
## Faculty of Mathematics and Computer Science

**Department of Programming Systems**


**Seminar 1919:**

**Modern Programming Techniques and Methods**

# *Artificial Intelligence in Software Testing*


**Student:**
***Maruna Derieg 6558020***

**Program:**
***B.Sc.Computer Science***

**Email:**
***maruna.derieg@studium.fernuni-hagen.de***

**Supervisor:**
***Dr.Daniela Keller***


**September 5, 2020**

# Contents

# List of Figures



# List of Tables

# 1 Introduction

Software testing is a crucial task in any software development project. It is estimated that at least half of the entire development cost can be attributed to software testing [15].

Artificial intelligence (AI) is penetrating almost every sector of technology [20]. It is thus not surprising, that many ask themselves whether AI will be able to solve some of the current challenges in software testing.

In a survey conducted in 2017 by Arbon [23], 328 professionals of the software testing industry have been responding to questions regarding their opinions about the future of AI in software testing. The results show that 94% believe AI to have a significant impact on software testing by 2025 [23].

In this paper we are going to discuss the current and potential impact of AI on software testing. First, the term AI and its sub-categories will be introduced. Unfortunately the term AI is often not properly defined, which can lead to confusion. Therefore we will not only define AI, but also discuss common misunderstandings about it. Having a brief look at the history of AI will provide a clearer picture about how AI has emerged as a research field.

Next the term software testing will be defined, and we will discuss the history and the current challenges of software testing. Understanding the current needs of software testing will make it more clear why developers have high hopes for AI to facilitate software testing.

We will then analyze the influence of AI on software testing from two different viewpoints: the micro and the macro perspective. In the micro perspective the question is asked, how AI is facilitating different sub-tasks of software testing. We will present recent research results and the claimed research activities of some software testing companies. The macro perspective is taking a step back, looking at AI in software testing from a bigger picture. Rather than focusing on the different areas of software testing, the macro perspective asks how the main goals of software testing can be achieved through the use of AI. This might include a change of the current paradigm of software testing. Approaches to software testing that have made sense until now might become inadequate with new AI-driven software testing solutions to come.

# 2 What is AI

When we speak about AI, it is sometimes not clear what exactly this term describes. The broad use of the term AI has lead to inconsistent ideas about what AI is and what it is not [12]. Further, the word 'AI' is often used for marketing purposes without clearly defining it. As a result, costumers have certain beliefs about the product they are buying, which do not always reflect reality. This is also the case with the advertisements of software testing programs. There is a discrepancy between the mainstream belief of what 'AI' is, and the use of the term 'AI' in scientific works [12].

We are going to give a brief history of AI. This will make it clear that AI is a research field of computer science that is not a new trend, and not limited to machine learning methods only. We will then present the definition of AI that we rely on in this paper, and also define the main sub-categories of AI. After a brief discussion about common

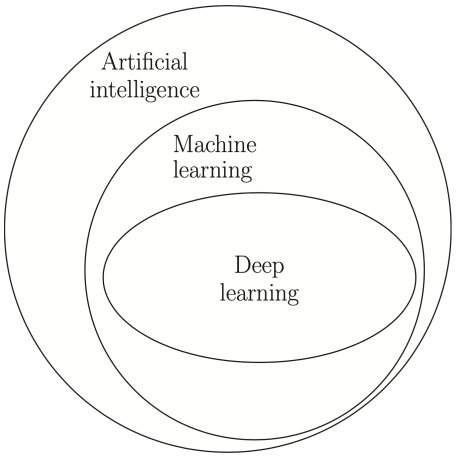


Figure 1: Venn Diagramm of AI Sub-classes [20]

misconceptions about what AI is, we will finish this section with a classification of all the AI methods mentioned throughout this paper.

## 2.1 History of AI

In the 1950's, researchers started to investigate how technology can be used to build computer systems which display intelligent behavior [10]. The findings of this ongoing research have brought a new discipline into life, called 'AI'. As this new field of research was at its very beginning, and the hardware capacities were very limited back then, the inventions made in the early days have not led to the desired outcome [43].

Deep learning techniques had in fact already been discovered in the 1950's [43]. Anyhow, research in the field of deep learning has been stagnate for a long time, as research sponsors did not recognize the full potential of this emerging field. Instead, AI researchers concentrated their efforts on other techniques such as expert systems, knowledge systems and fuzzy logic [10]. In the 1980's there was increased research being performed on machine learning methods such as support vector machines, decision trees, linear and logistic regression [34]. After 2010, AI attracted a lot of media attention due to the breakthroughs of many inventions that are based on deep learning techniques [20]. This has lead to a revival of neural network techniques within the AI research field [43].

The increased hardware capacities have made deep learning a usable tool with great success. Deep learning, however, is not the only methodology developed by the research discipline of AI. As we can see in Figure 1, deep learning belongs to the methods of 'Machine Learning' which in turn is a subset of AI [20]. People without a background in computer science do often not understand the difference between these 3 categories and use these terms interchangeably (see section 2.3). To avoid confusion it is crucial to clearly define these three terms.

## 2.2 Definitions

- **Artificial Intelligence:** Elaine Rich defines artificial intelligence as "the study of how to make computers do things at which, at the moment, people are better." [39].

There are of course other definitions of artificial intelligence in use, we believe this one to be the most suitable though. According to Wolfgang Ertel [10], this definition will remain valid even in 30 years from now.

- **Machine Learning:** Arthur Samuel [42] defined machine learning as "a field of study that gives computers the ability to learn without being explicitly programmed". The learning aspect is the central part of all machine learning techniques, and can easily be associated with the term 'intelligence'.
- **Deep Learning:** Deep learning is the subset of machine learning that focuses on deep neural network models [20]. Galbusera [12] defines deep learning as "the branch of machine learning which employs methods involving multiple layers of processing units, with the final aim of being able to capture different levels of abstraction". Thus, deep learning describes a family of neural network models [20].

### 2.3 Wrong Use of the Term AI

AI is a term, which can intuitively be understood by most people. As a result, people often use the term 'AI' when instead it would be much more precise to use the term machine learning or deep learning [12]. As machine learning and deep learning are subcategories of AI, this is as such not incorrect. The problem starts when people without a foundation in computer science believe that every invention based on 'AI' has to do with machine learning or deep learning. This is a wrong assumption, as the discipline AI had come up with a lot of tool sets, which have absolutely nothing to do with machine learning (see Figure 1). This kind of confusion can even be observed amongst people, that one might think would know the difference. For example Wolfgang Platz [37], who is the founder of the software testing company 'Tricentis', wrote in a 2017 article: "Given AI's buzzword status, it's all-too-commonly applied to innovations that are incredibly exciting and valuable - but are not AI because they do not involve self-learning". This statement makes it seem like Mr. Platz believes, that only innovations based on machine learning belong to the category of AI. From a scientific point of view this is not correct. We believe that it is due to the poorly defined use of the term AI that there is a lot of confusion about what AI is and what it is not.

### 2.4 Classification of AI Methods

Through our literature research and market analyses, we identified a sub-set of AI methods which are currently being researched for their possible use in software testing, or already in use by software testing vendors. The research field of AI is still in its beginnings and so it is to be expected that new methods will emerge, or that already invented methods become more relevant due to fine tuning or external factors such as changes in hardware resources. It could well be that the use of AI in software testing will look much different than what we are able to imagine right now.

In order to help the reader categorize the different methods of AI, we are going to classify all the presented tools into the three categories 'AI', 'Machine Learning' and 'Deep learning' (see figure 2). To give a more complete picture we further differentiate between supervised and unsupervised learning, and we also list some other methods which we consider main pillars of AI. For a detailed description of each method we suggest the works of [38] [34] [10] [12] [49], which are also the resources based on which we have made this classification. Note, that we have not included Natural Language Processing (NLP) and Visual Computing

