# Usage of Machine Learning in Software Testing

**Chapter** · January 2020

DOI: 10.1007/978-3-030-38006-9_3

Subhankar Mishra
National Institute of Science Education and Research

# Usage Of Machine Learning In Software Testing

Sumit Mahapatra[*]and Subhankar Mishra [†‡]

July 11, 2020

**Abstract**

Finding, pinpointing and correcting bugs in the software takes a lot of effort for software developers. Traditional testing requires humans to search and analyze data. Humans being prone to poor assumptions and hence skewed results leads to overlooked bugs. Machine learning helps systems to learn and apply the learned knowledge in the future which helps software testers with more accurate knowledge. Capability of several advanced machine learning techniques such as deep learning in a number of software engineering tasks such as code completion, defect prediction, bug localization, clone detection, code search and learning API sequences. A lot of approaches have been proposed by the researchers over the years at modifying programs automatically. Repairing programs by generate-and-validate techniques gain a lot by producing acceptable patches to the programmers and not overfitting through learning from past history and generating patches from correct code via probabilistic model. These approaches given the right environments play significant role in reducing the effort and time consumption as well as cost of the bug fixing for the software developers. In this chapter, various machine learning algorithms and their impact in software testing and bug fixing are explored. The last chapter concludes with the future of machine learning and predictive analysis and how they might be used for addressing the challenge of reacting faster to dynamic expectations of customers and their needs.

Machine Learning, Software Testing, Defect prediction, vulnerability analysis, Automated software testing

## 1 Introduction

In today's generation, computing devices play a pivotal role in modern human life among which especially software along with its variety has made the entire species of humans dependent on it. Although there are some standard practices that are followed in the software development; however they are build using a wild variety technologies. The wide array of base technologies come with their own positives and negatives; however also raises a concern about the security of the systems building using the software. This is in fact one of the paramount concerns in this vast industry of software engineering.

Software vulnerabilities are a major threat. Their varying degrees along with their impact areas, complexity and attack-surface are very well stated by the authors in [1]. More examples are described in this chapter to cite the significance of this issue. One of the major example of such vulnerabilities is the security and privacy issue of Internet users that is threatened by the browser extensions; Adobe Flash Player [2] and Oracle Java [3].

Out of the numerous vulnerabilities that are reported every year, the aforesaid examples are only a few. The gravity of these said security issues, these have been a area which has

[*]National Institute of Technology, Rourkela, Odisha-769008, India, Email-sumitmahapatra147@gmail.com
[†]National Institute of Science Education and Research, Bhubaneswar, Odisha - 752050 Email-smishra@niser.ac.in
[‡]Homi Bhabha National Institute, Training School Complex, Anushaktinagar, Mumbai 400094, India

undergone continuous investigation from the industry and academia likewise. A brief survey of the of approaches are reviewed in [4], the techniques from data science and artificial intelligence (AI) have been recently employed to mitigate and avoid the problem of software vulnerability discovery and analysis. However, a survey and analysis of such techniques are missing from the literature.

In this chapter, Section 2 presents a review of the data-mining and machine-learning approaches towards the discovery and analysis of the software vulnerabilities. Section 3 presents a brief overview of software metrics and the vulnerability prediction based on those metrics. Section 4 deals with various approaches for anomaly detection. Pattern recognition of vulnerable code is presented in Section 5. Finally a system that can used for automated software testing in presented in section 6 based on machine learning and the summary of the techniques is established in section 7. The conclusion is provided in Section 8.

## 2 Background: Software Vulnerability Analysis And Discovery

### 2.1 Definition

This section will begin with the definition of a software vulnerability which is the core focus of this entire chapter. In a PhD thesis on software susceptibility inspection, Ivan Krsul stated that:

***"an instance of an error in the specification, development or configuration of software such that its execution can violate the security policy."*** [5]

Although after acknowledging Krsul's statement, Ozment yet suggested a slight alteration and stated that:

***"A software vulnerability is an instance of a mistake in the specification, development or configuration of software such that its execution can violate the explicit or implicit security policy."*** [6]

Meanwhile, experts from industry also provided definitions that aligned with those stated above:

***"In the context of software security, vulnerabilities are specific flaws or oversights in a piece of software that allow attackers to do something malicious: expose or alter sensitive information, disrupt or destroy a system or take control of a computer system or program."*** [7]

Inferring from the aforesaid definitions, software vulnerabilities are defined using distinct idioms. To give a clear view about the idioms and choose the appropriate ones, the IEEE Standard Glossary of Software Engineering Terminology [8] is referred. Four key terms and their definitions are the center of focus here: "error," "fault," "failure," and "mistake". As claimed by IEEE Standards (1990) the connotation of error is: "the dissimilarity between theoretical values/conditions that is the correct ones and the computed values or can be said as the measured ones" [8]. Now a fault is basically: "a false tread or procedure or data connotation within a system software" [8]. Flaws or bugs are the common terms which are used to designate faults. A failure is: "the lack of ability of a module to execute its essential purpose within the bounds of stated executing requirements" [8]. Lastly, a mistake is: "An erroneous result as a product of human fallacy" [8]. The idea behind such a classification is to "differentiate between a human fallacy (an erroneous result), shortcoming because of either hardware or software (a fault), consequence of the shortcoming (a failure), difference in the aggregate outcome leading to erroneous conclusion (the error)" [8]. Software susceptibility is basically an occurrence of one or more flaws because of a mistake in the design, development or configuration of the software such that it can be exploited to violate some explicit or implicit security policy.

