IVVES

Industrial-grade Verification and Validation of Evolving Systems

Labelled in ITEA3, a EUREKA cluster, Call 5

ITEA3 Project Number 18022

D3.2 Validation methods and techniques for evolving systems considering use case requirements (version 1)

Due date of deliverable: December 30th 2020
Actual date of submission: December 21th 2020

Start date of project: 1 October 2019	**Duration:** 39 months
Organisation name of lead contractor for this deliverable:	CRIM
Author(s):	Omer Nguena-Timo, Alexandre Petrenko (CRIM, CAN), Matvey Pashkovskiy (F-Secure, FIN), Jesús Arce (KEYLAND, ESP), Paul Derckx (Philips, NLD), Mahshid Helali Moghadam, Ali Sedaghatbaf (RISE, SWE), Juan Leandro Sánchez (SII CONCATEL/NETCHECK, ESP), Elio Saltalamacchia (SII CONCATEL, ESP), Almira Pillay, Tia Nikolic (Sogeti, NLD), Pekka Aho (The Open University of The Netherlands, NLD), Tommi Mikkonen, Eero Kauhanen, Jukka K. Nurminen (University of Helsinki, FIN)
Status:	Draft
Version number:	V1.0
Submission Date:	21-December-2020
Doc reference:	IVVES_Deliverable_D3.2_Validation_methods_and_techniques _V1.0.docx
Work Pack./ Task:	WP3 T3.1, T3.2, T3.3
Description: *(max 5 lines)*	The document presents validation problems identified in use cases, anticipated contributions, and initial solutions.

Nature:	☑ **R**=Report, ☐ **P**=Prototype, ☐ **D**=Demonstrator, ☐ **O**=Other		
Dissemination Level:	**PU**	Public	**X**
	PP	Restricted to other programme participants	
	RE	Restricted to a group specified by the consortium	
	CO	Confidential, only for members of the consortium	

DOCUMENT HISTORY

Release	Date	Reason of change	Status	Distribution
V0.1	15/11/2020	Text review	Draft	Amongst partners
V0.2	17/12/2020	Review	Draft	Lead contractor
V1.0	21/12/2020	Approved by PMT, submitted to ITEA3	Final	ITEA Office Website

Table of Contents

Glossary

Abbreviation / acronym	Description
FSM	Finite State Machine
RL	Reinforcement Learning
ML	Machine Learning
NLP	Natural Language Processing
CI	Continuous Integration
ESG	Environmental, Social and Governance
GUI	Graphical User Interface
AI	Artificial Intelligence
TA	Test Automation
SUT	System Under Test
E2E	End-to-end

1. Executive summary

This document (D3.2) concerns initial validation methods and techniques for complex evolving systems in project use cases. The state of the art of the validation techniques for evolving systems appears in deliverable D3.1 referenced in this document. The document presents problems identified in use cases, anticipated contributions, and initial solutions. Further work, to be reported in upcoming D3.3 and D3.4, includes the elaboration of the anticipated contributions, enhancement of initial methods and tool development.

The document includes six sections corresponding to problems identified in validating complex evolving systems from IVVES partners. These problems meet the objectives of the three tasks of WP3, namely ML-based testing, testing under uncertainties, testing and monitoring. Each section provides an introduction to problems, anticipated contributions, approaches and claimed novelties.

Claimed novelties are methods or improvement of tools based on state-of-the-art techniques; they cover the three main stages for the continuous quality assurance process presented in D3.1. Claimed novelties in ML-based testing include scriptless GUI testing based on model inference, performance testing based on active deep learning and reinforcement learning, ML-based code quality prediction and an implementation of an efficient version of conformal prediction suitable for edge applications. The claimed novelty in testing under uncertainties is an approach for resolving uncertainties which can occur in generating tests from requirement documents. The claimed novelties in testing and monitoring include test prioritization based on code changes tracking in a version control system and continuous ML-based online monitoring for anomaly identification in AI systems.

2. Introduction

Complex evolving systems need adequate methods and tools to ensure their quality. In a previous deliverable, D3.1, we studied existing validation methods and techniques applicable to evolving systems. The goal of that study was to better understand the methods and techniques and foresee their applicability in use cases from IVVES industrial partners.

In this deliverable we present validation methods and techniques which we have been developing or anticipating to adapt to the use cases. We selected existing validation methods and techniques we plan to adapt and implement in existing tools developed by partners. We also propose new approaches to solve problems we identified in addressing the use cases.

The anticipated validation methods and techniques cover the design, the development and testing, and the operation phase in the continuous quality assurance process presented in D3.1.

In the design phase, we address the oracle mining problem, which amounts to building specifications useful to judge observations produced by executions of systems under automated testing. The problem is non-obvious especially if the data source for building the specification is requirement documents or logs of execution traces of a legacy version of the system. The anticipated contribution (see Section 4) is a novel semi-automated procedure to assist an expert in choosing a proper oracle from a set of potential oracles compactly presented by an imprecise oracle that can be built with NLP techniques.

In the development and test phases, we address test generation and prioritization problems. We propose testing techniques based on reinforcement learning and active deep learning for the performance analysis of ES (see Section 3.2) and the prediction of risky parts of code of ES (see Section 7). Performance is a quality characteristic which describes the time and resource bound aspects of a system's behavior. Systems can be executed with adequate tests to estimate the bound aspects, in particular after changes in the systems. We may have several performance measures and each test case may target a different measure. For example, we may have test cases to verify response time requirements, and also test cases that verify throughput requirements. Here, we consider each measure as a class in machine learning terminology and by classification, we mean labelling test cases with the measures that they verify and we employ ML to realize it.

We also anticipate developing and implementing a coverage-based test generation technique for enhancing a GUI testing tool (Section 3.1), yet another ML-based test prioritization

technique for an ESG-investment system (Section 3.3) and a coverage-based test prioritization technique suitable for continuous integration with version control systems (section 3.4). In the latter case, the idea is to map test cases to code modules utilizing coverage reports and/or errors stack trace analysis and the claimed novelty is the adaptation of this idea to an industrial context. We advocate using conformal predictive models for developing critical systems; indeed, conformal prediction allows us to devise more rigorous verification techniques for the systems. We propose an implementation of conformal predication that scales on edge applications (Section 6).

We also address the automation of test verdict production in a testing tool for GUI (Section 5). We propose leveraging finite state machine inference (passive learning) to build the specification of an application from logs and later use the built specification to judge observations made during the execution of the systems with test inputs.

In the operation phase, we anticipate developing an online monitoring system for anomaly detection in visual inspection systems deployed in a real industrial environment (Section 8). To the best of our knowledge, such a system needed by the partners does not yet exist.

The document is organized as follows. The next section presents our progress in the field of test generation and test prioritization. In Section 4, we present our progress on oracle mining from imprecise oracles. Section 5 suggests leveraging machine learning to automate the generation of test verdicts in an existing tool. Our implementation of conformal prediction able to run on edge applications is discussed in Section 6. In Section 7, we present an ML-based approach we anticipate to develop and apply for code defect risk prediction. In Section 8, we present our anticipated ML-based online monitoring approach for anomaly identification in AI systems.

3. Test generation and test prioritization for fault detection

3.1. Scriptless E2E test generation for ES with coverage analysis

Building a test automation pipeline of complex evolving systems (ES) is an elaborate task, especially on the end-to-end (E2E) level. One of the main challenges is the fact that the resulting test suites are difficult to maintain. We plan to address this challenge by applying both, non-ML and ML techniques to generate test suites with a high coverage level. Techniques, methodologies, and tools are needed to be able to:

- generate test cases automatically with a good level of interpretability for
  - different stages of test automation (TA);
  - different types of applications: standalone, web, and mobile applications;
- reuse generated test suites and inferred application models to optimize TA in terms of time and coverage.

3.1.1. State of the art

Traditionally, software test automation is based on scripts (pre-defined test sequences with test oracle checks) that are either automatically generated from models or written by a human. In scriptless test automation, the test sequences are generated dynamically during the execution, usually one step at a time, based on automatically detected available interactions that the end user could perform, or events from the environment. The execution of the action includes waiting for the reaction from the system under test (SUT). **Figure 3.1** depicts the process of scriptless testing at a high level.

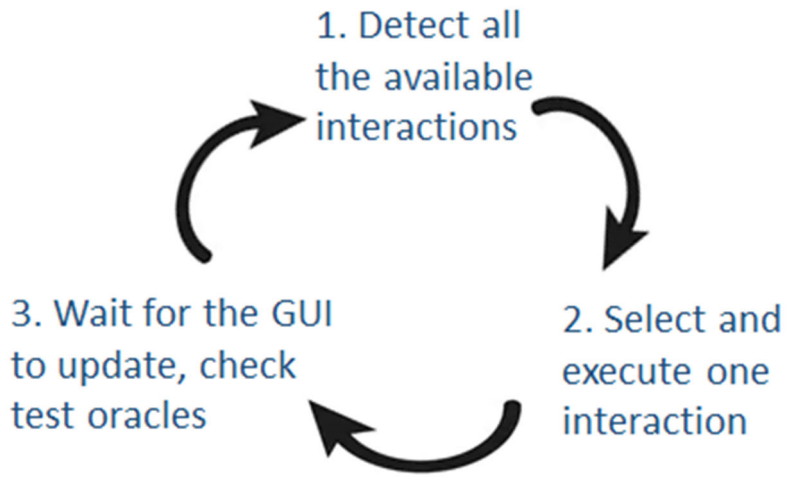


Figure 3.1: Scriptless testing process.

The action selection often involves some level of randomness, and therefore scriptless GUI testing is often called random or monkey testing. One of the research directions for trying to make monkey testing tools smarter has been using AI and machine learning for improving action selection. Usually, some kind of model inference approach has to be used for the learning process. Often, the reward or fitness functions have been connected to increased GUI or code coverage [7], [8]. This kind of strategy usually rewards visiting (i.e., covering) all GUI states at least once. However, visiting all GUI states does not mean that all paths or combinations of paths have been visited.

In IVVES project, we aim to investigate combinatorial coverage for improving action selection after all GUI states have been visited. One of the options is using n-switch coverage. In 1976, Pimont and Rault [13] introduced a criterion for covering pairs of edges called switch coverage. In 1978, Chow [14] generalized this and defined n-switch edge coverage for a specific graph. With a switch cover-tree, we can detect all possible n-switch walks for one vertex. We have to calculate an n-switch cover-tree of all vertices to be able to calculate the n-switch edge coverage of a particular walk. The n-switch edge coverage criterion is about the percentage of walks with length n+1 that a test set covered. This calculation can speed up knowing the cover-tree property that if we cover an n-switch walk, we also cover all lower n-switch walks. So, a 100% 0-switch coverage of the graph means that we visit every edge in the graph at least once. A 100% 1-switch coverage means we walk all possible back-tracking walks with 2 edges from all vertices at least once.

Inferring state models during automated GUI exploration has been researched with various approaches, for example, GUITAM [9], GUI Driver [10], and Crawljax [11]. However, automated change detection by comparing inferred models of consequent system versions has not been widely researched; Murphy tools [12] seems to be the only existing approach in the literature.

3.1.2. Anticipated contribution

Open source TESTAR tool will be used for scriptless graphical user interface (GUI) test generation and state model inference in continuous integration (CI) environment for selected software packages. TESTAR dynamically generates test sequences during the exploration of

SUT, one step at a time, based on detected available actions. TESTAR supports model inference during the automated exploration of the GUI of an application and uses the model for systematically (but in random order) trying out all the available actions in all the explored states.

TESTAR can infer state models based on the observed behavior of the system under testing (SUT) and use the inferred models for optimizing the action selection during test sequence generation. So far during the IVVES project, we have already demonstrated that TESTAR is able to automatically explore desktop and web applications of the industrial use case providers, and infer state models based on the observed SUT behaviour. Currently, we are improving the configuration of TESTAR and trying to find a suitable level of abstraction for the model inference. During the project, the state models will be used for reinforcement learning with various reward functions to improve action selection. Also, we plan to research automated change detection based on comparing the inferred state model of consequent SUT versions.

For Windows desktop applications, TESTAR uses Windows accessibility API to access the GUI information for detecting the state of the SUT and available actions. For web applications, TESTAR uses Selenium WebDriver. The plan is to extend TESTAR during IVVES project to support mobile applications by using Appium to connect to a mobile device emulator. Adding support for testing mobile applications with TESTAR is an important technical contribution that has been requested by several industrial partners of IVVES. The implementation effort has been started and the first proof-of-concept implementation has been demonstrated for the industrial partners of IVVES project.

3.1.3. Claimed novelty

Using inferred state models for machine learning to improve action selection in scriptless GUI testing has novelty and new publications are expected from the results.