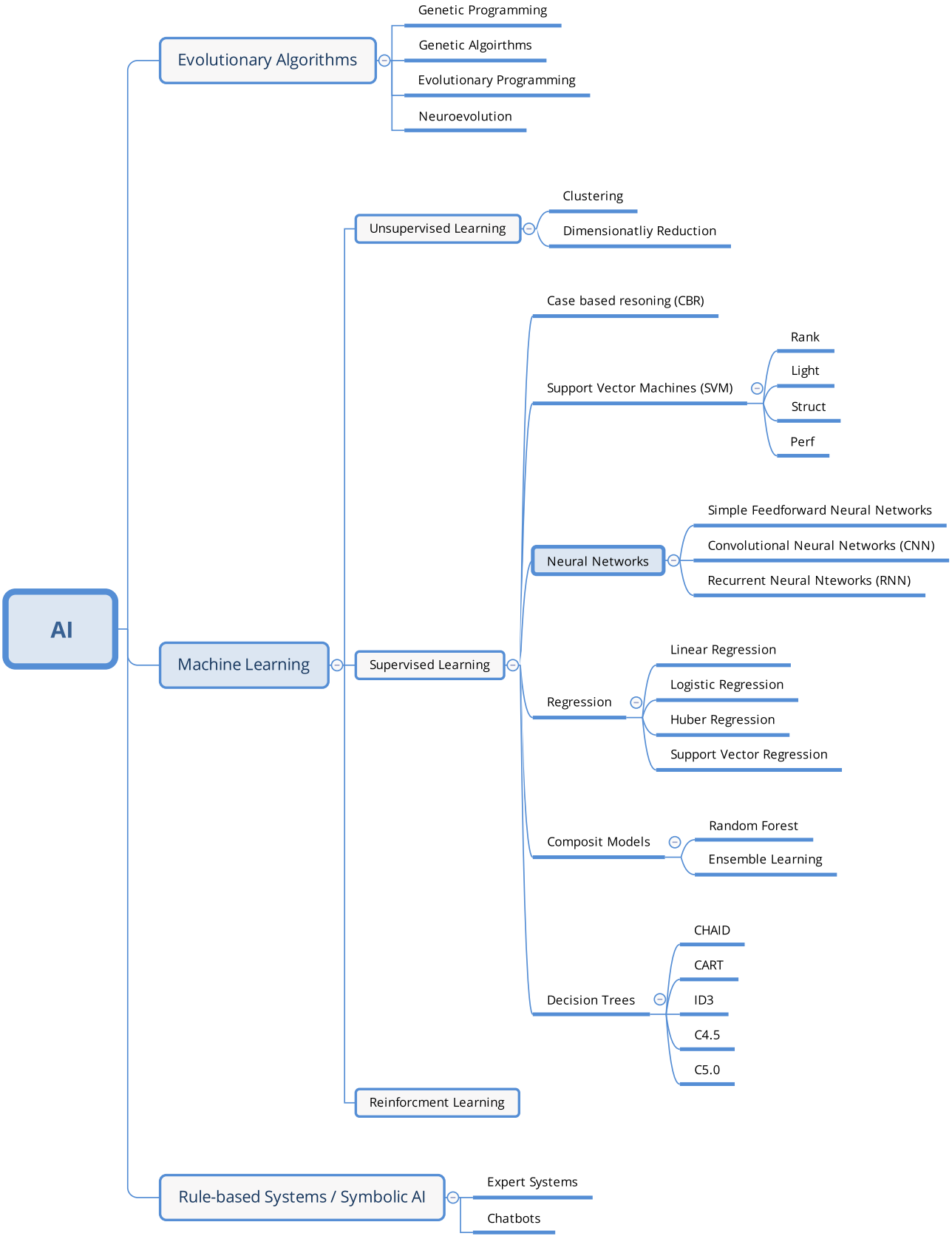


Figure 2: Classification of AI-Methods

in our classification, because we do not consider these to be methods of AI. Instead they can be understood as subfields of Computer Science which rely heavily on the use of AI methods, in particular the ones based on machine learning.

# 3 Software Testing

## 3.1 Definition

Sam Fatiregun defines software testing as "a process implemented in a controlled environment and which includes a set of activities which often require the use of tools and techniques to achieve multiple objectives such as risk measurement and risk reduction which are achievable by detecting faults and ensuring that faults are removed when possible" [11]. This elaborate definition contains many keywords which would require further specification. We will thus rely on a more simple yet precise definition given by Harunya et. al [16], who define Software testing as "the process of analyzing a software item to detect the difference between existing and required conditions". Differences between software requirements and the actual state of a software can also be called 'faults' or 'bugs' [24]. Thus the former definition basically states that software testing is the activity of detecting faults in the software.

## 3.2 History

The hopes that AI will have a positive impact on software testing are high amongst professionals of this field [23]. One reason for this is, that software testing is currently facing a lot of challenges, which are mostly due to the decreasing time span of software release cycles [37]. To get a more profound understanding of how these challenges have emerged, we will first discuss the history of software testing.

Throughout time, the models underlying software testing have drastically changed [44]. The only thing that seems to be consistent, is the fact that the time to market is continuously decreasing over time [47].

**1980 - 1990:** The widely used methodology in software development was the waterfall model and manual testing was the standard [44]. It took several years before software was ready to be released [47].

**1990-2000:** New approaches to development emerged, such as the V-model, and first automation tools were available [11]. This allowed a speed up in the time to market, which still was multiple months if not years [47].

**2000-2010:** More robust automation tools and open source frameworks for software testing became available. The old models of development were largely replaced with agile software development [44]. This new approach and the increased availability of automation tools allowed faster release cycles. It was now possible to release updates and new features within weeks [47].

**2010-2020:** The concept of continuous testing, crowd testing and cloud testing got introduced and DevOps appeared as a new approach to software development [44]. In the best case scenario, it takes less than a week to release new features [47]. The goal of

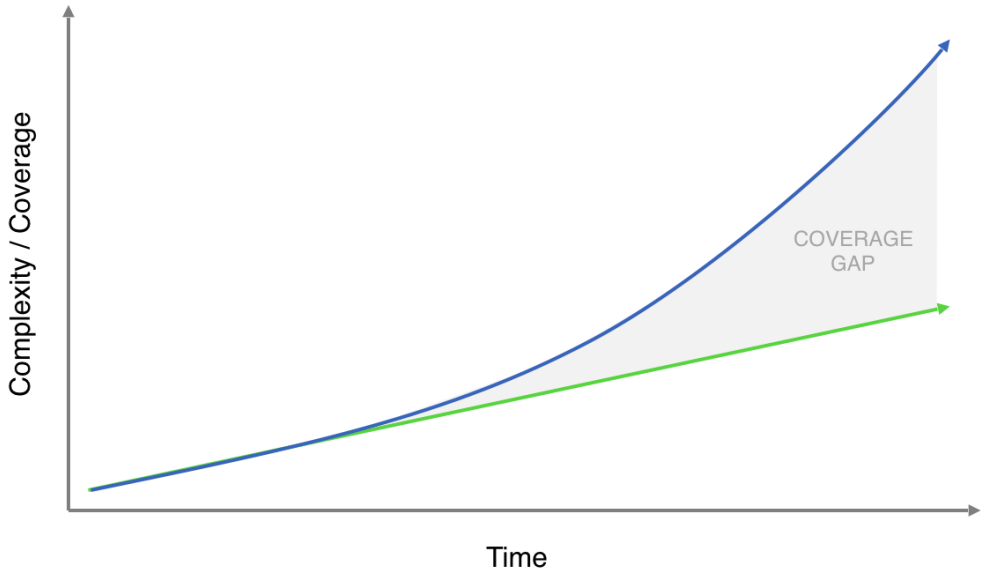


Figure 3: Coverage Gap [3]

further decreasing the time to market however is currently facing a few obstacles, which we will discuss in the next section.

**Future:** Speculations are that methods of AI will define software testing in the future [23]. Further, collaborative testing could proof as an important add-on to the current methodologies of software testing [37]. Immediate releasing of new software and features could eventually become a reality.

### 3.3 Challenges

**Coverage Gap:** One current issue in software testing is that the complexity of the features increase exponentially over time, whereas the test coverage will not be able to increase more than linearly [3]. As more features are added over time, the complexity of the system increases exponentially; due to the increasing amount of interactions between features. Test coverage, on the other hand, will only grow linearly; because the resources for testing are constant and tests are added one at a time. This results in a 'coverage gap' (see figure 3). The more release cycles a software project has gone through, the more difficult it will be to meet predefined coverage goals [3].

**Technical Complexity versus Cycle Time:** Over Time the delivery cycle time has continuously been decreasing whereas the technical complexity of software has been continuously increasing. Further, the rate at which programming paradigms are changing is increasing as well. This has caused a disruption between two expectations: one to deliver at increasing speed, the other to implement increasingly complex technology [37].

A hope to meet this disruption is seen in continuous testing, which many software companies are currently trying to implement. At some point, however, this will also not solve the

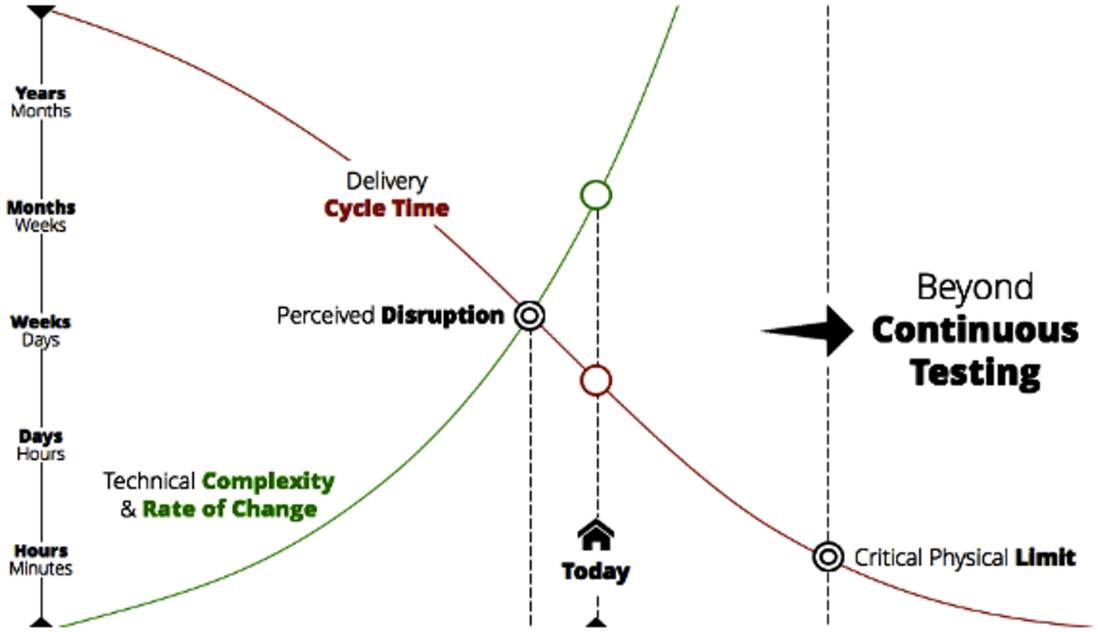


Figure 4: Technical Complexity versus Cycle Time [37]

problem. There will come the time, where the critical physical limit to how fast delivery can happen is reached (see figure 4). Platz [37] states, that the only way to move beyond this limit is by going beyond continuous testing. And this is where AI comes into the picture. According to Platz's predictions it will only be possible through the use of AI to go beyond a certain limit of delivery cycle time [37].

