How to Adapt Machine Learning into Software Testing

Mesut Durukal
IOT Division
Siemens AS
Istanbul, Turkey
e-mail: mesut.durukal@siemens.com

***Abstract*—Software testing cycles have several difficulties, such as coverage of a dense scope in a limited time, due to dynamic product development approaches. Researchers try to use new techniques to overcome these difficulties. This paper presents the utilization of Machine Learning (ML) in software testing stages with its effects and outcomes. Practical applications and advantages are analyzed. The main goal is to make insights about what can be done in different stages of software testing by employing ML and discuss benefits and risks.**

Keywords-artificial intelligence; machine learning; software testing; test automation.

I. INTRODUCTION

Nowadays, software applications have very comprehensive features and usages. Most of them interact with other applications and connect to various platforms, which results in a remarkably wide scope and complexity [1].

Comprehensive and competitive features are required for products to survive in the modern world. Products should adapt to new functionalities and be compatible with emerging technologies. On the other hand, they should respond to rapid changes to be one of the firsts in the market and not to be old fashioned.

Figure 1 depicts these challenges by illustrating decreasing delivery time against increasing complexity.

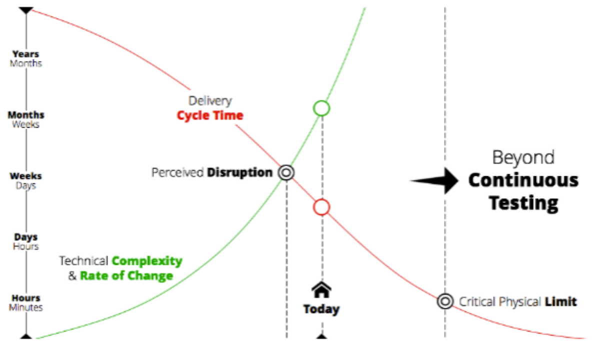


Figure 1. Delivery time versus complexity of products [2].

New challenges in product development have reflections in software testing as well. It is mandatory to take quick actions against gaps introduced by complexity and fast changes in testing cycles. In this manner, new approaches in testing have been applied to overcome these raising challenges. One of the most exciting candidates is the application of machine-based intelligence into testing [3]. ML practices in testing promise for additional coverage and saving on time thanks to their design capable of understanding the system and finding the best patterns. Machines work faster than human beings on analyzing big data and deciding on the most optimum solution. Therefore, faster, better and cheaper processes are expected to be achieved by the usage of ML. Consequently, huge budget will be allocated on adaptation of ML into software lifecycles. Figure 2 exhibits the estimation for ML projects budgets by 2025, which is $90BN.

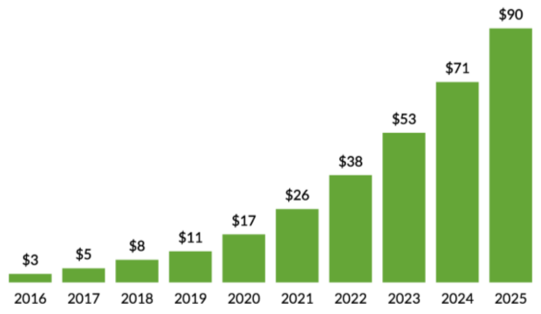


Figure 2. ML Projected Revenue in $ Billion [4].

In this paper, possible ML practices on software testing stages are investigated. Section II describes ML working principles. Section III explains several applications and their outcomes are analyzed in Section IV. Finally, summary of the work is given in Section V.

II. BACKGROUND

To reduce manual effort, several automation processes are integrated into software development projects. However, human intervention is still needed for the following activities [5]:

- acquiring the knowledge needed to test the system,
- defining testing goals,
- designing and specifying detailed test scenarios,
- writing the test automation scripts,
- executing scenarios that could not be automated,
- analyzing the results to determine threads.

Machines are mainly programmed to follow explicit instructions whereas humans learn a lot through observation and experience. ML is the key factor to fill the gap caused by the difference between the learning processes of machines and humans as much as possible and thereby to reduce human intervention.

ML is defined by Arthur Samuel in 1959 as “the subfield of computer science that gives computers the ability to learn without being explicitly programmed” similar to human beings. If the performance of a machine improves with experiences, it means that it is learning [5].