## 2.2 Completeness, Soundness and Undecidable

Program susceptibility inspection is used basically in order to categorise given programs whether known security vulnerabilities are present in it or not. Similarly undecidable signifies that there is no proper or exact result to the issue that exists [9] [10] [11].

A susceptibility inspection system is proper if no assailable program(s) is/are passed in context of software vulnerability. If no erroneous susceptibilities are passed or in other terms all error free programs are approved then it is termed as exact. By repercussion, a proper and exact susceptibility analysis system can accept every error free program and reject every susceptible program [11]. However, such proper and exact system does not exists [12].

Now let us take a look at what program vulnerability discovery system basically is. Aside from what already is known about vulnerability analysis, a program vulnerability discovery system is deemed as the most pragmatically useful. Contradicting to the general consensus about the vulnerability analysis system which keeps a check on the aspects of security of a given program by either approving it or disapproving it, a program vulnerability discovery system accounts for reporting of much detailed information like category, location, etc. for all vulnerabilities located in a specified program. Software developers and engineers who play a major role in detecting and fixing vulnerabilities are the ones who are aided most by this system and hence it is a desirable system in the software industry.

## 2.3 Conventional Approaches

Regardless of the monotony essence of the problem that persists in the field of software security, its analysis and discovery, numerous viewpoints have been recommended by executants in the academic fraternity as well as in the software field owing to the vital significance of the matter. The recommended viewpoints are all automatic approximates, each of them lacking in some or other form. Consequently, owing to a particular strand of the procedure of software inspection and locating, researchers are trying to propose upgraded solutions in resemblance to the prior solutions. For example: inspection coverage, discovery precision, run-time efficiency and so on. All inspecting advances are categorized into basically three major divisions:

1. Static Analysis
2. Dynamic Analysis
3. Hybrid Analysis
4. Fuzz-Testing [Other Known Method]
5. Static Data-Flow Analysis [Other Known Method]

### 2.3.1 Static Analysis:

In this analysis, the source code of a given program is analyzed without actually executing it. The techniques here employ a approximate pattern in order to inspect the attributes of a program (i.e.at best false vulnerabilities may be reported and there are no missed vulnerabilities). Hence, precise hypothesis leads to less claimed erroneous software vulnerabilities. Basically, a proper balance has to be made in between the accuracy of correct examining and methodical number crunching.

### 2.3.2 Dynamic Analysis:

In this analysis the run-time functioning of a given program is monitored with the help of some specific input data. In this technique, the properties of a program are analyzed using test cases and in fact the number of test cases can abruptly vary including the number of inputs as well as the run time states, hence it is not within the bounds of this analysis system to analyse the program's behaviour. Thus the best case scenario a practitioner can expect is the analysis system to be complete. However, instead of having such a drawback, it is widely used in software industry.

#### 2.3.3 Hybrid Analysis:

It is a combination of static analysis and dynamic analysis techniques. One of the misconception as per above descriptions regarding static and dynamic analysis might lead to conclude that hybrid analysis be a better analysis compared to others. However it is false, and while hybrid analysis techniques can be benefited from the advantages of both inspections, they also suffer from the limitations of both approaches. Thus this inspection technique can either root out erroneous susceptibilities or keep a check on test case preference and inspection procedure via static analysis system and hybrid analysis system respectively.

Some other known analysis include Fuzz-Testing and Static Data Flow analysis.

#### 2.3.4 Fuzz-Testing:

It is also termed as random-testing and accounts for examining of failures. Here well proportioned input data are aimlessly altered and then the program is put to test using these inputs [13] [14]. These inputs are usually fed at large amounts in regard to keep in check about the number of failures.

#### 2.3.5 Static Data-Flow Analysis:

Termed as "Tainted Data-flow Analysis", it is an analysis where the technique accounts for tagging the sceptical data from known/input sources as the corrupt ones. Then the flow to sinks (which stands for the sensitive programs) is observed as an indicator of vulnerability [15] [16] [17] [18]. This analysis approach is widely used in the software industry.

