Artificial Intelligence Techniques in System Testing

Michael Felderer, Eduard Paul Enoiu, Sahar Tahvili

Abstract System testing is essential for developing high-quality systems, but the degree of automation in system testing is still low. Therefore, there is high potential for Artificial Intelligence (AI) techniques like machine learning, natural language processing, or search-based optimization to improve the effectiveness and efficiency of system testing. This chapter presents where and how AI techniques can be applied to automate and optimize system testing activities. First, we identified different system testing activities (i.e., test planning and analysis, test design, test execution, and test evaluation) and indicated how AI techniques could be applied to automate and optimize these activities. Furthermore, we presented an industrial case study on test case analysis, where AI techniques are applied to encode and group natural language into clusters of similar test cases for cluster-based test optimization. Finally, we discuss the levels of autonomy of AI in system testing.

1 Introduction

The effectiveness and efficiency of system testing, which is conducted on a complete, integrated system to evaluate the systems' compliance with its requirements,

Michael Felderer
University of Innsbruck, Austria,
Blekinge Institute of Technology, Sweden
e-mail: michael.felderer@uibk.ac.at

Eduard Paul Enoiu
Mälardalen University, Sweden
e-mail: eduard.paul.enoiu@mdh.se

Sahar Tahvili
Ericsson AB, Sweden,
Mälardalen University, Sweden
e-mail: sahar.tahvili@ericsson.com

is essential for developing high-quality software-intensive systems. However, the degree of automation in system testing is still low (especially when compared to unit testing) and system tests are performed manually to a large extent. The main reasons for this circumstance are that system testing typically relies on less formal artifacts, especially requirements, than unit testing, which is directly linked to source code, also resulting in the fact that test automation is more complex on the system level.

Thus, Artificial Intelligence (AI) techniques like machine learning, natural language processing, or search-based optimization can significantly improve the effectiveness and efficiency of system testing. For instance, natural language processing has a high potential to increase the degree of automation in system testing, which typically relies on natural language requirements.

This chapter provides an overview of the state-of-the art and future trends on the application of AI techniques in different system testing activities (i.e., test planning and analysis, test design, test execution, and test evaluation). Furthermore, a concrete industrial case study is presented in which natural language processing, unsupervised machine learning, and search-based techniques have successfully been applied to automate and optimize system testing activities.

To support our concrete industrial study, we provide a project that can be used for automated functional dependency detection from natural language test specifications. The provided code is written in Python 3, so it assumes you have this installed on your local machine to execute the examples. The code repository is available at *https://github.com/leohatvani/clustering-dependency-detection*. Its README file comprises instructions on how to download and execute the code.

The chapter is organized as follows: In Sect. 2 we provide an overview about system testing. In Sect. 3 we discuss the application of AI in system testing. In Sect. 4 we present an industrial case study on the application of selected AI techniques (i.e., natural language processing, clustering, and search-based techniques) to system testing. In Sect. 5 we reflect on the levels of autonomy of AI in system testing. Finally, in Sect. 6 we conclude this chapter.

2 System Testing

According to the IEEE standard glossary of software engineering terminology [17], *testing* is the process of operating a system or component (or unit) under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component. The definition (and this chapter) refers to dynamic testing, where the system is executed (in contrast to static testing) and considers unit- and system-level testing. This classification based on the test level, where units of the system are tested either in isolation or together as the whole system, is important in testing. The IEEE standard glossary of software engineering terminology [17] defines *system testing* as testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. So system testing explicitly refers to requirements as a specification. The requirements can refer to functional or

extra-functional system properties. A system often comprises hardware and software; however, we focus only on the software testing aspects.

Conceptually, system-level testing differs in several respects from unit-level testing. The following items differentiate system testing from unit testing:

- What constitutes a test input changes and is rather input via a user interface or a system configuration.
- Testing extra-functional properties like performance, security, or usability is more important as they can typically only be tested on the system level.
- Domain knowledge becomes more important on the system level, which requires the integration of domain experts into the test process.
- Classically, system testing is the part of testing (differing from unit testing) that is not anymore in the plain responsibility of developers as domain aspects and the entire system are essential.
- Test automation is multi-faceted, often only partially possible requires specific test engineering know-how, and tests might intermediately fail.
- The oracle problem becomes more complex, i.e., deciding on a pass or fail is more difficult and often even not possible.
- System specifications are not provided in code but in informal, semi-formal or formal requirements (models).

Testing in general and system testing, in particular, comprises several activities. The generic testing process comprises the following activities:

- *Test Planning and Analysis*. This activity comprises planning of the means and schedule for test activities as well as the analysis of test artifacts to identify test conditions.
- *Test Design*. This activity comprises the derivation and specification of test cases from test conditions like coverage criteria.
- *Test Execution*. This activity comprises the preparation of the test environment and running tests on the component or system under test.
- *Test Evaluation*. This activity comprises decisions on test results and their reporting.

In the following, we present important system testing approaches and to what test activities they relate.

Risk-based testing [11] takes explicit risk assessments into account to steer all test activities. Therefore, it is linked to all test activities mentioned before. However, in practice, risk-based testing is primarily used to steer test planning (i.e., distribution of resources) and test execution (i.e., test case prioritization and selection) purposes.

Model-based testing [36] relies on explicit behavior models that encode the intended behaviors of a system under test and/or the behavior of its environment. The main focus of model-based testing lies in automated test case generation, i.e., on test design. However, test models can also support test planning and analysis, test execution, and test evaluation.