Other novel research directions include using the state models of previous TESTAR runs to optimize the current GUI exploration, and as a coverage metric to change from exploration strategy to combinatorial strategy (for example n-switch) after each state and action has been visited at least once. Also, the use of AI/ML approaches for improving TESTAR action selection will be researched.

3.2. Machine learning-assisted automated performance testing

3.2.1. Performance testing

Performance, which is also called efficiency in some taxonomies, is a quality characteristic which describes the time and resource bound aspects of a system's behavior and is of great importance for the success of many products. It's measured in terms of some metrics like response time, throughput and resource utilization.

Performance analysis is typically done to measure performance metrics and detect performance-related issues. Performance issues could be referred as any violation of performance requirements or any functional problems emerged under special performance-related conditions such as heavy workload and limited resource availability. Several methods have been developed to analyze the performance of software products. Generally, these methods can be categorized into modeling and testing groups. Modeling methods [15, 16, 17, 18, 19] are mainly based on the performance models extracted from the system model or the source code of the target system.

However, in performance testing, test cases are applied during the execution the SUT to find potential scenarios which lead to the emergence of performance issues. Many of the common performance testing approaches such as techniques based on source code analysis [20], system model analysis [21, 22], use case-based [23], and behavior-driven [24] design approaches mostly rely on source code or system models.

Performance testing challenges. Software performance testing to find performance issues upon new changes mainly occurred within CI/CD practice is always a challenging task. Therefore, automated performance testing is of importance in this regard and subsequently one of the primary concrete challenges in software performance testing is generating appropriate test cases (test scenarios) in an efficient and cost-effective way. Performance test scenarios are intended to detect performance degradation issues. By detecting performance degradation issues at an early phase, the changes leading to the degradation is easier to localize and could be actively decided upon, and consequently the extra cost at the customer side due to degraded system performance can be avoided.

Currently, the performance test is done manually for many software products. However, due to the increasing complexity and diversity of the products and the services requested by customers, the need to automate the performance testing process is highlighted.

In the manual testing process, several test cases have to be executed upon each change in the software. Not only, the manual process is time-consuming and laborious, but also it is error-prone and does not provide any correctness guarantee since it relies on expert knowledge.

On one hand in many cases in order to generate test cases, we usually have a large input space to explore which makes manual test case generation a time-consuming and laborious task. On the other hand, we usually do not have any knowledge about the internal structure and dynamics of the SUT, and the interaction with its public interface is the only way to learn about the SUT's performance characteristics. Last but not the least, we usually have a limited test budget, which means that we are not free to execute a large set of test cases hoping that some of them will reveal performance defects. Instead, we need to be careful about the test cases that we select for running on the SUT.

Taking advantage of the recent advances in machine learning, machine learning-assisted techniques have been used widely for meeting the need for automated performance testing. In our work, we propose machine learning-assisted approaches which learn from the behavior of the SUT and data collected from previous tests to generate promising/effective test cases automatically and efficiently. We propose two approaches based on active deep learning and reinforcement learning for efficient generation of optimized test cases to analyze the performance of a control software in a robotic system. The details of the proposed techniques are presented in the following sections.

3.2.2. Active deep learning-assisted performance testing

Active learning [26] is a kind of supervised learning suitable for situations where data labeling is expensive or difficult. So, in cases that we do not have access to a large amount of labeled data, we can iteratively choose small portions of unlabeled data, label them and then update the classifier with the labeled data. The iteration continues until the accuracy of the classifier reaches a desirable threshold. Since we can choose only a small number of unlabeled data in each iteration, it is very important that we follow a strategy for picking the most informative data for labeling. Uncertainty sampling, query-by-committee and variance reduction [26] are among the well-known strategies that can be used for this purpose.

Deep learning is a sub-field of machine learning where deep neural networks are used for supervised, unsupervised or semi-supervised learning purposes. We consider a neural network deep if its architecture includes multiple layers of neurons. Deep learning has been applied to

several domains (e.g., natural language processing, bioinformatics, and machine vision) and has achieved significant results.

In the proposed method, we apply an active learning sampling strategy to a variant of generative deep neural networks. We use a generative deep learning model which consists of two neural networks. The first network is responsible for generating new test cases and the second one classifies the generated test cases with respect to the satisfaction level of the performance test objective. For test case generation, we assume that there is a test case template which becomes executable when supplied by test input data. The responsibility of the generator network is to generate executable test cases by feeding the template with data samples from the input space.

Regarding the limited test budget, active learning helps us put less effort into test execution while acquiring more useful information about the system. Meanwhile, using a generative model we can automatically generate optimized test cases without putting any manual effort into it.

Generally, the following steps describe the procedure of test case generation in the proposed approach:

1. Train the classifier network with the existing test history.
2. Generate new test cases by sampling from the input space (which includes target position, motion speed and zone, and hardware configuration) by the generator network
3. Classify the generated test cases based on the targeted performance measures, which can be one of response time, CPU utilization and cycle time in the ABB example.
4. Rank the classified test cases based on the confidence level of the classifier.
5. Regarding the test budget, select the least confident test cases for executing on the SUT (or labeling in active learning terminology).
6. Use both the confident and executed test cases to update the weights of the classifier network.
7. Go to step 2 if the generator network cannot generate good test cases. To analyze the quality of the generated test cases, their distance to the classified test cases is measured.

In this process, the deep model is iteratively updated with knowledge about the SUT until it becomes ready for generating promising performance test cases automatically. After the convergence, the generator network is used to generate effective test cases for the performance test of the system based on a desired performance measure.

Applicability to DevOps. In DevOps we are to continuously integrate new changes with the SUT and deliver it with a high level of quality. In this situation, a part of the test cases which satisfy the performance test objective for the current version, might not work properly for the next one. Therefore, the deep model needs to continuously interact with the SUT and learn about its performance-related behavior. We can easily update the deep model with respect to new changes. The update procedure includes the following steps:

1. Regarding the test budget, pick the latest test cases applied in testing the previous version of the SUT.
2. Apply the latest test cases to the new version.
3. Compare the execution results with the results recorded for the previous version.
4. If the results were the same:

   Continue using the old generator network to generate test cases for the new version.

   Else if the results were different:

   a. Update the classifier model with the new execution results.
   b. Go to step 2 of the test case generation process and continue the loop until the generator network can generate good test cases again.

3.2.3. Reinforcement learning-assisted performance testing

Reinforcement learning (RL) [27] is a fundamental machine learning paradigm which is mainly intended to address decision making problems. Inspired by human's learning, RLis used to find the optimal way to make decisions. The learning procedure is quite different from supervised and unsupervised learning algorithms. The learning is based on continuous interaction between a smart RL agent and the problem environment which is system under test (SUT) in our research case. At each step of interaction, the smart test agent observes the status of the environment and makes a decision. The decision is generating a test case, e.g., based on changing the variables involved in forming the test case. Then the SUT is tested under the recommended test scenario, and the test agent receives a reward signal indicating the effectiveness of the recommended test case. One of the main differences between RL and other learning paradigms is that there is no supervisor in RL, i.e., the agent just receives a reward signal from the environment, and the agent goes through the environment based on a sequential decision-making process.

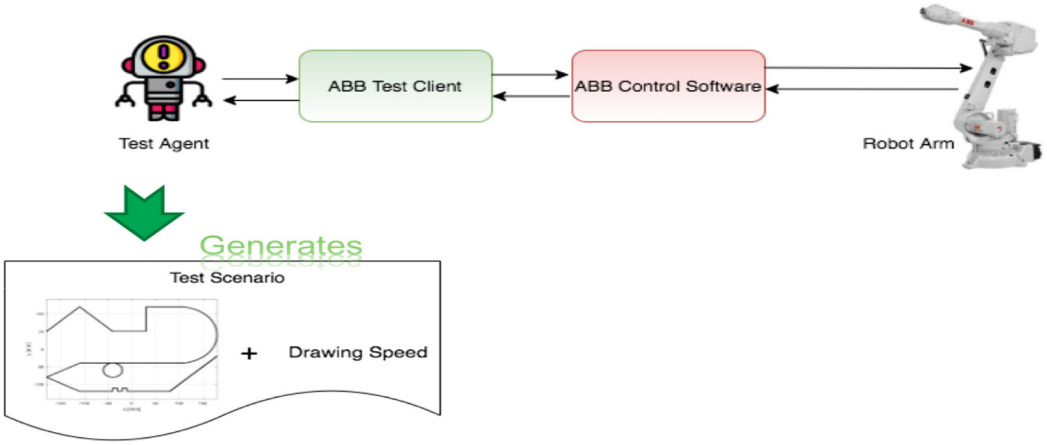
With regard to the characteristic of RL, we proposed that if the optimal policy (way) for accomplishing the intended performance test objective could be learned by a test agent, the intended test could be done automatically without need to access to source code or performance/system models. Moreover, once the optimal policy is learned, it can be reused in further testing situations, for example, regression performance testing of one SUT or performance testing of SUTs with similar performance sensitivity to resources [28]. Therefore, the capability of knowledge formation and reusing the gained knowledge in further situations is a key feature leading to test efficiency improvement. Based on this idea, we have proposed an RL-assisted performance testing framework that learns the optimal policy to accomplish the intended test objective without access to system model or source code of SUT. Once it learns, it is able to reuse the learned policy in further testing cases [28, 29, 30]. The proposed framework consists of two performance testing tools: SaFReL [31] and RELOAD [28, 32].

SaFReL, as a self-adaptive fuzzy reinforcement learning test agent which generates performance platform-based test cases, learns how to tune the resource availability to reach an intended performance breaking point for different types of SUTs with different levels of sensitivity to resources. It assumes two phases of learning: initial and transfer learning phases. First, it learns the optimal policy to reach the intended performance breaking point for different types of SUTs, i.e., CPU-intensive, memory-intensive and disk-intensive software. Once learning the optimal policy, it replays the learned policy on further similar SUTs. The conducted experimental evaluation shows that SaFReL can perform efficiently and adaptively on different software programs., i.e., CPU-intensive, memory-intensive and disk-intensive SUTs running on various hardware configurations and with different response time requirements. SaFReL accomplishes the intended test objective, i.e., finding performance breaking point, more efficiently in comparison to a typical stress testing technique which generates performance test cases in an exploratory way. SaFReL leads to reduced cost in terms of computation time by reusing the learned policy upon the SUTs with similar performance sensitivity [31].

RELOAD is an adaptive RL-driven load testing agent which effectively learns how to tune the load of transactions in the submitted workload to the SUT to accomplish the test objective (e.g., reaching a certain performance breaking point). It learns the optimal policy to generate an efficient workload to meet the test objective during an initial learning, then it is able to reuse the learned policy in later tests, e.g., within a continuous testing context. RELOAD

generates a more accurate and efficient workload to accomplish the intended objective compared with a baseline load testing technique, without access to source code or system models. Moreover, once it learns, it is able to reuse the learned policy in further situations and keeps the improved efficiency over later test activities [28, 32]

In our work in IVVES WP3, we extend the notion of RL-assisted performance testing by developing a smart test agent based on a scalable model-free temporal difference learning algorithm, i.e., DQN, for conducting performance test on a control software for a robotic arm belonging to ABB Robotic company. The proposed smart agent learns how to generate the effective test scenarios which result in undesired performance behavior. The performance requirements are defined in terms of the associated performance metrics such as resource (CPU) utilization, and/or response time. Test scenarios are coded as RAPID programs and target position, motion speed and zone are the primary parameters defining the test scenarios for the control software. The following figure shows the interaction between the smart test agent and the test environment for the robotic arm.



3.3. Test generation and prioritization for ESG-investment

The irruption of game-changing innovations and open-source technologies in NLP is changing the way that companies work with text. Unstructured text is being used as input data for many industrial domains (i.e., predicting market trends based on sentiment analysis). Data Analytics companies are curating and collating text information from diverse sources to feed AI models (Figure 3.2) and provide trends and insights. Its combination with other AI techniques applied to numerical data is fostering the integration of NLP into regular Data Analysis.