### 2.4 Categorizing Previous Work

A lot of research studies have been presented in previous years that accounts for the investigation using machine learning techniques in the sphere of inspection of software susceptibility and detection. There are three main categories, which are defined as follows:

#### 2.4.1 Vulnerability Prediction Models based on Software Metrics:

This category deals with the works where the main focus lies on the creation of a prediction model established on familiar software metrics known as the feature set and then the susceptibility status is assessed based on this model. In this category software metric plays a key role in locating the vulnerabilities in the software.

#### 2.4.2 Anomaly Detection Approaches:

This technique is used to detect vulnerabilities by employing an unsupervised learning method. In this method a model is extracted based on the unsupervised learning and anomalous behavior is checked based on the source code. Here anomalous behavior plays a major role in identifying as well as detecting the vulnerabilities.

#### 2.4.3 Vulnerable Code Pattern Recognition:

In this classification, the major focus is to draw out patterns based on supervised machine learning technique in order to compare and detect vulnerabilities in source code based on previous samples. Drawing out patterns and then localisation of the vulnerabilities makes this technique differ from the rest.

## 3 Vulnerability Prediction Based On Software Metrics

Susceptibility prediction models basically utilize statistical-analysis, machine-learning and data-mining techniques for identifying vulnerable program loopholes based on common metrics. The term metric is basically defined as a quantitative measure of the extent to which a system, component or process possesses a given feature. Given the limited resources available for investigation and authentication of software program, hence this process helps in bringing a model into play that accounts for enabling well organized software examining plans. These prototypes have gained high interest for research purposes in the academic community along with the industry fraternity as they deal with only a particular type of fault.

## 4 Anomaly Detection Approaches

The following section helps us in giving a view of section of works that utilize an anomaly detection approach using machine-learning and data-mining techniques for software vulnerability analysis and discovery. These recognizing approaches accounts for detecting framework that do not match with the standard patterns and are categorized as anomalies [19].

These methods are aimed at identifying software susceptibility by rooting out the spots in program that do not match with the anticipating patterns. Basic examples are in the domain of API usage which include but not restricted to function call pair of malloc and free, lock and unlock. Apart from these known sequences, every API has its individual set of procedures which might be skillfully written but too complex. However not being in line with usage patterns would result in software susceptibilities.

Among many other areas where anomaly detection approaches are pragmatic for software calibre, one such area is of identifying the overlooked states or mislaid inspections. As a matter of fact, mislaid inspections are considered to be the root of numerous software defects. These inspections are broadly subdivided into two categories: (1) inspections needed for genuine API running and (2) inspections that use program reasoning. Both of the categories are a prime reason for software defects or software vulnerabilities. An inspection on the input data taken as stipulations of an API function call can be cited as an example of the first type. Absence of alike inspections can lead to undefined, or unexpected actions by the software (e.g., division by zero). This loophole can also have adverse impact on security (e.g., buffer-overflow, SQL-injection, etc.). Failure to inspect the authorization of a topic while retrieving a resource object are cited as the ones of second type. These reasonable flaws are cited as the prime reason behind security reverberations leading to security logic vulnerabilities (e.g., confidentiality and integrity access control).

One of the vital features required for identifying abberant aspects is via automated extrication of detailed rules and patterns. The automated extrication of normal deportment is vital to the applicability and success of these techniques. However if the normal requirements were to be assigned by human users it would highly hinder the functioning of the approach since: (1) writing requirements is a hard and tedious task and (2) human errors can lead to inaccurate requirements that result in incorrect conclusions.

## 5 Vulnerable Code Pattern Recognition

In this segment, an overview of another category of works that employ data-mining and machine learning techniques via pattern-matching procedures in order to draw out attributes and figures of susceptible segments of the code and locate software vulnerabilities at the same time is described. Well this section of techniques inspect as well as draw out attributes from the source code be it high level or the binary one. But this group of techniques endeavours in order to draw out models and sketches of susceptible code in contrast to the anomaly detection group where the sole aim was to draw out models and rules of normality. However, in this category of studies the main focus is always tended to gather a large data-set of software vulnerability samples and then process them to extract feature vectors from each sample and then utilize machine-learning algorithms (mostly supervised learning) to automatically learn a pattern acknowledging model for software vulnerabilities.

## 6 System And Method For Automated Software Testing Based On Machine Learning

Previous inventions provide a system as well as method for automated software testing based on machine learning. ML based automated bug triaging, filing and notification system is a new approach in software testing field. The invention makes the present-day test management life cycle more productive, efficient and a self-decision-making system, which is fast and reliable. The invention is based on supervised / semi - supervised learning to make all processes automated and time saving. Algorithms and programs control quality so that any number of defects can be handled in an efficient manner.