# 4 AI in Software Testing: Two Perspectives

A look at the history and challenges of software testing has given the impression, that the main motivation behind AI in software testing is the potential of increased software delivery speed.

To judge the potential use of AI in software testing only with regards to speed improvements, would be a very incomplete perspective however. As we will see in the following discussion, AI will hopefully not only help reducing the time to market, but also improve the quality of software, the user friendliness, the amount of faults found in beta version products, security, and more. On the downside, AI is often accompanied by certain risks such as violations to user privacy or the risk that current skills of software testing professionals will become obsolete in a new paradigm of software testing based on AI [44].

During our analysis of how AI can be applied to software testing, we have noticed that this topic can be approached either from a micro- or a macro perspective.

One way of using AI in software testing is by looking at the various tasks done by software testers. It can then be investigated which of these tasks could be either fully replaced by AI or assisted by AI. Methods that could be used for this purpose are currently researched and to some degree already implemented by software testing companies. We call this ap-

proach the micro perspective, as we are focussing on small sub-sets of software testing tasks.

Another approach would be to look at the general goal of software testing. We will discuss if and how it could be possible to apply AI to software testing in a grander scale, with out limiting one-selves to specific subtasks of software testing. Instead we will look at the main goals of software testing, and how we could use AI to achieve these goals. We consider this approach to be the macro perspective.

# 5 Micro Perspective

## 5.1 Literature Review

In the following section we present a small excerpt of research findings regarding AI in software testing. The researches which we decided to summarize here have been chosen in accordance to their quality and relevance. Further, we attempt to demonstrate that a wide variety of AI methods can be applied to various tasks of the software testing process.

### 5.1.1 Test Case Reduction

Software testing is costly and thus software testing activities should be planned carefully [11]. Ideally resources should be concentrated primarily to software modules, which have a higher risk for failure. The goal of the research conducted by Khoshgoftaar et al. [22] was to classify the modules of a software system into the three categories low-risk, medium-risk and high-risk. The classification method which Khoshgoftaar et al. decided to be most appropriate for this use case was the 'Case Based Reasoning' (CBR) algorithm(see figure 2). This method, however only allows a two-group classification, whereas [22] aimed for a three-group classification. To solve this problem, the researchers developed an algorithm which applies the CBR algorithm three times. This allowed them to classify software modules into three risk categories. The researchers mention, that instead of the CBR one could also use other classification techniques, such as decision trees, neural networks, logistic regression or genetic programming (see figure 2). Which one of these algorithm yields best results would have to be further researched. The classification model built by [22] showed promising results in terms of classification accuracy and model-stability. Classifying modules into risk categories will allow test planners to plan testing resources accordingly.

Another approach to reduce the amount of test cases is proposed by [17]. Instead of analyzing the source code or specification of the software under test, this research focused on system-level manual testing. With a simple linear regression model (see figure 2) the researchers were able to accurately predict test case failures. This was achieved with a non-code/specification-based heuristics for test selection, prioritization and reduction. The method of linear regression belongs to the category of supervised machine learning (see figure 2), thus features are required for the learning phase. The linear regression model presented by [17] is using methods of natural language processing (NLP) for feature extraction in combination with history based features (previous test execution results). The research has compared their model for predicting failure with other approaches, and concluded that NLP-based features allow an improved prediction model.

A very similar research has been conducted by [25] but with the use of the algorithm SVM Rank instead of linear regression (see figure 2). The goal of this research was to

help prioritization of test cases during manual system-level regression testing. Data used to train the model was black-box meta-data, such as test case history, as well as natural language test case descriptions [25]. The evaluation of the resulting SVM Rank model showed, that the failure detection rate could be improved significantly.

### 5.1.2 Evaluating and Re-engineering of Test Suites

According to [8], open-source test suites are often too generic for a specific testing project, which results in testing redundancies or insufficient test coverage. To solve this problem [8] present a methodology which they call MELBA (machine learning based refinement of black-box test specification). The goal of this heuristics is to help software engineers analyze the weaknesses and redundancies of existing test suites and iteratively improve them. To achieve this, [8] are making use of machine learning by applying the C4.5 Decision Tree Algorithm (see figure 2). The researchers have evaluated MELBA by comparing open-source test suites with re-engineered versions of these test-suites (created with the help of MELBA). The results show a significant increase in fault detection while requiring only modest size increase of the test suites [8].

### 5.1.3 Predicting Branch Coverage

Developers usually specify requirements regarding the minimal test coverage that should be achieved by testing activities [11]. Machine learning techniques can be used to predict the branch coverage achieved by a certain test data generation tool. This can assist test planing activities and the evaluation of automatic test data generation tools. The research of [14] investigates multiple machine learning methods with regards to their usability for this task. The three supervised machine learning techniques investigated were Huber Regression, Support Vector Regression and Artificial Neural Networks (see figure 2). One of the main challenges the researchers faced was the definition of features, which is an important phase of most learning based algorithms [14]. The results of the research show that predicting branch coverage with machine learning techniques is viable and feasible. However, future research is needed to investigate more sophisticated features, a feature selection analysis to remove redundant or irrelevant features, and a broader comparison of possible methods [14].

### 5.1.4 Automatically Documenting Unit Testing

Writing documentations of testing activities is time consuming. Automatically documenting unit test cases would make it easier for developers to maintain them. An approach called 'UnitTestScribe' [26] have proposed a methodology to automatically document unit test cases. 'UnitTestScribe' generates a documentation in natural language. The methods used to make this possible are static analysis, natural language processing, backward slicing, and code summarization techniques. According to an online survey conducted by the researchers, the resulting documentations are complete, concise and easy to read [26].

### 5.1.5 Estimating Required Testing Efforts

Planning costs and time needed to test software is an important part of test planning [11]. Machine learning algorithms can be used to facilitate predictions. Silva et. al [45] have researched the application of neural networks, support vector regression and linear regression (see figure 2) to estimate execution efforts of functional testing. Which one of

these three methods would perform best was tested in two case studies.

In machine learning it is common for most methods to utilize a cost function during the training phase [34]. The model that is being trained can learn from the output of the cost function how well it has classified a certain input. This information is then used to adjust internal parameters in a way that minimizes the output of the cost function. By modifying the cost function one can influence the final performance of a classifier [34]. When it comes to estimating efforts of testing activities, it is better to overestimate efforts than to underestimate them [45]. For this reason, Silva et. al. have applied a modified cost function during the training phase of their neural network. This allowed a predictive model biased to overestimate efforts, instead of underestimating them.

The findings of the project were not as promising as the researchers had hoped for [45]. During two case studies the performance of the three models (linear regression, support vector regression and neural network) has been evaluated and compared. The results of the case studies showed discrepancies in performance of the models for different use cases, making a clear interpretation difficult. The researchers conclude that the effort estimation problem is of high complexity and that more research is needed in this field [45].

### 5.1.6 Summary of Literature Review

Table 1 summarizes the presented works. There are many more researches in this field which we have reviewed but are not presented here. Further, we probably have not been able to retrieve every relevant research report existing about AI in software testing. Our goal for this review was not to give a detailed and exhaustive summary, but rather to demonstrate the range of possibilities. As we can see, there are many software testing tasks that can be facilitated with AI-driven methods, especially during the planning stage of software testing. The AI algorithms that have been applied in the presented researches mostly belong to the AI category of machine learning. Oftentimes there exists more than one possible algorithm for the realization of a certain goal; which AI method is most suitable has to be individually researched for each use case.

## 5.2 Commercial Software Testing Products

An increasing number of software testing product vendors are starting to offer tools that are based on AI. In the following section we will give a summary of some AI based software testing products that are commercially available.

Upon researching the market for AI based software testing tools we faced two problems; first the lack of self criticism and second the lack of transparency.

A company who wants to sell a product will advertise itself accordingly. White-papers, research papers, articles, etc. written by representatives of such companies should thus be read with caution. The term ‘AI’ is the big topic everyone speaks about nowadays, thus it has also become an important marketing strategy to advertise products as ‘using AI’. It is rare for company representatives to publish a truly honest and self-critical report about the software product they are trying to sell.

