Briand et al. [55] (PS12), for instance, developed a ML-based approach to help testers analyze the

[10]Test case prioritization is often termed test-suite selection.

[11]Test case refinement is also known as test-suite reduction.

strengths, weaknesses, and redundancies of black-box test specifications and test suites and iteratively improve them. This partially automated approach is based on abstracting test suite information by transforming test cases into specifications at a higher level of abstraction. More specifically, test cases are interpreted as categories and choice combinations, as defined by the black-box text specification technique category-partition. Hence, test suites are transformed into abstract test suites, which are much more amenable to use in a ML algorithm. A ML algorithm is then used to learn this abstract representation of the test suite, taking into account the relationships between input properties and output equivalence classes. As Briand et al. state, this allows the tester to better understand the strengths and the drawbacks of the test suite. In addition, this can be used when a given test suite needs to be improved but there is no test specification nor rationale (e.g., reusing open source software). Also, it is possible to use this approach to carry out a black-box testing process in which a test specification is created (e.g., using category-partition) and then test cases are generated from this specification.

Chen et al. [66] (PS23) devised an approach aimed at effective automating regression testing by means of clustering algorithms: distance measures and clustering algorithms are employed to group test cases into clusters. In this context, test cases in the same cluster are considered to have similar behavior and characteristics. The novelty of their approach is that they introduced a semi-supervised clustering method (semi-supervised K-means, SSKM) to enhance cluster selection. The limited supervision used by their clustering method is in the form of pairwise constraints: (i.e., *must-link* when two test cases are must be assigned to the same cluster or *cannot-link* when two test cases must belong to different clusters). These pairwise constraints are extracted from previous test case executions and test selection results. Chen et al. claim that they were the first to apply a semi-supervised clustering algorithm to test case selection. They believe that their study has the potential to foster developments in this area as well as help establish the basis for a greater understanding of how semi-supervised clustering algorithms can be applied to solve similar software testing related problems.

5) Test cost estimation: As mentioned, software testing accounts for a significant proportion of the total cost of software development. Therefore, testers have to come up with ways to effectively test software systems while avoiding setbacks and staying within the allotted time and budget. Some ML-based approaches were proposed to help testers to better estimate circumstances that can affect the cost of software testing efforts. Zhu et al. [53] (PS10), for instance, developed an approach to estimate the effort to execute test suites. Their approach characterizes test suites as a three-dimension vector that combines the number of test cases, execution complexity, and the experience of the tester in charge of executing the test suite. Examples are presented to show the usefulness of their proposed approach.

Cheatham et al. [57] (PS14) investigated how ML algorithms can be used to determine the factors that are important in predicting testing time. More specifically, a ML algorithm was used to learn the most important attributes that influence testing time from a database containing data on 25 software projects. The resulting classification tree was then used to predict the testing time for new software systems. Silva et al. [61] (PS18) also employed a ML-based approach towards estimating the execution effort of functional test suites. PS18 is the only primary study in this category that through experimental evidence provides a stronger case for using ML to predict the effort involved in testing-related activities. In a more recent study, Badri et al. [90] (PS47), set out to employ ML algorithms to predict test code size for object-oriented software in terms of test lines of code (TLOC), which is a key indicator of testing effort. Badri et al. used different ML algorithms to build the prediction models. To predict testing effort in terms of TLOC, Badri et al. used several metrics as input to the ML algorithms. According to their results, their metric-based approach yields accurate predictions of TLOC.

6) Mutation testing automation: From a research viewpoint, mutation testing is a mature technique [39]. This technique is centered around the idea of devising test data for uncovering artificially introduced faults. These faults are slight syntactic changes made to a given program. Each modified version of the original program is a mutant. Mutation operators dictate how mutants are created: a hallmark of the changes introduced by mutation operators is that they are analogous to mistakes that programmers make. Mutation testing is often used as a "gold standard" to compare testing approaches. Due to its effectiveness, mutation testing is widely used as an experimental research technique. In effect, some of the primary studies have experimentally used mutation testing to compare the effectiveness of their proposed approaches (e.g., PS6, PS12, PS20).

Despite the effectiveness of this technique, manually carrying out mutation testing entails a lot of human effort. Even when taking into account moderate-sized programs, mutation testing yields hundreds of mutants. Hence, mutation testing hinges on the existence of tools. In fact, mutation testing is costly and time-consuming even when automated. Recently, researchers have been trying to overcome these hurdles to the widespread adoption of this technique by using ML algorithms to expedite some steps of the process, e.g., mutant execution [44, 50].

Strug and Strug [44](PS1) put forward an approach to reduce the computational cost associated with mutation execution. In their approach a randomly selected number of mutants is run and the performance of the mutants that were not selected is assessed on the basis of their similarity to the executed mutants. To measure the similarity among mutants, they are turned into a graph representation, which is then analyzed by a ML algorithm. This approach to classifying mutants thus reduces the number of mutants that need to be executed by evaluating the quality of the test suite without running it against all generated mutants. Also with the purpose of reducing the cost of executing mutants, Jalbert and Bradbury [50] (PS7) devised a ML-based approach tailored towards predicting the effectiveness of given test suite based on a combination of source code and test suite metrics. Zhang et al. [78] in PS7 propose an approach to predicting mutation testing results without having to run the mutants. Their approach creates a model that is based on features related to mutants and tests.

Such a model is then used to predict whether a mutant can be killed by the current test suite.

E. Mapping primary studies according to ML algorithm

As mentioned, there are a plethora of ML algorithms. Most of these algorithms fall into one of two broad learning categories: supervised or unsupervised learning. Supervised learning is used when for each input variable (i.e., X) there is a corresponding output variable (i.e., Y). In such scenario, with the purpose of predicting the outputs for future inputs or better understanding the relationship between the input and the output, an algorithm is used to learn the mapping function (i.e., model) from the input to the output: $Y = \mathcal{F}(X)$ [20]. Put simply, supervised algorithms "learn" by generalizing from known examples. These algorithms find ways to produce the desired output based on the pairs of inputs and desired outputs provided by the user. In contrast, unsupervised learning is when only input data is available, so the goal is to understand the underlying relationship between the inputs. In this setting, unsupervised learning often is concerned with clustering problems, in which the goal is to determine whether the inputs fall into distinct groups [20]. According to the results of our mapping study, the vast majority of software testing issues have been formulated and tackled as supervised learning problems (Table V). Only three unsupervised learning algorithms were used to automate software testing.

It is worth noting that some ML algorithms do not fit in the classification that groups them into supervised and unsupervised. When only a subset of the input data has output data associated to it, the problem lies between supervised and unsupervised learning. This is often referred to as semi-supervised problem. In this setting, algorithms have to incorporate into the analysis the input data for which the associated output data are available as well as the input data for which there are no corresponding output data [20]. Interestingly, our results indicate that semi-supervised algorithms have been used more often than unsupervised algorithms. One of the primary studies also investigated an algorithm that can be used in a supervised or semi-supervised fashion: the Expectation-Maximization (EM) algorithm was used in PS9. The classification scheme in Table V also includes the category meta-algorithm: a meta-learning algorithm combines the predictions of several different ML algorithms in some way so as to utilize the strengths of each algorithm [16]. Only one primary study falls into this category, this primary study evaluated the performance of a meta-algorithm (i.e., AdaBoost) when applied to support software testing: PS8.

Table V does not list all primary studies, so the numbers in the table do not amount to the total number of ML algorithms investigated by the set of selected studies. Some studies describe more than one solution, thus some primary studies (e.g., PS9 and PS44) fall into more than one category. However, we believe that the taxonomy presented in Table V is useful because it gives an insight into the types of input data that need to be taken into account at the intersection of these two research areas as well as how software testing issues are more naturally formulated as ML problems. Although a useful classification scheme, there are still algorithms that do not quite fit into the five categories in Table V. For example, ANNs can be trained in either a supervised or unsupervised fashion. Therefore, we further classified the primary studies according to the similarity of the ML algorithms that they investigate. Stated more formally, we grouped the selected studies based on the function of the ML algorithms. This classification scheme is detailed in the next subsection.