### 6.1 System for Automated Software Testing

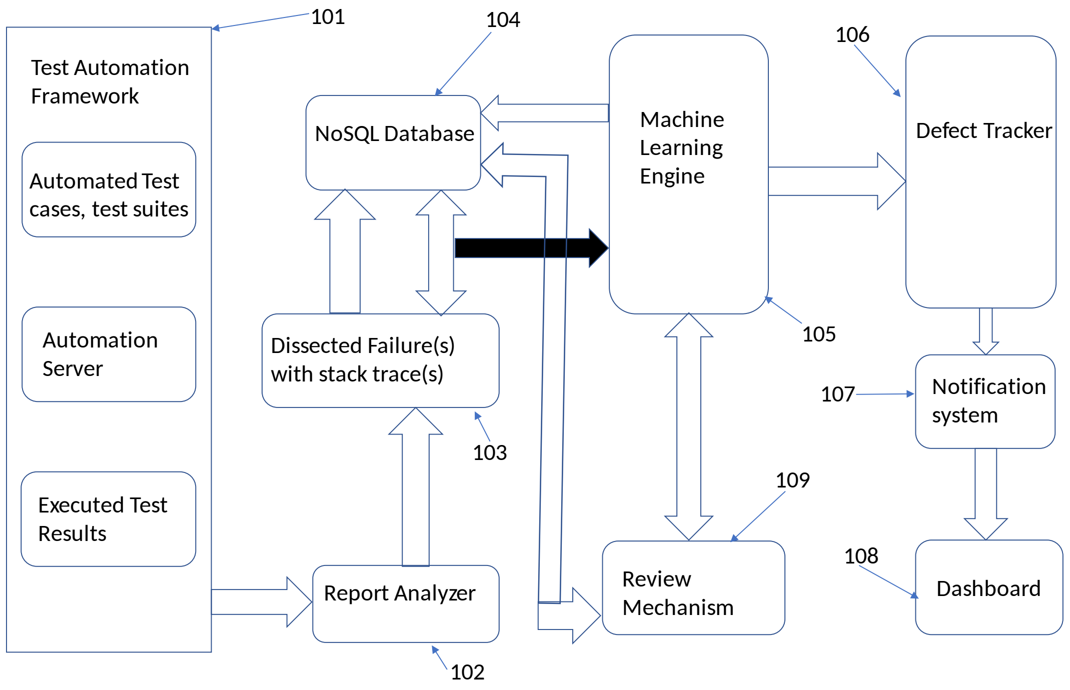


Figure 1: Illustrates the system for automated soft ware testing based on ML.

Software test automation framework 101 contains automated test cases and automated test suite. The automated test case codes may be written in any preferred programming language be it Python or any other. Automated test suite is basically a collection of automated test cases, page objects, helper or utility programs. The test suites are run through an automation server. Automated server is a server which acts as an interface or tool to easily configure or bring together the requirements / dependencies / frameworks / test suites to execute programs to obtain the desired results. One can set the time for these jobs as per as desired schedule or preferences either to run automated tests or to build soft ware, mobile applications etc. In a preferred paradigm of the invention, when the automated tests are run they provide the results in a pre-determined format.

Report parser parses these results to capture failures or exceptions with their respective stack traces. Report parser 102 is a program, which goes through whole content of the result file and searches for certain attributes or keywords. In a paradigm of the invention, the test automation results are obtained in any desired format, preferably a XML format. The system picks the XML file with records results from the Jenkins server location and by using a preferred programming language ( e.g. Python based parsing scripts in this case ), it goes through the length and breadth of the XML file to filter out all the respective failures or exceptions for respective tests with respective stack traces 103.

The recording of the parsed results is done in a NoSQL database 104 table where these will be compared with historical results for further processing. In a preferred paradigm of the invention, failures or exception and the list of historically captured failures or exceptions that already has defect / bug tickets, are stored in a NoSQL database 104 to facilitate the current comparison of theirs with the newly encountered failures or exceptions. In an embodiment of the invention, once the failure or exception is found in the NoSQL database, the ML engine 105 detects whether a defect ticket is created in the past for the automated test. In such cases, the failure or exception is not new but known and is already filed / recorded and considered for debugging .In a preferred embodiment of the invention, when ML engine 105 is unable to find the current failure or exception in the historical data then it recognizes that it is a new problem that has been found and it needs to be filed as new defect.

ML engine 105 uses the test management tool and its Application Programming Interfaces ( APIs ) to file the failure or exception as a new defect and upload all the related stack traces or exceptions in the ticket. In a paradigm of the invention, the severity of the bug and the priority of the test case that have failed remains the same if it is a failure that was historically encountered. But if the failure is new, then ML engine 105 gets the priority of the automated test case for which the failure occurred and the severity of the defect can be mapped in two ways. In the first way, if the test case that failed has high priority then the bug is considered to be of high severity and if the test case failed, it is of low priority which means that the bug is of low severity. In this way, the system has a mechanism to map the severity of the failure to the defect ticket. In the second way, the ML engine 105 provides the ability to learn by looking at the failure or exception and stack traces that when such an exception occurred and what was the severity of the defect ticket. This learning is unsupervised as the ML takes historical data to learn the same. Preferably, in case the severity predicted by ML algorithm is not correct as per test engineer he can change the mapping of severity for such types of failure or exception in the machine learning NoSQL database 104 mapping structure. This is the feedback that helps in making the ML more accurate and precise.