Exploratory testing [1] is simultaneous learning, test design, and test execution. It is therefore linked to test design and execution, but also impacts test planning and evaluation.

Agent-based testing [22] is defined as the application of AI-infused agents (i.e., software bots, intelligent agents, autonomous agents, multi-agent systems) to software testing problems. Most of these approaches are used for test planning and analysis as well as test design and execution.

Hardware-in-the-loop and *Software-in-the-loop* [14] use simulations to test hardware and software, respectively, especially in the context of embedded and cyber-physical systems. The focus lies on test execution.

3 Application of AI in System Testing

In this section, we first present a classification scheme for the application of AI in system testing and discuss selected approaches that are classified in the defined scheme. Then, we discuss the optimization of various test activities based on AI.

3.1 Classification Scheme

Given the current expansion of the use of AI in software testing and the large number of ways AI can be applied during software development, we aim to outline the categories of AI applications according to their application point on the system testing process, i.e., test planning and analysis, test design, test execution as well as test evaluation.

Feldt et al. [12] proposed a taxonomy that categorizes the different ways of applying AI in software engineering according to their point of AI application, the type of AI technology used and the automation level allowed. When using their classification scheme on some existing work, the authors found that AI application to software engineering focused on supporting stakeholders during the development process, but did not directly affect the source code or the runtime behavior of the systems. In this chapter, we focus on the dimension where AI techniques are applied in the software testing process.

Here the focus of the AI is to support or optimize the verification that the system as a whole can accomplish its task.

Even though AI can be applied in different ways during system testing, we argue that a simpler taxonomy focusing on the generic steps in a system-level testing process can help in the understanding of the points of applications during system-level testing. Many AI approaches are applied to more than one system testing step and the results can be combined.

The following steps where AI is applied to system testing are outlined in Fig. 1:

- *Test Planning and Analysis*. AI can be applied in this activity by determining what is going to be tested and optimizing a test plan by analysing the test artifacts created during software development (e.g., requirement documents, test specifications).
- *Test Design*. AI can be applied in this activity for automating system-level test design. This has been proposed to allow test cases to be created with less effort. The goal is to automatically find a small set of test cases that check the correctness of the system and guard against (previous as well as future) faults.
- *Test Execution*. After test cases have been generated, AI can be applied during the test selection and execution process in order to make the following determinations: which test cases to execute during regression testing, which each test will evaluate system configurations, and which test setups are available for the actual running of each test case.
- *Test Evaluation*. As the test cases execute, valuable data is generated that AI can exploit through data mining techniques to evaluate the test result and localize suspicious program behavior as well as to cluster similar and independent faults.

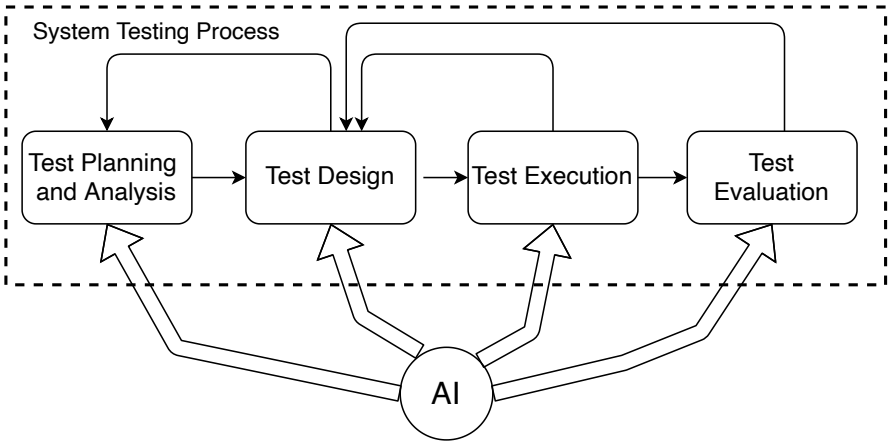

Fig. 1 Ways of Applying AI in a Generic System Testing Process. These activities are based on the generalized test process outlined by [6].

The main criterion for classifying the application of AI in system testing is the “when", i.e., which system testing activity is supported by AI. This criterion is linked to the “what", i.e., the software and test artifacts and data the application of AI depends on and refers to. This can be test cases, system interfaces, GUI elements, natural language requirements, defects, etc.

To give an overview and apply the classification scheme defined before, we provide selected approaches that vary in the types of AI techniques applied and the targeted test activity based on the classification scheme presented before.

Ramler and Felderer [29] integrate machine-learning-based defect prediction based on classifiers into risk-based testing. The goal of the approach is to predict risk

probability, which is used to plan system testing in general and the depth of testing for different system components in particular.

Tahvili et al. [32] provide a test analysis approach that derives test cases' similarities and functional dependencies directly from the test specification documents written in natural language, without requiring any other data source. The approach uses natural language processing to detect text-semantic similarities between test cases and then groups them using different clustering algorithms. The approach is further discussed in the case study in Sect. 4.

Adamo et al. [2] apply reinforcement learning for automating test design and execution of GUI testing of Android apps. The authors use a test generation algorithm to systematically select events and explore the GUI of an application under test without requiring a preexisting abstract model.

Briand et al. [7] apply search-based approaches to automate test design for stress testing of real-time systems. Genetic algorithms were used to search for the sequence of arrival times of events for aperiodic tasks that would cause the most significant delays in the execution of the target task. The fitness function was expressed in an exponential form, based on the difference between the deadline of an execution and the execution's actual completion.

Memon et al. [28] use AI plan for automated test design. For example, given a set of operations, an initial state and a goal state for a GUI, a planning system produces a sequence of operations that transforms the initial state to the goal state.