TABLE V
ALGORITHMS USED IN THE PRIMARY STUDIES CLASSIFIED INTO FIVE BROAD CATEGORIES. THE MAJORITY OF THE PRIMARY STUDIES EMPLOYED SUPERVISED LEARNING ALGORITHMS.

Learning Category	Primary Studies	Total
Meta-algorithm	PS8	1
Semi-supervised	PS4, PS22, PS23, PS31, and PS38	4
Supervised	PS1, PS2 (two algorithms), PS3 (three algorithms), PS5, PS6, PS7, PS8 (four algorithms), PS9, PS10, PS11, PS12, PS13, PS18, PS22, PS24, PS30, PS32, PS33, PS34, PS35, PS36, PS37, PS39 (three algorithms), PS40, PS41, PS42, PS44, and PS47 (six algorithms)	41
Supervised and semi-supervised	PS9	2
Unsupervised	PS9 (two algorithms), PS14, and PS44	4

1) Classifying the algorithms according to their function: To classify the existing research spectrum and give a better idea of the ML algorithms that have been most used to automate software testing, we decided to further classify the algorithms in terms of their function. We studied the terminology used in the ML literature and proposed eight categories that attempt to capture the essence of the function of different ML algorithms. These eight categories are the following: ANNs, Bayesian, clustering, decision tree, ensemble algorithm, instance based, learning finite automata, and regression.

a) ANNs: This group includes studies that employ models designed to resemble biological neural networks. These models can be described as directed graphs whose nodes represent neurons and edges correspond to links between them. Each neuron performs computations that contribute to the learning process of the network. In this setting, neurons receive as input a weighted sum of the outputs of the neurons connected to them [22]. Put simply, ANNs are a parallel information-processing structure that learns and stores knowledge about their environment. This learning paradigm has been mostly used to cope with the test oracle problem (as discussed in Subsection V-D2). This is the largest category, comprising 16 primary studies (Table VI). The learning process, or training, of ANNs can be guided by a number of different algorithms. Moreover, there are many types of ANNs. Considering the 16 primary studies, five employed back-propagation, two were multilayer perceptron, one used cascade feed forward, one used feed-forward, and one was a radial basis function. Some studies do not describe the specific algorithm they used to realize their

ANNs.

TABLE VI
ALGORITHMS USED IN THE PRIMARY STUDIES CLASSIFIED ACCORDING TO THEIR FUNCTION.

Similarity/Function	Primary Studies	Total
ANN	PS8, PS18, PS20, PS29, PS30 (three algorithms), PS32, PS33 (two algorithms), PS34, PS39, PS40, PS44, PS45, and PS47	16
Bayesian	PS3 (two algorithms), PS8, and PS47	4
Clustering	PS9 (three algorithms), PS14, and PS23	5
Decision Tree	PS3, PS6, PS8 (two algorithms), PS9, PS12, PS13, PS24, PS30, PS32, PS35, PS36, PS42, and PS47	14
Ensemble Algorithm	PS8	1
Instance Based	PS1 and PS48	2
Learning Finite Automata	PS31 and PS46	2
Regression	PS3, PS8, and PS47	3

b) Bayesian: Studies in this group investigate algorithms that apply Bayes' Theorem [40]. PS3 investigated Bayesian Network and Naive Bayes. PS8 examined the performance of a Naive Bayes learner and predictor. PS47 investigated a Naive Bayes learner can be used to predict the amount of TLOC for object-oriented software.

c) Clustering: This category comprises the five primary studies that report on algorithms concerned with organizing the data into groups having as much commonality as possible. COBWEB and K-means are commonly used for clustering, these algorithms appear in two studies each. Usually, these algorithms are trained in an unsupervised fashion. An exception is PS23, which employed K-means in a semi-supervised manner. As mentioned, EM appears in one study (PS9).

d) Decision tree: Based on actual values in the training data, decision tree learners yield a model in the form of a tree structure. The resulting trees are made up of logical decisions, which can be interpreted as follows. Nodes represent decisions to be made on a given attribute, branches indicate the possible decision's choices and leaf nodes denote the result of following a sequence of decisions [16]. Given that the generated trees are essentially flowcharts, these learners lend themselves well to creating models that need to be interpreted by users. Also, another advantage of these algorithms is that they are applicable to numerical or categorical data. Because of these advantages, these algorithms have been widely used to automate software testing tasks: 14 primary studies outline approaches based on decision tree learners. Of these 14 primary studies, five do not describe the specific decision tree algorithm they implemented, seven used the C4.5 implementation, one employed the C5.0 implementation, and one approach was built on M5-based implementation. Thus, we can presume that C4.5 is the most commonly adopted implementation of this algorithm.

e) Ensemble algorithm: The algorithmic approaches[12] in this category are concerned with combining somewhat independent models (termed base learners) of multiple learners into an ensemble [17]. The key idea stems from the observation that an ensemble performs significantly better than a single model provided that there is a significant number of independent models [18]. Only one primary study employed an ensemble algorithm (i.e., AdaBoost): PS8.

f) Instance based: This category was created to group the algorithms that build up a database of example data. To find the best match or make predictions, algorithms in this group compare new data with data in a database using similarity measures, instead of performing explicit generalizations. This category has only two studies: PS1, which used the k-nearest neighbors (k-NN) algorithm, and PS48, which used the KStar (K*) algorithm to analyze GUI elements and generate functional test cases.

g) Learning finite automata: This is a topic that has been explored in various forms by lots of researchers since the early days of computer science [18]. One study looked into how this type of learning algorithm can be harnessed to automate software testing activities: PS31. In PS31 [74], Hungar et al. make a theoretical contribution towards enhancing the practicality of learning models so that these models can be used to steer testing efforts of real-world systems. In PS46 [89], Groz et al. describe a method called hW-inference, whose purpose is to infer finite state machine (FSM) models from non-resettable systems. To this end, the authors combine learning methods with conformance testing.

h) Regression: this category gathers primary studies that employ ML algorithms that are based on regression. Regression is a widely used statistical tool for modeling the relationship between variables and predicting a quantitative response. When using regression algorithms, the model is progressively refined using measures of error in the predictions made by the model. Three primary studies employed algorithms that are based on regression: PS3 (i.e., logistic regression), PS8 (i.e., additive regression), and PS47 (i.e., linear regression).

F. Classifying the studies according to the information learned

According to our results, several primary studies use ML algorithms as data analysis tools to extract information from a number of software artifacts. Many primary studies propose ML-enhanced approaches that use source code or source code related metrics as input to their learning algorithms: PS7, PS9, PS18, and PS39. Some studies take into account artifacts at a higher abstraction level as, for instance, graphs or graph-like representations. PS1 employs ML to extract information from graph kernels generated from hierarchical control flow graphs (HCFG). Similarly, PS11 [54] also takes into account graph kernels.

In this mapping, we found that the software artifacts most frequently taken into account by ML-based approaches are test cases (i.e, sets of inputs and outputs) and test suite metrics. Many studies extract information from test cases: PS3, PS7, PS8, PS9, PS10, PS12, PS14, PS15, PS18, PS25,

[12]This algorithm is ofter referred to as ensemble method.

PS28, PS29, PS30, PS34, PS40, and PS42. Although information on test cases is widely used, this sort of information is seldom considered in isolation. Often, other information sources are also used during the learning process.

Aside from source code, graphs, and test cases, ML algorithms have also been used to analyze regular expressions (i.e., PS2), features from polygonal models (i.e., PS6), formal specifications (i.e., PS13 [56]), abstract GUI models (i.e., PS21), GUI elements (i.e., PS48), false positives and false negatives yielded by oracles (i.e., PS24 [67]), Web logs (i.e., PS26), and images (i.e., PS32). An overview of the inputs (i.e., elements learned) and outputs (i.e., resulting models) of each ML-based approach is provided in Appendix C.