In a preferred paradigm of the invention, ML is not equipped to deal with all kinds of scenarios and exceptional cases as it is built upon the idea that it would learn based on the data provided, and to learn in a supervised and an unsupervised manner. The feedback mechanism 109 helps in fine tuning or adjusting the ML algorithm such that one can make some of its learning more precise and targeted towards providing with outputs that one can make sense of. Further, the feedback mechanism is provided in the system to fine tune the quality. Since all the processes are auto mated, it saves time of the concerned persons. Hence, the algorithms and programs control the quality so any number of defects can be handled. Yet another paradigm of the invention, prediction of failures includes creation of bug tickets in the defect - tracking tool. Ticket creation means logging the failure or exception as a defect or bug in the bug / defect tracking tool 106 i.e. for e.g. JIRA , Bugzilla etc., so that a developer can look at it to fix it. In a preferred paradigm of the invention, if thefailure is new then a new bug ticket is created and the status of the new ticket is kept as " open ". If the failure is already present in the defect tracking tool 106 and the status of the bug ticket is " closed " then the status of the bug ticket is changed as " reopen ". If the failure is already present in the defect tracking tool and the status of the bug ticket is either " known " , " deferred " , " open " , " reopen " then the system will add a comment with current failure details. All this is done through the defect / bug tracking system's Application Programming Interface(APIs).

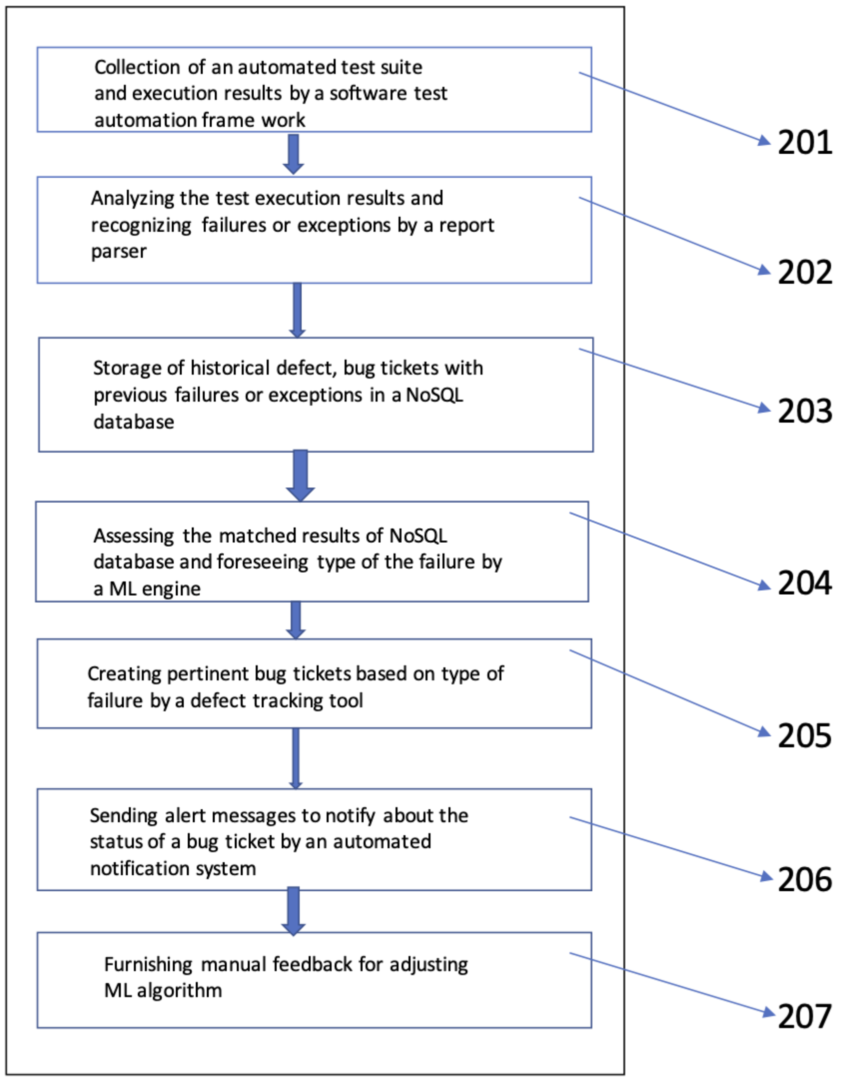
Preferably, the timeline of the release can be predicted based on the number of defect or bugs that have filed in the form of new or reopened tickets. The timeline of the release is directly proportional to number of new tickets created or reopened. The developer who fixes the issue does this estimate manually. ML engine is intended to predict the time a developer takes to close ticket based on certain type of failure or exception using same unsupervised learning based ML techniques have been used in the bug triaging and filing system. In a preferred embodiment of the invention , the notification of the stakeholders is done by an automated notification system 107, when the ticket is created or reopened in the defect tracking system. The notification is done via email or instant messaging about the respective ticket. In a preferred embodiment of the invention, dash board 108 is configurable to graphical user tools / programs / interfaces by which one can easily access the results, logs, failures, key performance indicators etc. in the form of histograms, pie graphs etc. They access data from the database usually.