Another issue is the lack of transparency. The source-code of commercial software products is usually not publicly available. As a result, it is not possible to closely investigate how and

| What | Researchers | Year | Goal | Used AI Methods | Classification | Reference |
|---|---|---|---|---|---|---|
| **Test Case Reduction** | Khoshgoftaar and Seliya | 2004 | Classifying the modules of a software system into the three categories low-risk, medium-risk and high-risk. | Case-based Reasoning (CBR) | Machine Learning | [22] |
| | Hemmati and Sharifi | 2018 | Reducing the amount of test cases during manual system-level testing, using NLP for feature extraction in combination with history based features. | Linear Regression | Machine Learning | [17] |
| | | | | Natural Language Processing (NLP) | Multiple | |
| | Lachmann et al. | 2016 | Prioritization of test cases during manual system-level regression testing. | Support Vector Machine (SVM) | Machine Learning | [25] |
| **Evaluating and Reengineering of Test Suites** | Briand, Labiche and Bawar | 2008 | Analyzing the weaknesses and redundancies of existing test suites and iteratively improving them with the help of 'MELBA'. | C4.5 Decision Tree | Machine Learning | [8] |
| **Predicting Branch Coverage** | Grano et al. | 2018 | Using machine leaning techniques to predict the branch coverage achieved by certain test data generation tools. | Huber Regression | Machine Learning | [14] |
| | | | | Support Vector Regression | Machine Learning | |
| | | | | Artificial Neural Networks | Deep Learning | |
| **Automatically Documenting Unit Testing** | Li et al. | 2016 | Generating a documentation of the unit test cases in natural language with the help of 'UnitTestScribe'. | Natural Language Processing (NLP) | Multiple | [26] |
| | | | | Code-summarization Techniques | Multiple | |
| **Estimating Required Testing Efforts** | Silva, Jino and T. de Abreu | 2010 | Estimating execution efforts of functional testing with a modified cost function during the training phase. | Artificial Neural Networks | Deep Learning | [45] |
| | | | | Support Vector Regression | Machine Learning | |
| | | | | Linear Tegression | Machine Learning | |

Table 1: Summary of Literature Review

to what extent a product is implementing certain AI methods. A product could be largely based on non-AI methods, and nevertheless be advertised as a AI driven product; just because in some small part of the software an AI method is applied. The non-availability of source-code makes it impossible to judge to what extent a software product is based on AI.

Some popular vendors of software testing tools which claim to use AI are: Tricentis [36], Testsigma [48] and Applitools [7].

### 5.2.1 Tricentis

Tricentis is a software testing company offering software for automated testing. One small step toward AI in testing has been made by Tricentis with the creation of NEO; a neural optical engine which is basically a neural network that identifies objects on a user interface (UI) [9]. Tricentis states that this technology enables them to offer tools for faster test creation and automated test executions.

Further, Tricentis claims that they are currently researching other areas where AI could be applied to testing. Ingo Philipp [36] summarizes in a whitepaper the different areas where testers hope to get help through AI, and to what extent an AI-driven solution is feasible in the near or distant future. Philipp [36] states, that Tricentis is actively conducting research in the following areas:

- Redundancy prevention
- Risk coverage optimization
- Automated exploratory testing
- Portfolio inspection
- False-positives detection
- User experiences analysis
- Self-healing automation

Except for the automated exploratory testing, Philipp believes that all these goals can be achieved with the use of AI today already. Tricentis does not disclose any detailed information about their research activities. It is unclear, to what extent Tricentis is performing research in the mentioned areas, what AI methodologies they use and what the findings are.

### 5.2.2 Testsigma

Testsigma is a company offering cloud-based automation testing tools for continuous testing. They claim that they offer an "AI-driven continuous testing platform for continuous delivery". Testsigma states that they use AI to "create stable and reliable tests faster than ever and to speed-up the maintenance of [..] automated tests" [48].

Unfortunately, we were not able to identify how exactly their product is based on AI. Testsigma is a very good example for a company that claims to be using AI, but is not transparent about the methods used. This makes it impossible to evaluate how much AI Testsigma is really applying to their product.
Testsigma [48] claims, that they are:

- Using natural language processing to write stable and reliable automated tests, faster than ever
- Using AI to suggest affected or relevant test to be executed during regression testing
- Using AI to identify other tests that might have been affected in case of a test failure
- Using AI to help identifying all relevant test cases to be included in a test run

In addition to these points, the CEO of the company posted an article on Testsigma's Website about how AI can be used in software testing [33]. He does mention some interesting points about how AI could facilitate testing, however these points are all theory based. The article does not disclose if Testsigma is researching or applying any of the suggestions made.

### 5.2.3 Applitools

Applitools is developing software for functional app testing through the UI. One of the features they offer is called 'AI-based image comparison' [2]. Applitools supports a specific approach to regression testing in app development, in which the state of the UI is captured before and after a change is made to the software. Ideally, the only changes identified should be the ones that were purposefully made. Any other changes that were not intended are then classified as faults. The task of comparing the UI before and after the change can be done manually. This however, requires a lot of time and a human eye is prone to oversee details. To make the task more efficient and complete, Applitools is applying AI computer vision algorithms to automatically identify changes on the UI before and after the software update [1]. This could also be done with a simple pixel-based approach, and in fact there are numerous open-source programs for automated testing available which are doing exactly this [31]. The problem with these pixel-based approaches is that they have a large false positive rate if the browser version or hardware change over time. The advantage of using AI instead of a pixel-based approach is that even if the browser or display size changes, faults are ideally still correctly identified[1].

In an evaluation study of their product, Applitool conducted a hackathon during which 288 quality engineers from 101 different countries participated [7]. According to the project report, the participants were able to create test cases 5.8 times faster and test code stability increased 3.8 times [7]. With the increase of efficiency the software testers were able to produce higher coverage leading to more faults being identified. The results of this report sound very promising. Since the evaluation has been conducted by Applitools itself, the findings have to be interpreted with caution. The report results have so far not been published in a scientific magazine, thus it is currently unknown if the evaluation would satisfy scientific standards.

# 6 Macro Perspective

## 6.1 Goals

Paul Jorgenson [19] defines the two main goals of software testing as:

- To make a judgement about the quality and acceptability of software
- To identify and fix faults in software

To reach these goals, software developers have come up with a vast amount of methodologies which might not necessarily be suited for the usage of AI methods. However, as long as the main goals of software testing are reached, there is no reason to stick to current ways of doing things. For making the maximum use out of AI in software testing, it is crucial to investigate if methods can be replaced by new ones which are better suited for the use of AI. What has worked so far might not make much sense in future with the new possibilities AI is making possible.

## 6.2 Neural Networks

John Kelleher [20] states that most of the AI breakthroughs that we hear about in the media are based on deep learning. Many new inventions such as self driving cars, speech recognition, YouTube suggestions of video clips and many more are based on deep learning. Given the potential this methodology has, it seems logical to investigate whether deep learning could be applied to software testing as well.

In this following section we will present the illustrations of two 'professionals' about the potential use of neural networks in software testing. To understand better why the ideas of these two experts cannot be applied to reality, we are going to give a practical example of how a simple feedforward neural network works. We will then discuss what is wrong with the presented ideas and what is currently stopping us from using neural networks to replace human software testers.

### 6.2.1 The Statements of Two Experts about the Potential Use of Neural Networks in Software Testing

Jason Arbon [4] is the CEO of an app testing company called 'test.ai' and founder of the 'AI for software testing association' (AISTA). In an interview at the Starwest conference hosted by Techwell in 2017, he said:

> "The thing that testers don't realize is that AI is perfectly suited to replace testing activities. [...] All you need is the input and the output. If you have those things, guess what you can do? Train a machine to do it. That's literally the fundamental thing about machine learning. So what do testers do? [...] They commit test inputs, and they check the outputs. And what are they documenting all day? [...] Test inputs, and [...] the expected outputs. So the only question really is, how much of that data do you need to train an application? [...] Out of all the professions that are most in need of help for automation, [software testing] [..] is also the most ripe one for automation with AI."

Another interesting explanation of how neural networks can be used in software testing is given by Prashant Kumar [24]. He has over 14 years of experience in the area of QA and Management in software testing. In 2018 during a live webinar about 'ML and AI in software testing' he said:

> "What is software testing? [...] Testing means applying certain kind of input, doing some kind of process, and finding some output [...]. Actual output and expected behavior, if they both are matching, then we'll say our tests are passing. This is known as software testing. And, what is software testing using AI? We do the same kind of thing wherein we have input, we have an inner

> layer wherein we provide different kind of data, parameters, and then we have a hidden layer which does the different kind of processing of data [...]. So it is same. We have set of input, and some actions are taken, [...], and then we have output. [...] If the expected output is equal to the actual behavior then we say tests are passing, [...] if at all there is any difference then the tests are failing and we report it as bad. This is what is software testing using AI" [sic!].

These two statements leave the impression that human software testers might soon be replaced by AI with the use of neural networks. Realistically, it sounds too good to be true. As we will see in the following discussion, neural networks cannot be applied to software testing in the way it has been described by Arbon [4] and Kumar [24].

### 6.2.2 A Practical Example of How Neural Networks Work

Presenting the theory of neural networks in detail would be far too extensive for the intended scope of this paper. We will thus only give a practical example of how to construct a simple feed forward neural network. There are other types of neural networks as well (see figure 2), the basic functionality of neural networks however can be best illustrated with the simple feed forward neural network [20].
The process of creating and using such a neural network can be divided into the following phases [34]:

1. Defining the classes
2. Feature definition
3. Getting labeled training data
4. Training the network
5. Using the network

We are going to give a short descriptions of how we would construct a neural network that serves as an email spam classifier. The goal of this neural network is to classify emails as 'spam' or 'no spam'.

**Defining the Classes:** The number of nodes in the output layer of a neural network equals the amount of classes [34]. In our example classification problem we have two classes, 'spam' and 'no spam'. Thus we would design a neural network with two nodes in the output layer (see figure 5); each one of the output nodes representing a class.