G. Advantages and drawbacks of using ML algorithms to automate software testing

As discussed in the previous sections, ML algorithms are appealing for automating a wide range of software testing activities. Most selected studies use ML algorithms to "synthesize" test-related artifacts (e.g., test cases) into a form that is suitable for decision-making, be it either fully-automated or with human interaction. There are a plethora of ML algorithms and they differ in terms of their function, some of which seem to be more suitable to automate certain testing activities than others. In this section we discuss the advantages and disadvantages of applying ML to solve software testing activities.

ANNs have been widely used due to their ability to solve multiple types of problems related to test oracle automation. Nevertheless, as pointed out by Anderson et al. [77], issues tend to surface when it is needed to extract the model or interpret what an ANN has learned. As Louridas and Ebert [21] remark, ML algorithms lie on a spectrum based on the easy of understanding their results. ANNs, for example, do not yield anything that can be interpreted by users: the network itself embodies all learned information. On the other end of the spectrum, some ML algorithms yield human-readable models. For example, one of the advantages of decision tree algorithms is that they produce flowchart-like tree structures that are a somewhat straightforward to interpret by humans. However, according to the results of our mapping, it turns out that the decision trees generated by these algorithms are not always intuitive. In PS24, Sprenkle et al. reported that the decision trees yielded by their approach led to decisions (i.e., in this case concerning oracle combinations) that were non-intuitive and contrary to what they were expecting.

Given that most software testing activities are challenging, we were interested in investigating to what extent software testing activities can be automated by ML algorithms as well as how practical it is to apply ML-based approaches. As pointed out by Briand [59], one of the few key limitations of ML algorithms that impact their usefulness for supporting certain testing activities is that testers have to ensure that relevant data is available. In effect, the available data must be in a form that facilitates the learning process using ML algorithms. For instance, in the approach to refining test cases proposed by Briand et al. [55], a human expert has to provide inputs in the form of categories and choices. This sort of pre-processing, which is often necessary, is one of the main disadvantages of relying on standard ML algorithms. It is worth noting, however, that this disadvantage is inherent to the use of some ML algorithms.

We conjecture that a possible obstacle to the adoption of ML algorithms is that, if these algorithms are to be used effectively, software testing efforts will have to include informed testers at all levels. These testers will have to be able to deploy and interrogate the outcomes of ML-based approaches. In many cases, this will entail having people who might not have an in-depth understanding of the code under test but know how to work knowledgeably with the strengths and weaknesses of ML algorithms. More specifically, the adoption of ML algorithms might blur the roles of testers and data scientists. Testers will not be able to truly leverage the benefits of ML algorithms without understanding the assumptions and implications of these algorithms.

H. Most prolific researchers in the area

Upon analyzing the primary studies, we also counted the number of primary studies published by each author as a way to evaluate the author's impact. We found that only five researchers have published more than one paper: Lionel Briand (i.e., PS12 and PS16) and Abraham Kandel and Mark Last (i.e., PS20 and PS33), and Neil Walkinshaw and Gordon Fraser (i.e., PS8 and PS43) Although the rate of papers published in the area seems to have increased since 2008, our results would seem to suggest that there is no research group specifically dealing with ML and software testing.

VI. Potential Research Directions

Researchers have been able to successfully harness ML algorithms to automate a number of software testing activities. While a fair amount of research has been carried out in this direction, we found that most research efforts are not methodologically sound and some issues remain unexplored. In this section, we present several potential directions for exploring the synergy between ML and software testing.

We posit that applying ML algorithms to a wider range of software testing problems could be a useful trend to follow. In particular, a number of approaches have been developed for automating mutation testing [39]. However, according to our results, not much has been done in terms of drawing on ML algorithms to expedite mutation testing. For example, ascertain whether a program and one of its mutants are equivalent is an undecidable problem. Consequently, this activity is often carried out by humans. Although this has drawn the attention of many researchers over the years, resulting in many theoretical contributions, this is still an open challenge. Along the lines of what we have previously argued, ML algorithms have the potential to outperform current approaches to detecting possible equivalent mutants as well as automating other facets of this test criterion. However, as discussed in the previous sections, there has not been much research work in this direction.

ML algorithms have become instrumental in automating activities in other fields. Although these algorithms have come a long way, software testing researchers and practitioners have

only started to tap into the potential of these algorithms. Given that automation is seen as a practical approach to coping with the increasing demand for reliable software systems, we believe that the overarching motivation for research in this area should be automating most software testing activities. However, despite future advances in ML, some human collaboration will still be needed. Thus, ML-based tools should not be designed as black-box solutions. Researchers should seek to provide solutions that allow users to easily interrogate the model behind the outputs. An effective step towards addressing this challenge would be to carry out research efforts that bridge the gap between academia and industry; we believe that this will increase the chances of coming up with solutions that can be translated into tools that are useful in industrial settings.

Finally, as pointed out by Briand [59], there is very little empirically grounded evidence supporting the cost-effectiveness of the existing applications of ML algorithms in software testing. Thus, more empirical research is needed to examine how ML models perform in software testing settings.

VII. Threats to validity

In this section we discuss the factors that can threaten the validity of our systematic mapping study. When carrying out systematic mappings, threats arise from the design, conduct, analysis, and interpretation. There are several classification schemes for different types of threats to the validity of empirical studies. In this section, we follow the classification scheme proposed by Campbell and Stanley [41] and followed by many software engineering researchers [42]. As Campbell and Stanley state, threats to validity can be categorized into four major categories: construct validity, internal validity, conclusion validity, and external validity. More specifically, considering our mapping study, the main factors might have introduced threats to the validity of our study are the following: selection of digital libraries, definition of search string, the time frame we chose, researcher bias during study selection, inaccurate data extraction, and researcher-biased data synthesis. These factors are discussed in the following subsections.

A. Construction validity

Construct validity has to do with whether the concepts being investigated are interpreted correctly and whether all relevant studies were selected. In this mapping study, the main concepts under consideration are ML algorithms and software testing activities. In hopes of ensuring the correct interpretation of these concepts, we checked their definitions and discussed them among the authors to reach a consensus. The soundness of the categorization schemes we created during data extraction hinge on how we interpreted the concepts in both areas. Due to the interdisciplinarity of the subject, we cannot rule out the possibility that some primary studies might have been misclassified. To mitigate this issue, the categorization schemes underwent several reviews by the authors.

B. Internal validity

The two main threats to the internal validity of our study are the following: *(i)* missing relevant papers and *(ii)* researcher-bias during paper selection. Although mapping studies cover a broad scope, the search is often restricted to one or more databases [31]. In this study, we also restricted our search to the most widely used databases. To mitigate the threat of failing to include relevant papers, we searched the digital libraries that are most likely to cover most of the literature on software engineering and a general indexing system. We believe that the set of primary studies accounts for most of the relevant papers on applying ML algorithms to support software testing activities. However, we cannot rule out the possibility that we may have missed several relevant studies during the conduction of the automated search in these databases. To cope with the interdisciplinarity of the subject area, we derived the keywords for our search string from the RQs and based on the keywords used in the quasi-gold standard. We also used the quasi-gold standard to assess the completeness of automated searches we performed.

Researcher bias during data extraction can potentially lead to inaccuracies in data extraction, which may affect the classification and analysis of the selected studies. In hopes of mitigating this issue we took some preventive measures. First, all DIs extracted during this mapping study were discussed among the researchers so that an agreement on the meaning of each DI was achieved previous to data extraction. Second, as mentioned, to ensure that the two researchers in charge of data extraction had a clear understanding of all DIs, they pilot-tested the data extraction form. The results of the pilot data extraction were then discussed so as to reach a consensus. Third, when needed, a third researcher went over the data extraction results to settle any disagreement.