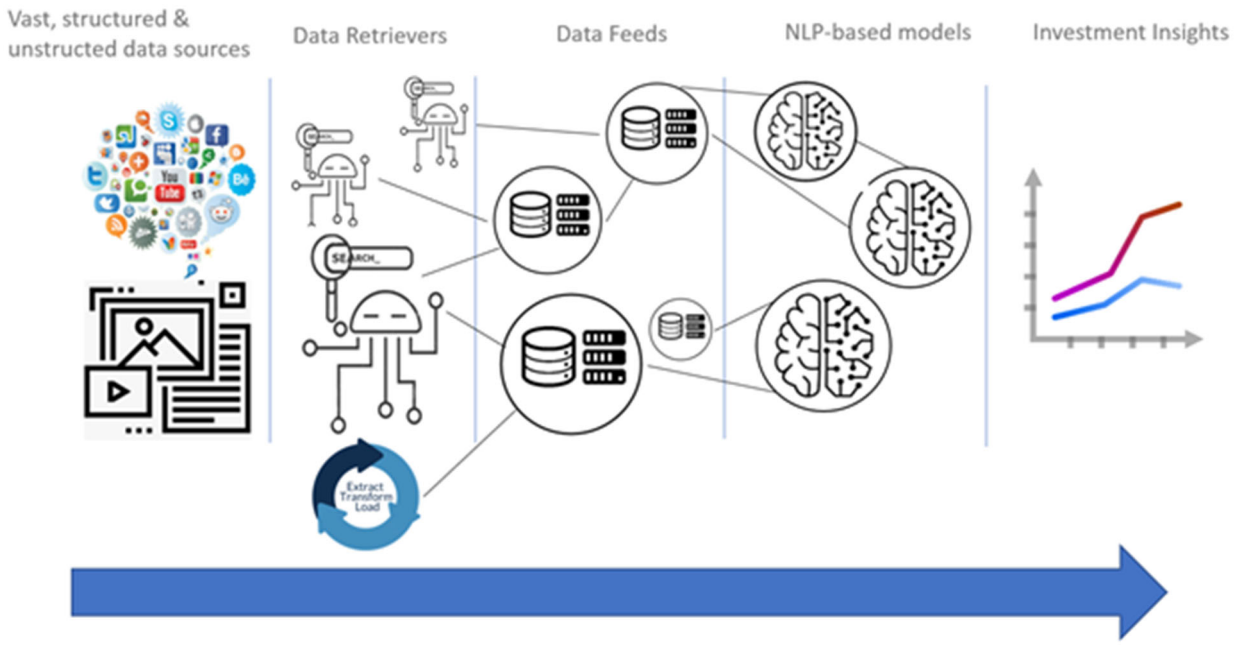


Figure 3.2 General workflow for NLP-based models to provide trends and insights in finance sector

ESG (Environmental, Social and Governance) investing refers to a class of investing that is also known as "sustainable investing." This is an umbrella term for investments that seek positive returns and long-term impact on society, environment and the performance of the business. To assess a company (an asset) based on environmental, social, and governance (ESG) criteria, investors look at a broad range of behaviours to set the ESG score for a given asset.

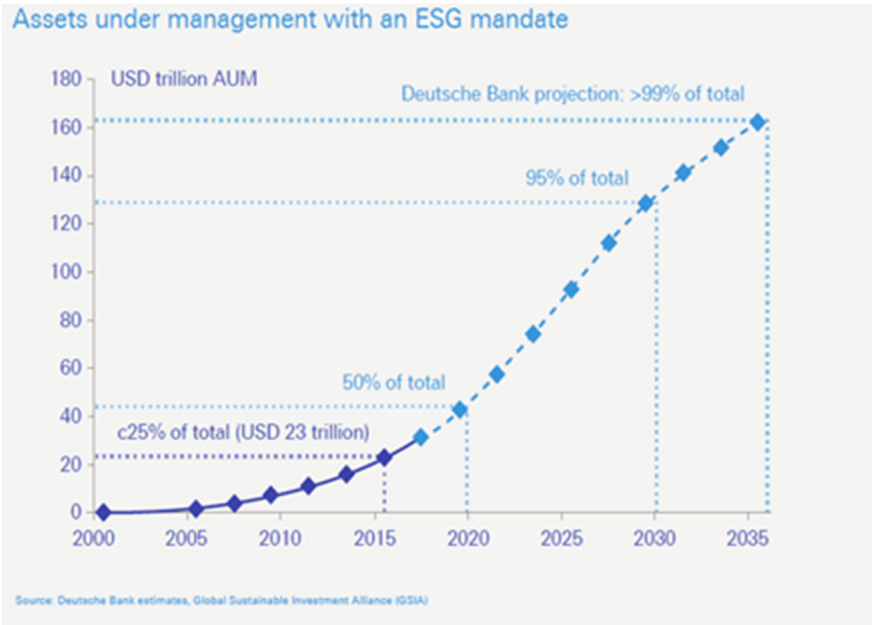


Figure 3.2: Estimates of assets under management with an ESG mandate. Source: Deutsche Bank.

The compilation of these scores is based on the analysis of a vast amount of fast changing alternative data sources, including non-structured information (websites, news, corporate reporting...) that can't be processed with traditional keyword searches and manual analysis. Since ESG investment is a solid trend that is increasingly impacting the market (figure2), the data sources to analyze are growing fast. Given the vast amount of data and information available, that analysis can only be reliably carried out with artificial intelligence. New, powerful AI-based systems are now on scene that can potentially reduce the manual tasks and increase the efficiency. However, new V&V techniques are required:

- The growth of AI-based analysis of sources, has also impacted the way that companies communicate with the external audience, being savvier with their wording. This is causing the appearance of biased content, that must be taken into account before applying NLP-based techniques -heavily relying on sentiment analysis- to get insights and trends. Hence, a systematic and continuous analysis of source credibility and content credibility must be implemented.
- The AI-based systems must continuously adapt to a great variability in sources and content, that are constantly changing. Information sources evolve, mutate and topics related to ESG change over time. Hence, test maintenance and prioritization are challenging.

Added to this, since the outcomes of the evolving systems are insights that may impact investment decisions, these systems are subject to regtech (regulatory technology) constraints that must be taken into account.

3.3.1. State of the Art

There are different approaches for test case Reinforcement Learning-based test case prioritization. Zhaolin, et al. [80], provides a reference for test case prioritization to save computing resources in Continuous Integration. In [80], a novel reward function is proposed, by using partial historical information of test cases effectively for fast feedback and cost reduction. The approach is focusing in reducing the huge cost in terms of time and resource availability defining the Average Percentage of Historical Failure with time Window (APHFW), as a novel reinforcement learning reward function, that utilizes a time window to filter recent historical information to calculate reward value.

3.3.2. Anticipated contribution

The main technical contribution is supporting testing based on Reinforcement Learning for NLP-based ESG evolving systems. Specifically, given an ESG-investment-focused ES and a set of rules defined by an expert for scoring securities with respect to ESG criteria, develop masked language models.

Figure 3.3: Templating with masked language models. [78]

The RL-based test generation and prioritization is based in a time window-based reward function that will also take into account the most effective Metamorphic Relations for a given case. For the selection of effective Metamorphic Relations, the model will initially interact with the templates generated in WP2 with masked language models (Figure 3.3), and eventually will control “Plug” operators.

As an outcome, the most effective Metamorphic Relations will be selected to generate the optimal test cases to execute, taking also into account performance.

3.3.3. Claimed novelty

The novelty relies in the dynamic metamorphic testing for NLP-based ESG evolving systems based on templates and Knowledge Graphs combined with approaches for test prioritization based on performance.

3.4. Coverage-based tests prioritization

Complex ES consists of a huge codebase and a big number of test cases covering it. In order to deliver new functionality with high confidence, a common approach is to run all the tests against every change in the codebase. That causes high infrastructure costs and delays in the development process. One of the approaches to solve this problem could be smart test selection

and/or prioritization. That could be solved by mapping test cases to code modules utilizing coverage reports and/or errors stack trace analysis.

Devising techniques and tools for providing a fine-grained mapping between test cases and code modules are needed.

3.4.1. State of the art

In a large software project, running the full test set can take a long time. In particular, when working with the modern DevOps style, developers are frequently checking in their modifications, and the resulting updates are tested and deployed on the fly. Having to wait -- often only for some minutes, but in extreme case hours -- at each check-in slows down development and is frustrating for developers. Furthermore, it consumes computing resources. For example, Mozilla estimates each check-in to cost over $25 in Amazon Web Services fees [33] while Google suggests that their annual continuous integration (CI) system execution is in millions of dollars [34].

Regression test selection (RTS) has been studied for a long time (see e.g. [35]) but its importance grows in modern DevOps and MLOps environments. Identifying and rerunning only the relevant tests after code changes are necessary for productivity and efficient resource use in CI.

A big part of past RTS studies has focused on Java and other compilable languages, e.g., [36]. At present Python and other interpretable languages are increasingly popular. One of the reasons for that increase of popularity is the growth of MLOps culture where Python plays a key role along with (Py)Spark framework for data analysis and building of ML models. RTS can bring even more benefits to MLOps process because testing data transformation logic and model building are extremely expensive operations. Interpretable languages present new challenges in terms of language constructs and dynamic operations that are missing from older languages.

3.4.2. Anticipated contribution

Our technique uses Git version control system and Coverage.py for tracking changes in the code. For the test runner, PyTest is used. The procedure of the technique is as follows: First, an initial run of all tests is performed. While performing the first run of the tests, a locally stored SQLite

database is constructed with the coverage data provided by Coverage.py. The database contains six tables:

- 'source file' which contains the full path to the source file, along with an id
- 'test file' which contains the full path to the test file, along with an id
- 'test function' which contains the test function name extracted from PyTest and information about where it's located, such as test file id and start and end line numbers in that test file
- 'test map' which contains information about which test function ran a specific line in a specific source file
- 'new tests' which contains information about newly added tests after the previous run
- 'last update hash' which contains information when the database was last updated

Let’s consider a scenario where the tool is used in a developer’s environment (local machine), see Figure 3.4. After the initial full test suite run, the tool is ready to be used. When changes are made to the target project's files, the tool checks for changes in the Git working directory. The tool first constructs a list of changed files according to Git and checks which of those files are either source code files or test code files in our local database. After the tool has determined which files are taken into consideration, it checks the Git diff -output for each of those files. From this 'diff', the tool can determine which lines have changed and which lines have shifted from their original position. Then the tool can query all the test functions from our database according to the list of line numbers for the changed lines and run them with PyTest. No database state updating is performed during this. If a user wishes to make these changes final, a Git commit operation is required. When the changes are committed, a CI server starts RTS process shown on the Figure 3.5 by calling the tool: the tool checks whether the current Git HEAD hash differs from the one that is marked as the last update hash. If so, the tool queries the changes and tests for those changes almost as before. As a small addition, it does two additional things. The tool calculates how unchanged lines have shifted in the files and performs a database update based on this information. It also checks for newly added test functions by checking what test functions PyTest can find and comparing it to the current state in the database. When the tests are run after this, new coverage data is collected and inserted into the database.

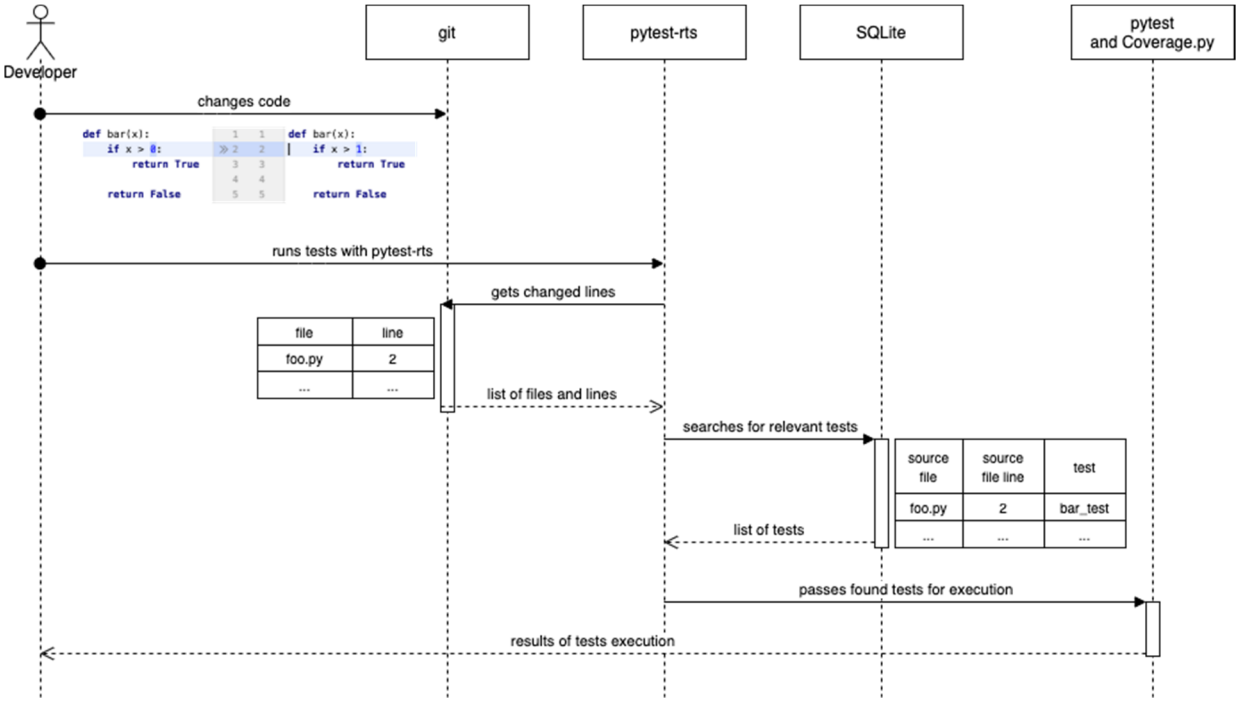


Figure 3.4: Usage of RTS tool during development.