## 6.2 Method for Automated Software Testing

The method for automated software testing based on ML comprises the steps of collecting a test suite and test execution results by a software test automation framework on a regular basis, which could be hourly , nightly , weekly etc., at step 201. At step 202, the report parser parses the test execution results generated from the software test automation framework to provide the failures or exceptions with their respective stack traces. At step 203, NoSQL database stores historical defect, bug tickets with past failures or exceptions. At step 204, ML engine evaluates the matching results of NoSQL database and predicts the type of the failure or exception. Further, defect-tracking tool creates relevant bug tickets based on the type of failure or exception at step 205. An automated notification system gives notifications to notify the stake holders about the status of a bug ticket at step 206. Furthermore, a manual feedback mechanism provides for adjusting ML algorithm and NoSQL database table entries at step 207. In accordance to one embodiment of the invention, the method for predicting the failure or exception further comprises the steps of creating a new ticket. The steps include keeping the status as " open " if the failure is new. Changing the status of the bug ticket to " reopen " for existing failure, if the status of the ticket is " closed " and adding comment with the failure details for the known and deferred failures after changing the status of bug ticket to " reopen ".

In an embodiment of the invention, once the bug ticket is created or reopened, the system will automatically notify the stakeholders via email or instant messaging about the status of the bug ticket. Hence, whole process will be automated. Therefore, there will no human intervention unless the algorithm or the ML engine has to adjust via feedback. This makes the system more precise and consumes less time in debugging the errors. Compared to the current state of the art software testing methods, the present invention offers faster operation and higher quality results due to a more focused analysis, ML algorithm and feedback mechanism. In comparison to static software analysis tools, the present invention offers a way to scale the analysis to find deep and intricate potentially multi - threaded software bugs. Furthermore by using ML to learn program behavior, the present approach provides heuristics to automatically mine for hypothetical program-dependent properties.

The present principles related to computer software development and testing, in particular for improved software testing based on ML incorporate a report parser, NoSQL database, feedback mechanism and a dashboard. It would be easy to glance through the quality of the automated test suites via dashboard. Hence, the overall quality can be checked and immediate action can be taken based on the observation. Further, the techniques described in the various paradigm discussed above result in efficient, robust and cost-effective management and testing of software production incidents subsequent to release of a software product. The techniques described in the paradigms discussed above provide an automated process of testing a defect that is found in production, thereby ensuring a consistent and predictive delivery of software product quality. ML and feedback mechanism of the process ensure that the process keeps improving its efficiency after putting to use. The feedback mechanism further receives feedback from the users for continuous improvement of the process. Additionally, the techniques described in the paradigms discussed above analyze the production defects and learn from the pattern, correlate



Collection of an automated test suite and execution results by a software test automation frame work
201
Analyzing the test execution results and recognizing failures or exceptions by a report parser
202
Storage of historical defect, bug tickets with previous failures or exceptions in a NoSQL database
203
Assessing the matched results of NoSQL database and foreseeing type of the failure by a ML engine
204
Creating pertinent bug tickets based on type of failure by a defect tracking tool
205
Sending alert messages to notify about the status of a bug ticket by an automated notification system
206
Furnishing manual feedback for adjusting ML algorithm
207

Figure 2: Illustrates a method for automated software testing based on ML, in accordance to one or more embodiment of the present invention.

the defects with the existing test cases and ensure a smooth build, run and re-installed into production. Further, the techniques described in the paradigm discussed above are easy to build and use and can be integrated with any system. The description of the present system has been presented for purposes of illustration and description but is not intended to be exhaustive or limited to the invention in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art without departing from the scope and spirit of the invention. The embodiment was chosen and described in order to best explain the principles of the invention and the practical application and to enable others of ordinary skill in the art to understand the invention for various paradigms with various modifications as are suited to the particular use contemplated.