C. Conclusion validity

Conclusion validity is mainly concerned with the degree to which the conclusions we reached are reasonable. We answered the RQs and drew conclusions based on information gleaned from the primary studies, e.g., number of papers investigating test data generation using ML algorithms. The conclusion validity issue lies in whether there is a relationship between the number of studies we found and current research trends in the subject area. We cannot rule out this threat. Another potential threat is that some primary studies might have been misclassified. To mitigate this threat, data extraction and synthesis were undertaken as a team, with two reviewers working together to reach agreements concerning the extracted data and classification thereof: as mentioned, two reviewers worked together to create the classification schemes presented in the previous sections. However, we cannot fully rule out this threat because of the qualitative nature of this systematic mapping, which makes data extraction and synthesis (i.e., classification) more susceptible to bias.

D. External validity

External validity is concerned with the extent to which the outcomes from our systematic mapping can be generalized to the intended population of interest. Therefore, one potential issue stems from assessing whether the primary studies are representative of all the relevant studies in the subject area. To mitigate this issue we followed a comprehensive search process

during which we tried to be as inclusive as possible. Although we did not take into account studies written in languages other than English, we believe that the primary studies we selected contain enough information to give researchers and practitioners an insight into how ML has been employed to support software testing.

Some primary studies do not provide all the information needed to fill out the extraction form. Thus, we often had to infer some information concerning some DIs during data synthesis. For example, some primary studies mention that their ML approach results in several advantages without elaborating on these advantages in the text. Similarly, some primary studies do not mention the drawbacks of their approaches.

VIII. Concluding remarks

ML and software testing are two broad areas of active research whose intersection has been drawing the attention of researchers. Our systematic mapping focused on making a survey of research efforts based on using ML algorithms to support software testing. We believe that our mapping study provides a valuable overview of the state of the art in ML applied to software testing, which is useful for researchers and practitioners looking to understand this research field either for the goal of leveraging or contributing to that field.

We posed the following RQs and provided answers for them through the analysis of the results of our systematic mapping study:

a) RQ_1: What is the intensity of the research on ML applied to software testing?

According to our results, ML algorithms have been applied to tackle software testing problems since 1995, but only very recently ML algorithms caught the interest of researchers and practitioners. Our results suggest that interest in applying ML algorithms to solve software testing problems has spiked in the last few years. In effect, this renewed interest in ML-related approaches to software testing started in 2010 and has been more pronounced since 2016 onwards.

b) RQ_2: What types of ML algorithms have been used to cope with software testing issues?

The vast majority of the approaches described in the primary studies automate software testing using supervised learning algorithms. According to our results, ANNs and decision trees are the most widely used algorithms.

c) RQ_3: Which software testing activities are automated by ML algorithms?

ML algorithms have been used mainly for oracle construction and for test case generation, refinement, and evaluation. Another application that seems to be gaining traction is using ML algorithms to predict the cost of testing-related activities.

d) RQ_4: What trends can be observed among research studies discussing the application of ML to support software testing activities?

A trend we observed is that the oracle problem tends to be tackled by employing either ANN- or decision tree based approaches. Interestingly, these approaches lie on opposite ends of a spectrum based on how easy it is to understand their results. ANNs do not yield anything that can be interpreted by users: the network itself embodies all learned information. On the other end of the spectrum, decision trees yield flowchart-like tree structures that are easily interpretable by humans.

e) RQ_5: What are the drawbacks and advantages of the algorithms when applied to software testing?

The main advantage of the ML-based approaches described in the primary studies is that most approaches are likely to scale very well, thus we believe they can be used to support increasingly complex testing activities. Another advantage is that most approaches require minimal human intervention. As for the drawbacks, upon analyzing our results, we found that a key limitations of ML algorithms is that testers have to ensure that relevant data is available. Moreover, the available data must be in a form that facilitates the learning process using ML algorithms. Therefore, pre-processing all the available data is an inherent disadvantage of some ML algorithms. Another drawback that has the potential to hinder widespread adoption of ML algorithms is that, if these algorithms are to be used effectively, software testing efforts will have to include informed testers at all levels. These testers will have to be able to deploy and interrogate the outcomes of ML-based approaches. In many cases, this will entail having people who might not have an in-depth understanding of the SUT but know how to work knowledgeably with the strengths and weaknesses of ML algorithms. We conjecture that the adoption of ML algorithms might in a way blur the roles of testers and data scientists. Testers will not be able to truly leverage the benefits of ML algorithms without understanding the assumptions and implications of these algorithms.

f) RQ_6: What problems have been observed by researchers when applying ML algorithms to support software testing activities? Basically, the two problems faced by researchers when trying to apply ML algorithms to solve software testing problems are *(i)* Most ML algorithms need a substantial amount of training data and *(ii)* data quality is key for ML algorithms to function as intended.

g) RQ_7: To what extent have these ML-based approaches been evaluated empirically?

We found that the body of empirical research available at the intersection of ML and software testing leaves much to be desired, specially when compared with the level of understanding and body of evidence that have been achieved in other fields. Although most selected studies present sections that were termed *"experiment"*, we found that these evaluations could not be strictly considered as such: these sections lack the necessary rigor to be considered an experiment description. More specifically, most experiments were not described well, methods of data collection and analysis were often poorly described, and few studies elaborate on validity issues.

h) RQ_8: Which individuals are most active in this research area?

Our results would seem to suggest that there is no research group specifically dealing with ML and software testing.

A. Implications for future research and practice

We believe that a number of implications can be drawn from the results of our systematic mapping study. For research, our

results indicate that, even though most selected studies provide preliminary empirical evidence to substantiate their claims, there is a clear need for more sound empirical studies. Thus, an important research challenge is to increase the quality of studies on ML algorithms applied to support software testing.

As mentioned, most solutions proposed in the selected studies scale well, and we believe that some problems currently being faced by the software testing industry could benefit from ML-based solutions. Nevertheless, few primary studies evaluated their ML-based solutions in industrial settings or using industry-grade software. We think that researchers and practitioners should collaborate to create a research agenda that will guide progress in this area. It is beyond the scope of this article to propose such an agenda, however our overview of the literature on how researchers have harnessed ML algorithms to automate software testing may provide a foundation for the creation of one.

For practitioners, the results of this systematic mapping show that many promising approaches that employ ML algorithms to automate software testing activities have been reported. It would be interesting if practitioners could participate in research projects in the future in order to provide additional experimental evidence to help validate these ML-based approaches. In this context, action research [43] could be used to guide the collaboration between researchers and practitioners.

REFERENCES

[1] P. E. Ceruzzi, *Computing: A Concise History (The MIT Press Essential Knowledge series)*. The MIT Press, 2012.

[2] J. C. Westland, "The cost of errors in software development: Evidence from industry," *Journal of Systems and Software*, vol. 62, no. 1, pp. 1–9, 2002.

[3] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge University Press, 2016.

[4] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Wiley, 2011.

[5] M. J. Harrold, "Testing: A Roadmap," in *Proceedings of the Conference on The Future of Software Engineering*. ACM, 2000, pp. 61–72.

[6] C. A. Welty and P. G. Selfridge, "Artificial Intelligence and Software Engineering: Breaking the Toy Mold," *Automated Software Engineering*, vol. 4, no. 3, pp. 255–270, 1997.

[7] M. Harman, "The Role of Artificial Intelligence in Software Engineering," in *First International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, 2012, pp. 1–6.

[8] T. Xie, "The Synergy of Human and Artificial Intelligence in Software Engineering," in *Second International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, 2013, pp. 4–6.

[9] J. Bell, *Machine Learning: Hands-On for Developers and Technical Professionals*. Wiley, 2014.

[10] D. Zhang and J. J. Tsai, "Machine Learning and Software Engineering," *Software Quality Journal*, vol. 11, no. 2, pp. 87–119, 2003.