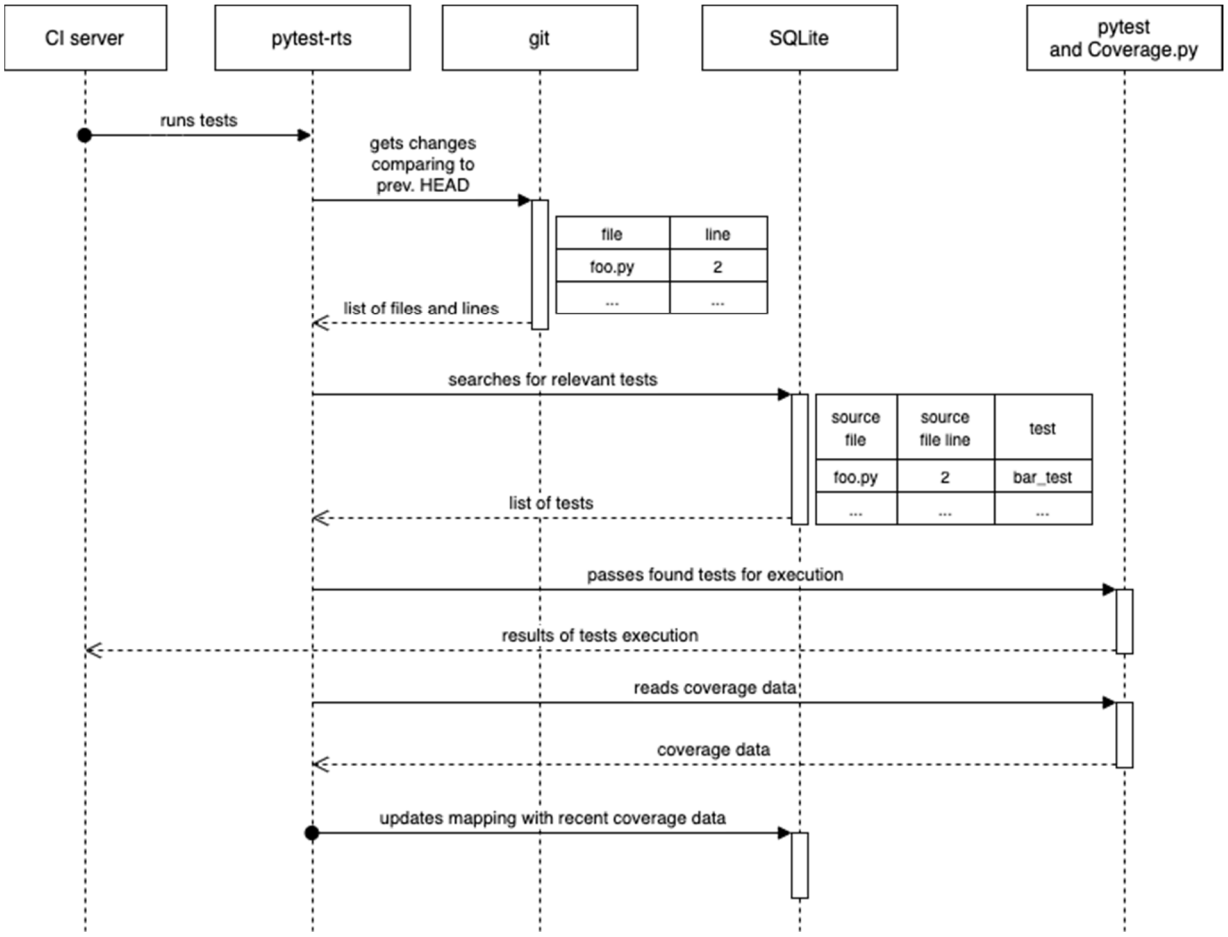


Figure 3.5: Usage of RTS tool on CI server.

3.4.3. Claimed novelty

Pytest-rts aims to be a tool that integrates into the version control system of the project under development. This allows regression test selection to be performed in a typical workflow where new features are developed in separate version control branches and a CI-pipeline is configured to run the tests. Currently, two types of tools are available for regression test selection: code coverage based and history-based. The current code coverage-based tools, such as Ekstazi and Pytest-testmon, only work on the developers' local machine by creating a mapping database completely independent from any version control data. Without the version control data linked to the mapping database, selecting correct tests between software versions becomes extremely difficult. The history-based tools, such as ChangeEngine, require data from previous test runs and the output of such tools can be inaccurate if the data is scarce.

3.4.4. Evaluation of the approach

The proper evaluation of the tool requires a diverse list of Python projects that have a test suite runnable with PyTest. We collected the following preliminary list of projects for our needs: flask, rich, pytest, chardet, dateutil, idna, python-rsa, urllib3, and wheel. The difficulty of choosing the right projects arises from the fact that open-source python projects seem to have greatly varying ways of handling dependencies and installation of the project for testing. Also, many projects require specific environments for passing their tests and that is often handled with tox.

We plan to evaluate the tool in four different ways. For each type of evaluation, we plan to examine four different metrics as proposed by Rothermel and Harrold:

- Inclusiveness, which measures if the selected tests are able to find the same faults as the full test set.
- Precision, which measures the ability to only execute relevant tests and avoid the execution of unnecessary tests.
- Efficiency, which measures the computation cost of the technique.
- Generality, which measures how well the approach is able to deal with different language constructs and complex code changes.

First, we attempt to use the historical version control data found for open-source python projects in GitHub. By going through the commits, we attempt to find out how well our tool

performs in real-life development scenarios. A difficulty regarding this approach is that projects might have to change ways of installing dependencies and running the full test suite.

The second way of evaluation is using an approach where a random source code line is deleted and our system attempts to find the relevant tests. For each iteration of the process, we can extract the line-level and file-level test sets for the change. Then we can run all the three test sets (tests for line-level and file-level changes and the full test set for the project) and compare the PyTest exit codes. The benefit of this approach is that a very large number of iterations can easily be executed. A drawback is that deletions of single lines are only one of the many possible ways how changes in a real project happen.

The third way of evaluation is very similar to the random line deletion but here we use mutation testing approaches instead. By mutating the original code with existing mutation testing tools, we attempt to produce changes to the code that better reflect real-life scenarios than just simply deleting a line.

In the fourth type of evaluation, we attempt to use the tool in a development process and collect data while doing so.

Data from the evaluation could be saved in CSV format or in a relational database. If CSV format is selected, the files could be stored in a Git repository or as artifacts in GitHub actions CI storage. If a relational database is picked for storing evaluation results, an external service could be used for storing the database.

Early evaluation results with random source line deletion indicate situations where the tool does not operate correctly. These situations include removing:

- a line defining a function. This will possibly also affect the situation when the parameters or name of the function are changed.
- An import command resulting in missing references.
- A decorator as they are not part of the executing code and thus not seen by py-test

The planned further evaluation of the tool with more realistic examples is expected to give us a better view of the limitations. This is naturally followed by experiments to improve the tool. Finally, tighter integration of the tool to the continuous integration process is expected to solve some issues but, at the same time, bring up new problems.

4. Oracle mining

4.1. Introduction

Automating the test generation from requirements expressed in ambiguous natural languages is challenging, even for controlled ones. It can be decomposed into two steps: the automatic generation of formal specifications and test generation from formal specifications. Formal specification plays the role of an oracle in testing, i.e., it specifies the relation between the inputs and the expected outputs. However, constructing formal specifications is a very challenging task. CS Canada and IVVES industrial partners are facing this challenge in testing (critical) evolving systems. Requirements describe features and functionalities of systems in terms of constraints that must hold on variables that represent concepts (e.g., input, outputs, and states) of the systems. In addition to the variable names, ambiguous words and punctuation marks from the natural languages (e.g., when, if, after, while, where, and, or, do, make, set, etc.) appear in the constraints. The ambiguity of the meaning of some words and marks, the usage of multiple variable names for the same concepts introduce uncertainty in the requirement analysis. For example, a part of a requirement can be enhanced by connecting it to a new part of the requirement via the usage of the word "where"; it is not obvious to determine the connected parts of requirements. The uncertainty leads to various interpretations of each ambiguous part of the requirements and combinations of the interpretations result in a possible vast number of plausible specifications. Approaches are needed to choose proper specifications.

4.2. State of the art and anticipated contribution

Most of the approaches to generate precise oracles or tests from requirement documents are fully automated and aims at producing precise oracles [2, 6, 4, 1, 5]. The direct translation approach makes correspondence between certain patterns in the modelling language for precise oracles and their possible representations in natural languages. The machine translation-based approach uses examples of translated requirements either to infer formal grammar for the requirements or to train a translation model with ML techniques. The approach in [4] automatically generates a precise oracle, which is compared to a manually generated precise oracle for validation of the automatic precise oracle generation procedure. In case the generated oracle is not the expected one, the approach does not propose another version to the expert.

Our contribution is a two-steps approach to generate precise oracles, as illustrated in Figure 4.1. The first step consists in generating imprecise oracles representing a set of plausible precise oracles (specifications). The second step is mining a precise oracle from the imprecise one. Mining a precise oracle corresponds to the resolution of uncertainty in the imprecise oracle in order to choose one of the plausible precise oracles. We suggest involving an expert in realizing this task.

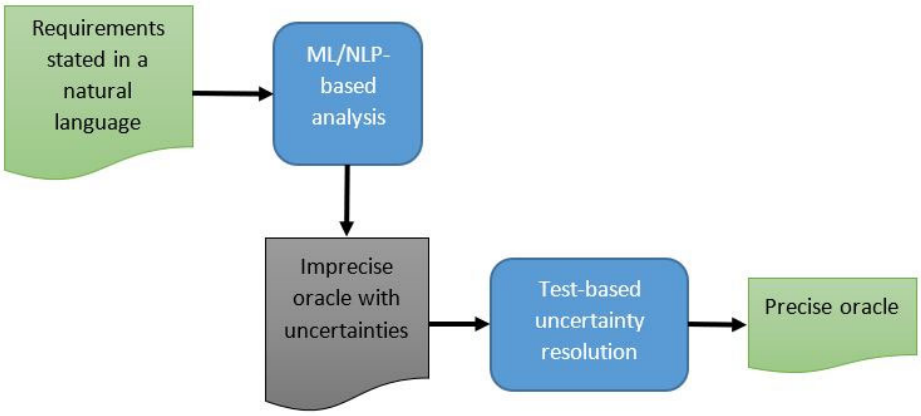


Figure 4.1: Two-steps approach to generate precise oracles

The next section presents our approach to mining a precise oracle from an imprecise one.

4.3. Proposed approach

In our approach, we represent uncertainty with nondeterministic transitions in a finite state machine. Finite state machine has been used as a formal model for evolving systems [1] and used in developing verification and validation techniques for evolving systems such as model-based testing and model-checking.

We represent a set of plausible precise oracles, called the imprecise oracle, with an input complete and non-deterministic finite state machine (FSM). A plausible precise oracle is then a deterministic and input complete submachine of the imprecise one. Each precise oracle produces a single output sequence in response to an input sequence. A test is nothing else but an input sequence. Precise oracles are distinguishable if they produce different output sequences for the same test; otherwise, they are indistinguishable.

Our approach to assisting an expert in choosing a proper precise oracle from an imprecise one works as follows:

- Randomly generate a test
- ***Loop:*** Determine the outputs which can be produced by the executions of the plausible precise oracles with the test; this is done by exploring paths of the imprecise oracle
- Ask an expert to choose the expected output; we assume that one of the outputs is expected.
- Remove from the imprecise oracle the precise ones that do not produce the chosen output sequence
- If the imprecise oracle contains only indistinguishable precise oracles, then ***return*** any one of the precise oracles as expected and ***exit***
- Else generate a test that distinguishes two precise oracles in it and **goto loop**

Let us illustrate our contribution. The nondeterministic FSM in Figure 4.2 represents an imprecise oracle for a system. It has 11 transitions t1, t2 …, t11. Its inputs a and b can represent a Boolean assertion over variables used in requirements. The outputs are 0 and 1. Uncertainty is modelled with nondeterministic transitions in states. For example, in state 3, it is uncertain whether the output is 0 or 1 on input a. The imprecise oracle defines eight plausible precise oracles. Two of them appear in Figure 4.3.