**Feature Definition:** First we have to define features which will allow the neural network to classify new emails into the two categories, 'spam' and 'no spam'. When defining features, it makes sense to analyze what the indicators are that help human beings to make the classification [34]. Possible features could be word counts of certain key words such as 'money', 'bank', 'heritage', 'help', ... each one of these word counts could be one feature. The amount of features we define will equal the number of nodes we design in the input layer (see figure 5). The number of hidden layers and nodes within each hidden layer can be freely chosen by the network designer [34].

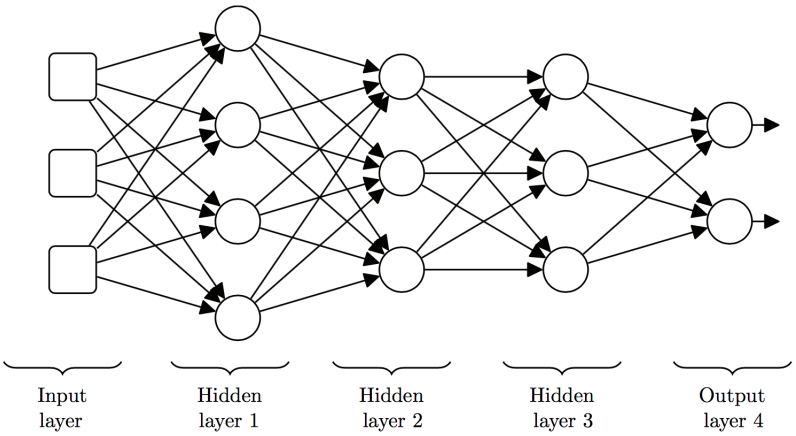


Figure 5: Topological Illustration of a Simple Feedforward Neural Network [20]

**Getting Labeled Training Data:** Next we need labeled data, which we will later use to train the network. Getting the data required for the training phase is often the biggest obstacles and can be very time consuming, especially if data has to be labeled manually [34]. In our case of a spam classifier, we would need emails which are labeled as 'spam' and 'no spam'. For each of these emails we would place the features values into a 'feature-vector' that contains a numeric value for each feature. Doing this for each email that we have already classified into 'spam' and 'no spam', we will get a feature matrix (every row containing the feature values of one email), and an output vector (each entry containing the numeric classification of the email).

**Training the Neural Network:** This training data can now be used to train our network. During this process, the network will adjust some randomly initialized values of the connections between each network layer in a such a way, that the network will eventually classify most emails of the training data correctly [20].

**Using the Neural Network:** Once we have a trained network, we can take any email that we want to classify, extract the feature values of this email (e.g. word counts of our defined key words) and put this feature vector at the input of the neural network. This will result in the network calculating the output, which is in fact the classification result. As we have mentioned already, each output node represents a class. The values of the output nodes tells us how likely an email belongs to the class 'spam' or 'no spam'. The correctness of this classification depends on multiple factors such as network design, used features, amount of data used to train the network and distribution of the data [34].
Now that we have seen how a neural network can be used to solve a classification problem, we look further into the question of whether a neural network can be used to replace human software testers. Generally it is much easier to prove things wrong than right. So we will start by showing that using a neural network in the way it has been suggested by Arbon [4] and Kumar [24] is not possible.

### 6.2.3 What is Erroneous with the Ideas Presented by Arbon and Kumar?

Without being aware of it, Arbon [4] is actually explaining to us in his interview how we could train a network to replace the software that is under test. The training dataset

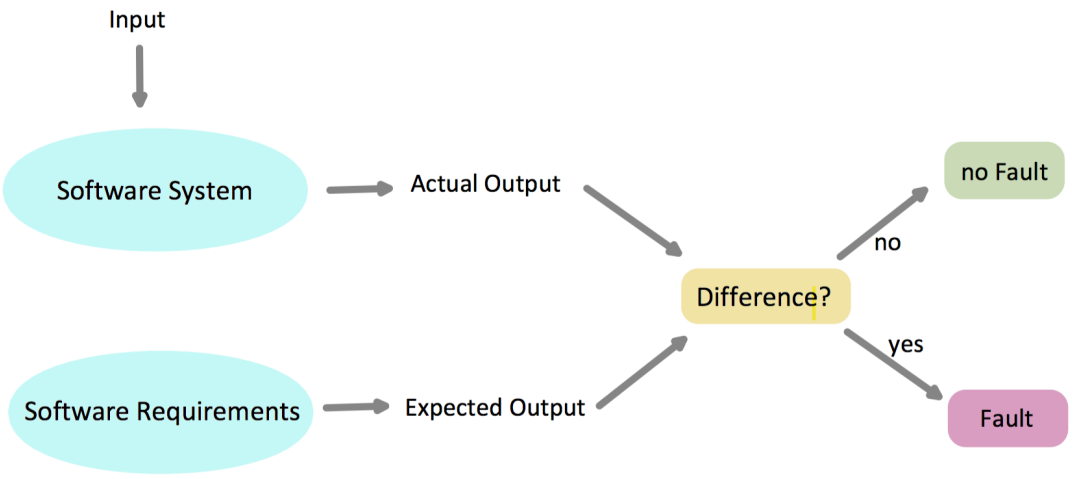


Figure 6: Classification during Software Testing

that Arbon describes consists of the inputs that testers commit, and the expected outputs. Arbon suggests that we use this data to create our training data set. Were we to train a neural network with such data, we would end up having a neural network that has memorized what a software program should be producing as an output in response to a certain input. Assuming we had data for every possible scenario of input, our trained network would not only be able to replace the software, but it would also do so without any faults in the output. This is the case, because the network has been trained to produce the expected outputs. Arbon's idea could lead to a new paradigm of software development; one that will not require functional software testing. Thus, the realization of such a network would in fact solve the problem of software testing indirectly, because if there are no faults in the first place we do not need to test for them.

Whether or not the realization of such a neural network is feasible and worth the effort is another question which we are not going to discuss in this paper. Arbon fails to give a solution to the problem of how we could apply AI to software testing. He presents an interesting idea of how to use AI in software development, but the network he describes will not be of use in software testing activities.

With the statement of Kumar [24] it is more difficult to understand what exactly his idea is. We understood from his explanations that in software testing we have input and output, and in neural networks we have input and output, and as a conclusion we can just use neural networks to do what human software testers are doing. Unfortunately, Kumar does not illustrate how he comes to this conclusion.

A human software tester is giving the software system some input, and observes the output. If the actual output is the same as the expected output, then the test passes. If not, then the test fails (see figure 6) [11]. So the tester is making a classification of the output into the two classes 'no difference=no fault' and 'difference=fault'.
As we have shown in section 6.2.2, a neural network takes an input and performs a classification of this input based on what has been learned during the training phase. A human tester on the other hand is making a classification of the software output. This

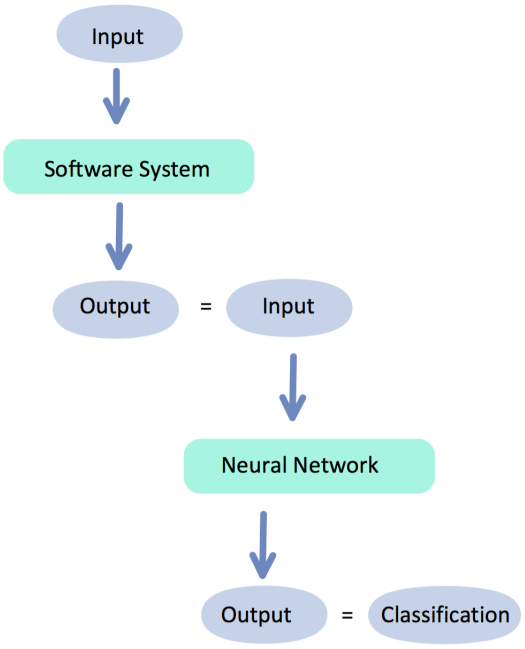


Figure 7: Neural Networks for the Classification of the Software System Output

output is produced by the software that is being tested as a response to a series of inputs. It is true that in both cases we have an input and an output. The classification in software testing however, is made on the output, where as in neural networks it is made on the input.

### 6.2.4 Two Obstacles that Stop Us from Replacing Testers with Neural Networks

We could in theory take the output of a software system, extract meaningful features and use these values as an input for a neural network, which will then perform the classification that is currently done by human software testers (see figure 6). In that case we first had an input to the software system, then an output of the software system, then the feature extraction from this output which becomes the input of the neural network, and finally the output of the neural network which is the classification (see figure 7).
There are two obstacles, which prevent us from doing this:

1. How do we get a representative and large training data-set?
2. Which features of a software system output will help the neural network make the classification?

**Getting a Representative and Large Data-set:** Neural networks belong to the group of supervised learning methods and therefore we need labeled data in the training phase [34]. Getting this data is often the biggest obstacle in deep learning, not only in the area of software testing [20]. In our scenario however, we have some additional challenges.

What we would need is a very large dataset of software system outputs which are each labeled with 'fault' and 'no fault'. This dataset would have to be a representative for

all possible outputs. Only data which resembles something that the network has seen before in the training phase can be classified meaningfully [34]. Further, we would also have to ensure that the data-set is large enough, so that our trained network will generalize well to data that has not been used in the training phase [34]. How will we get this data?