[11] S. R. Vergilio, J. A. C. Maldonado, and M. Jino, "Infeasible paths in the context of data flow based testing criteria: identification, classification and prediction," *Journal of the Brazilian Computer Society*, vol. 12, pp. 71–86, 06 2006.

[12] B. Beizer, *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold Co., 1990.

[13] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys (CSUR)*, vol. 29, no. 4, pp. 366–427, 1997.

[14] S. Rapps and E. J. Weyuker, "Data flow analysis techniques for test data selection," in *Proceedings of the 6th International Conference on Software Engineering*. IEEE, 1982, pp. 272–278.

[15] A. Orso and G. Rothermel, "Software testing: A research travelogue (2000–2014)," in *Proceedings of the on Future of Software Engineering*, ser. FOSE 2014. ACM, 2014, pp. 117–132.

[16] B. Lantz, *Machine Learning with R*, 2nd ed. Packt Publishing, 2015.

[17] M. Bowles, *Machine Learning in Python: Essential Techniques for Predictive Analysis*. Wiley, 2015.

[18] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*. The MIT Press, 2012.

[19] P. Flach, *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press, 2012.

[20] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: with Applications in R*. Springer Texts in Statistics, 2013.

[21] P. Louridas and C. Ebert, "Machine learning," *IEEE Software*, vol. 33, no. 5, pp. 110–115, 2016.

[22] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[23] C. Anderson, A. V. Mayrhauser, and R. Mraz, "On the Use of Neural Networks to Guide Software Testing Activities," in *International Test Conference*, 1995, pp. 720–729.

[24] H. Singh, M. Conrad, and S. Sadeghipour, "Test case design based on z and the classification-tree method," in *Proceedings of the IEEE International Conference on Formal Engineering Methods*, 1997, pp. 81–90.

[25] J. Bowring, J. M. Rehg, and M. J. Harrold, "Active Learning for Automatic Classification of Software Behavior," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2004, pp. 195–205.

[26] L. C. Briand, "Novel applications of machine learning in software testing," in *The Eighth International Conference on Quality Software (QSIC)*, 2008, pp. 3–10.

[27] W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2013, pp. 623–640.

[28] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive Mutation Testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 342–353.

[29] M. Noorian, E. Bagheri, and W. Du, "Machine Learning-based Software Testing: Towards a Classification Framework," in *International Conference on Software Engineering and Knowledge Engineering (SEKE)*. Knowledge Systems Institute Graduate School, 2011, pp. 225–229.

[30] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for Conducting Systematic Mapping Studies in Software Engineering: An Update," *Information and Software Technology*, vol. 64, pp. 1–18, 2015.

[31] B. A. Kitchenham, D. Budgen, and P. Brereton, *Evidence-Based Software Engineering and Systematic Reviews*. Chapman and Hall/CRC, 2015.

[32] B. A. Kitchenham, D. Budgen, and O. P. Brereton, "Using Mapping Studies as the Basis for Further Research – A Participant-observer Case Study," *Information and Software Technology*, vol. 53, no. 6, pp. 638–651, 2011.

[33] V. R. Basili, G. Caldiera, and H. D. Rombach, "The Goal Question Metric Approach," in *Encyclopedia of Software Engineering*. Wiley, 1994.

[34] H. Zhang, M. A. Babar, and P. Tell, "Identifying Relevant Studies in Software Engineering," *Information and Software Technology*, vol. 53, no. 6, pp. 625–637, 2011.

[35] C. Wohlin, "Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE. ACM, 2014, pp. 38:1–38:10.

[36] R. Wieringa, N. Maiden, N. Mead, and C. Rolland, "Requirements Engineering Paper Classification and Evaluation Criteria: A Proposal and a Discussion," *Requirements Engineering*, vol. 11, no. 1, pp. 102–107, 2005.

[37] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[38] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing Test Cases For Regression Testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[39] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[40] J. V. Stone, *Bayes' Rule: A Tutorial Introduction to Bayesian Analysis*. Sebtel Press, 2012.

[41] D. T. Campbell and J. Stanley, *Experimental and Quasi-Experimental Designs for Research*. Wadsworth Publishing, 1963.

[42] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2012.

[43] E. T. Stringer, *Action Research*, 4th ed. SAGE, 2013.

APPENDIX A
PRIMARY STUDIES

This appendix lists all primary studies.

[44] **PS1:** J. Strug and B. Strug, "Machine learning approach in mutation testing," in *Proceedings of the International Conference on Testing Software and Systems*. Springer, 2012, pp. 200–214.

[45] **PS2:** R. Gove and J. Faytong, "Identifying Infeasible GUI Test Cases Using Support Vector Machines and Induced Grammars," in *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2011, pp. 202–211.

[46] **PS3:** D. Cotroneo, R. Pietrantuono, and S. Russo, "A Learning-based Method for Combining Testing Techniques," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 142–151.

[47] **PS4:** G. Xiao, F. Southey, R. C. Holte, and D. Wilkinson, "Software Testing by Active Learning for Commercial Games," in *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2*. AAAI Press, 2005, pp. 898–903.

[48] **PS5:** F. Wang, L. W. Yao, and J. H. Wu, "Intelligent Test Oracle Construction for Reactive Systems without Explicit Specifications," in *International Conference on Dependable, Autonomic and Secure Computing (DASC)*, 2011, pp. 89–96.

[49] **PS6:** W. K. Chan, J. C. F. Ho, and T. H. Tse, "Finding Failures from Passed Test Cases: Improving the Pattern Classification Approach to the Testing of Mesh Simplification Programs," *Software Testing, Verification and Reliability*, vol. 20, no. 2, pp. 89–120, 2010.

[50] **PS7:** K. Jalbert and J. S. Bradbury, "Predicting Mutation Score Using Source Code and Test Suite Metrics," in *Proceedings of the International Workshop on Realizing AI Synergies in Software Engineering*. IEEE, 2012, pp. 42–46.

[51] **PS8:** G. Fraser and N. Walkinshaw, "Assessing and Generating Test Sets in Terms of Behavioural Adequacy," *Software Testing, Verification and Reliability*, vol. 25, no. 8, pp. 749–780, 2015.

[52] **PS9:** A. R. Lenz, A. Pozo, and S. R. Vergilio, "Linking Software Testing Results with a Machine Learning Approach," *Engineering Applications of Artificial Intelligence*, vol. 26, no. 5-6, pp. 1631–1640, 2013.

[53] **PS10:** X. Zhu, B. Zhou, L. Hou, J. Chen, and L. Chen, "An Experience-Based Approach for Test Execution Effort Estimation," in *The International Conference for Young Computer Scientists*, 2008, pp. 1193–1198.

[54] **PS11:** U. Kanewala, J. M. Bieman, and A. Ben-Hur, "Predicting metamorphic relations for testing scientific software: A machine learning approach using graph kernels," *Software Testing, Verification and Reliability*, vol. 26, no. 3, pp. 245–269, 2016.

[55] **PS12:** L. C. Briand, Y. Labiche, Z. Bawar, and N. T. Spido, "Using machine learning to refine category-partition test specifications and test suites," *Information and Software Technology*, vol. 51, no. 11, pp. 1551–1564, 2009.

[56] **PS13:** H. Singh, M. Conrad, and S. Sadeghipour, "Test case design based on z and the classification-tree method," in *Proceedings of the IEEE International Conference on Formal Engineering Methods*, 1997, pp. 81–90.

[57] **PS14:** T. J. Cheatham, J. P. Yoo, and N. J. Wahl, "Software Testing: A Machine Learning Experiment," in *Proceedings of the ACM Annual Conference on Computer Science*. ACM, 1995, pp. 135–141.

[58] **PS15:** L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "Automatic Testing of GUI-based Applications," *Software Testing, Verification and Reliability*, vol. 24, no. 5, pp. 341–366, 2014.

