University of Economics in Katowice

Journal of Economics and Management

ISSN 2719-9975 Vol. 47 • 2025

sciendo

Mieczysław Lech Owoc
https://orcid.org/0000-0003-1578-6934
Department of Business Intelligence in Management
Faculty of Business and Management
Wroclaw University of Economics and Business, Poland
mieczyslaw.owoc@ue.wroc.pl

Adam Stambulski
https://orcid.org/0009-0002-9099-2471
Department of Business Intelligence in Management
Faculty of Business and Management
Wroclaw University of Economics and Business, Poland
adam.stambulski@gmail.com

Software quality management: Machine learning for recommendation of regression test suites

Accepted by Editor Ewa W. Ziemba | Received: December 17, 2023, | Revised: May 24, 2024; November 15, 2024; January 14, 2025 | Accepted: February 10, 2025 | Published: March 11, 2025.

Abstract

Aim/purpose – This study aims to demonstrate machine learning (ML) applications to enhance software development quality management, specifically through optimizing regression test suites. This research aims to demonstrate how ML can predict and prioritize the most relevant regression tests based on software changes and historical testing data, thereby reducing unnecessary testing, assuring software quality, and leading to significant cost savings.
Design/methodology/approach – The methodology of this study involves developing and training a ML model using historical data on software modifications and test executions. The model analyzes the data to predict and prioritize the most relevant regression tests for new software builds. This approach is validated through a comparative analysis, whereby the recommendations from the ML model are benchmarked against traditional regression testing methods to evaluate their efficiency and cost-effectiveness. The results demonstrate the practical advantages of integrating ML into software quality management processes.
Findings – The conclusions indicate that implementing ML to optimize regression testing has the potential to significantly improve test efficiency and reduce operational costs. The ML model effectively prioritized crucial test cases, reducing the number of unnecessary tests by 29.24% while maintaining the required quality assurance level and focusing efforts on areas with the highest impact. This optimization not only streamlines the testing process but also significantly improves the allocation of resources and cost-effectiveness in software development practices.

Cite as: Owoc, M. L., & Stambulski, A. (2025). Software quality management: Machine learning for recommendation of regression test suites. *Journal of Economics and Management, 47*, 117-137. https://doi.org/10.22367/jem.2025.47.05

Research implications/limitations – The research indicated that future studies should adopt more advanced ML algorithms, test these methods on a range of software products, and adopt a more diverse approach to testing. Such an expansion of research may provide better results and a deeper understanding of the role of ML in quality assurance, with the potential to optimize software development processes more broadly. Furthermore, establishing a more robust link between software code and specific tests within the scope of regression tests could enhance the effectiveness of ML-driven recommendations for regression test suites.
Originality/value/contribution – Integrating ML into regression testing selection represents a novel approach to the software development process, offering enhanced efficiency and cost savings. This research exemplifies the potential for transforming traditional testing methodologies, thereby making a valuable contribution to the field of software quality assurance. The study demonstrates how advanced technologies can optimize software development processes, reducing costs while maintaining an assured level of software product quality.

Keywords: Quality management, machine learning, software testing, regression test suite.
JEL Classification: C61; C63; C91.

1. Introduction

Development of new applications and, more generally, software packages is still a big challenge for programmers and software companies. It is a matter of changing technologies, growing users' expectations (e.g., adding new functionalities), conditions of use, and many other determinants. To be positively assessed by recipients, software should be of high quality. Therefore, software quality management (SQM) is a must; it relies on the systematic and organized process of all necessary activities to ensure software quality throughout the entire development life cycle. It embraces various processes, techniques, and methodologies used to monitor, control, and improve the quality of software products. This paper focuses on testing as the essential phase in the development life cycle, considering the usability of machine learning (ML) as a supportive technique of SQM (Kalech et al., 2021; Nama, 2024). In the field of software development, continuous integration (CI) is a software development practice that involves regularly integrating code changes from multiple developers into a shared code repository. This practice allows for early detection of integration issues and ensures that the developed software remains stable and functional (Arachchi & Perera, 2018). A critical aspect of CI is the execution of a regression test suite. The regression test suite is a collection of tests designed to verify that previously developed and tested software still functions correctly after introducing new changes. By running the regression test suite as part of the CI process, developers and testers can quickly identify any unintended side effects

caused by the recent code changes and ensure that new code changes do not break existing functionality and that the software continues to meet the desired quality standards. These tests are vital for ensuring that previously delivered functionalities continue to work correctly. This process is ongoing and executed with each new software version (Mårtensson et al., 2019). The choice of a software development life cycle (SDLC) model depends on factors like organizational size and industry specifics, shaping the overall software product life cycle (Rai & Seth,2014; Soares et al., 2022).

The continuous addition of new features increases the number of regression tests in a phenomenon known as the snowball effect. Each new software release adds additional regression tests to the scope. The growing number of regression tests necessitates constant analysis and optimization, yet it is a technically complex and time-consuming activity that requires knowledge and experience. The growing number of regression tests performed for each new software version increases the cost of product development. It extends the time to validate the product for delivering subsequent versions of the new software version to the customers (ISTQB Testing, 2018; Rai & Seth, 2014).