ML algorithms run in two stages: training and execution. First, machine learns the system, or in other words, it models the system. This stage is called training. Then, the execution is performed by the prediction of next steps according to learnt experiences. In short, what was learned in the past is applied to new data by machines. ML types can be classified as Supervised, Unsupervised, Semi-Supervised and Reinforcement Learning.

A. *Supervised Learning*

Supervised ML algorithms use labeled examples to learn and then to predict future events. Starting from the analysis of a known training dataset, the algorithm builds a model to make predictions about the output values as shown in Figure 3.

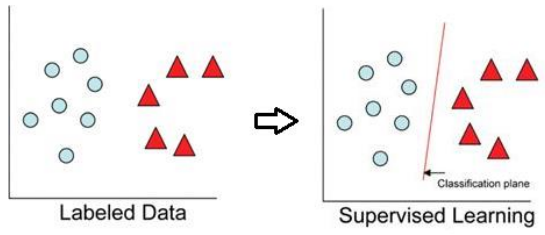


Figure 3. Supervised learning [6].

B. *Unsupervised Learning*

Unsupervised ML algorithms are used when training information is neither classified nor labeled. Under these conditions, system builds a model from unlabeled data to describe a hidden structure. The system is not expected to estimate the right output, but it explores the data, draws outcomes from datasets and finally describes hidden structures from unlabeled data [1].

C. *Semi-supervised Learning*

Semi-supervised ML algorithms fall somewhere in between supervised and unsupervised learning, since they use both labeled and unlabeled data for training. Figure 4 illustrates a sample modeling.

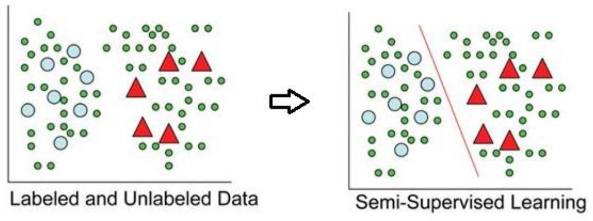


Figure 4. Semi-supervised learning [6].

D. *Reinforcement Learning*

Reinforcement ML algorithm is a learning method that interacts with its environment by producing actions and discovers errors or rewards. Simple reward feedback is required for the machine to learn which action is the best, which is known as the reinforcement signal. Figure 5 exhibits the execution of Reinforcement Learning.

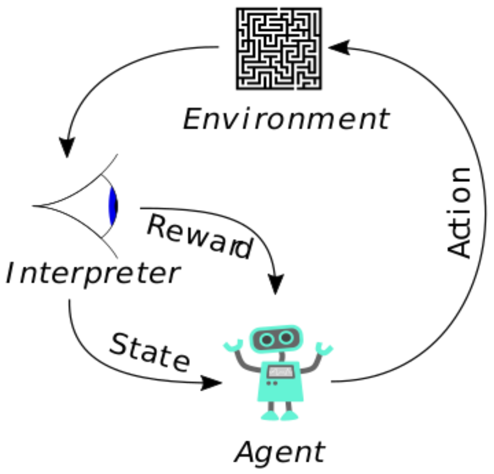


Figure 5. Reinforcement Learning [7].

III. Methodology

Lots of applications are developed with ML algorithms in various models, such as Artificial Neural Networks (ANN), Support Vector Machines (SVM), k Means Clustering, Random Forest (RF) and k Nearest Neighbors (kNN) as shown in Figure 6.

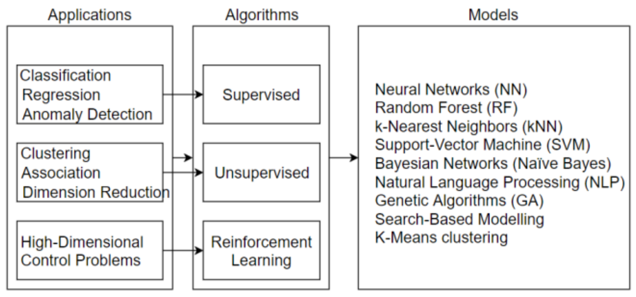


Figure 6. Models to Develop ML algorithms for various applications.