[59] **PS16:** L. C. Briand, "Novel applications of machine learning in software testing," in *The Eighth International Conference on Quality Software (QSIC)*, 2008, pp. 3–10.

[60] **PS17:** M. Noorian, E. Bagheri, and W. Du, "Machine Learning-based Software Testing: Towards a Classification Framework," in *International Conference on Software Engineering and Knowledge Engineering (SEKE)*. Knowledge Systems Institute Graduate School, 2011, pp. 225–229.

[61] **PS18: D**. G. Silva, M. Jino, and B. T. d. Abreu, "Machine Learning Methods and Asymmetric Cost Function to Estimate Execution Effort of Software Testing," in *International Conference on Software Testing, Verification and Validation*, 2010, pp. 275–284.

[62] **PS19:** D. Zhang, "Machine Learning in Value-Based Software Test Data Generation," in *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, 2006, pp. 732–736.

[63] **PS20: M**. Vanmali, M. Last, and A. Kandel, "Using a Neural Network in the Software Testing Process," *International Journal of Intelligent Systems*, vol. 17, no. 1, pp. 45–62, 2002.

[64] **PS21:** W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2013, pp. 623–640.

[65] **PS22:** F. Aarts, H. Kuppens, J. Tretmans, F. Vaandrager, and S. Verwer, "Improving Active Mealy Machine Learning for Protocol Conformance Testing," *Machine Learning*, vol. 96, no. 1, pp. 189–224, 2014.

[66] **PS23:** S . Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng, "Using Semi-supervised Clustering to Improve Regression Test Selection Techniques," in *IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 1–10.

[67] **PS24:** S. Sprenkle, E. Hill, and L. Pollock, "Learning Effective Oracle Comparator Combinations for Web Applications," in *International Conference on Quality Software*, 2007, pp. 372–379.

[68] **PS25:** P. Tonella, P. Avesani, and A. Susi, "Using the Case-Based Ranking Methodology for Test Case Prioritization," in *IEEE International Conference on Software Maintenance*, 2006, pp. 123–133.

[69] **PS26:** J. Sant, A. Souter, and L. Greenwald, "An Exploration of Statistical Models for Automated Test Case Generation," in *Proceedings of the International Workshop on Dynamic Analysis*. ACM, 2005, pp. 1–7.

[70] **PS27:** A. S. Namin and M. Sridharan, "Bayesian Reasoning for Software Testing," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. ACM, 2010, pp. 349–354.

[71] **PS28:** F. Bergadano and D. Gunetti, "Testing by Means of Inductive Program Learning," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 119–145, 1996.

[72] **PS29:** H. Jin, Y. Wang, N. W. Chen, Z. J. Gou, and S. Wang, "Artificial Neural Network for Automatic Test Oracles Generation," in *International Conference on Computer Science and Software Engineering*, vol. 2, 2008, pp. 727–730.

[73] **PS30:** . Vineeta, A. Singhal, and A. Bansal, "Generation of Test Oracles Using Neural Network and Decision Tree Model," in *International Conference - Confluence The Next Generation Information Technology Summit (Confluence)*, 2014, pp. 313–318.

[74] **PS31:** H. Hungar, O. Niese, and B. Steffen, "Domain-Specific Optimization in Automata Learning," in *Proceedings of the International Conference on Computer Aided Verification*. Springer, 2003, pp. 315–327.

[75] **PS32:** N. Semenenko, M. Dumas, and T. Saar, "Browserbite: Accurate Cross-Browser Testing via Machine Learning over Image Features," in *IEEE International Conference on Software Maintenance*, 2013, pp. 528–531.

[76] **PS33:** D. Agarwal, D. E. Tamir, M. Last, and A. Kandel, "A Comparative Study of Artificial Neural Networks and Info-Fuzzy Networks as Automated Oracles in Software Testing," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 42, no. 5, pp. 1183–1193, 2012.

[77] **PS34:** C. Anderson, A. V. Mayrhauser, and R. Mraz, "On the Use of Neural Networks to Guide Software Testing Activities," in *International Test Conference*, 1995, pp. 720–729.

[78] **PS35:** J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive Mutation Testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 342–353.

[79] **PS36:** H. Felbinger, F. Wotawa, and M. Nica, "Empirical Study of Correlation Between Mutation Score and Model Inference Based Test Suite Adequacy Assessment," in *Proceedings of the International Workshop on Automation of Software Test*. ACM, 2016, pp. 43–49.

[80] **PS37:** B. Busjaeger and T. Xie, "Learning for Test Prioritization: An Industrial Case Study," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 975–980.

[81] **PS38:** J. Bowring, J. M. Rehg, and M. J. Harrold, "Active Learning for Automatic Classification of Software Behavior," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2004, pp. 195–205.

[82] **PS39:** G. Grano, T. V. Titov, S. Panichella, and H. C. Gall, "How high will it be? using machine learning models to predict branch coverage in automated testing," in *Proceedings of the Workshop on Machine Learning Techniques for Software Quality Evaluation*, 2018, pp. 19–24.

[83] **PS40:** H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 12–22.

[84] **PS41:** B. Hardin and U. Kanewala, "Using semi-supervised learning for predicting metamorphic relations," in *Proceedings of the International Workshop on Metamorphic Testing*. ACM, 2018, pp. 14–17.

[85] **PS42:** H. Felbinger, F. Wotawa, and M. Nica, "Empirical study of correlation between mutation score and model inference based test suite adequacy assessment," in *Proceedings of the International Workshop in Automation of Software Test*, 2016, pp. 43–49.

[86] **PS43:** N. Walkinshaw and G. Fraser, "Uncertainty-driven black-box test data generation," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2017, pp. 253–263.

[87] **PS43:** A. Balkan, P. Tabuada, J. V. Deshmukh, X. Jin, and J. Kapinski, "Underminer: A framework for automatically identifying nonconverging behaviors in black-box system models," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 1, pp. 20:1–20:28, 2017.

[88] **PS45:** H. Enişer and A. Sen, "Testing service oriented architectures using stateful service visualization via machine learning," in *Proceedings of the International Workshop on Automation of Software Test*. ACM, 2018, pp. 9–15.

[89] **PS46:** R. Groz, A. Simao, N. Bremond, and C. Oriat, "Revisiting ai and testing methods to infer fsm models of black-box systems," in *Proceedings of the International Workshop on Automation of Software Test*. ACM, 2018, pp. 16–19.

[90] **PS47:** M. Badri, L. Badri, W. Flageol, and F. Toure, "Investigating the accuracy of test code size prediction using use case metrics and machine learning algorithms: An empirical study," in *Proceedings of the International Conference on Machine Learning and Soft Computing*. ACM, 2017, pp. 25–33.

[91] **PS48:** A. Rosenfeld, O. Kardashov, and O. Zang, "Automation of android applications functional testing using machine learning activities classification," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*. ACM, 2018, pp. 122–132.

APPENDIX B
DATA EXTRACTION FORM

This appendix presents the data extraction form we used throughout the conduction of our systematic mapping study.