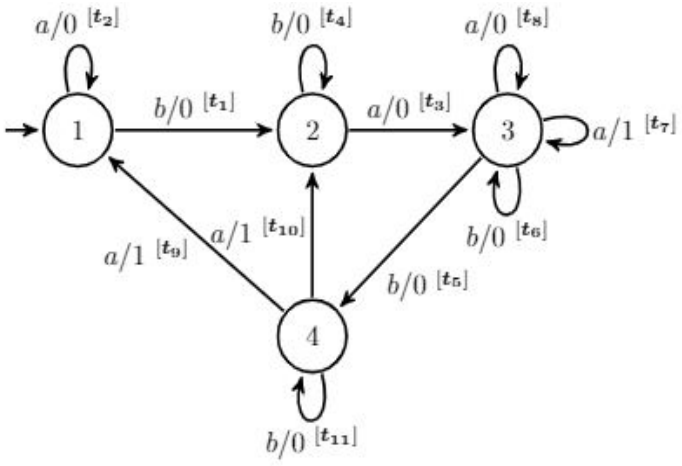


Figure 4.2: An imprecise oracle

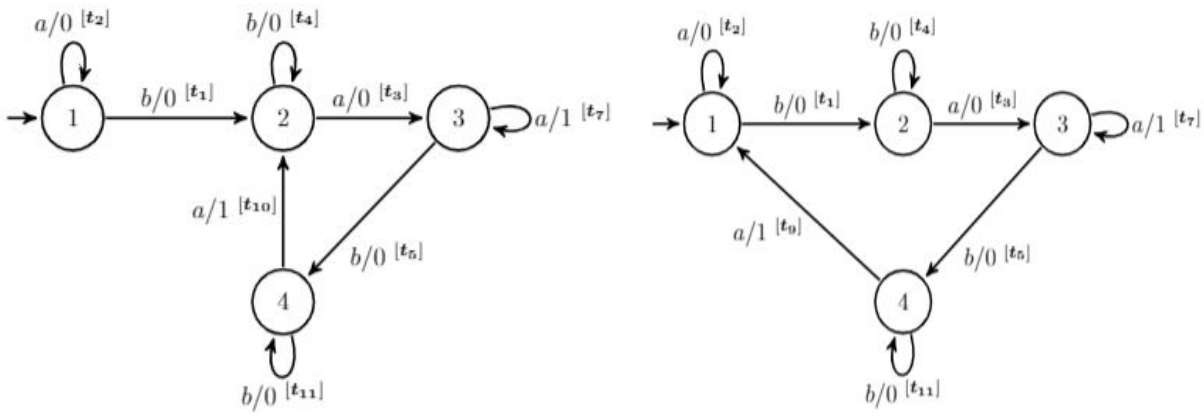


Figure 4.3: Two plausible oracles

The two plausible precise oracles are indistinguishable with test babaab because both produce output 000100 on the test. For input babaab, the eight precise oracles produce outputs 000100, 000110 and 000000. If the expert judges the output 000100 is expected, the procedure will generate the test babaaa that distinguishes between the two plausible precise oracles in Figure 2. The precise oracles produce 000101 and 000100 with the test babaaa. If an expert chooses 000101 as the expected output, then the leftmost precise specification is the expected one. Otherwise, a new test is automatically generated, and the expert is invited to estimate the plausible outputs.

Preliminary evaluation results: A preliminary version of the prototype tool is implemented in Java and it uses Z3 solver. The objective of the empirical evaluation is scalability. The empirical evaluation works as follows: We randomly generate an imprecise oracle and select a precise oracle in it who plays the role of the expert. Preliminary results indicate that the approach might scale for imprecise oracle with a limited number of states and uncertainties.

4.4. Claimed novelty

Our two-step approach to generating precise oracle is novel and suitable for evolving systems. We anticipate that it can be easier to automatically generate imprecise oracles without missing any complex interpretation of the requirements. In addition the proposed procedure for mining precise is based on the Boolean encoding of the imprecise oracle and constraint resolution. It avoids a one-by-one enumeration of every precise oracle. It proceeds by building partitions of the imprecise oracle, which is also a novelty as compared to our previous work [3].

Ongoing work includes enhancing a prototype tool for the proposed procedure and lifting the procedure to extensions of FSM with variables and complex operations on them. The variables can represent input and output ports of evolving systems on these variables. We are also investigating the ML/NLP based generation of imprecise specifications for requirement documents, especially requirements used by the IVVES industrial partners.

5. Automating test verdict generation via Model Learning

The Oracle problem [1] is the main reason why engineers have to be involved in QA process. Though it is possible to automatically detect application crashes and situations where SUT doesn't respond to commands, most of the defects are logical, for example display of wrong information or wrong calculations. Traditional testing of a given SUT uses the following basic types of verdicts, pass, inclusive, and fail, which are made considering the results of test execution. Considering the testing scenario when several models of an evolving SUT are inferred during the continuous process of test execution or online testing (monitoring the SUT), one should use different types of test verdicts, made after the models are compared. In particular, when one model is declared to be a so-called "reference model" then the detected deviation from it in another model could be classified either as a "defect" or "extension/enhancement". The undetermined change would result in the verdict "undetermined" to be later refined by the domain expert to a determined one.

5.1. State of the art

The earliest applications of ML to software testing date back to the pioneering work of Budd [38] and Weyuker [39]. During the 1990s and early 2000s, Inductive Logic Programming (ILP) was considered as a model learning paradigm for model-based test case generation [40] but it has been unclear what range of behaviors can be learned by ILP. Recently, alternative modeling and inference approaches have been considered, such as learning algebraic specifications [41] and learning decision trees [42]. Not all such approaches automate the important test oracle step (i.e. test verdict generation) e.g. Briand et al. [43] argues to keep the human in the loop. However, when test suites are large (e.g. > 1 million test cases) it seems clear that automation of the oracle step is also necessary. This is currently being tackled by methods such as metamorphic testing.

5.2. Anticipated contribution

Open source TESTAR tool, scriptless GUI testing, and state model inference during automated GUI exploration are described in "Test prioritization" section.

During this research, we would like to use a classical supervised classification approach rather than metamorphic testing. Classical supervised classification approach consists of three steps:

1. Data collection;
2. Samples (changes) labeling;
3. ML model training.

Data collection step consists of two parts: model inference for the current application version and storage of the model. When state models are automatically inferred from new SUT versions from a CI system, state models of consequent system versions can be automatically compared to detect changes. Detected changes are then labelled during the labeling step. A semi-automatic approach could be applied to it. All the changes are marked as "extension/enhancement" if all the tests in the main TA pipeline passed otherwise changes are visualized and presented to a QA engineer to identify changes that are defects and are not in the model.

Training is then performed to infer the ML model which is able to predict the type and importance of each change based on collected data. ML model training will be performed on different feature sets from different model levels which may include displayed information and screenshots of the current SUT screen or page, types of transitions between new and existing states.

5.3. Claimed novelty

Recently, TESTAR has been extended with the state model inference and proof-of-concept implementation of model comparison functionality. This new functionality has not been evaluated in an industrial environment yet. Finding the right set of features for model training from collected data on state machine changes and using machine learning for predicting the type and importance of the detected changes is a novel approach pushing the boundaries of the state-of-the-art.

6. Conformal prediction for edge applications

6.1. Introduction

Many machine learning systems involve making predictions, through estimating the value of a dependent variable with an *a priori* unknown ground truth. Verification of such predictive systems, in particular when applied in high-risk applications, is crucial. Traditional machine learning algorithms and means of validation, however, generally lack capabilities for establishing trustworthy verification. Established common-use validation procedures tend to be biased, leading to error frequency on production data not corresponding with error frequency on test data; and, established common-use validation procedures tend to perform verification on a macroscopic level (per-model) rather than a microscopic level (per-prediction), leading to difficulties in verification of individual predictions.

The conformal prediction framework offers an alternative method for constructing and evaluating predictive models that appears better suited than traditional predictive methods in applications where thorough verification is crucial, [44, Ekkono 2, 46, 47, 48, 49]. Whereas traditional predictive models output so-called *point predictions*—a single-valued best-guess prediction for the value of the dependent variable—conformal predictors output multi-valued *prediction regions* that represent a range of likely value assignments for the dependent variable, constrained by its domain. Any prediction region produced by a conformal predictor comes associated with a very specific, statistically valid, expectation: that the *a priori* probability of the prediction region containing the ground truth value of the dependent variable is fixed and known. Under these conditions, model and prediction verification becomes straightforward, as each prediction is guaranteed to contain the correct value of the dependent variable with a user-specified probability. Figure 6.1 presents the relation between classical training algorithms and conformal prediction algorithms. Calibration data are used to generate prediction regions encompassing prediction points.

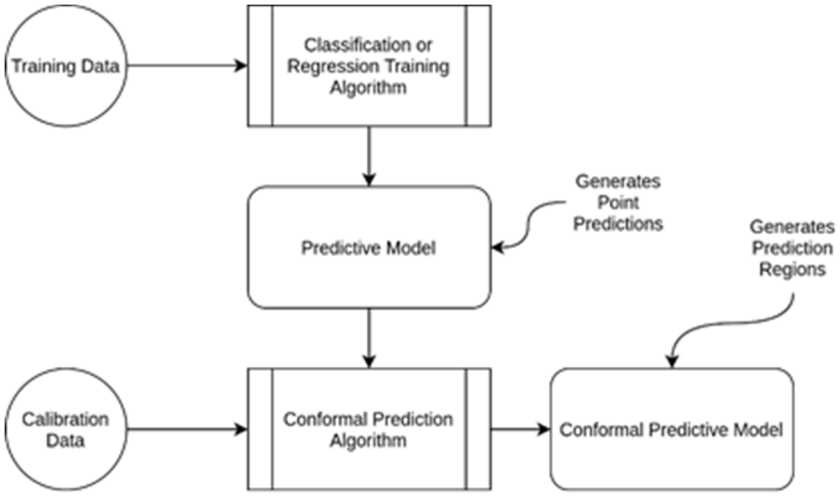


Figure 6.1: Relation between classical training algorithms and conformal prediction algorithm

6.1.1. Technical description

Conformal predictors are able to produce predictions on without any knowledge of the shape or parameterization of the data generating distribution (unlike, e.g., Bayesian methods); the only requirement is that the observed data sequence is exchangeable (a looser requirement than the typical i.i.d. assumption). In addition, the conformal prediction framework is model agnostic, in the sense that it can be applied on top of an arbitrary machine learning algorithm (e.g., classification or regression algorithm) and transform the point predictions of the underlying algorithm into prediction regions that exhibit the expected statistical guarantees.

Given a (calibration) set of instances $\{(x_i, y_i)\}_{i=1}^{n}$ and a predictive model h, the approach is to evaluate how good the individual predictions $h(x_i)$ are. To this end, a *non-conformity function*, f, is introduced and its role is to measure how unexpected is a pair (x_i, y_i). A common choice for the non-conformity function is simply the absolute difference between the prediction and the target, i.e., $f(x_i, y_i) = |y_i - h(x_i)|$.

Applying the non-conformity function on the calibration set will produce an empirical distribution, against which new data is compared to. For a new data point x_i, all possible outputs y^* are evaluated using the non-conformity function. The prediction region of x_i then includes all outputs y^* that together with x_i would form a plausible observation in comparison to the calibration set, or more precisely, those outputs y^* that can't be rejected through a standard statistical test based on $f(x_i, y^*)$. Figure 6.2 illustrates the process to compute the prediction region for an instance x_i.