Frounchi et al. [13] use machine learning to construct an oracle for test evaluation, which can then be used to verify the correctness of image segmentations automatically.

Arcuri et al. [4] use AI to generate whole-suite system-level test cases for RESTful API web services by using the Many Independent Objective (MIO) algorithm.

3.2 Test optimization

optimizing test activities, i.e., test planning and analysis, test design, test execution, or test evaluation, using AI is a process we call *test optimization*. Test optimization goes beyond plain automation and defines an explicit optimization function. The overall architecture of a generic AI-based system testing process used for test optimization is shown in Figure 2. This approach starts with only two inputs: the choice of the representation of the problem based on the available raw data and the definition of the heuristic function. With these two, an engineer can implement AI-based system testing using optimization algorithms and obtain results.

Test optimization helps make the test process more time and resources efficient without compromising test accuracy or coverage. Eliminating unnecessary steps and automating some steps can be considered the two main strategies in an efficient test optimization process to save time, reduce errors, and avoid duplicate work.

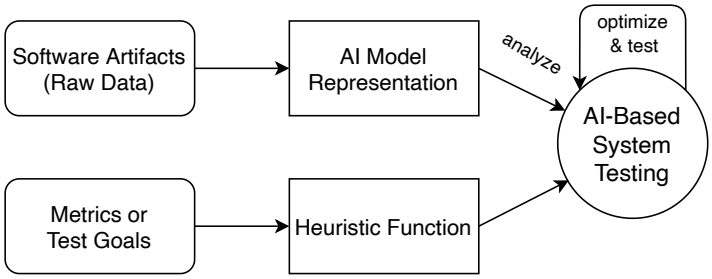

Fig. 2 Overall Architecture of AI-based System Testing Approaches adapted from [15].

Employing the test optimization in an iterative testing process (e.g., where regression testing is applied) can significantly improve system testing efficiency and effectiveness.

However, guiding a testing team to work more efficiently, supported by a proper test optimization approach, is a challenging task. Generally, optimizing a continuous testing process requires hard work and effort, in terms of designing and implementing the new solution without stopping the testing process.

In this regard, evaluating the efficiency of the investment (e.g., return on investment, internal rate of return) in all optimization process is a golden key that can guide the testing team to select a proper optimization approach. Therefore, employing artificial intelligence techniques for optimizing the testing process has received a great deal of attention recently. However, training a single artificial inference model can be an ongoing costly process since the data that feeds the AI models tends to change over time. Furthermore, there's a reciprocal relationship between big data and AI especially in a continuous testing process, where large data (e.g., test results) is generated after each execution. However, one of the main purposes of AI is to minimize the need for human intervention, which might end up having higher accuracy and less uncertainty and ambiguity.

Figure 3 illustrates an overall overview of employing AI technologies for optimization of a testing process. As one can see in Figure 3 the optimization process can be performed in three main phases:

1. **Analyze:** everything regarding the data, e.g., data gathering and data preprocessing, is being performed in this phase. The required data for applying an AI-based solution should be captured and be ready for training the AI model. The required data might be different based on the optimization goal. However, all system and test artifacts such as requirements specification, test cases (specification or script), and test results (log files) can be considered as data.
2. **optimize:** the AI model needs to be trained in this phase using the pre-processed data from the previous phase. Moreover, the hyperparameter tuning needs to be done in this phase. Tuning (also known as hyperparameter optimization) refers to choosing a set of optimal hyperparameters for a learning algorithm (see "*Improve*" arrow in Fig. 3). This improvement process (tuning) can be performed

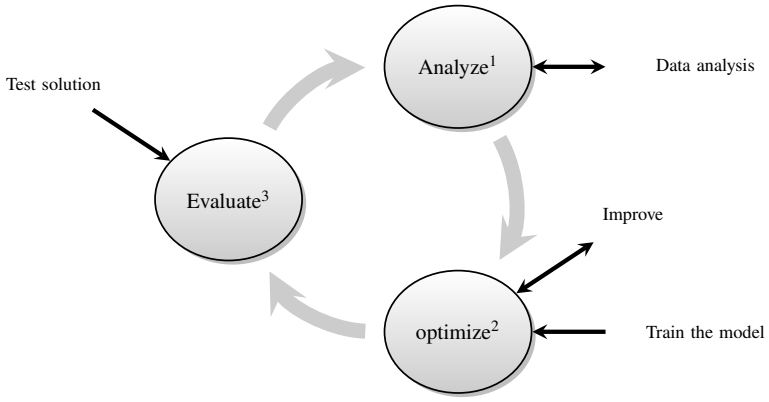

Fig. 3 Test optimization process using Artificial Intelligence.

via employing different approaches e.g., Grid search, Random search, Bayesian optimization, Population-based approaches, etc.

3. **Evaluate:** the proposed AI-based solution needs to be evaluated with respect to specific metrics in this phase.

However, in a continuous testing process, a large and new dataset might be generated, thus the optimization process should be connected to both *Analyze* and *Test* phases directly. The mentioned phases in Fig. 3 are exemplified in an industrial cases study in the next section.