What is claimed :

1. A system for automated software testing based on Machine Learning ( ML ), the system comprising :
   - (a) a software test automation framework configured to collect automated test suite and test execution results
   - (b) a report parser to parse the test execution results generated by the software test automation framework and configured to identify the failures or exceptions with their respective stack trace)
   - (c) a NoSQL database configured to hold historical defect, bug tickets with past failures or exceptions
   - (d) a ML engine to evaluate matching results of the NoSQL database and configured to predict type of the failure or exception
   - (e) a defect - tracking tool configured to create relevant bug tickets based on the type of failure or exception
   - (f) an automated notification system configured to notify the status of a bug ticket
   - (g) a dashboard to facilitate the access results, logs, failures, key performance indicators etc. in the form of histograms, pie graphs etc.
   - (h) a manual feedback mechanism for adjusting the machine learning algorithm and NoSQL database table entries
2. As per claim 1, wherein the test suite is running through an automation server.
3. As per claim 1, wherein the software test automation framework is further configured to test a web application , mobile application and any other type of software application to provide the test execution results in desired form.
4. As per claim 1, wherein the report parser parsing the test execution results file using a programming language.
5. As per claim 1, wherein the ML engine is configured to compare the parsed failures or exception with historical data in the NoSQL database to predict whether the failure or exception is new, deferred or known.
6. As per claim 5, wherein the ML engine is further configured to create a bug ticket for new failure and change the bug ticket status for known failure in the defect - tracking tool.
7. As per claim 1, wherein the auto mated notification system is configured to notify the stake holders via email or instant messaging about the respective status of the bug ticket.
8. A method for automated software testing based on Machine Learning ( ML ), the method comprising the steps of:
   - (a) collecting an automated test suite and test execution results by a software test automation framework ;
   - (b) parsing the test execution results generated from the software test automation framework and identifying failures or exceptions with their respective stack traces by a report parser;

(c) storing historical defect, bug tickets with past failures or exceptions in a NoSQL database;
(d) evaluating matching results of NoSQL database and predicting type of the failure or exception by a ML engine;
(e) creating relevant bug tickets based on type of failure or exception by a defect - tracking tool;
(f) sending notifications to notify about the status of a bug ticket by an automated notification system ; and
(g) providing manual feedback for adjusting ML algorithm and NoSQL database table entries.

9. As per claim 8, wherein the test suite is running through an automation server.
10. As per claim 8, wherein collecting the test execution results by testing a web application, mobile application and any other type of software application using the automated testing framework.
11. As per claim 8, wherein the report parser parses the test execution results file using a programming language.
12. As per claim 8, wherein the ML engine compares the parsed failures or exception with historical data in the NoSQL database to predict whether the failure or exception is new, deferred or known.
13. As per claim 12, wherein the predicting the failure or exception further includes method of creation of bug tickets in the defect tracking tool by :
    (a) creating a new ticket, if the failure is new with the status as " open ";
    (b) changing the status of the bug ticket to " reopen " for existing failure if the status of the bug ticket is " closed "; and
    (c) adding comment with the failure details for the known and deferred failures after changing the status of bug ticket to " reopen ".
14. As per claim 8, wherein sending notifications to notify the stakeholders about the status of a bug ticket is done via email or instant messaging.

## 7 Summary of the techniques

For detecting errors or code prone to defects, researchers have been working on the machine learning methods for around three decades now. This definitely comes in handy to the software test engineer when project is too huge to perform testing exhaustively and there is limited budget. Some of the techniques are compiled in the table below.

| Sl. | Main Technique | Sub-items |
|---|---|---|
| 1 | Static Attributes | • McCabe metric [20]<br>• Halstead metric [21] |
| 2 | Imbalance learning | • Cost-sensitive learning algorithms [24, 23]<br>• One-class learning [22] |
| 3 | Ensemble learning | • Ensemble Learning Methods [25, 26] |
| 4 | Multiple Classifiers | • SMOTEBoost [27]<br>• AdaBoost.NC [28, 29] |
| 5 | Resampling | • Resampling [30]<br>• Stratification [31] |

## 8 Conclusion

Software will continue to be a part of human life as far as the future can be envisioned. Hence, the current gravity of the situation where software vulnerabilities are becoming a critical threat, which directly and indirectly have a great impact on human life. Data mining and machine learning have made some great leaps in other domains of computer science, and the last decade has also been a witness to these techniques being used in software vulnerability analysis also. This chapter summarized the previous studies and current state-of-art steps, techniques in each category of the vulnerability analysis. This chapter and the effort put into it will help researchers have a grasp on the current best-practices and some well-known techniques.

## References

[1] Kartik Nayak, Daniel Marino, Petros Efstathopoulos, and Tudor Dumitras. 2014. Some vul- nerabilities are different than others. In Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'14). Spring er, 426–446

[2] US-CERT. 2015. Adobe Flash and Microsoft Windows Vulnerabilities. Retrieved from https://www.us-cert.gov/ncas/alerts/ TA15-195A

[3] US-CERT. 2013. Oracle Java Contains Multiple Vulnerabilities. Retrieved from https://www.us-cert.gov/ncas/alerts/ TA13-064A.