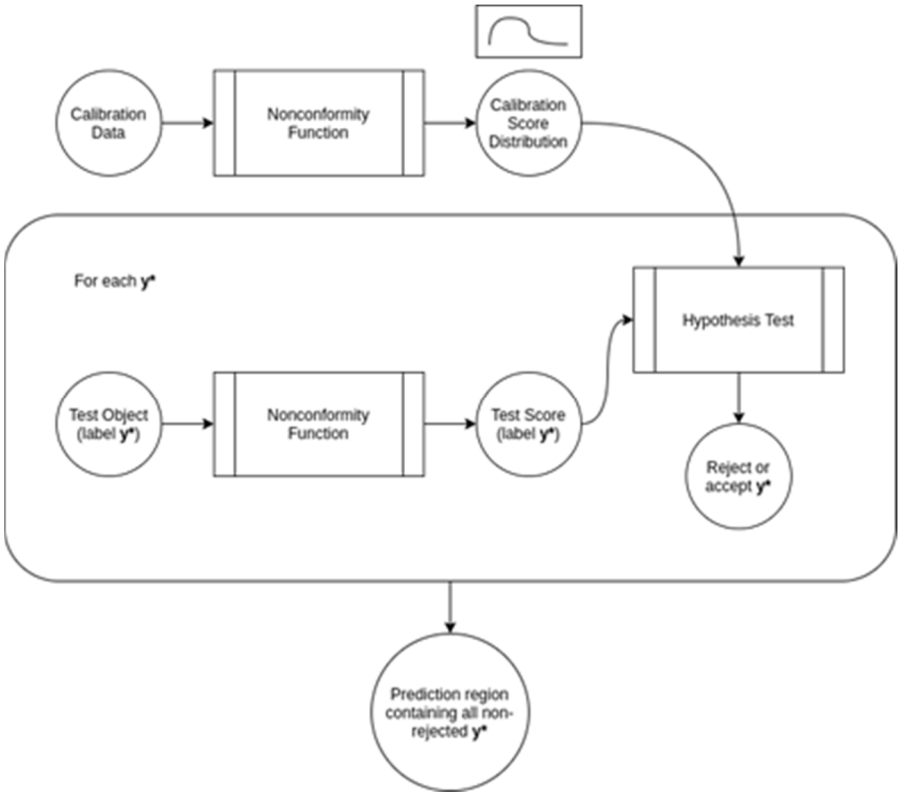


Figure 6.2: Computation of the prediction region

6.1.2. Example

To illustrate the process of conformal prediction we have looked at a dataset consisting of daily mean temperature from a weather station on mars, taken from NASA website [81]. The aim is to predict tomorrow's temperature based on the last few days of data and for that purpose we train a linear regression model, which can be seen in the left panel of Figure 6.3. It appears that the data is somewhat noisy and, without model evaluation, the point predictor from the linear regression gives no insight of how much we trust our predictions.

In the right panel of Figure 6.3, one can see the output of the conformal prediction algorithm when applied to the linear regression model. The chosen confidence level is 0.1, meaning that the prediction interval will contain the true value at on average 90% of the instances. Note that the width of the prediction interval varies with the accuracy of the underlying model. If error of the underlying model is particularly bad in certain regions, the prediction interval will be wider in those regions. Note also that the prediction interval is not necessarily symmetrical around the underlying model. In the first 100 or so days of the test set, the linear regression output is higher than the target, which will be accounted for by the

conformal prediction algorithm. Conformal predictors with this feature are called *balanced*, see for example [82].

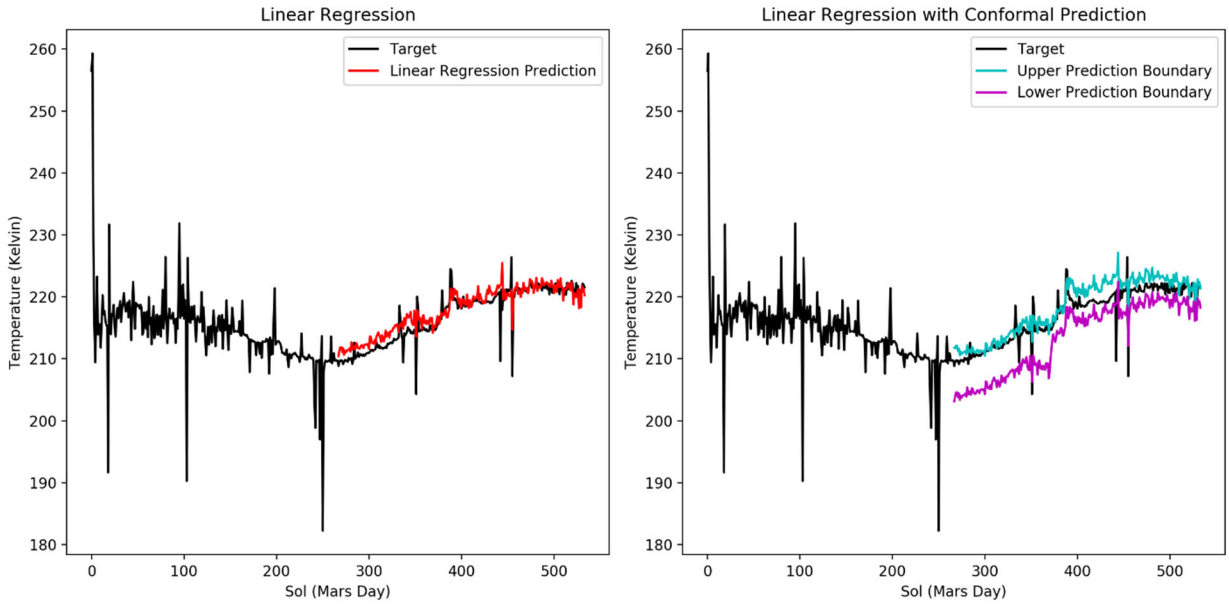


Figure 6.3: Target and prediction values for temperatures

6.2. State of the art and anticipated contribution

Conformal prediction has been successfully applied to problems in several domains: most notably that of drug development, see for example [50, 51, 52]. Developing a new pharmaceutical drug, from initial compound design through clinical testing, can cost as much as one billion dollars, and discarding non-promising candidates early in the process is key for economic sustainability. Conformal prediction offers an opportunity to perform statistical modelling of chemical compounds already in the design stage, and allows for early filtering of drug candidates of poor efficacy, while simultaneously controlling for false negative rate (i.e., limiting the likelihood of erroneously disqualifying efficacious drug candidates).

Pharmaceutical drug development is a rather particular industry, however,it has a high degree of maturity regarding the use of statistical and predictive modelling. In most other industrial application areas, with a lesser degree of maturity in statistical modelling, knowledge and use of conformal predictive systems is substantially rarer. Simultaneously, conformal predictors, in particular conformal regression models, appear well-suited for applications in a wide variety of industries, both as a method for providing enriched decision support in human-

computer systems and as a trustworthy component in autonomous or semi-autonomous digital or cyber-physical systems.

6.3. Claimed novelty

Since applications of conformal prediction mostly have been carried out within the pharmaceutical domain, implementations tend to focus on large scale solutions. Using our expertise in small scale solutions on the edge, we have implemented an efficient version of conformal prediction, able to run on edge applications. The implementation supports online re-calibration of the conformal predictor, to allow for individualized learning, and adapting to change. The provided implementation of conformal prediction will be further extended to support fully online training on the edge, without the need to supply any prior training data. We at Ekkono have spent resources developing our conformal prediction framework partly because we believe that it has potential to be impactful not just in the pharmaceutical industry but in many industrial settings were ML in general is applicable.

7. Code defect risk prediction

7.1. Introduction

Code review in software development is a key step for QA in finding errors that may have been missed during the development phase and to minimise the risk of bugs or defects in the production phase. Because code review is a complex task, especially with complex evolving software products; manual reviews and testing will never find every bug in the source code, so there is always a margin of code change that will not be checked as thoroughly it should. It has also been shown that certain parts of source code are more prone to defects than other parts, so there is a need for an automated, smart approach to recognise project features that are significantly at risk of defect. A more extensive review of state-of-the-art approaches to code anomaly detection has been provided in ITEA IVVES D 3.1, paragraph 3.2.1 [53].

We believe that by assessing risk earlier in development and more accurately, QA activities can be better organised and focused – such as extending code review in risk areas and effectively prioritising test cases. This will ultimately add value by enhancing the quality of the software while shortening the delivery/release cycle.

7.2. Anticipated contribution

There are many approaches that study automated code analysis and rate the quality of source code. Apart from human rating, like peer review and static code analysis (quality analysis of the code without executing the program), machine learning is the approach that we propose to use. Machine learning can be used to automatically identify well written or poorly written code based on a machine learning classification method. In this deliverable, the concept of code defect risk prediction using a machine learning approach will be explained.

Tools and methods used to collect necessary input data for the code risk prediction ML model are explained in the next paragraphs. They include:

1. Static code analysis packages and tools, which assess the **structural quality aspect** (non-functional requirements evaluated through analysis of the inner code structure); and
2. Code coverage and version control information, which tackle the **functional quality aspect** (how well the code fulfils its functional requirements and specifications).

In our approach, we use both static code analysis tools' output and linked bugs and defects to train the model.

Tools and methods used to collect necessary input data for the code risk prediction ML model are explained in the next paragraphs and include static code analysis packages and tools, which tackle the **structural quality aspect** (non-functional requirements evaluated through analysis of the inner code structure) and code coverage and version control information, which tackle the **functional quality aspect** (how well the code fulfils its functional requirements and specifications). In our approach, we use both data sources to train the model. To accommodate this, we suggest a specific DevOps workflow to be used to collect this data in future software projects (Figure 7.2). Additionally, we include an explainable layer to the model, for transparent results.

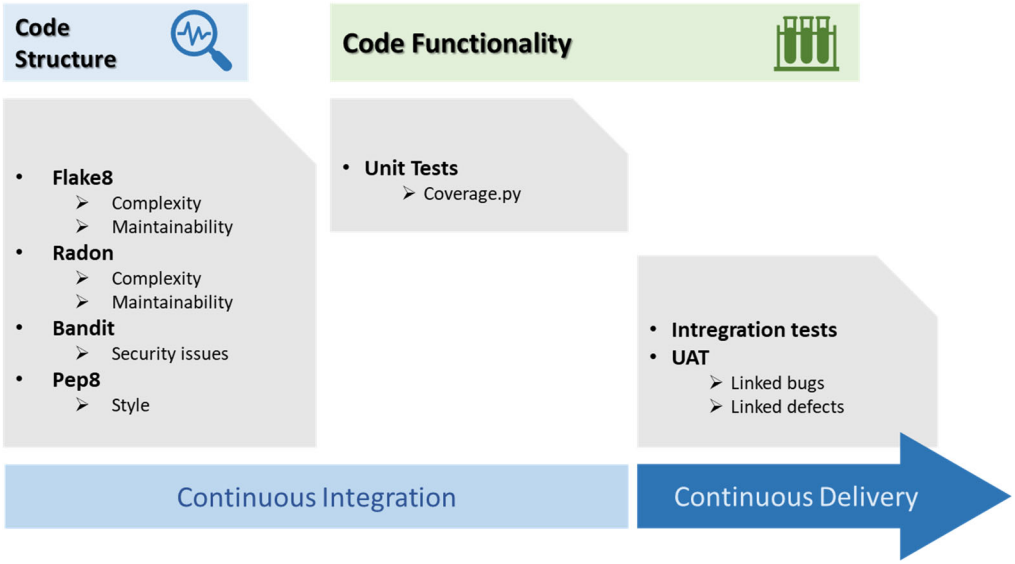


Figure 7.2: Structural and functional code quality data collection steps in the CI/CD workflow for code quality risk prediction

7.3. Proposed approach

7.3.1. Code coverage: static code analysis

When we evaluate the quality of the source code, we typically want to measure the following aspects of the system [54]:

1. Reliability: the probability that the system will run without failure.

2. Maintainability: how easily the software can be maintained relating to the size, consistency, structure and complexity of the codebase.
3. Testability: how well the software supports testing efforts – how you can control, observe, isolate, and automate testing.

Along with these metrics, we can use manual code review and static code analysers to measure the maintainability and understandability of the code.

Static code analysis is a method of evaluating the complexity and possibility of erroneous code. Code coverage tools give insight into effectiveness of a test suite for an application. These methods are executed at a granular level and require developer action [55]. Luckily, they are easily implemented in the development workflow. Please note, the suggested packages and tools are for Python, but the methods can be applied to any programming language.

We propose using **Flake8** [56], a wrapper around PyFlakes, pycodestyle and McCabe script, with numerous plugins available for static code analysis and code coverage [57], like **Radon** [58]**, Bandit** [59], and a popular code coverage tool **Coverage.py** [60].

Types of static code analysis methods that these packages assess can be classified in the following way:

1. Code complexity assessment;
2. Raw metrics assessment, including lines of code, comments, blank lines;
3. Halstead metrics assessment, metrics for SW code assessment set in 1977 to evaluate measurable properties of software [61];
4. Maintainability;
5. Style assessment;
6. Security issues like passwords, use of assert, binding to interfaces, weak cryptography, SQL injection and others;
7. Code coverage measurements like statement coverage (number of code lines executed by the test suite and the number that haven't), function coverage (how many functions have been ran, reveals dead code), branch coverage (in if statements, have the conditions been met), and condition coverage (has every Boolean sub-expression within a program been evaluated to both true and false during the execution of a test suite) [62];
8. Compiler anomaly detection can be considered as one of possible techniques to improve code quality assessment. Python is an interpreted, not a compiled language, however, it does contain a compilation part. The code gets compiled to bytecode and stored to a .pyc

file to increase the speed of the execution. If the code doesn't change, the .pyc file doesn't change either [63].

7.3.2. DevOps pipeline: integrating code coverage tools

Static code analysis is an efficient way to assess the code that was built locally, before deployment to other environments. Since the state of the software industry requires quick deployments from local (feature) branches to develop, master and release ones, it is of crucial importance to do checks between them and report any commits that may lead to bugs. This can be done through several Python packages that need to be integrated in the CI/CD flow in order to accomplish this.