Considering all mentioned issues, especially all aspects of test automation, especially in the context of regression testing, have a high potential for test optimization. In fact, instead of regressing the same test cases each time, this process can be fully automated. Therefore, test automation can be a suitable test optimization approach in large industries where some of the features of the software hardly change when a new build is made [21]. However, the maintenance of test scripts needs to be considered a recurring expense that is usually a low cost. Regression testing tasks that benefit in particular from test optimization based on AI techniques are the following:

- *Test case selection:* refers to identifying a subset of test cases for execution which has to be sufficient for achieving given requirements.
- *Test case prioritization:* ranks the test cases based on some priority score to guide testing activities.
- *Test suite minimization:* keeps the test case precise and unique and eliminates non-effective test cases which reduces regression testing time and lowers costs while maintaining quality.

In addition, utilizing *exploratory testing* by employing on-the-go testing in order to make the testing process faster and to make use of domain expertise has a high potential for test optimization. This topic touches on the different levels of autonomy of AI in system testing and is discussed in Sec. 5.

4 Industrial Case Study

In this section, we describe a replicable industrial case study on system test analysis. We first motivate it and define its industrial context. Then we present the underlying Natural Language Processing approach for detecting test case similarity. Afterwards, we discuss clustering similar test cases using unsupervised machine learning. Then, we discuss the data processing and visualization. Finally, we sketch test optimization application scenarios.

4.1 Motivation

Testing a software product in large industries is a time and resource-consuming process. Manual testing is still an essential approach to system testing, especially in safety-critical domains. In a purely manual testing process, all test cases are created and executed manually by the tester without using any automation tool. In fact, a testing team consisting of several testers creates and later executes manual test cases based on the requirement specifications (that takes the different stakeholders' perspectives into account).

System (integration) testing is recognized as one of the most complex and critical levels of testing, where the interaction of all subsystems and the entire system should be tested. Therefore, the number of test scenarios that need to be defined for testing the interaction between modules might be very high. Considering a high number of test cases that need to be tested manually, highlights the need for test optimization. However, manual test creation by different testers might lead to having similar or even duplicated test cases. Similar test cases can refer to test cases designed to test the same function or require the same system setup, preconditions, or post-conditions. Identifying similar or duplicate test cases in an early stage of a testing process can help one to apply different test optimization approaches such as test case selection, prioritization, scheduling, and test suite minimization. For example, assume that we have clusters of similar test cases. We can then use these clusters for test optimization, e.g., by eliminating some clusters or just selecting some test case samples from the clusters and ranking them for test execution.

Clustering in general is an *unsupervised machine learning* approach that scans data and groups it without any labels. Therefore, unsupervised (machine) learning can be considered a suitable approach for solving the industrial problem of identifying similar test cases and using this information to guide system testing.

However, detecting similar test cases for a large-size product is a challenging task that might suffer from human judgment, uncertainty, and ambiguity. Since the input for the similarity section is the natural-language test case specification (due to manual testing). Thus, employing AI techniques such as natural language processing (NLP), unsupervised machine learning, and deep learning can be considered a proper approach.

4.2 Industrial Context

In the following, we provide an industrial case study conducted at Bombardier Transportation (BT) in Sweden[1]. This case study focuses on finding the similarity and duplication between manual system integration test cases. It, therefore, covers a test analysis scenario, where system and test artifacts are analyzed. In this regard, the case study provided in this section can be employed for different test optimization purposes such as test case selection (selecting a subset of test cases for test execution or automation) or test scheduling (ranking the test cases for test execution). In other words, the running industrial case study in this section can easily be reused for achieving several test optimization goals simultaneously. The case study is performed in the following steps:

1. One ongoing testing project is selected as a case under study.
2. A total number of $1,748$ test specifications are extracted from the database at BT.
3. Each test case is saved in a separate file in the comma-separated values (.CSV) format.
4. Some irrelevant information such as the name of the tester, date, and time is removed from the test specification.
5. The .CSV files are utilized later as input to the natural language processing model.
6. The generated outputs of the NLP approach are then clustered.
7. The obtained clusters are later removed or ranked for the test execution.

4.3 Detecting Test Case Similarity using Natural Language Processing

As stated earlier, in a manual testing process, a large number of test specifications with short or long text are processed. The provided test specifications include the testing procedure, requirements specification, system setup, etc., which are generally subjective and also semantics-oriented. Detecting the similarity between created test cases is the problem of finding the score to which test cases have similar content. Usually, two text documents are similar either lexically or semantically if the two text documents are used in the same context or have the same meaning [26]. Due to a wide application of semantic similarity detection (e.g., text summarization [20], topic extraction [18], finding duplication [23] and text document clustering and classification [34]) it always plays a vital role in the NLP domain. The following approaches can be considered as the main solutions for detecting the similarity problem between texts:

[1] Please note that the authors have approval from the third party for the utilized case study and its code provided in the Git repository *https://github.com/leohatvani/clustering-dependency-detection*. The permission to use the code is also granted to the reader.

- *String distance algorithms* such as: token-based, edit-based, and string-matching
- *Normalized compression distance* such as: bzip2, gzip, zstd
- *Deep learning* such as: Doc2vec and SBERT

However, most of the mentioned solutions for similarity detection deliver the extracted features as input to machine learning algorithms, especially clustering algorithms. In this regard, the deep learning approach can execute feature engineering by itself compared to other existing solutions.

We especially want to highlight Doc2vec, which has been applied in our approach. It is a deep learning model for representing documents as a vector which was first proposed by Le and Mikolov [27]. Doc2vec has excellent scalability and has filled gaps of previously proposed approaches in the test analysis domain where the semantics of the words were ignored (see [34]). Doc2vec reads different text documents as input, transfers them to the high dimensional vectors, and measures the cosine distance between them. The dimension of each vector is directly related to the text size. Via applying a proper clustering method on the obtained vectors provided by Doc2vec, we can divide the similar text (test specification in this study) into different clusters [33].