The question arises, whether we could not just have all software testers save an entry in a cloud based database each time they assign the label 'fault' or 'no fault' to a specific software output? Like this we would have a constantly growing dataset of labeled data. If all software testers worldwide would share their data of how they classified certain software outputs as 'fault' or 'no fault' on a cloud, then we would end up having a lot of data. The resulting dataset could then be made available open-source and machine learning engineers could use it to train their models.

The problem here is that a bug is not just a bug [40]. A certain output could be a fault in one software project, but not in another. Further, it could be a fault at some stage within the life cycle of a software project, but not anymore at a future point. According to our definition of software testing (see section 3.1) it is the difference between the existing and the expected output that defines if we have a fault in the code or not [16]. That which is expected depends on the specified requirements, which are different for each software project and can also change throughout the life cycle of a project itself.

**Feature Definition:** Another problem arises with the task of defining meaningful features. A neural network can only work with numerically coded feature values and therefore it is necessary to define simple features whose values can be represented numerically [34]. When we define features, we usually look for traits which help a human intellect to perform the classification. According to Andrew Ng, [34] these will also be the features which allow a neural network to perform a meaningful classification. One very obvious indication that there is a fault in the software is a program crash. One possible feature could therefore be, whether the outcome has led to a software crash or not. Obviously it is no problem to find a numerical coding for this feature (1=crash, 0=no crash).

What if the program does not crash in response to a certain input, but contains a fault nonetheless? What key indications of an output do we have that helps human testers label the output as 'fault' or 'no fault'? For human intellect, it seems obvious that regardless of software requirements the output shown in figure 8 indicates a bug. It is however, very difficult to find simple key indications that make a human tester do this classification. The knowledge behind the classifications performed during software testing is very complex, and thus it seems difficult to deduct key information that could be used as features.

### 6.3 The Oracle Problem

That which is currently stopping technology from fully replacing human testers with AI methods is the oracle problem. Barr et al. [6] define the oracle problem as "the challenge of distinguishing the corresponding desired, correct behavior from potentially incorrect behavior". As we will see in the next section, there exist methods that allow fully automated testing to some extent. These techniques however, can currently only reveal faults that cause crashes or uncaught exceptions [35]. Pastore et. al [35] say that the oracle problem is extremely relevant, because solving it would enable full automation of software testing. Only if the oracle problem is solved can AI fully replace humans in

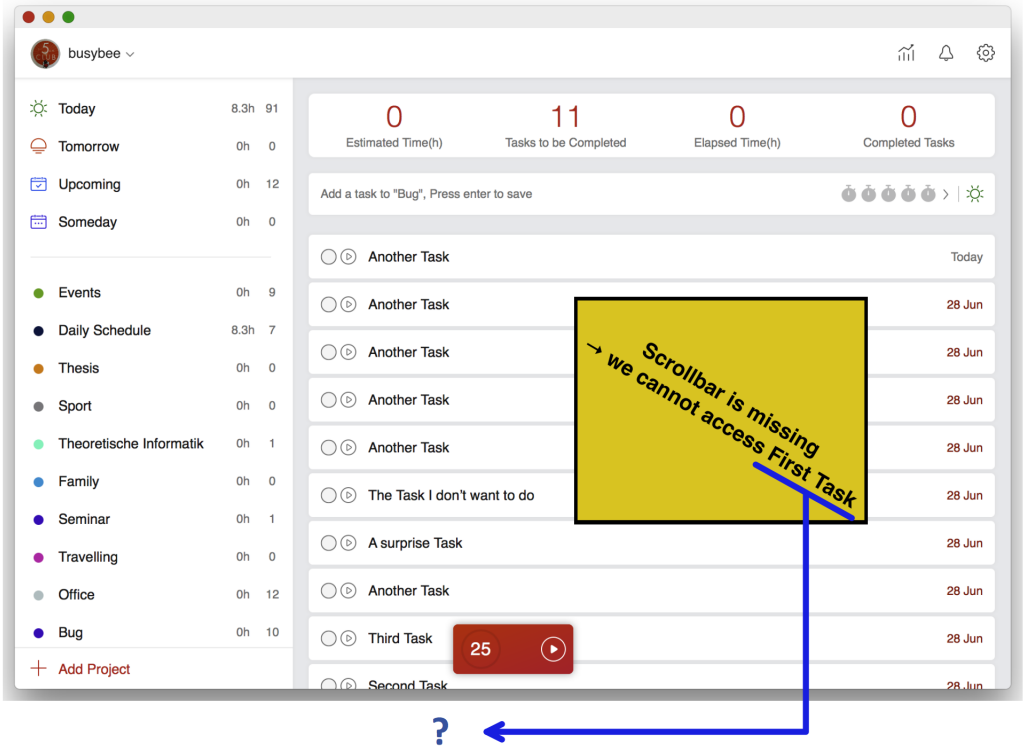


Figure 8: Example of a Bug that does not Result in a Crash

the task of software testing.

## 6.4 Monkeys and AI in GUI Testing

Despite the fact that the oracle problem could not be solved until now, there are promising methods which allow the replacement of human testers with AI at least to some extent. The infinite monkey theorem applied to GUI testing is one such example. In this section we will discuss what a 'monkey program' is, how it can be used in GUI testing, and how AI can contribute to making this method more efficient. Further, we will present two applications that implement intelligent monkey programs and we will discuss the limitations of these tools.

### 6.4.1 The Infinite Monkey Theorem

Would we train a monkey to sit in front of a computer and randomly move the cursor and press any sequence of keys, our monkey would eventually create every possible input until the system crashes. This is called the infinite monkey theorem [5]. There is no possible scenario of input that the monkey will not produce at some point, if it has infinite amount of time to randomly provide input.

There are multiple providers of such monkey programs which are used in automated software testing. In Visual Studio for example we have the 'Android Monkey' which allows automated testing of apps through a 'monkey', in the manner we have just described [46].

The main problem with this approach is that it is not very efficient [28]. It can take a very long time until the monkey coincidently provides the input that leads to a system

failure. The fact that a system crashes however, is not necessarily an indication that system requirements are not met. The monkey is likely to reveal some input sequences that lead to a system crash. Chances are high that this sequence of input would unlikely be produced by a human user of the system. According to [40] an error in the code is only a fault, if the requirements are not met because of this error. We can say that it is a general requirement for every system that it should not crash. What if a system does not crash if it is used by in an intelligent way by a human, but it crashes under the random use of a monkey? Would this system fulfill its requirements or not?

No matter what our answer to this question is; it makes sense to first focus on the input scenarios that an intelligent human is more likely going to produce. Fixing faults is costly and time consuming. Capacities are usually limited, and thus it is crucial to focus on the most relevant system failures first. A standard monkey program however is not able to do this for us. This is where AI can help making the monkey more intelligent.

### 6.4.2 Making the Monkey Smarter

There are multiple possibilities to increase the efficiency and intelligence of the monkey program which not all fall into the categories of AI.

**Identifying Buttons and Input Fields:** One such possibility is the identification of buttons and input fields. Instead of clicking randomly on the screen or providing text input without being in a text input field, the monkey could learn only to click on buttons and only to provide text after clicking on a text input field. This would result in a tremendous speed-up. Since the design of buttons and input fields follows a general standard it is possible to train a neural network to identify these. As long as the data-set used to train the neural network is sufficiently large and representative, this network could very likely identify buttons and text input fields on a majority of GUI applications. Tricentis [9], a software testing company, has developed such a program which they named NEO (neural optical engine). NEO is based on a neural network and enables the automatic detection of buttons and input fields on a GUI. Instead of using AI, one could also use other techniques of visual computing, such as edge detection or trait-based approaches [21]. Which methods are best suited would have to be further analyzed.

Giving the monkey the ability to identify buttons and input fields enables the monkey to work much faster. However, what the monkey can not do yet is choose an intelligent series of input; one that a human will likely perform. Providing a general solution to this problem that works for any program seems to be unfeasible to achieve, because every program is unique with regards to its use. Yet, throughout our research we came across two applications that claim to do exactly this; navigating the GUI in a sophisticated way without having to be fine tuned to the specific system under test. These two applications are called AutoBlackTest [29] and Sapienz [28].

**AutoBlackTest:** The University of Milano-Biocca is currently working on a learning system for automatic black-box testing on system level, called 'AutoBlackTest' [29]. Through the use of reinforcement learning (see figure 2) AutoBlackTest has been trained to interact with the GUI in an intelligent manner. The benefit of reinforcement learning in AutoBlackTest is that there is no need for the user to demonstrate to the model specific workflows of the program [29]. This makes the application usable for a vast amount of

applications without the need of pre-training for each system that is tested. Ingo Philipp [36] from the software testing company Tricentis, however, argues that AutoBlackTest will not be of much use because of the ways reinforcement learning works. Reinforcement learning cannot be compared with learning based on a labeled dataset, like it is the case with Neural Networks (see section 6.2.2). Instead, models based on reinforcement learning learn with rewards. For each action taken, the system receives a feedback and based on this feedback it will optimize its behavior in a way that leverages positive feedback [34]. Philipp claims that the application AutoBlackTest is going to optimize itself in such a way that the longest possible paths through an application are found. The longest input series however, are not the most relevant ones [36]. Intelligent users are more likely to perform simple and short input sequences, than long and arbitrary ones. For this reason, Phillipp [36] does not predict a big future for AutoBlackTest, should it become commercially available.