Several applications are developed for software testing purposes as well. As far as the adaptation of ML into Software Testing Life Cycle (STLC) is concerned, the whole process is handled in a structured manner in order to make it easily trackable. STLC is managed in three major stages [8] as shown in Figure 7:

- Definition
- Implementation
- Maintenance

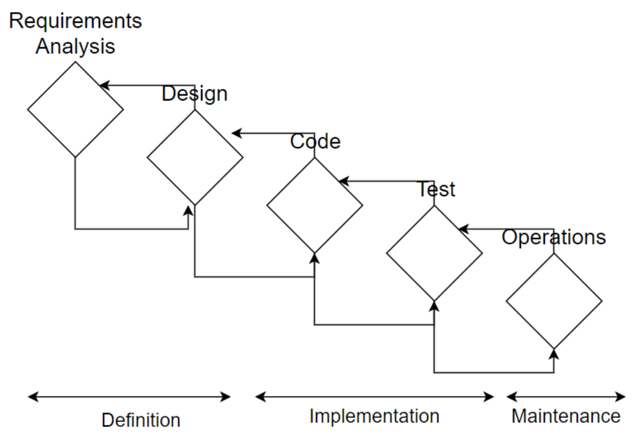


Figure 7. Software Testing Stages [8].

ML is utilized in all of these stages. In Section IV, application methodologies are investigated in detail. Table I summarizes sample tools or models used in the stages.

TABLE I. ML APPLICATIONS IN TESTING

Stage	Application	Tool/Model
Definition	Test Case Generation	AIST [5]
Implementation	Code Generation & Completion	DeepCoder [9]
		TabNine [10]
	Execution	Applitools [11]
Maintenance	Refactoring	DeepCode [12]
	Prioritization	ANN, GA models
	Suite Generation	Search-Based Models
	Bug Handling • Classification • Addressing • Scoring	Naïve Bayes, K-Means clustering models

IV. APPLICATIONS

In this section, ML applications in software testing stages are discussed.

A. *Test Definition*

In this stage, test scenarios are defined to cover all use cases to ensure product quality. ML improves effectiveness and reduces manual effort in the test definition stage in different ways. One of them is letting the machine learn the use cases of the system by observing actions and reactions. In this way, the mandatory parameters and expected inputs are learnt. Similarly, error messages in negative scenarios are also observed. At the end of the learning phase, a model of the system is created. Afterwards, test cases are generated to verify expected results and behaviors according to the model. A commercial example for this approach is Artificial Intelligence (AI) for Software Testing Association (AISTA) [5].

If the working principle is further investigated, it can be understood that the machine observes the responses to requests to model the data structure. Any of the algorithms mentioned in Section III can be applied to generate the model. Then, a set of parameter inference rules are defined to generate the input data required by the test cases [13]. Figure 8 [14] visualizes the model generation.

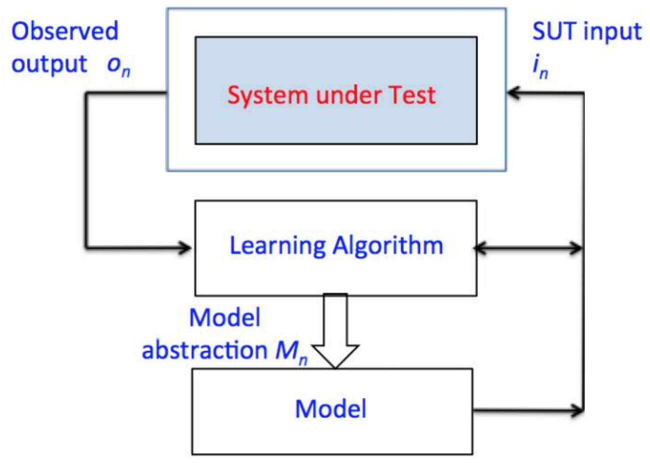


Figure 8. ML based model generation [14].

Offutt et al. [15] followed the same approach to learn the system. They illustrate the algorithm over a sample eXtensible Markup Language (XML) response:

```
<books>
    <book>
            <ISBN>0-672-32374-5</ISBN>
            <price>59.99</price>
            <year>2002</year>
    </book>
    <book>
            <ISBN>0-781-44371-2</ISBN>
            <price>69.99</price>
            <year>2003</year>
    </book>
</books>
```

As the machine trains the behaviors of the system, it learns the fields of entities and supported data types. For instance, after training, the machine knows that a book has properties “ISBN”, “price” and “year” in data types “string”, “double” and “integer”. Finally, test cases are generated by forming request according to this model with perturbated data values. Data values are smartly selected (e.g., boundary values). Table II [15] illustrates a sample set of cases. They constructed 100 test cases, which found 14 faults out of 18, implying that the success rate is 78%.