4.4 Clustering Similar Test Cases using Unsupervised Machine Learning

Clustering large data sets that have thousands of dimensions is a complex and challenging problem. Most of the proposed solutions in the state-of-the-art focus on dimension reduction. In that process, data from a high-dimensional space needs to be transferred into a low-dimensional space, which can be handled by most of the clustering and classification algorithms [24]. However, during dimensionality reduction, some properties of the data might be lost, whereas some meaningful properties of the original data might remain [34].

The clustering algorithms can mainly contain the following commonly used ones:

- *Hierarchical-based* such as the Agglomerative algorithm.
- *Partition-based* such as the Affinity algorithm.
- *Density-based* such as the HDBSCAN algorithm.

In order to keep all data properties, we recommend employing new types of clustering algorithms that can handle high-dimensional data sets. Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN). It is a hierarchical clustering algorithm that extracts a flat clustering-based on the stability of clusters. Moreover, HDBSCAN can produce a cluster of non-clustered data points that need to be interpreted accordingly based on the application. For instance, in our setting, non-clusterable data points can be considered as non-similar test cases.

Furthermore, Fuzzy C-Means Clustering (FCM) is another alternative for solving the clustering problem. FCM generally considers each object a member of every cluster, with a variable degree of membership [33], which makes sense in the context

of system testing. In fact, each test case can be similar to one or more test cases. Listing 1 shows a Python code snippet for selecting and applying HDBSCAN and FCM.

```
if opt_method=='hdbscan':
    clusterer = hdbscan.HDBSCAN().fit(values)
    mylabels = clusterer.labels_
elif opt_method == 'fcm':
    values_rotated = np.rot90(values)
    cntr, u, u0, d, jm, p, fpc = skfuzzy.cluster.cmeans(
    values_rotated, opt_nclusters, 2, error=0.005, maxiter=1000,
    init=None)
    mylabels = np.argmax(u, axis=0)
else:
    print("Clustering method unknown")
```

Listing 1 Python script for clustering

4.5 Data Processing and Visualization

In this case study, we combine Doc2vec with the HDBSCAN clustering algorithm in order to divide all test cases into several clusters based on their semantic similarity in the test specifications. In order to rerun this case study with your own data, the following steps need to be performed[2]:

1. All irrelevant information such as testing date, time, station, testers ID should be eliminated from the test case specifications. In other words, just the relevant information which indicates the testing purposes, requirements, steps, and testing procedure need to be kept.
2. For improving the accuracy of the model, different data pre-processing techniques such as tokenization, which is a way of separating a piece of text into smaller units called tokens, should be applied to the test case specifications.
3. The pre-processed data (i.e., the test case specifications) are the input for Doc2vec providing numeric representations of the test case specifications.
4. Since the output of the Doc2vec is a large set of high dimensional data, the HDBSCAN algorithm can be employed for clustering, which relaxes the pressure of dimensional reduction. However, if one wants to use other clustering algorithms, then the high dimensional data (output from Doc2vec) might be processed by dimensionality reduction algorithms such as t-SNE. t-distributed stochastic neighbor embedding (t-SNE) is a statistical method for high-dimensional reduction that gives each datapoint a location in a two or three-dimensional map. Listing 2 shows a Python code snippet to apply t-SNE.

```
def get_tsne(_values):
```

[2] The source code of our work can be found online at [16], together with anonymized feature vectors and a test case graph.

```
    tsne = TSNE(n_components=2, random_state=0)
    return tsne.fit_transform(_values)
```

Listing 2 Python script for t-SNE based dimensionality reduction

5. As HDBSCAN utilizes an unsupervised learning approach, the algorithm itself decides on the number of clusters (including non-clusterable data points). However, the end-user can decide the minimum number of data (test cases) for each cluster.

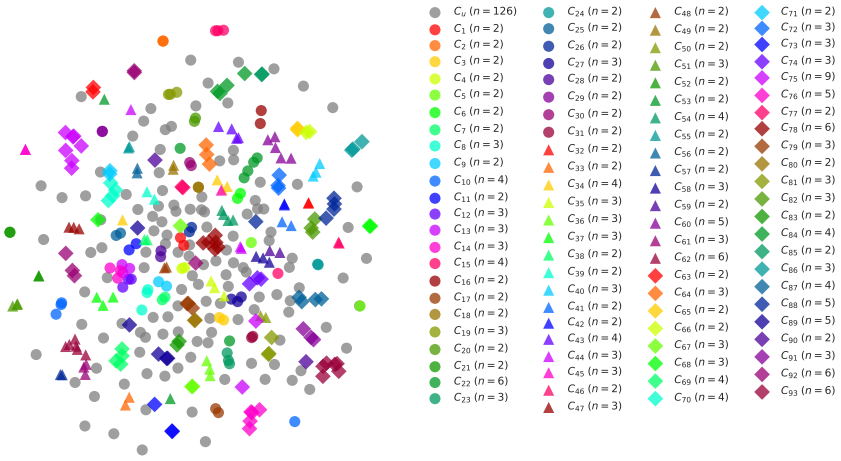

Fig. 4 The clustered test case using Doc2vec and HDBSCAN, where C_u represents non-clusterable data points and n indicates the size of each cluster.