#	Data Item (DI)	Description	RQ(s)
DI_1	ID		**RQ**$_1$
DI_2	Year		**RQ**$_1$
DI_3	Venue	Name of the venue in which the primary study was published.	**RQ**$_1$
DI_4	Publication Type	Journal, conference, workshop, or book chapter.	**RQ**$_1$
DI_5	Study Type	Solution proposal, validation and evaluation research, philosophical, opinion, and personal experience papers.	**RQs**$_{1,7}$
DI_6	Title		
DI_7	Author(s) Name(s)		**RQ**$_8$
DI_8	Keywords		**RQ**$_4$
DI_9	Citation Count[13]		
DI_{10}	Proposed Approach	A summary of the proposed approach to using ML to support software testing.	
DI_{11}	ML Algorithm(s)	ML algorithms employed by the proposed approach.	**RQs**$_{2,4}$
DI_{12}	Testing Activity(ies)	Testing activity supported/automated by the proposed approach.	**RQs**$_{3,4}$
DI_{13}	Advantages / Strengths	Brief description of the advantages of the proposed approach.	**RQ**$_5$
DI_{14}	Disadvantages / Weaknesses	Brief description of the drawbacks of the proposed approach.	**RQ**$_5$
DI_{15}	Problem(s)	Problems faced when adapting the ML algorithm to support software testing activities.	**RQ**$_6$
DI_{16}	Empirically Evaluated?	A yes or no question as to whether the approach was empirically evaluated.	**RQ**$_7$
DI_{17}	Research Strategy	The research strategy used to evaluate the approach (e.g., experiment, case study).	**RQ**$_{7.1}$

APPENDIX C
SUMMARY OF THE SELECTED STUDIES

This section gives an overview of the research at the intersection of software testing and ML by providing a brief summary of each study. Since most ML algorithms are centered around learning a mapping from inputs (i.e., data points) to outputs, we tried to describe each ML-based testing approach in terms of its inputs (e.g., information that is fed into the learning model) and outputs (e.g., how the resulting mapping or model is used to make predictions about some software testing activity).

PS1: Strug and Strug [44] propose an approach to reducing the amount of mutants that needs to be executed during mutation testing. Their KNN-learner receives mutants as input, which are represented as a hierarchical graph. As output, their model can be used to make predictions on whether a given test is able to kill a certain mutant.

PS2: Gove and Faytong [45] employ SVM and grammar induction learners to eliminate infeasible GUI test cases. The learner receives as input the test case as a sequence of event

[13]Citation analysis was performed using only Google Scholar.

IDs. The results yielded by their learner can be used to make predictions on whether a given test case is infeasible or not.

PS3: Cotroneo et al. [46] aim to improve the selection of testing techniques for a given test session. The predictor is fed with historical data on features (i.e., metrics) of test sessions and their outcomes. As output, the predictor yields information on the performance of each technique for a given test session.

PS4: Xiao et al. [47] propose an approach to generating tests for commercial games. The learner receives as input samples of input/output pairs extracted from the game engine. As output, it yields a model of the game's expected behavior.

PS5: Wang et al. [48] set out to devise test oracles for reactive systems. Feature vectors generated from test traces are used as input to the proposed approach. An oracle-like model is yielded by the approach.

PS6: Chan et al. [49] present a methodology that integrates ML and metamorphic testing to build a test oracle for mesh simplification programs of 3D polygonal models. The learner receives as input features of polygonal models. Once the learner has been built, it can be used to predict whether a test case will fail or not.

PS7: Jalbert and Bradbury [50] propose an approach to improve the performance and reduce the cost of mutation testing. The learner receives as input source code's and test suite's metrics for a given unit. As output, the learner estimates the mutation score for an unknown unit of code as low, medium, or high.

PS8: Fraser and Walkinshaw [51] aim to evaluate test suites by using behavioral coverage instead of syntactic adequacy metrics as branch coverage. The learner receives as input input/output pairs observed by a test generation tool. As a result, a model aimed at predicting the behavior of the program under test is generated.

PS9: Lenz et al. [52] propose an approach to ranking the results of different testing techniques into functional clusters. The results of such ranking can be used to support the selection and priorization of test cases. The learner receives as input test cases, structural coverage information, number of mutants killed, and mutation score associated to each mutation operator. As output, the approach groups the data into clusters that can be seen as functional equivalence classes.

PS10: Zhu et al. [53] describe an approach to supporting the estimation of test execution effort. The input to the proposed approach includes the number of test cases, the complexity of executing the test cases, and the tester that will execute the test cases (testers are classified according to their experience and knowledge of the target application). As output, the approach generates a model tailored to predicting the test execution effort.

PS11: Kanewala et al. [54] propose an approach to support testing activities without the need for test oracle automation by predicting metamorphic relations for scientific software. The input to their ML-based approach comprises graph kernels obtained from control flow graphs. The results can be used to make predictions on metamorphic relations.

PS12: Briand et al. [55] introduce an approach and a tool to support the refinement of test cases in Category-Partition testing. The learner receives as input abstract test cases obtained from the test suite and a Category-Partition specification. As output, the learner predicts rules that relate pairs (e.g., category, choice).

PS13: Singh et al. [56] detail an approach to generating test cases from Z specifications for partition testing. The learner receives as input the functional specification in Z. As output, the approach produces a classification tree describing high level test cases.

PS14: Cheatham et al. [57] investigate factors that affect the prediction of testing costs, mainly testing time. The approach takes as input metrics of code complexity, programmer and tester experience, adoption of software engineering practices, and statistics on test execution. A model that lends itself well to make predictions on testing time is produced as output.

PS15: Mariani et al. [58] present a technique to generate new test cases for GUI-based applications from GUI-driven tests performed manually. The learner receives as input an initial test suite, GUI actions, and observed states obtained by the tool. As output, this GUI-based testing approach produces a behavioral model from which new test cases can be created.

PS16: Briand [59] gives an overview of the state of the art and reports on the diverse ways in which ML has been applied to automate and support software testing activities. Thus, this study does not focus on a specific ML-based approach for software testing.

PS17: Noorian et al. [60] outline a classification framework that can help to systematically review research in the ML and software testing domains. No specific ML-based approach for software testing is detailed.

PS18: Silva et al. [61] propose an approach aimed at supporting the estimation of test execution effort. Their ML-based approach takes as input metrics related to the test suite, testers, use cases, and source code. According to Silva et al. [61], the resulting model is able to predict the effort (in person-hours) required to run the test cycle.

PS19: This study does not detail a specific ML-based approach for automating software testing. Instead, Zhang [62] describes a general framework for value-based test data generation.

PS20: Vanmali et al. [63] present an approach whose main purpose is to create an oracle from a software system's test suite. The test cases of the SUT serve as input for the proposed approach. The resulting oracle-like model can be used to predict the outcomes produced by new and possibly faulty versions of the SUT.

PS21: Choi et al. [64] introduce a tool that automatically generates sequences of test inputs for Android apps. The learner receives as input sequences of actions extracted from the app's GUI. The output can be seen as a model representing the GUI of the application under test.

PS22: Aarts et al. [65] investigate how active learning can be employed to support protocol conformance testing. Sequences of input/output pairs are used as input to the proposed approach. The outcome of the approach is a Mealy machine model representing the behavior of the SUT.

PS23: Chen et al. [66] present an approach for test selection during regression testing. The learner receives as input function call profiles of test cases and pairwise constraints. As output,

the approach produces clusters of test cases (considered to have similar behaviors) from which sampling strategies can be employed to reduce the test suite for regression testing.

PS24: Sprenkle et al. [67] introduce an approach to identify the most effective HTML oracle combinations for web application testing. The learner receives as input test results for each oracle, application behavior, and expected results. According to Sprenkle et al., the output predicts the most effective oracle combination.

PS25: Tonella et al. [68] propose a test case prioritization technique that takes advantage of user knowledge. Their ML-based approach receives as input test cases, the prioritization indexes, and a sample of user-defined pairwise priority relations on test cases. As a result, the approach iteratively refines the test case ordering.

PS26: the approach proposed by Sant et al. [69] creates test cases for web applications from logged user data. Web logs from user sessions are used as inputs. The output is a Markov model from which test cases can be derived.

PS27: Namin and Sridharan [70] give an overview of Bayesian reasoning methods and discuss their applicability to software testing. No specific ML-based approach for automating software testing is discussed.

PS28: Bergadano and Gunetti [71] introduce an approach to generate test cases that distinguish a given program from a set of alternative programs. The approach is based on the inductive learning of programs from a finite set of input/output examples. More specifically, their approach receives as input the program, the set of alternative programs, and input/output examples. As output, the approach induces an alternative program that is consistent (equivalent) with the original, taking into account the provided input/output examples.