TABLE II. GENERATED TEST CASES BY MACHINE [15]

Original Value	Perturbated Value	Test Case
<price>59.99</price>	2^{63}-1	Maximum Value
	-2^{63}	Minimum Value
	0	Zero

After the deployments of new features, changes on User Interface (UI) are detected and images removed from the application are noticed. Consequently, the machine starts to learn about the application and relations between the modules. New test cases are generated according to these relations. In summary, whenever there are changes in the system under test, additional test cases are created by means of the approach explained.

It can be concluded that, ML improves the efficiency of testing activities in terms of coverage, time, effort and cost. Instead of analyzing the model and constructing test cases manually, the machine performs these operations. Thus, risks of manual work (e.g., skipping some cases) are minimized.

B. Implementation

In continuous testing environments, no one would refuse an increase in test implementation and execution speed. There are many ways to do this.

1) Code Generation & Completion

Coding is one of the biggest tasks in software lifecycles including development and testing activities. Thus, ML is an opportunity to improve or fasten the coding practices.

For robots, a way to write code is, first understanding the problem and then applying the solution. When a problem is defined with inputs and outputs, the needed operations are predicted and the related codes are generated by the machines. DeepCoder [9] follows the same approach. Here is an example of input and output in a scenario, in which negative numbers are filtered and listed in a reserve order after multiplied with 4:

For the input:
[-17, -3, 4, 11, 0, -5, -9, 13, 6, 6, -8, 11]

Expected output is:
[-12, -20, -32, -36, -68]

Figure 9 shows that DeepCoder predicts needed operations after checking inputs and outputs:

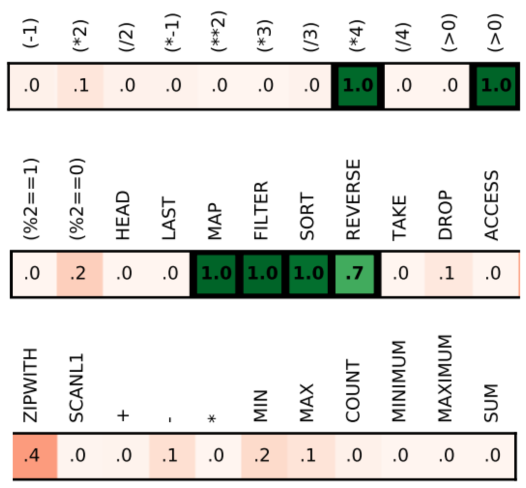
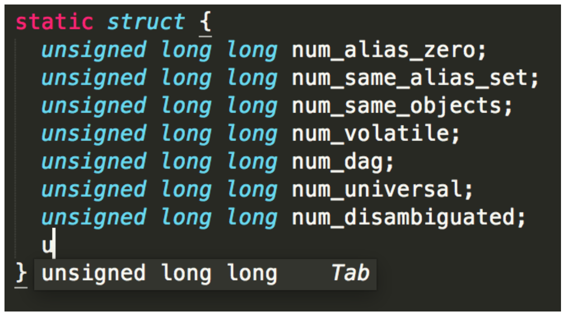


Figure 9. Predicted possibilities of operations [9].

Recognition of patterns between input and output values is achieved by passing them through hidden layers with an ANN model. As a result, they reached a speedup of up to 467 times [9].

Beyond generating a code from scratch, another way to improve the prcess is to automatically complete code. After the patterns the most frequently used are learned, the machines propose the subsequent codes during implementation. As shown in Figure 10, Tabnine [10] is an application, which facilitates test implementation.



```
static struct {
  unsigned long long num_alias_zero;
  unsigned long long num_same_alias_set;
  unsigned long long num_same_objects;
  unsigned long long num_volatile;
  unsigned long long num_dag;
  unsigned long long num_universal;
  unsigned long long num_disambiguated;
  u
} unsigned long long    Tab
```

Figure 10. An auto-completion application: Tabnine [10].