**Sapienz:** Facebook has developed an intelligent software testing tool called Sapienz, which is used for testing Android and iOS applications of the Facebook product family [28]. Facebook's Software Engineers claim that Sapinez allows them to test new code within hours, sometimes within minutes of the code being written [28]. Sapienz was developed with the intention that it would optimize input sequences during automated UI testing. Thus, the input sequences are aimed to be minimal in length while simultaneously maximizing coverage and fault revelation [28]. To reach these goals, Sapienz is combining multiple techniques. Mao et. al [28] name the unity of these methods a 'pareto-optimal multi-objective search-based testing approach'. In detail, this approach combines random fuzzing, systematic and search-based exploration, string seeding and multi-level instrumentation [28]. Due to the pareto optimality, long input sequences are not discarded if they are the only ones revealing a certain fault. Nevertheless, whenever possible longer sequences are replaced with shorter ones.

In their research Mao et. al [28] compare the performance of Sapienz with the automated app testing programs 'Android Monkey' [46] and 'Dynodroid' [27]. The program Android Monkey is one of the most used automated testing tools for Android apps [28]. It functions like the non-intelligent monkey program as described in section 6.4.1. Dynodroid is extending the random testing approach of the 'Android Monkey' by two feedback-driven biases: 'BiasedRandom' and 'Frequency [27]. Mao et. al [28] have tested the three software testing tools Android Monkey, Dynodroid and Sapinez with the 1000 most popular Google Play apps and compared their performances. In 70% of the tests Sapinez outperformed Android Monkey and Dynodroid in coverage and fault detection. In addition, the input sequence lengths of the test cases created by Sapinez were consistently shorter than the ones produced by the other two programs.

### 6.4.3 Limitations

Unfortunately, neither Sapienz nor AutoBlackTest are currently available to the public. Sapienz is used only company internally by Facebook and AutoBlackTest is still in its RD phase. Not having access to the source code or documentations of these programs makes it difficult to judge their true potential for software testing. Facebook claims to be testing its apps automatically through the use of Sapienz at remarkable scale [28]. In fact, Facebook goes even a step further by saying that it is also able to automatically fix bugs that Sapienz has identified with a tool called 'Sapfix' [18]. Whether or not this is true, we

cannot judge by the information available to us. Anyone can claim to be doing anything, and for marketing purposes things are often presented in a distorted manner. To what degree Facebook is using AI in software testing and debugging, we can not know as long as the software source code is not publicly available.

What we do know from logic reasoning are the following two limitations:

- Compared to the unintelligent monkey, coverage does not increase
- Only crashes can be detected

The unintelligent monkey will sooner or later produce every possible input that is every possible test case will eventually be created [5]. Thus, the limited random monkey already reaches maximum coverage. No matter how intelligent our monkey becomes, coverage and fault detection will not increase, if we assume that we have unlimited time. By adding the factor of intelligence the monkey does not find more faults, but it finds them faster and ideally it will find those faults first which matter the most.

The second limitation goes back to the oracle problem [35]. As long as no solution to the oracle problem is found, our monkeys will only be able to detect faults that lead to a system crash of the program. Both, Sapienz and AutoBlackTest, cannot test a program to its completeness, because faults do not necessarily lead to a crash [40]. To find all types of faults, additional human testing is necessary.

# 7 A New Paradigm of Software Testing

AI is penetrating every sector of IT, which will result in major changes that influence one another [20]. Even if we would not use AI in software testing at all, software testing would have to change nevertheless, because other disciplines such as software development, debugging, non-ethical hacking, etc. are starting to implement AI.

**AI in Software Development:** In section 3.2 about the history of software testing we have seen that the advances of software testing are marked by increasing speed of delivery time. One way of decreasing the time to market is by speeding up the process of software testing with the use of AI. Another way is by facilitating the process of software development with the use of AI [32]. As a result software testing will not be required to the extent it is at present. If a fault is not created in the first place, then there is no need to test for it.

**Testing for AI Software:** Software testing also has to adjust to new software testing requirements emerging from the use of AI in programs. Software that is based on AI differs from regular software to such an extent, that traditional methodologies of software testing might not be sufficient or obsolete with these new kinds of code [13]. Thus, software testing needs to adjust and come up with new testing approaches suited for AI software.

**AI and Security:** One of many reasons why we do testing is security. Especially with web applications it is crucial that the code does not break, as hackers can and will make use of these security loop holes. AI is already used to identify spam emails, phishing websites, etc. so we can hope that security issues will decrease with the use of AI [30]

[50] [41]. Unfortunately, the defense system is not the only one that is getting smarter. The attacker will use AI as well, and thus the requirements for security are currently higher than ever before. Software testing has to adjust so that it can meet the increasing demands for security.

Traditional approaches to software testing might not make sense anymore with the new requirements. Further, we might want to replace some traditional approaches to testing with new ones, which are especially well suited for the use of AI in software testing.

# 8 Conclusion

There a multiple ways how AI can facilitate software testing. Looking at the topic from a micro perspective there are many sub-tasks in software testing which can be assisted or even fully replaced by methods of AI. As we have seen in our literature review, most of these methods fall into the category of machine learning. Further, multiple software testing companies claim to be using AI in their products, unfortunately these companies do not disclose which AI methods they are implementing. Labeling a product as 'AI-driven' is a good marketing strategy. As long as the source-code is not publicly available, it is difficult to judge to what degree a program is really implementing AI methods.

Approaching the topic from a macro perspective we conclude that a complete replacement of human testers by AI is most likely not going to happen any time soon. We have discussed several obstacles which prevent this from happening. The biggest one is in our opinion the oracle problem. Teaching a machine what is considered a fault and what not is especially difficult in the field of software testing, because the knowledge used for this classification is very complex. Further, the definition of what a fault is and what not depends on the specified requirements, which differ for each software project.

Despite these limitations, there are AI-based programs such as Sapienz or AutoBlackTest which claim to test a GUI automatically without the help of human input. These tools are based on the infinite monkey theorem and AI methods, which allow the monkey program to interact with the GUI in an intelligent manner. The big drawback of this approach is that only faults that result in a system crash can be detected. An intelligent monkey program will no doubt be of great help when testing a software that has a GUI; but it will not be sufficient. Unless the only requirement of a software is that it should not crash, testing with an intelligent monkey will be incomplete and additional testing by humans is necessary.

How exactly and to what extent AI will penetrate the discipline of software testing is currently unknown. One thing that is sure though, is the fact that software testing will have to change. Since AI is not only influencing software testing but many other areas of IT as well, it is unavoidable that software testing will have to adjust to new requirements. Further, the ways software testing is currently done might simply not allow a maximum benefit from the use of AI methods. To fully leverage the potential of AI, it might proove necessary to replace current software testing approaches with new ones that AI can better be applied to. We thus predict a new paradigm of software testing; one that addresses new requirements and maximizes the use of AI in software testing.

# References


[1] Applitools. *Applitools Features*. URL: https://applitools.com/features/# (visited on 06/20/2020).

[2] Applitools. *Pricing*. URL: https://applitools.com/pricing/ (visited on 06/20/2020).

[3] Jason Arbon. *AI for Software Testing*. Mar. 2, 2016. URL: https://medium.com/app-quality-and-testing/ai-for-software-testing-44052eb0d834 (visited on 06/22/2020).

[4] Jason Arbon. *The Future of Software Testing with AI*. TechWell. Aug. 21, 2017. URL: https://youtu.be/MR6GVsWoPak (visited on 05/21/2020).

[5] Christopher R. S. Banerji, Toufik Mansour, and Simone Severini. "A notion of graph likelihood and an infinite monkey theorem". In: *Journal of Physics A: Mathematical and Theoretical* 47.3 (2013).

[6] Earl T. Barr et al. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* 41.5 (May 1, 2015), pp. 507–525.

[7] Addie Ben-Yehuda. *The Impact of Visual AI On Test Automation*. Applitools. 2020. URL: https://applitools.com/blog/visual-ai-webinar/ (visited on 06/21/2020).

[8] Lionel C. Briand, Yvan Labiche, and Zaheer Bawar. "Using Machine Learning to Refine Black-Box Test Specifications and Test Suites". In: *2008 The Eighth International Conference on Quality Software*. IEEE, 2008.

[9] David Colwell and Michael Keeley. *Introducing NEO - AI driven test automation*. Tricentis. 2020. URL: https://www.tricentis.com/resources/introducing-neo-ai-driven-test-automation/ (visited on 06/21/2020).

[10] Wolfgang Ertel. *Grundkurs Künstliche Intelligenz: eine praxisorientierte Einführung*. 4th ed. Wiesbaden: Springer Verlag, 2016.

[11] Mr Olusola Samuel Fatiregun. *A Practical Learning Guide to Software Testing*. CreateSpace Independent Publishing Platform, 2017.

[12] Fabio Galbusera, Gloria Casaroli, and Tito Bassani. "Artificial intelligence and machine learning in spine research". In: 2.1 (2019).

[13] Jerry Gao et al. "Invited Paper: What is AI Software Testing? and Why". In: *International Conference on Service-Oriented System Engineering*. IEEE, 2019.

[14] Giovanni Grano et al. "How high will it be? Using machine learning models to predict branch coverage in automated testing". In: *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation*. IEEE, 2018.

[15] B. Hailpern and P. Santhanam. "Software debugging, testing, and verification". In: *IBM Systems Journal* 41.1 (2002), pp. 4–12.