Continuous integration refers to merging code changes frequently to master. For this to be established, a pipeline configuration tool is used. We propose the use of either:

- **Git CI**. It relies on a .yml file which creates a pipeline and runs for changes to the code in the repository. This pipeline has one or more stages that run and can contain one or more jobs that run in parallel, executed by GitLab running agent [64].
- **Jenkins**, an open-source, MIT licenced tool with easy configuration [65].

Since we are using either Git CI or Jenkins, we will be focusing on tools that can be integrated with them:

1. **Wily**, an application for tracking the complexity of Python code in tests and applications. Those measures include cyclomatic complexity, Halstead metrics, raw metrics and maintainability. Wily can be integrated in the CI pipeline and used to compare scripts between different branches, making it very useful for assessment of code between environments [66].
2. **Flake8**, with its plugins [57].

Static code analysis pipeline integration can be used to stop the commit changes from being pushed to develop, release or master branches in case of risky, flaky code. This is an immediate effect, and, in case this happens, it is developer's responsibility to attend to faulty code. It is good to note that the sensitivity of the pipeline stages can be adjusted, meaning the project can be configured for most efficient deployment. In case the build succeeds, CI tools will output a log or xml file with the static code and unit test results. This file is stored locally or fed into a database for later use in the ML model.

7.3.3. Traceability: version control and defects

Continuous integration is followed by a continuous delivery pipeline, which ensures quick deployment into an environment where further testing is done. Usually, testing includes integration and user acceptance tests. Unlike the continuous integration pipeline, which focuses on structural and functional changes, where latter is assessed through unit test runs, integration and user acceptance tests evaluate only the functional aspect of code changes.

Linking defects and bugs to commits depends heavily on the project DevOps configuration. Jira and Jenkins integration can be used. It is important to make a clear link between a commit and a reported bug so that this information can be used to train the ML model. The approach in data collection allows for the possibility of classifying code commits into risky and non-risky ones, with a possibility of future improvements in the ML model classification granularity. We aim to classify code changes by types of risk they lead to.

7.3.4. Machine learning: predicting code quality

To improve the assessment of source code quality before testing begins, we propose using a **Random Forest Classifier** as an innovative method to detect the probability of faults and thus determine if the quality of code is risky or not [67]. We will use defects as our objective for our machine learning outcome.

To predict the probability of defect occurring in a feature, we need to understand where bugs are introduced. Source code changes and how they are associated to features will be significant in analysing risk areas. We will **predict code risk and thereby probability of defect, by predicting the features that will be affected by a change in source code or file.** The expected outcome of the algorithm is a distribution of probabilities of the file that effects different software features. This will **quantify the risk of new code changes** [68]**.**

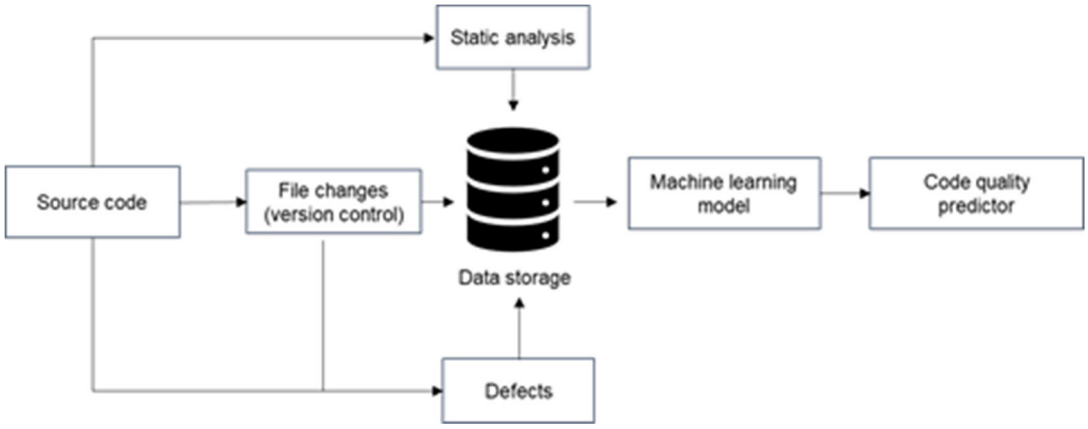


Figure 7.2: Code defect risk prediction model architecture

Data sources:

To create an algorithm that identifies code risk features we use a dataset that contains:

- File changes associated with features – version control commit metadata, including static code and code coverage analysis results;
- Defect metadata – identified bugs linked to feature.

Risks [69]:

- Historical defect data: to construct any defect predictor, we need access to historical data. Oftentimes historical data is not available for new projects which can make successful prediction challenging. We can mitigate this by using available data from similar projects.
- Data quality: quality and accessibility can be a constraint to the effectiveness and accuracy of the model. data traceability will be key to integrating multiple types of data that will allow us to form statistically significant conclusions about the cause of defects in different features.

Risk mitigation:

To address the above risks and ensure that we have a robust and high performing code quality model, we will use public data from open-source projects. This way we will have more observations of commits linked to bugs, and we can use this data to train our model in identifying risky features. By using similar open-source projects, we can then apply the model to new and existing projects that do not have historical data or high data quality.

Assumptions:

- Workflow is DevOps, with a setup CI/CD pipeline;
- There are change files associated with the code change and the bug fix for a given feature;
- There should be enough data points to the required level of granularity of the features;
- Automated changes that are generated by automated systems or scripts and not by a developer should be excluded as they do not represent code changes from a developer.

Model interpretability:

For the software developer to interpret why the model predicted a risk area and likelihood of defect, we can implement an explainable AI method to understand the results and provide transparency so that the developer or tester can make better decisions.

Our approach is based on a random forest model which consists of many deep trees. Each tree is trained using a random selection of features. These models are seen as black-boxes as each tree can have thousands of nodes, and getting a full understanding of the model's process by examining each tree, is not feasible. However, we can turn a random forest algorithm into a white-box by **computing the sum of feature contributions** which shows how the features led to an individual prediction [70]. This is a straightforward and easy to implement method that can make the model more interpretable. We can run the treeinterpreter (python library) [71] interpretation algorithm on python's scikit-learn's [72] decision tree and random forest model, which will explain the local predictions. Thus, the developer or tester will be able to understand the model's decision process better and make more effective decisions regarding code and test coverage.

7.4. Claimed novelty

As industrial systems evolve and become increasingly complex, we need to find new ways to manage the sustainable growth of development and testing operations. Traditional code coverage and testing approaches will require large amounts of resource investment to maintain quality. However, we still would not be able to cover all risks thoroughly. Thus, it's imperative that we implement a smart system, in combination with static code analyser and IDE tools, that will enable us to move toward risk-based testing. By implementing an automated machine learning

code risk model, we can more accurately predict the quality of our code and where the high-risk features are, which will enable the prioritisation of test cases and better allocation of resources. This will result in compliant development and a more efficient and effective QA cycle. This type of smart testing pipeline integration tool along with a machine learning model to predict code quality, has not been seen on the market or rarely implemented into CI pipelines. This is mainly because of legacy systems, data quality and collection issues and a non-agile way of working. Most of the tools that claim to predict code risk, are rule-based and do not use machine learning methods. Our proposed technical solution is a novelty in the software industry and will shift the way of working towards compliant & quality development.

8. Unsupervised anomaly detection for visual inspection in industrial environments

Grupo Antolin holds a 50% shareholding in Keyland. Grupo Antolin is one of the largest manufacturers of vehicle interiors in the world, and achieved sales amounting to €5,214 million in 2019. Keyland is an ICT (Information and Communication Technology) provider for the automotive sector in Spain, being a key partner to generate innovative solutions.

The complexity of car interiors -with surfaces made up of many more layers, with a lot more choice in colors- is increasing. The greater demand for personalization from end users is pushing the providers to adapt fast and efficiently to this continuously changing environment. Optical quality inspection is still one of the common methods to ensure quality control. This is preventing full automation and integration towards the Industry 4.0 concept. The increasing variability in colors, shapes, textures, positions, etc. renders the optical inspection impracticable, which has led this sector to look for automating defect inspection. Artificial intelligence (AI) is a promising technique for automated inspection.

Grupo Antolin Inspection Operator

Grupo Antolin QA Operator

The growing demand for automation in this context, together with the great variability is stressing ICT providers, that must face a great flexibility. The software must easily evolve and adapt to different production lines, with shortened lifecycles. This variability makes the identification of anomalies both in datasets and AI system behavior more challenging. Continuous online monitoring of AI systems is required to ensure reliability during operation phases.

8.1. State of the art

Anomaly detection is focused on two main common types: outliers and mislabelled data. At this stage we focus on detection of outliers, both in datasets and in the outcomes of the AI system. A good reference illustrating the benefits of incorporating anomaly detection into real-world ML systems is the Out-of-Bag anomaly detection [73]. Its main goal -integrating anomaly detection as a pre-processing step- is to improve the accuracy of the ML models. This approach could reduce uncertainty when a new product line must be set up, and changes in textures, colors, materials… must be faced. Other reference directly focused on image analysis [74], proposes a method to detect anomalies by directly isolating anomaly pixels from background. Anomaly detection also has the capability to summarize the status of a complex system with a single indicator (anomaly score). In this context, Functional Isolation Forest exhibits a great flexibility in detecting anomalies for a variety of tasks [75].

8.2. Anticipated contribution

KEYLAND is focused in online monitoring of AI systems for automated inspection of car interiors, exploring techniques for anomaly detection. For this, Isolation Forest is being used. Isolation Forest [76] is similar to Random Forest. It is based on decision trees. However, Isolation Forest is an unsupervised learning algorithm that explicitly identifies anomalies, instead of profiling normal data points. It identifies anomalies or outliers, isolating observations by randomly selecting a feature and then randomly selecting a split value between the minimum and maximum values of that selected feature.

Since fully unsupervised learning may not achieve enough reliability due to increasing variability, we are considering the integration of expert knowledge in the detection algorithms, remarking occurrences (i.e., unforeseen events, uncommon states or configuration...) as “relevant”. This is in line with other ongoing works [77], though directly focused on visual inspection.

8.3. Claimed novelty

So far, we haven't identified any solution designed for anomaly detection for visual inspection systems deployed in a real industrial environment. KEYLAND is focused in generating a workflow that can be applied in a real, industrial environment. We believe that our contributions could eventually provide more reliable and trustworthy systems, addressing the challenges identified for visual inspection in different stages within the automotive sector.

9. References

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, S. Yoo, “The Oracle Problem in Software Testing: A Survey” IEEE Transactions on Software Engineering, 2015

[2] I. Buzhinsky, "Formalization of natural language requirements into temporal logics: a survey," 2019 IEEE 17th International Conference on Industrial Informatics (INDIN), Helsinki, Finland, pp. 400-406, 2019.

[3] O. Nguena Timo, Alexandre Petrenko, S. Ramesh, “Using Imprecise Test Oracles Modelled by FSM”. ICST Workshops, pp. 32-39, 2019.

[4] J. Galvani Greghi, E. Martins, A. Maria Brito, R. Carvalho, “Semi-automatic Generation of Extended Finite State Machines from Natural Language Standard Documents”, DSN Workshops, pp. 45-50, 2015.

[5] F. Pudlitz, F. Brokhausen, A. Vogelsang, “Extraction of System States from Natural Language Requirements” RE 211-222, 2019

[6] C. Gustavo, S. Augusto, “Formal Specification Generation from Requirement Documents”, Electronic Notes in Theoretical Computer Science. 195. 171-188, 2008.

[7] S. Bauersfeld, T. Vos, “A reinforcement learning approach to automated gui robustness testing”, In Fast Abstracts of the 4th Symposium on Search-Based Software Engineering (SSBSE), IEEE, pp. 7–12, 2012.

[8] A. Esparcia-Alcázar, F. Almenar, T. Vos, U. Rueda, “Using genetic programming to evolve action selection rules in traversal-based automated software testing: results obtained with the TESTAR tool”, Memetic Computing 10(3): 257-265, 2018.

[9] Y. Miao and X. Yang, “An FSM based GUI Test Automation Model”, the 11th International Conference on Control, Automation, Robotics and Vision (ICARCV), pp. 120-126, 2010.

[10] P. Aho, N. Menz, T. Räty and I. Schieferdecker, "Automated Java GUI Modeling for Model-Based Testing Purposes," In Eighth International Conference on Information Technology: New Generations, pp. 268-273, 2011.

[11] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes”, ACM Trans. Web 6, 1, Article 3 (March 2012), 30 pages. DOI:https://doi.org/10.1145/2109205.2109208

[12] P. Aho, M. Suarez, T. Kanstrén and A. M. Memon, "Murphy Tools: Utilizing Extracted GUI Models for Industrial Software Testing," Seventh IEEE International Conference on Software Testing, Verification and Validation Workshops, pp. 343-348, 2014.