In short, ML not only reduces the effort and duration spent on code implementation, but also suggests the most frequently used patterns previously. In this way, standardization is also improved.

2) Execution

In terms of execution, ML helps with:

- Exploratory Testing
- Usability & Efficiency Checks
- Execution Analysis

ML bots perform exploratory testing by clicking every button on the application to test the functionalities. Adam Carmi, co-founder of Applitools [11], states: “We want to make sure that the UI itself looks right to the user and that each UI element appears in the right color, shape, position, and size.” ML algorithms are used in their tool Applitools to perform usability and efficiency tests. The system is modeled by the machine according to the defined use cases. Parameters for difficult and easy paths are extracted, and new designs are oriented by these trainings.

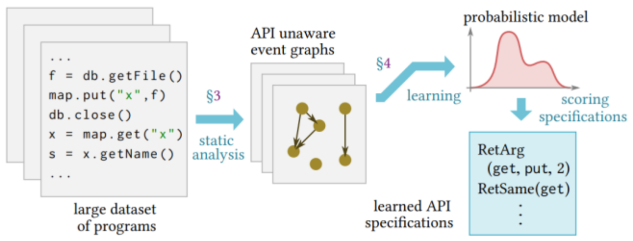
Furthermore, execution evaluation is performed by analyzing execution data with ML algorithms. During test executions, ML algorithms learn patterns and user tendencies by collecting data, taking screenshots, downloading the content of web pages and measuring loading times. Then, properties of new features are estimated, and the deviations are detected accordingly. For instance, if loading time of a new page is longer than predicted, a warning is raised. Some outlier detection algorithms are applied with Info Fuzzy Network [16] for ML based test execution purposes.

Results show that algorithms can automatically produce a set of nonredundant test cases covering the most common functional relationships existing in software. A significant saving is achieved from required human effort in this way, which means that benefits of ML are not limited to time only, but also cost and quality.

C. *Maintenance*

1) *Refactoring*

According to learnt patterns, some applications like DeepCode [12] propose solutions against code smells. It alerts about critical vulnerabilities needed to be solved in the code. Figure 11 shows how the model of the API is constructed with unsupervised learning algorithms. [17]



```
...
f = db.getFile()
map.put("x",f)
db.close()
x = map.get("x")
s = x.getName()
...
```

Figure 11. Learning of API Specifications in DeepCode [17].

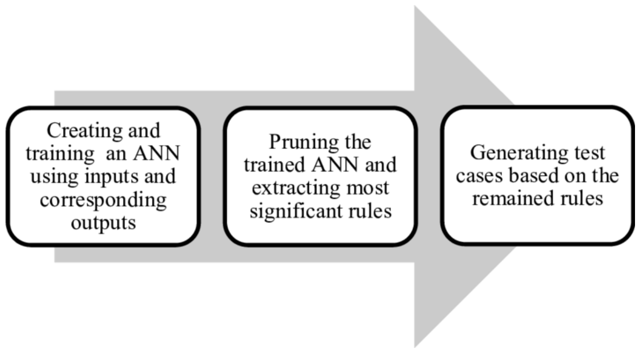
Bugs are not allowed to go to production thanks to findings. Thus, saving on time is achieved [17].

2) *Prioritization*

Infinite testing is impossible. With limited resources, prioritization among the test cases has critical importance. Priority is decided according to [18]:

- The probability to find an error,
- Uniqueness in terms of scope,
- Complexity or simplicity,
- Fitness for the regression activity.

For prioritization, test cases are evaluated according to the learnings, which are collected from the labelled training sets. Algorithms are developed with various approaches, such as ANN [19] and Genetic algorithms [GA] [20]. Figure 12 shows how the most significant cases are selected with ANN.



Creating and training an ANN using inputs and corresponding outputs
Pruning the trained ANN and extracting most significant rules
Generating test cases based on the remained rules

Figure 12. Test prioritization and reduction with ANN [19].

Thanks to prioritization, the number of tests cases to be executed is significantly reduced and less time is consumed on execution. Additionally, adaptation against immediate changes is quickly ensured since regression suite can be generated by ML algorithms.