PS29: Jin et al. [72] tackle the automated creation of test oracles by employing ANNs. The input to the their ANN-based approach is test cases. As output, their approach is able to predict the expected behavior of new test cases.

PS30: Vineeta et al. [73] set out to generate test oracles. The learner receives as input test cases and, as a result, it can be used to predict the expected behavior of new test cases.

PS31: Hungar et al. [74] investigate automata learning to support the testing of complex reactive systems, mainly from the telecommunication domain. The approach receives as input stimuli and responses from the SUT. The resulting learning model can be used to predict I/O automata.

PS32: Semenenko et al. [75] report an experience on building a tool for cross-browser compatibility testing. The ML-based approach receives as input image features of regions of interest in the web pages. The resulting model can be used to point out potential incompatibilities among multiple browser-platform combinations.

PS33: Agarwal et al. [76] compare IFNs and ANNs to build automated test oracles. Test cases are fed into the learning model. As output, the model is then used to determine whether a new input is correct or not.

PS34: Anderson et al. [77] present empirical results on the adoption of ANNs to prune test suites while keeping their effectiveness. The learner receives as input test case metrics like length, command frequency, and parameter use frequency. As output, their ANN predicts the fault detection capabilities of a given test case.

PS35: Zhang et al. [78] propose an approach to reduce the execution cost of mutation testing. The inputs to their approach are source code's and test suite's features related to execution, infection, and propagation of a given mutant. The resulting model predicts whether a mutant is killed by some test case.

PS36: Felbinger et al. [79] present an approach to evaluating a test suite adequacy with respect to an inferred model. The learner receives as input the test cases as input/output pairs. As output, the approach builds a state machine model that can be used to calculate the similarity with the SUT.

PS37: Busjaeger and Xie [80] introduce a novel approach for test prioritization in industrial environments. Inputs to the approach are code coverage information, text path similarity, text content similarity, failure history, and test age. As output, the resulting model yields an effective prioritization of the test cases.

PS38: Bowring et al. [81] investigate the use of Markov models to evaluate and augment test suites for future versions of the SUT. The learner receives as input test cases, event-transition profiles, and its behavior label. Markov models that are clustered into effective behavior classifiers are produced as output.

PS39: Grano et al. [82] conducted a preliminary study to look at how ML models can be used to predict the branch coverage achieved by test data generation tools. The learner receives as input code metrics and, as output, it predicts branch coverage achieved by test data generator tools.

PS40: Spieker et al. [83] propose a ML-based approach for test case selection and prioritization. The approach receives as input information on test cases: the test case duration, the last execution, and a failure history. As output, their approach yield a model that tailored to prioritize error-prone test cases under guidance of a reward function and by taking into account previous executions.

PS41: Hardin and Kanewala [84] propose a semi-supervised ML approach to detecting metamorphic relations that are applicable to a given code base. The learner receives as input paths through methods control flow graph. The resulting model can be used to predict metamorphic relations.

PS42: Felbinger et al. [85] propose a method for evaluting the effectiveness of test suites that is based on inferring models from the test suites. The input to their approach is the test suite being evaluating and information concerning the current state and previous output of the SUT. As a result, the approach yields an inferred model from the test suite.

PS43: Walkinshaw and Fraser [86] apply a technique known in ML parlance as "query strategy framework" that entails inferring a behavioral model of the SUT and selecting test cases which the inferred model is "least certain" about. It is assumed that running these tests on the SUT will further help to inform the learner, That is, the underlying assumption is that by providing information that the learner has not processed yet (i.e., test cases that are not present in the training set) this uncertainty-driven approach is able to form a effective basis for test case selection. The learner receives as input the JSON

specification of the SUT's interface. As output, it produces new test inputs.

PS44: Balkan et al. [87] introduce a framework named Underminer that aims to find parameter values and inputs for black box system models that lead to undesirable behaviors. The learner receives as input an existing system model or design behaviors labelled as convergent or nonconvergent. As output, it predicts inputs and parameter values that cause outputs related to undesirable or nonconvergent behavior.

PS45: Enişer and Sen [88] employ ML to support the virtualization of stateful services. The learner receives as input traces with request-response pairs. As output, it predicts the behavior of the virtual service.

PS46: Groz et al. [89] propose a method called hW-inference to infer FSM models from non-resettable systems; to do so, they combine learning-based methods with conformance testing. The learner receives as input the known inputs and outputs of the system. An FSM model is generated as result.

PS47: Badri et al. [90] investigate the early prediction of test lines of code for object-oriented software using use case metrics. The learner receives as input use case metrics and use case points. As output, it predicts the amount of test lines of code.

PS48: Rosenfeld et al. [91] describe a approach for the automation of functional testing in mobile software by leveraging ML techniques and reusing generic test scenarios. The proposed approach receives as input elements of the GUI of Android apps. As output, it generates functional test cases.

APPENDIX D
DISTRIBUTION OF THE SELECTED STUDIES ACCORDING TO THEIR PUBLICATION SOURCES

#	Venue	Type	No.	%
1	ACM Annual Conference on Computer Science	Conference	1	2.08
2	ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)	Conference	1	2.08
3	ACM Transactions on Software Engineering and Methodology (TOSEM)	Journal	1	2.08
4	Conference on Software Engineering and Knowledge Engineering (SEKE)	Conference	1	2.08
5	Engineering Applications of Artificial Intelligence	Journal	1	2.08
6	IEEE Transactions on Systems, Man, and Cybernetics: Systems	Journal	1	2.08
7	Information and Software Technology (IST)	Journal	1	2.08
8	International Conference - Confluence The Next Generation Information Technology Summit	Conference	1	2.08
9	International Conference for Young Computer Scientists	Conference	1	2.08
10	International Conference on Computer Science and Software Engineering	Conference	1	2.08
11	International Conference on Computer-Aided Verification	Conference	1	2.08
12	International Conference on Dependable, Autonomic and Secure Computing (DASC)	Conference	1	2.08
13	International Conference on Formal Engineering Methods	Conference	1	2.08
14	International Conference on Quality Software	Conference	2	4.17
15	International Conference on Software Engineering (ICSE)	Conference	1	2.08
16	International Conference on Software Maintenance	Conference	2	4.17
17	International Conference on Software Testing, Verification and Validation (ICST)	Conference	2	4.17
18	International Conference on Testing Software and Systems (ICTSS)	Conference	1	2.08
19	International Conference on Tools with Artificial Intelligence (ICTAI)	Conference	1	2.08
20	International Journal of Intelligent Systems	Journal	1	2.08
21	International Symposium on Foundations of Software Engineering	Symposium	1	2.08
22	International Symposium on Software Testing and Analysis (ISSTA)	Symposium	3	6.25
23	International Test Conference	Conference	1	2.08
24	International Workshop in Automation of Software Testing (AST)	Workshop	4	8.33
25	International Workshop on Dynamic Analysis	Workshop	1	2.08
26	International Workshop on Realizing AI Synergies in Software Engineering	Workshop	1	2.08
27	Machine Learning	Journal	1	2.08
28	National Conference on Artificial intelligence (AAAI)	Conference	1	2.08
29	Software Testing, Verification and Reliability	Journal	4	8.33
30	Software Testing, Verification and Validation Workshops (ICSTW)	Workshop	1	2.08
31	Workshop on Future of Software Engineering Research	Workshop	1	2.08
32	Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)	Workshop	1	2.08
33	International Workshop on Metamorphic Testing	Workshop	1	2.08
34	ACM Transactions on Embedded Computing Systems	Journal	1	2.08
35	International Conference on Machine Learning and Soft Computing	Conference	1	2.08
36	International Conference on Mobile Software Engineering and Systems	Conference	1	2.08

Continued on next column

Continued from previous column

Concluded