Listing 3 shows the Python code snippet to generate the graph shown in Fig. 4

```
def clustered_graph(file_name, labels, label2cluster, X_tsne):
    sns.set_context('paper')
    sns.set_style('white')
    sns.set_color_codes()

    plot_kwds={'alpha':0.75, 's':40, 'linewidths':0}
    fig = plt.figure()
    fig.set_dpi(72)
    fig.set_size_inches(6, 6)
    ax = fig.add_subplot(111)

    sns.despine(top=True, bottom=True, left=True, right=True)
    ax.set_xticks([])
    ax.set_yticks([])

    maxcolors = ceil((max(label2cluster.values())+1)/3)
    pal = my_palette(maxcolors)
```

```
    fltrdX = []
    fltrdY = []
    for i in range(0,len(labels)):
        if label2cluster[labels[i]] == -1:
            fltrdX.append(X_tsne[i, 0])
            fltrdY.append(X_tsne[i, 1])
    plt.scatter(fltrdX, fltrdY, c=mpl.colors.rgb2hex(
    (0.5,0.5,0.5) ), label="$C_u$ $(n={})$".format(len(fltrdX)),
    **plot_kwds)

    for cluster in range(0,max(label2cluster.values())+1):
        fltrdX = []
        fltrdY = []
        for i in range(0,len(labels)):
            if label2cluster[labels[i]] == cluster:
                fltrdX.append(X_tsne[i, 0])
                fltrdY.append(X_tsne[i, 1])
        marker = 'o'
        if (cluster >= maxcolors):
            marker = '^'
        if (cluster >= 2*maxcolors):
            marker = 'D'

        plt.scatter(fltrdX, fltrdY, c=mpl.colors.rgb2hex(pal[
    cluster%maxcolors]), marker=marker, label="$C_{{{}}}$ $(n={})
    $".format(cluster+1, len(fltrdX)), **plot_kwds)

    lgd = plt.legend(bbox_to_anchor=(1.05, 1), borderaxespad=0.,
    prop = {"size": 6})
    fig.tight_layout()
    plt.savefig(file_name, format="pdf", bbox_extra_artists=(lgd
    ,), bbox_inches='tight')
    plt.clf()
```

Listing 3 Python script for clustering graph

As mentioned before, in our case study in total 1,748 natural language test case specifications were analyzed. Figure 4 visualizes the initial results with a total of 93 clusters obtained from applying HDBSCAN. However, as one can see in Fig. 4 the size of each cluster is different. Note that with hyperparameter tuning we set the minimum cluster size to 2. Moreover, in total 126 test cases are detected as non-clustered data points (see C_u in Fig. 4), which represent the non-similar test cases. The source code of our work can be found online at [16], together with anonymized feature vectors and a test case graph.

4.6 Test Optimization Strategies

The results shown in Figure 4 can be utilized in several ways and also for different test optimization purposes. Cluster-based test optimization strategies can be

applied based on the company policies, resources, and constraints. In our context the following cluster-based test optimization strategies were applied:

- *Test case selection:* for that purpose, one or more test cases are selected from each cluster. Since the grouped test cases inside of the obtained clusters are similar, some test cases can be selected for test execution purposes. However, to keep the test coverage and to avoid unnecessary failure, some parameters need to be checked, and some conditions need to be satisfied. For instance, the dependency between test cases [31], requirement coverage, and execution time [30] should be checked in advance. In this regard, a good test candidate from each cluster can be a test case that is independent or has the highest requirements coverage compared to other test cases of the same cluster. This strategy can also be used for the *test suite minimization* and also *test automation*, where some test cases from each cluster or even some clusters can be eliminated from the test suite. Moreover, since we are working with the manual test cases, some of the test cases from each cluster (or even some clusters) can be selected for the test automation. In fact, knowing similar test cases in advance can help the testing team to select some test specifications for test script generation. However, for the resulting clusters shown in Fig. 4, we recommend executing all non-similar test cases in cluster C_u as the setting is safety-critical.
- *Cluster prioritization:* the resulting clusters in Fig. 4 can also be ranked for execution. In this regard, those clusters which have a bigger size (see n in Fig. 4) can be high or low ranked for the execution. However, the mentioned constraints (dependency, requirement coverage) need to be evaluated in this strategy as well.
- *Functional dependency detection between test cases:* the entire proposed solution in this chapter can also be employed for detecting the functional dependencies between system integration test cases. Previously (see [34], [33]), we proved that two test cases can be functionally dependent on each other if there is a semantic similarity between their test specification. This hypothesis is evaluated several times against a conducted ground truth at Bombardier Transportation. The utilized performance metrics (F1-Score[3] and AUC[4]) indicate promising results.

To sum up, we first pre-processed test specification data, then applied Doc2vec to encode the natural language text into vectors, and later applied clustering algorithms to cluster test cases into groups of similar test cases. These groups formed the basis for cluster-based test optimization.

5 Levels of Autonomy of AI in System Testing

The approaches to applying AI in the previous sections, i.e., search-based approaches, natural language processing, and machine learning, were applied by supporting

[3] is a measure of a model's accuracy on a dataset.

[4] The Area Under Curve provides an aggregate measure of performance across all possible classification thresholds.

human testers in performing testing activities. However, the AI did not operate autonomously but was triggered by humans and relied on their knowledge.

In this section, we reflect on the potential of autonomously operating AI for system testing. The term "autonomy" is generally used to mean the capacity of an artificial agent to operate independently of human guidance [19]. Agents or bots act autonomously, perceive their environment, persist over a prolonged time period, adapt to change, and create and pursue goals. Software agents are called softbots or software robots, and it is a natural approach to use them in system testing to simulate real users or perform exploratory testing. Autonomous agents or softbots have already been used to automate different aspects of software development [38, 37, 10]. Several researchers have proposed different approaches for using agents specifically in software testing [35, 9], by considering different aspects that relate to test optimization. A recent study [22] suggests that autonomous agents are predominantly used at system-level testing functional, non-functional and white-box testing.