3) *Suite Generation*

Whenever there is a change in the software, at least the regression suite is executed. ML algorithms train the relations between the test cases and the features and decide the related test suite for the newly added feature.

It is possible to construct a group of tests, which are similar, by observing the coverages of tests during their executions [21]. The main idea of the analysis performed by the machine is to understand which tests are contextually close enough to each other to construct suites. After the similarities between test case contexts are analyzed, they are grouped by their coverage.

a) Branch Coverage: According to the number of hits to a branch, the algorithm calculates the distances of executions to the target branch and the relation between a test and a branch is estimated.

b) Line Coverage: Distances are calculated with the number of covered lines in the code after the execution of a test.

c) Exception Coverage: Exception coverage is a kind of reinforcement learning and aims to reach as much exceptions as possible. Tests, which throw more exceptions, are rewarded.

d) Method Coverage: Method coverage approach applies the same algorithms over methods. Tests are evaluated according to whether they call methods or not.

4) *Bug Handling*

Bugs are of great importance since they contain valuable information about the product. According to bugs, useful analysis can be done, such as:

- constructing bug classes in relation with features,
- learning relations between bug contexts and severity,
- learning relations between bug contexts and assignees.

Bug classification provides hints about the weaknesses of the product. For example, if bugs mostly heap together on a feature, some actions can be taken accordingly. In such a case, related tests are prioritized to investigate the feature deeper.

Additionally, scoring of the bugs is very important since they are handled according to their severities. Bugs with the highest severity levels are fixed primarily, then the rest is handled in order. If a critical bug is not scored correctly (e.g., with a low severity), it may be postponed since it is not regarded as a priority. As a result, the regarding fix is not done as soon as the bug identified, which leads to additional costs.

Another point to mention is, for big teams, it is not easy to know each assignee for all features. In such cases, the assignee of a bug can be proposed by the machine according to the previously addressed bugs. Correctly assigned bugs are labeled, and the system is modeled by the machine. Then addresses for next bugs are estimated.

Table III summarizes results from 3 different studies.

TABLE III. ACHIEVEMENTS ON BUG CLASSIFICATION BY ML

Study	Assignment	Scoring
1 [22]	50%	
2 [23]	51.4%	72.5%
3 [24]		72%-98%

To sum up, ML algorithms contribute to bug handling processes, substantially. In terms of quality and scope, coverage is extended by means of the ML observations. If the machines detect any other weakness after bugs' analysis, related test cases are determined and added. Additionally, ML improves the management of bugs, since it helps with correct triage and assignment.

V. RELATED WORK

In this section, an application is discussed on bug severity estimation. For this application, bugs of MindSphere [25] are used. MindSphere is the cloud-based IoT open operating system from Siemens. In the project, bugs are labeled with severity classes:

- Severity 1: Safety
- Severity 2: Critical
- Severity 3: Major
- Severity 4: Minor

Severity 1 is for the cases, which are related to human life and safety issues. Currently, there is not a Severity 1 bug in the project. For the other severity classes, severity assignment to a bug is important in terms of prioritization. Additionally, in the project, Severity 2 bugs are especially tracked, since they are regarded as release blocker issues. Therefore, decision for a bug whether it is Severity 2 or not, affects the progress of the release.

For two different purposes, 889 customer bug entries are collected from Jira. On this data, estimation of a bug severity for 3 classes (Severity: 2,3,4) and decision on a bug whether it is a release blocker or not (Severity 2 or not) are tested. Figure 13 shows the distribution of severity of bugs.

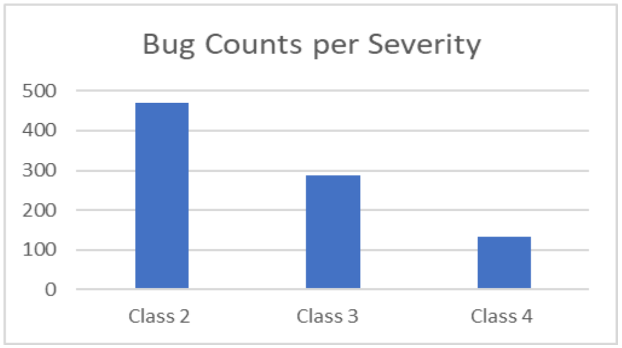


Figure 13. Bug Counts per Severity.