[4] Hossain Shahriar and Mohammad Zulkernine. 2012. Mitigating program security vulnerabilities: Approaches and challenges. ACM Comput. Surveys (CSUR) 44, 3 (2012), 11

[5] Ivan Victor Krsul. 1998. Software Vulnerability Analysis. Ph.D. Dissertation. Purdue Univer- sity

[6] Andy Ozment. 2007. Improving vulnerability discovery model s. In Proceedings of the 2007 ACM workshop on Quality of Protection (QoP'07). ACM, 6–11

[7] Mark Dowd, John McDonald, and Justin Schuh.2007. The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Addison-Wesley Professional

[8] AIEEE Standards. 1990. IEEE Standard Glossary of Software E ngineering Terminology. IEEE Std. 610.12-1990.

[9] William Landi. 1992. Undecidability of static analysis. A CM Lett. Program. Lang. Syst. (LOPLAS) 1, 4 (1992), 323–337.

[10] Thomas Reps. 2000. Undecidability of context-sensitive dat a-dependence analysis. ACM Trans. Program. Lang. Syst. (TOPLAS) 22, 1 (2000), 162–186. St uart Russell and Peter Norvig. 2009. Artificial Intelligence: A Modern Approach (3 rd ed.).

[11] Yichen Xie, Mayur Naik, Brian Hackett, and Alex Aiken. 20 05. Soundness and its role in bug detection systems. In Proceed- ings of the Workshop on the Evalua tion of Software Defect Detection Tools (BUGS'05).

[12] Ranjit Jhala and Rupak Majumdar. 2009. Software model check ing. ACM Comput. Surveys (CSUR'09) 41, 4 (2009), 21.

[13] Patrice Godefroid. 2007. Random testing for security: Bl ackbox vs. whitebox fuzzing. In Proceedings of the 2nd International Workshop on Random Test ing (RT'07). ACM,1.

[14] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox fuzzing for security testing. Queue 10, 1 (2012), 20.

[15] David Evans and David Larochelle. 2002. Improving securit y using extensible lightweight static analysis. IEEE Software 19, 1 (2002), 42-51.

[16] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manue l Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. 2004. Right ing software. IEEE Software. 21, 3 (2004), 92–100.

[17] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using static analysis to find bugs. IEEE Softw. 25, 5 (2008) , 22-29.

[18] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Se th Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few bil lion lines of code later: Using static analysis to find bugs in the real world. Commun . ACM (CACM) 53, 2 (2010), 66-75.

[19] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. ACM Comput. Surveys (CSUR) 41, 3 (2009), 15.

[20] T. J. McCabe, "A complexity measure," IEEE Trans. Software Eng., vol. 2, no. 4, pp. 308–320, Feb. 1976.

[21] M. H. Halstead, Elements of Software Science.. New York, NY, USA: Elsevier, 1977.

[22] N. Japkowicz, C. Myers, and M. A. Gluck, "A novelty detection approach to classification," in Proc. IJCAI, 1995, pp. 518–523.

[23] H. He and E. A. Garcia, "Learning from imbalanced data," IEEE Trans. Knowledge Data Eng., vol. 21, no. 9, pp. 1263–1284, Sep. 2009.

[24] Z.-H. Zhou and X.-Y. Liu, "Training cost-sensitive neural networks with methods addressing the class imbalance problem," IEEE Trans. Knowledge, Data Eng., vol. 18, no. 1, pp. 63–77, Jan. 2006.

[25] G. Brown, J. L. Wyatt, and P. Tino, "Managing diversity in regression ensembles," J. Mach. Learn. Res., vol. 6, pp. 1621–1650, 2005.

[26] K. Tang, P. N. Suganthan, and X. Yao, "An analysis of diversity measures," Mach. Learn., vol. 65, pp. 247–271, 2006.

[27] N. V. Chawla, A. Lazarevic, L. O. Hall, and K. W. Bowyer, "SMOTE- Boost: Improving prediction of the minority class in boosting," Knowledge Discovery in Databases: PKDD 2003, vol. 2838, pp. 107–119, 2003.

[28] S.Wang, H.Chen,and X.Yao," Negative correlation learning for classification ensembles," in Proc. Int. Joint Conf. Neural Netw., WCCI, 2010, pp. 2893–2900.

[29] S. Wang and X. Yao, "Negative Correlation Learning for Class Imbalance Problems" School of Computer Science, University of Birmingham, 2012, Tech. Rep.

[30] L. Pelayo and S. Dick, ”Applying novel resampling strategies to software defect prediction,” in Annu. Meeting North Amer. Fuzzy Inf, Process Society, 2007, pp. 69–72.

[31] L. and S. Dick, ”Evaluating stratification alternatives to improve software defect prediction,” IEEE Trans. Rel., vol. 61, no. 2, pp. 516–525, Jun. 2012.