Similar to the levels of autonomy in automotive engineering, we define and discuss several levels of autonomy when applying AI in system testing. Autonomy in itself can be described through two dimensions, self-sufficiency, i.e., ability to fulfill a task without outside help, and self-directness, i.e., the ability to decide upon one's own goals as well as the involvement of an external human actor.

Overall, we can distinguish four levels of autonomy of AI in system testing[5] from Level 0 to not apply AI at all to Level 3 of full autonomy. In the following, we describe each level in more detail and provide examples:

- Level 0 - *AI is not applied*: System testing tasks are performed by humans or automated without AI. For example, test design at the system level can be automated using fuzzing, model-based testing, or combinatorial techniques [3].
- Level 1 - *AI algorithms assist humans by performing (semi-)automate testing tasks*: AI algorithms support test analysis, design, execution, and evaluation activities, as described in the previous section for test case dependency detection and execution. Another such Level 1 approach is the adaptive test management system (ATMS) [25], which aims at selecting an appropriate set of test cases to be executed in every test cycle using test unit agents and fuzzy logic.
- Level 2 - *AI replaces or mimics human behavior (e.g., agents that replace users)*: On this level not specific testing activities, but human behavior that is an integral part of testing is (partially) replaced by intelligent agents. Typical human behavior that is replaced are users of system under test or testers performing exploratory testing. For instance, in [8] intelligent agents are applied for real time testing of insider threat detection systems. As the number and variety of system interfaces increases, e.g., via image or voice recognition, the potential and need for the application of bots for system testing further increases. A different approach is presented in the work of Tang et al. [35]. Their study aims at automating the whole testing life cycle by using four types of agents: requirement agent, construct agent, execution, and report agent.

[5] Based on the levels shown by Synopsis https://www.synopsys.com/automotive/autonomous-driving-levels.html

- Level 3 - *System testing is done fully automated by AI agents*: This is the fully autonomous level, where system testing is performed fully automated by intelligent agents. Currently, this is just a vision and it is even not clear whether this is achievable both from a theoretical or practical point of view. Several approaches [9, 5] have shown some incipient but promising results on how this vision can be attained.

These levels of autonomy differ on how humans are involved and we can consider that this categorization is a continuum of autonomy rather than a dichotomy. Similar to this, Feldt et al.[12] used the Sheridan-Verplanck taxonomy to categorize the levels of AI automation based on decision and action selection performed by different actors. Nevertheless, more research is needed to characterize AI-infused system-level testing techniques that enable autonomous behavior.

6 Conclusion

This chapter presented where and how AI techniques can be applied to automate and optimize system testing. We first provided an overview of system testing, where testing is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. Then, we identified different system testing activities (i.e., test planning and analysis, test design, test execution, and test evaluation) and indicated how AI techniques like optimization algorithms, natural language processing and machine learning could be applied to automate and optimize these activities. Furthermore, we presented an industrial case study on test case analysis. In the case study, natural language test cases are - based on natural language processing with Doc2vec and clustering - grouped into clusters of similar test cases that formed the basis for cluster-based test optimization. Finally, we discuss levels of autonomy of AI in system testing from Level 0 to not apply AI at all to Level 3 of full autonomy.

Acknowledgements This work was partially supported by the Austrian Science Fund (FWF): I 4701-N and the project ConTest funded by the Austrian Research Promotion Agency (FFG). Eduard Enoiu was partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 957212.

References

1. Abran, A., Moore, J., Bourque, P., Dupuis, R., Tripp, L.: Software engineering body of knowledge. IEEE Computer Society (2004)
2. Adamo, D., Khan, M.K., Koppula, S., Bryce, R.: Reinforcement learning for android gui testing. In: Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, pp. 2–8 (2018)