Since the bugs are collected from a real-life project, sample count is not very high. 5% of data is separated for testing and training is performed with SVM on the rest. Figure 14 exhibits the confusion matrix for 3 classes. The accuracy is found to be 64% for this case.

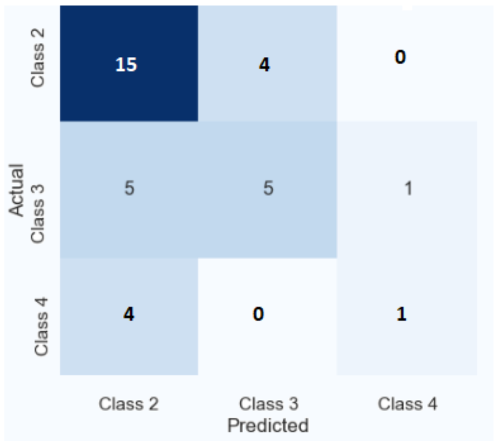


Figure 14. Confusion Matrix for 3 classes.

For two classes (Severity 2 or not), accuracy is 77%. Related confusion matrix can be seen in Figure 15.

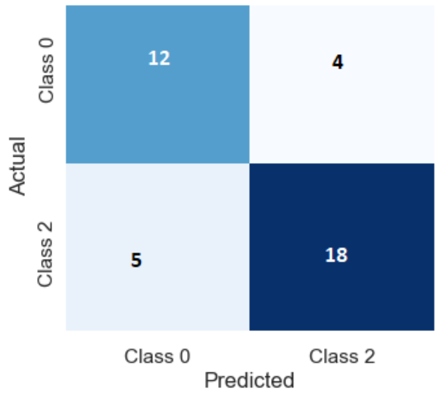


Figure 15. Confusion Matrix for 2 classes.

Severity estimation is not an easy task for bugs. For the same service, a bug can be both Severity 4 and Severity 2. Similarly, the same failure, i.e., data upload failure, can imply different severity levels depending on the conditions or input parameters. Therefore, it is very likely to face difficult decision cases. Considering these challenges, the results can be evaluated as successful.

VI. DISCUSSION

ML is applicable in all stages of software testing cycles. The usage of ML in testing activities has lots of advantages. Test coverage is improved by automatic test generation by machines. For the machine generated test cases, machines' success rate in detecting faults is reported as 78%.

In addition, ML applications provide extra speed in all stages of testing. Compared to humans, machines decide much faster. At least for the rough estimations, AI results can provide a quick feedback. As presented in Section IV, 467 times faster implementation is achieved.

Moreover, manual effort is obviously reduced. Instead of manual tasks, the machines work for defining, executing and maintaining tests. Outliers are detected by algorithms during execution. In this way, the risk off missing bugs is minimized and the cost is reduced with early fixes.

Advantages of ML applications in testing are unneglectable, however, potential risks should not be ignored. Performance, security, control and social risks can be faced in failure cases. Error cases can result in misleading actions, including security risks or fatal consequences [26]. Furthermore, if ML goes out of control, or is abused by people, some ethical and social concerns can arise. In short, it can be concluded that ML is a safe and beneficial tool only when it is under control.

VII. CONCLUSION AND FUTURE WORK

Rapidly improving software world grows a great rivalry and creates a pressure on stakeholders in terms of time, cost, scope and quality. Besides development processes, these challenges are faced also during the testing cycles. Thus, any effort that can overcome these challenges is welcomed. In this respect, ML is probably the most promising discipline to improve testing by making better and faster decisions.

Even though it is assumed that ML can never fully replace human beings, it is already surpassing humans in several tasks, such as playing games and providing recommendations. As far as these advances are concerned, the goal is to make use of ML in testing as much as possible.

ML algorithms provide a remarkable benefit on testing activities. It contributes with test coverage improvement, manual effort reduction, better conclusion and addressing.

As a future work, it is aimed to develop an algorithm to support bug assignment. Improving the algorithm for triage is on future agenda and finally, comparison of results with the studies in literature will be performed.

ACKNOWLEDGMENT

I am very grateful to Ms. Berrin Anil Tasdoken who has reviewed the paper and guided me for the improvements.