[16] B. Harunya, R. Deepa N., and S. Dr.Gunasekaran. "Survey on Fault Tolerance and Residual Software Fault of the System by Using Fault Injection". In: *International Journal of Scientific and Engineering Research* 6.4 (2015).

[17] Hadi Hemmati and Fatemeh Sharifi. "Investigating NLP-Based Approaches for Predicting Manual Test Case Failure". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018.

[18] Yue Jia, Ke Mao, and Mark Harman. *Finding and fixing software bugs automatically with SapFix and Sapienz*. Facebook Engineering. Sept. 13, 2018. URL: https://engineering.fb.com/developer-tools/finding-and-fixing-software-bugs-automatically-with-sapfix-and-sapienz/ (visited on 06/21/2020).

[19] Paul Jorgensen. *Software testing a craftsman's approach*. Boca Raton, FL: CRC Press, Taylor & Francis Group, 2014.

[20] John D. Kelleher. *Deep learning*. The MIT press essential knowledge series. Cambridge, Massachusetts: The MIT Press, 2019.

[21] Jochen Kerdels and Gabriele Peters. *Interaktive Systeme 1: Konzepte und Methoden des Computersehens - Kurseinheit 3 - Verarbeitung visueller Signale*. Hagen: FernUniversität in Hagen, Fakultät für Mathematik und Informatik, 2019.

[22] Taghi M. Khoshgoftaar and Naeem Seliya. "Three group software quality classification modeling using an automated reasoning approach". In: *Series in Machine Perception and Artificial Intelligence*. WORLD SCIENTIFIC, 2004, pp. 133–174.

[23] Tariq M. King et al. "AI for Testing Today and Tomorrow: Industry Perspectives". In: *2019 IEEE International Conference On Artificial Intelligence Testing*. IEEE, 2019.

[24] Prashant Kumar. *Live Webinar - AI and ML in Software Testing — Future of Software Testing — AI and ML Video Tutorial*. Apr. 15, 2018. URL: https://youtu.be/XTO3MBJf_-Y (visited on 06/21/2020).

[25] Remo Lachmann et al. "System-Level Test Case Prioritization Using Machine Learning". In: *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2016.

[26] Boyang Li et al. "Automatically Documenting Unit Test Cases". In: *2016 IEEE International Conference on Software Testing*. IEEE, 2016.

[27] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. "Dynodroid: an input generation system for Android apps". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. ACM Press, 2013.

[28] Ke Mao. *Sapienz: Intelligent automated software testing at scale*. May 2, 2018. URL: https://engineering.fb.com/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/ (visited on 05/16/2020).

[29] Leonardo Mariani et al. "AutoBlackTest: Automatic Black-Box Testing of Interactive Applications". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012.

[30] Ericsson Marin, Mohammed Almukaynizi, and Paulo Shakarian. "Reasoning About Future Cyber-Attacks Through Socio-Technical Hacking Information". In: *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2019.

[31] Alex McPeak. *Making Your Site Pixel Perfect with Visual Regression Testing*. Aug. 2, 2018. URL: https://crossbrowsertesting.com/blog/visual-testing/visual-regression-testing/ (visited on 06/16/2020).

[32] Farid Meziane and Sunil Vadera. *Artificial Intelligence Applications for Improved Software Engineering Development*. Business Science Reference, Feb. 25, 2011.

[33] K. R. Naidu. *AI-Driven Test Automation*. Testsigma. Jan. 11, 2018. URL: https://testsigma.com/blog/ai-driven-test-automation/ (visited on 05/19/2020).

[34] Andrew Ng. “Machine Learning Yearning”. 2018. URL: https://www.dbooks.org/machine-learning-yearning-1501/ (visited on 06/18/2020).

[35] Fabrizio Pastore, Leonardo Mariani, and Gordon Fraser. “CrowdOracles: Can the Crowd Solve the Oracle Problem?” In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. Luxembourg: IEEE, Mar. 2013, pp. 342–351.

[36] Ingo Philipp. *AI in Software Testing: A Reality Check*. Tricentis, 2018. URL: https://www.tricentis.com/resources/ai-in-software-testing-reality-check/ (visited on 05/28/2020).

[37] Wolfgang Platz. *What's beyond continuous testing? AI*. SD Time. Sept. 15, 2017. URL: https://sdtimes.com/ai/whats-beyond-continuous-testing-ai/ (visited on 05/12/2020).

[38] David L. Poole and Alan K. Mackworth. *Artificial Intelligence. Foundations of Computational Agents*. 2nd ed. Cambridge University Pr., Sept. 25, 2017.

[39] Elaine Rich, Kevin Knight, and Shivashankar B. Nair. *Artificial Intelligence*. 3rd ed. Gardners Books, 2009.

[40] Jeremias Rossler, Alessandro Orso, and Andreas Zeller. “When Does My Program Fail?” In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011.

[41] Sanjiban Sekhar Roy et al. “Spam Email Detection Using Deep Support Vector Machine, Support Vector Machine and Artificial Neural Network”. In: *Soft Computing Applications*. Springer International Publishing, 2017, pp. 162–174.

[42] A. L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers”. In: *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229.

[43] Andreas Scherer, Hermann Helbig, and Wolfram Schiffmann. *Kurs 1834: Künstliche Neuronale Netze*. Hagen: Fernuniversität in Hagen, Fakultät für Mathematik und Informatik, 2019.

[44] Mukesh Sharma. *Software Testing 2020*. Taylor & Francis Inc, 2016.

[45] Daniel G. Silva, Mario Jino, and Bruno T. de Abreu. “Machine Learning Methods and Asymmetric Cost Function to Estimate Execution Effort of Software Testing”. In: *2010 Third International Conference on Software Testing*. IEEE, 2010.

[46] Android Studio. *UI/Application Exerciser Monkey*. URL: https://developer.android.com/studio/test/monkey (visited on 06/28/2020).

[47] Testim. *How AI is Changing the Future of Software Testing*. June 11, 2018. URL: https://www.testim.io/blog/ai-transforming-software-testing/ (visited on 04/21/2020).

[48] Testsigma. *AI-Driven Automated Testing Tool for Continuous Testing*. URL: https://testsigma.com/ai-driven-test-automation (visited on 05/28/2020).

[49] Pradnya A. Vikhar. “Evolutionary algorithms: A critical review and its future prospects”. In: *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*. IEEE, 2016.

[50] Erzhou Zhu et al. “OFS-NN: An Effective Phishing Websites Detection Model Based on Optimal Feature Selection and Neural Network”. In: *IEEE Access* 7 (2019).

# A Unedited Citations

**Unedited Excerpt of Jason Arbon's [4] Interview at the Starwest Conference hosted by Techwell in 2017:** "The thing that testers don't realize is that AI is perfectly suited to replace testing activities. And the reason is, fundamentally AI is just a way to train software or let software train itself, if you have a bunch of input data and you have a bunch of output data examples. So all you need is the input and the output. If you have those things, guess what you can do? Train a machine to do it. That's literally the fundamental thing about machine learning. So what do testers do? Quick Quiz. They commit test inputs, and they check the outputs. And what are they documenting all day? They are actually documenting the data in their test-case databases of how to replace themselves. Here is my test inputs, and here is the expected outputs. So the only question really is, how much of that data do you need to train application, or to train a bot to test a application, but really like of all the professions that is most in need of help for automation it is also the most ripe one for automation with AI. So AI is not this mysterious thing, it is actually a tool. But it is perfectly suited for software testing, and people are waking up to that idea."

**Unedited Citation of Prashant Kumar [24] during his Webinar about ML and AI in Software Testing in 2018:** "What is software testing? If you ask a layman or if you ask somebody who is not having any experience or if you ask somebody who is having a lot of experience, this is what you are going to get as an answer which is common from both the persons, which is: Testing means applying certain kind of input, doing some kind of process, and finding some output, and this output should be verifying this output, checking this output, that this output are as per the expectations. Means actual output and expected behavior, if they both are matching then we'll say our tests are passing. This is known as software testing. And, what is software testing using AI? We do the same kind of thing wherein we have input, we have an inner layer wherein we provide different kind of data, parameters, and then we have a hidden layer which does the different kind of processing of data. Now when I say processing of data, like for example in supervised learning we had information from past and then we compare the new data depending up on the past training data and figure our test compared or is processed in terms of the training data and then we figure out our decision making process and try to this outcome of it the output and in unsupervised learning we do not have any thing, we collect the data on our own depending on different kind of situations depending on different kind of parameters we figure our own data and we figure out our own learning and then we come up with a decision which is output. So it is same. We have set of input, and some actions are taken, or some algorithm which does the kind of processing, and then we have output, and this output, if the expected output is equal to the actual behavior then we say tests are passing, if actual and expected behavior is same the tests are passing, if at all there is any difference then the tests are failing and we report it as bad, this is what is software testing using AI."

# B Non-plagiarism Declaration

I confirm that this assignment is my own work, is not copied from any other person's work (published or unpublished), and has not been submitted for assessment either at Fernuniversität in Hagen or elsewhere. I further declare that the text, pictures and bibliography reflect the sources I have consulted. Sections with no source referrals are my own ideas, arguments or conclusions.

**Name:** Maruna Derieg
**Matriculation Number:** 6558020
**Date and Place:** 5.9.2020, Kollam

**Signature:**