3. Anand, S., Burke, E., Chen, T.Y., Clark, J., Cohen, M., Grieskamp, W., Harman, M., Harrold, M., McMinn, P., Bertolino, A., et al.: An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software **86**(8), 1978–2001 (2013)
4. Arcuri, A.: Test suite generation with the many independent objective (mio) algorithm. Information and Software Technology **104**, 195–206 (2018)
5. Baral, K., Offutt, J., Mulla, F.: Self determination: A comprehensive strategy for making automated tests more effective and efficient. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 127–136. IEEE (2021)
6. Bath, G., Van Veenendaal, E.: Improving the Test Process: Implementing Improvement and Change-A Study Guide for the ISTQB Expert Level Module. Rocky Nook, Inc. (2013)
7. Briand, L.C., Labiche, Y., Shousha, M.: Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. Genetic Programming and Evolvable Machines **7**(2), 145–170 (2006)
8. Dutta, P., Ryan, G., Zieba, A., Stolfo, S.: Simulated user bots: Real time testing of insider threat detection systems. In: 2018 IEEE Security and Privacy Workshops (SPW), pp. 228–236. IEEE (2018)
9. Enoiu, E., Frasheri, M.: Test agents: The next generation of test cases. In: International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 305–308. IEEE (2019)
10. Erlenhov, L., Oliveira Neto, F.G., Scandariato, R., Leitner, P.: Current and future bots in software development. In: International Workshop on Bots in Software Engineering (BotSE), pp. 7–11. IEEE (2019)
11. Felderer, M., Schieferdecker, I.: A taxonomy of risk-based testing. International Journal on Software Tools for Technology Transfer **16**(5), 559–568 (2014)
12. Feldt, R., Oliveira Neto, F.G., Torkar, R.: Ways of applying artificial intelligence in software engineering. In: International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), pp. 35–41. IEEE (2018)
13. Frounchi, K., Briand, L.C., Grady, L., Labiche, Y., Subramanyan, R.: Automating image segmentation verification and validation by learning test oracles. Information and Software Technology **53**(12), 1337–1348 (2011)
14. Garousi, V., Felderer, M., Karapıçak, Ç.M., Yılmaz, U.: Testing embedded software: A survey of the literature. Information and Software Technology **104**, 14–45 (2018)
15. Harman, M., McMinn, P., Souza, J.T., Yoo, S.: Search based software engineering: Techniques, taxonomy, tutorial. In: Empirical software engineering and verification, pp. 1–59. Springer (2010)
16. Hatvani, L., Tahvili, S.: Clustering dependency detection. https://github.com/leohatvani/clustering-dependency-detection (2018)
17. IEEE: Ieee standard glossary of software engineering terminology. IEEE Std 610.12-1990 pp. 1–84 (1990)
18. Jiang, J.Y., Zhang, M., Li, C., Bendersky, M., Golbandi, N., Najork, M.: Semantic text matching for long-form documents. WWW '19. Association for Computing Machinery, New York, NY, USA (2019)
19. Johnson, M., Bradshaw, J., Feltovich, P., Jonker, C., Van Riemsdijk, B., Sierhuis, M.: The fundamental principle of coactive design: Interdependence must shape autonomy. In: International Workshop on coordination, organizations, institutions, and norms in agent systems, pp. 172–191. Springer (2010)
20. Joshi, A., Fidalgo, E., Alegre, E., Fernández-Robles, L.: Summcoder: An unsupervised framework for extractive text summarization based on deep auto-encoders. Expert Systems with Applications **129**, 200–215 (2019)
21. Kane, M.: Validating the interpretations and uses of test scores. Journal of Educational Measurement **50**, 1–73 (2013)
22. Kumaresen, P., Frasheri, M., Enoiu, E.: Agent-based software testing: A definition and systematic mapping study. In: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 24–31. IEEE (2020)

23. Liang, D., Zhang, F., Zhang, W., Zhang, Q., Fu, J., Peng, M., Gui, T., Huang, X.: Adaptive multi-attention network incorporating answer information for duplicate question detection. SIGIR'19, p. 95–104. Association for Computing Machinery, New York, NY, USA (2019)
24. van der Maaten, L., Postma, E., Herik, H.: Dimensionality reduction: A comparative review. Journal of Machine Learning Research - JMLR **10** (2007)
25. Malz, C., Jazdi, N.: Agent-based test management for software system test. In: International Conference on Automation Quality and Testing Robotics (AQTR), vol. 2, pp. 1–6. IEEE (2010)
26. Mansoor, M., Rehman, Z., Shaheen, M., Khan, M., Habib, M.: Deep learning based semantic similarity detection using text data. Information Technology And Control **49** (2020)
27. Markov, I., Gómez-Adorno, H., Posadas-Durán, J.P., Sidorov, G., Gelbukh, A.: Author profiling with doc2vec neural network-based document embeddings. In: O. Pichardo-Lagunas, S. Miranda-Jiménez (eds.) Advances in Soft Computing, pp. 117–131. Springer International Publishing, Cham (2017)
28. Memon, A.M., Pollack, M.E., Soffa, M.L.: A planning-based approach to gui testing. Proceedings of The 13th International Software/Internet Quality Week (2000)
29. Ramler, R., Felderer, M.: Requirements for integrating defect prediction and risk-based testing. In: 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 359–362. IEEE (2016)
30. Tahvili, S., Afzal, W., Saadatmand, M., Bohlin, M., Ameerjan, S.H.: Espret: A tool for execution time estimation of manual test cases. Journal of Systems and Software **161**, 1–43 (2018)
31. Tahvili, S., Bohlin, M., Saadatmand, M., Larsson, S., Afzal, W., Sundmark, D.: Cost-benefit analysis of using dependency knowledge at integration testing. In: The 17th International Conference On Product-Focused Software Process Improvement (2016)
32. Tahvili, S., Hatvani, L., Felderer, M., Afzal, W., Bohlin, M.: Automated functional dependency detection between test cases using doc2vec and clustering. In: 2019 IEEE International Conference On Artificial Intelligence Testing (AITest), pp. 19–26. IEEE (2019)
33. Tahvili, S., Hatvani, L., Felderer, M., Afzal, W., Saadatmand, M., Bohlin, M.: Cluster-based test scheduling strategies using semantic relationships between test specifications. In: Proceedings of the 5th International Workshop on Requirements Engineering and Testing, pp. 1–4 (2018)
34. Tahvili, S., Hatvani, L., Ramentol, E., Pimentel, R., Afzal, W., Herrera, F.: A novel methodology to classify test cases using natural language processing and imbalanced learning. Engineering Applications of Artificial Intelligence **95**, 1–13 (2020)
35. Tang, J.: Towards automation in software test life cycle based on multi-agent. In: International Conference on Computational Intelligence and Software Engineering, pp. 1–4. IEEE (2010)
36. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Software testing, verification and reliability **22**(5), 297–312 (2012)
37. Winikoff, M.: Future directions for agent-based software engineering. IJAOSE **3**(4), 402–410 (2009)
38. Wooldridge, M.: Agent-based software engineering. IEE Proceedings-software **144**(1), 26–37 (1997)