REFERENCES

[1] M. Durukal, "Practical Applications of Artificial Intelligence in Software Testing", International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), Volume 5 Issue 4, pp. 198-205, July-August 2019, doi : 10.32628/CSEIT195434.

[2] W. Platz, "What's beyond continuous testing? AI," SD Times, 2017.

[3] W. Murray, P. Karuppiah, and C. Stancombe," On the way to smart, intelligent, and cognitive QA," World Quality Report 2017-18, 9th edition, 2017.

[4] "Which Industries Are Investing in Artificial Intelligence?," Splunk, Priceonomics Data Studio, 2018.

[5] T. King, "AI Driven Testing: A New Era of Test Automation," Japan Symposium on Software Testing JaSST, pp. 1-30, 2019.

[6] A. R. Shah, C. S. Oehmen, and B. Webb-Robertson, "SVM-HUSTLE—an iterative semi-supervised machine learning approach for pairwise protein remote homology detection," Bioinformatics, Volume 24, Issue 6, 15 March 2008, pp. 783–790, doi: 10.1093/bioinformatics/btn028

[7] Reinforcement learning [Online] Available from: https://en.wikipedia.org/wiki/Reinforcement_learning 2019. 11.05

[8] M. M. Lehman, "Programs, life cycles, and laws of software evolution," in Proceedings of the IEEE, vol. 68, no. 9, pp. 1060-1076, Sept. 1980. doi: 10.1109/PROC.1980.11805

[9] M. Balog, A. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "DeepCoder: Learning to Write Programs," Proceedings of ICLR'17, March 2017

[10] Tabnine [Online] Available from: https://tabnine.com/ 2019.11.05

[11] Applitools [Online] Available from: https://applitools.com/ 2019. 11.05

[12] DeepCode [Online] Available from: https://www.deepcode.ai/ 2019. 11.05

[13] H. Ed-douibi, J. L. Cánovas Izquierdo and J. Cabot, "Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach," 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC), Stockholm, 2018, pp. 181-190. doi: 10.1109/EDOC.2018.00031

[14] K. Meinke and P. Nycander, "Learning-based testing of distributed microservice architectures: Correctness and fault injection," SEFM 2015 Collocated Workshops, pp. 3-10, 2015.

[15] J. Offutt and X. Wuzhi "Generating test cases for web services using data perturbation." ACM SIGSOFT Software Engineering Notes 29.5, pp. 1-10, 2004.

[16] M. Last and M. Freidman, "Black-Box Testing with Info-Fuzzy Networks," World Scientific, City, 2004.

[17] J. Eberhardt, S. Steffen, V. Raychev and M. Vechev, "Unsupervised learning of API aliasing specifications." In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 745-759, 2019.

[18] P. Saraph, M. Last, and A. Kandell, "Test case generation and reduction by automated input-output analysis," Institute of Electrical and Electronics Engineers Inc., City, 2003.

[19] Dr. A. P. Nirmala, Md Shajahan, Somnath K, "Impact of Artificial Intelligence in Software Testing," International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), ISSN : 2456-3307, Volume 3, Issue 3, pp.1519-1526, 2018.

[20] S. Dhawan, K. S. Handa, and R. Kumar, "Optimization of software testing using genetic algorithms," In Proceedings of the 11th WSEAS international conference on Mathematical and computational methods in science and engineering (MACMESE'09), World Scientific and Engineering Academy and Society (WSEAS), pp. 108-112, 2009.

[21] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," Springer International Publishing, Search-Based Software Engineering, volume 9275 of Lecture Notes in Computer Science, pp. 93–108, 2015.

[22] J. Anvik, L. Hiew and G. C. Murphy, "Who should fix this bug?." In Proceedings of the 28th international conference on Software engineering, pp. 361-370, 2006.

[23] V. Stagge, "Categorizing Software Defects using Machine Learning." LU-CS-EX, 2018.

[24] Imran, "Predicting Bug Severity in Open-source Software Systems Using Scalable Machine Learning Techniques." PhD diss., Youngstown State University, 2016.

[25] MindSphere [Online] Available from: https:// https://siemens.mindsphere.io/en/ 2019.11.05

[26] S. Levin and J. C. Wong, "Self-driving Uber kills Arizona woman in first fatal crash involving," The Guardian, March. 19, 2018.