[13] S. Pimont and J. Rault, “A software reliability assessment based on a structural and behavioral analysis of programs”, In Proc. of the 2nd international conference on Software engineering (ICSE). IEEE, pp. 486–491, 1976.

[14] T. Chow, “Testing Software Design Modeled by Finite-State Machines”, IEEE Trans. on sw. eng., vol. SE-4, no. 3, 1978.

[15] V. Cortellessa, A. Di Marco, P. Inverardi, “Model-based software performance analysis”, Springer Science & Business Media, 2011.

[16] M. Harchol-Balter, “Performance modeling and design of computer systems: queueing theory in action”, Cambridge University Press, 2013.

[17] K. Kant, M. M. Srinivasan, “Introduction to computer system performance evaluation”, McGraw-Hill College, 1992.

[18] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni, “Model-based performance prediction in software development: A survey”, IEEE Transactions on Software Engineering, 295-310, 2004.

[19] H. Koziolek, “Performance evaluation of component-based software systems: A survey. Performance evaluation”, pp. 634-658, 2010.

[20] P. Zhang, S. Elbaum, M. B. Dwyer, “Compositional load test generation for software pipelines”, In Proceedings of the International Symposium on Software Testing and Analysis pp. 89-99, 2012.

[21] V. Garousi, “A genetic algorithm-based stress test requirements generator tool and its empirical evaluation”. IEEE Transactions on Software Engineering, 36(6), 778-797, 2010.

[22] M. B. da Silveira, E. D. M. Rodrigues, A. F. Zorzo, L. T. Costa, H. V. Vieira, F. M. de Oliveira, “Generation of Scripts for Performance Testing Based on UML Models”, In SEKE, pp. 258-263, 2011.

[23] C. Lutteroth, G. Weber, “Modeling a realistic workload for performance testing”. In 12th International IEEE Enterprise Distributed Object Computing Conference, pp. 149-158, 2008.

[24] H. Schulz, D. Okanović, A. van Hoorn, V. Ferme, C. Pautasso, “Behavior-driven load testing using contextual knowledge-approach and experiences”, In Proceedings of the ACM/SPEC International Conference on Performance Engineering, pp. 265-272, 2019.

[25] V. Ferme, C. Pautasso, “A declarative approach for performance tests execution in continuous software development environments”, In Proceedings of the ACM/SPEC International Conference on Performance Engineering, pp. 261-272, 2018.

[26] B. Settles, “Active learning literature survey”, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[27] R. S. Sutton, A. G. Barto, “Reinforcement learning: An introduction”, MIT press, 2018.

[28] H. M. Moghadam, “Machine Learning-Assisted Performance Assurance”, Licentiate Thesis, Mälardalen University, 2020.

[29] M. H. Moghadam, M. Saadatmand, M. Borg, M. Bohlin, B. Lisper, “Poster: Performance Testing Driven by Reinforcement Learning”, In IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pp. 402-405, IEEE, 2020,

[30] M. H. Moghadam, “Machine learning-assisted performance testing”. In Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1187-1189, 2019.

[31] M. H. Moghadam, M. Saadatmand, M. Borg, M. Bohlin, B. Lisper, “An Autonomous Performance Testing Framework using Self-Adaptive Fuzzy Reinforcement Learning”, arXiv preprint arXiv:1908.06900, 2019.

[32] G. Hamidi, “Reinforcement Learning Assisted Load Test Generation for E-Commerce Applications”, Master thesis, Mälardalen University, 2020.

[33] J.O’Duinn, The financial cost of a check in (2013). Accessed 2020-08-11.

[34] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, “Usage, costs, and benefits of continuous integration inopen-source projects,” in2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 426–437, 2016.

[35] G. Rothermel and M. J. Harrold, “Analyzing regression test selection techniques” IEEE Transactions on software engineering, vol. 22, no. 8, pp. 529–551, 1996.

[36] B. G. Ryder, F. Tip, “Change impact analysis for object-oriented programs,” in Proceedings of the 2001ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp. 46–53, 2001.

[37] M. Gligoric, L. Eloussi, D. Marinov, “Ekstazi: Lightweight test selection,” in2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2. IEEE, pp. 713–716, 2015.

[38] T. A. Budd, D. Angluin, “Two notions of correctness and their relation to testing”, Acta Informatica 18(1), pp. 31–45, 1982.

[39] E. J. Weyuker, “Assessing test data adequacy through program inference. ACM Transactions on Programming Languages and Systems (TOPLAS) 5(4), pp. 641–655, 1983.

[40] R.D. King, K.E. Whelan, F.M. Jones, P.G. Reiser, C.H. Bryant, S.H. Muggleton, D.B. Kell, S.G. Oliver, “Functional genomic hypothesis generation and experimentation by a robot scientist”, Nature 427(6971): pp. 247–252, 2004.

[41] J. Henkel, A. Diwan, “Discovering algebraic specifications from java classes”, In: European Conference on Object-Oriented Programming, Springer, pp. 431–456, 2003.

[42] P. Papadopoulos, N. Walkinshaw, “Black-box test generation from inferred models”, In: Proceedings of the Fourth International Work- shop on Realizing Artificial Intelligence Synergies in Software Engineering, IEEE Press, pp. 19–24, 2015

[43] L. C. Briand, Y. Labiche, Z. Bawar, N. T. Spido, “Using machine learning to refine category-partition test specifications and test suites”, Information and Software Technology 51(11): pp. 1551–1564, 2009.

[44] V. Vovk, A. Gammerman, G. Shafer, “Algorithmic learning in a random world”, Springer Science & Business Media; 2005.

[45] G. Shafer, V. Vovk, “A tutorial on conformal prediction”, Journal of Machine Learning Research, vol 9, pp. 371-421, 2008.

[46] C. Saunders, A. Gammerman, V. Vovk, “Transduction with confidence and credibility”, pp. 712-726, 1999.

[47] V. Balasubramanian, S.S. Ho, V. Vovk, editors. “Conformal prediction for reliable machine learning: theory, adaptations and applications”, Newnes, 2014.

[48] U. Johansson, H. Boström, T. Löfström, H. Linusson, “Regression conformal prediction with random forests”, Machine Learning, vol 97(1-2), pp. 155-76, 2014.

[49] H. Linusson, U. Norinder, H. Boström, U. Johansson, T. Löfström, “On the calibration of aggregated conformal predictors”, In Conformal and probabilistic prediction and applications, pp. 154-173, 2017.

[50] J. Alvarsson, S.A. McShane, U. Norinder, O. Spjuth, “Predicting with confidence: Using conformal prediction in drug discovery”, Journal of Pharmaceutical Sciences, 2020.

[51] N. Bosc, F. Atkinson, E. Felix, A. Gaulton, A. Hersey, A.R. Leach, “Large scale comparison of QSAR and conformal prediction methods and their applications in drug discovery”, Journal of cheminformatics, vol 11(1), p. 4, 2019.

[52] M. Eklund, U. Norinder, S. Boyer, L. Carlsson, “The application of conformal prediction to the drug discovery process”, Annals of Mathematics and Artificial Intelligence, vol 74(1-2), pp. 117-32, 2015.

[53] M. Pashkovisky, et. al, “State of the art of validation methods and techniques for complex evolving systems,” ITEA, 30-Jun-2020. [Online]. Available: https://itea3.org/project/ivves.html. [Accessed: 2020].

[54] R. Bellairs, “What Is Code Quality? And How to Improve Code Quality,” Perforce Software, 2019. [Online]. Available: https://www.perforce.com/blog/sca/what-code-quality-and-how-improve-code-quality. [Accessed: 01-Dec-2020].

[55] A. VanTol. “Python Code Quality: Tools & Best Practices.” Real Python, Real Python, 7 Nov. 2020, realpython.com/python-code-quality/.

[56] I. S. Cordasco, flake8 Documentation. (2020) [online] Available at: < https://flake8.pycqa.org/_/downloads/en/latest/pdf/> [Accessed 23 November 2020].

[57] GitHub. 2020. Dmytrolitvinov/Awesome-Flake8-Extensions. [online] Available at: <https://github.com/DmytroLitvinov/awesome-flake8-extensions> [Accessed 23 November 2020].

[58] Radon.readthedocs.io. 2020. Welcome To Radon'S Documentation! — Radon 4.1.0 Documentation. [online] Available at: <https://radon.readthedocs.io/en/latest/> [Accessed 23 November 2020].

[59] Bandit.readthedocs.io. 2020. Welcome To Bandit'S Developer Documentation! — Bandit Documentation. [online] Available at: <https://bandit.readthedocs.io/en/latest/> [Accessed 23 November 2020].

[60] Coverage.readthedocs.io. 2020. Coverage.Py — Coverage.Py 5.3 Documentation. [online] Available at: <https://coverage.readthedocs.io/en/coverage-5.3/> [Accessed 23 November 2020].

[61] M. H. Halstead, "Elements of Software Science", Elsevier, vol 7, 1977.

[62] K. Pijanowski, "Improve Code Quality Using Test Coverage." CODE Magazine, www.codemag.com/article/1701081/Improve-Code-Quality-Using-Test-Coverage.

[63] Docs.python.org. 2020. Py_Compile — Compile Python Source Files — Python 3.9.0 Documentation. [online] Available at: <https://docs.python.org/3/library/py_compile.html> [Accessed 23 November 2020].

[64] "GitLab CI," GitLab. [Online]. Available: https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/. [Accessed: 07-Dec-2020].

[65] Jenkins User Documentation. [Online]. Available: https://www.jenkins.io/doc/. [Accessed: 07-Dec-2020].

[66] Wily. [Online]. Available: https://wily.readthedocs.io/en/latest/. [Accessed: 07-Dec-2020].

[67] M. Madera, R. Tomoń, "A case study on machine learning model for code review expert system in software engineering", In Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 1357-1363, 2017.

[68] P. Deep Singh, A. Chug, "Software defect prediction analysis using machine learning algorithms", In 7th International Conference on Cloud Computing, Data Science & Engineering - Confluence, pp. 775-781, 2017.

[69] V. Barstad, M. Goodwin, T. Gjøsæter, "Predicting Source Code Quality with Static Analysis and Machine Learning", In Norsk IKT-konferanse for forskning og utdanning. 2014.

[70] “Interpreting random forests,” Diving into data, 19-Oct-2014. [Online]. Available: http://blog.datadive.net/interpreting-random-forests/. [Accessed: 01-Dec-2020].

[71] GitHub. 2020. Andosa. [online] Available at <https://github.com/andosa/treeinterpreter

[72] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, “Scikit-learn: Machine Learning in Python”, JMLR 12, pp. 2825-2830, 2011.

[73] E. Klevak, S. Lin, A. Martin, O. Linda, E. Ringger, “Out-Of-Bag Anomaly Detection”, arXiv preprint arXiv:2009.09358, 2020.

[74] K. Zhang, X. Kang, S. LI, “Isolation Forest for Anomaly Detection in Hyperspectral Images” In IEEE International Geoscience and Remote Sensing Symposium *(IGARSS), IEEE, pp. 437-440, 2019.

[75] G. Staerman, P. Mozharovskyi, S. Clémençon, F. d'Alché-Buc, “Functional Isolation Forest”, arXiv preprint arXiv:1904.04573, 2019.

[76] F. T. Liu, K. M. Ting, Z.-H. Zhou, “Isolation forest”, In Eighth IEEE International Conference on Data Mining, IEEE, pp. 413-422, 2008.

[77] V. Vercruyssen, “Designing Anomaly Detection Algorithms that Exploit Flexible Supervision”, PhD Thesis, 2020.

[78] M. Tulio Ribeiro, T. Wu, C. Guestrin, S. Singh, “Beyond Accuracy: Behavioral Testing of NLP Models with CheckList”. arXiv preprint arXiv:2005.04118, 2020.

[79] A. Chan, L. Ma, F. Juefei-Xu, X. Xie, Y. Liu, Y. S. Ong, “Metamorphic relation based adversarial attacks on differentiable neural computer”. arXiv preprint arXiv:1809.02444, 2018.

[80] WU, Zhaolin, et al., “A Time Window based Reinforcement Learning Reward for Test Case Prioritization in Continuous Integration”, In Proceedings of the 11th Asia-Pacific Symposium on Internetware, p. 1-6, 2019.

[81] NASA. [Online]. Available: https://atmos.nmsu.edu/data_and_services/atmospheres_data/INSIGHT/insight.html. [Accessed: 15-Dec-2020].

[82] H. Linusson, U. Johansson, T Löfström, “Signed-error conformal regression”, In Pacific-Asia Conference on Knowledge Discovery and Data Mining, pp. 224-236, Springer, 2014.