The introduction of new functionalities into the software product necessitates the creation of new tests to validate them. These new tests are executed and become candidates for a regression test suite. As a result of this process, the scope of regression tests is constantly increasing. Implementing a constant optimization process is necessary to prevent the regression test suites from becoming too large. This involves selecting, prioritizing, and occasionally arguing test cases to reach the most efficient and effective scope for regression tests (Forgács & Kovács, 2024).

In the context of this article's focus on regression test scope optimization, applying ML based tools to recommend regression test cases stands out as a strategic approach. These tools prove instrumental in achieving a more efficient and effective regression test scope by delving into software changes and historical test results. This streamlines the testing process and contributes to enhanced software quality and resource utilization, which is crucial for SQM (Marijan, 2023).

This article aims to investigate the effectiveness of ML techniques in optimizing the scope of regression testing in software development and to assess their impact on SQM. The research hypothesis is to validate whether applying ML techniques in regression testing can significantly improve the efficiency and effectiveness of SQM by optimizing the scope of regression tests. The following sections of the paper address specific aspects of ML research in software regres-

sion testing. Section 2 provides a comprehensive literature review and research method, summarizing previous studies and focusing on software quality assurance using artificial intelligence (AI) and ML techniques. Section 3 outlines the SQM stages, aspects of software development processes, and their impact on quality. Section 4 presents the research methodology, including the experimental design, data collection methods, and statistical analysis techniques. Section 5 describes two key performance indicators (KPIs) for measuring the effectiveness of ML models in recommending regression test suites. Section 6 discusses the significance of the findings in the broader scientific context, drawing connections to existing theories and suggesting potential avenues for further research. Finally, Section 7 concludes the article by summarizing the main findings, discussing their implications, and suggesting potential applications or future directions for the field.

2. Literature review and research methods

Literature research indicates many possibilities for applying ML to specific aspects of software testing. Software engineering uses the following methods and techniques of AI (ISTQB AI Testing, 2021; Khaliq, 2022; Virvou et al., 2022; Yaraghi, 2022).

Fuzzy logic and probabilistic methods in AI represent a nuanced approach to dealing with the inherent uncertainties and probabilistic nature of real-world problems. Applying fuzzy logic and probabilistic methods in AI for software testing brings a level of sophistication that aligns with real-world systems' unpredictable and probabilistic nature, offering a more adaptive and comprehensive approach to quality assurance.

The classification, learning, and prediction, particularly when embodied by ML, offer a robust set of tools for various applications within software testing, like defect management, defect prediction, and user interface testing (Theissler et al., 2021).

Computational search and optimization techniques offer valuable contributions to software testing, particularly in automating test case generation, identifying the smallest number of test cases that achieves a given coverage criterion, optimizing regression testing, and navigating complex test spaces. The adaptability and overlap with other AI technologies contribute to a holistic approach to addressing the multifaceted challenges of software testing (Russell & Norvig, 2020).

In some areas, AI/ML tools can significantly impact software quality assurance (ISTQB AI Testing, 2021).

- Analysis of reported defects: AI/ML tools can scrutinize reported defects, offering insights into patterns, severity, and potential root causes. This aids in prioritizing and addressing critical issues efficiently (Kim et al., 2021).
- Test case generation: ML algorithms can assist in the automated generation of test cases, streamlining the testing process and ensuring comprehensive coverage of functionalities (Tuncali et al., 2019).
- Optimizing regression test suites: AI-based tools analyze past test results, defects, and recent changes to intelligently select, prioritize, and augment test cases (Da Roza, 2022; Marijan et al., 2018; Pan et al., 2022; Sutar, 2020).
- Defect prediction: Predictive analytics powered by ML can anticipate potential defects by analyzing historical data, enabling proactive measures to be taken before issues escalate (Pachouly et al., 2022).
- Testing the user interface (GUI): AI/ML tools play a pivotal role in automating and optimizing the testing of user interfaces, ensuring robust performance and user experience (Stige et al., 2023)
- Requirements analysis: AI/ML tools help in analyzing and understanding requirements, ensuring alignment with stakeholder needs, and minimizing misinterpretations (Liu et al., 2022).
- Performance monitoring and optimization: AI/ML tools monitor application performance, identify bottlenecks, and suggest optimizations, contributing to enhanced software performance (Gill et al., 2022).
- Log analysis: Efficient debugging by ML-powered log analysis tools automate the identification of issues in logs, expediting debugging processes (Shash, 2021; Yang et al., 2021).

Integrating AI/ML in software quality assurance introduces intelligent automation, data-driven decision-making, and enhanced adaptability, ultimately contributing to higher software quality and more efficient testing processes.

Quality management in software engineering involves systematic processes and practices to ensure that software products meet predefined quality standards and user expectations (Kobyliński, 2021; Laporte & April, 2018).

The key aspects of the quality covering phases or steps of the system development life cycle, mostly:

- Requirements analysis: Clearly defining and understanding customer requirements is crucial. It sets the foundation for the entire development process.

- Testing and quality assurance: Rigorous testing throughout the development life cycle is essential. This includes unit testing, integration testing, system testing, and acceptance testing. Automated testing tools can enhance efficiency.
- Code reviews: Regular code reviews help identify and address issues early in the development process. This promotes better code quality and reduces the likelihood of defects.
- Version control: Using version control systems (e.g., Git) ensures that changes to the codebase are tracked, reversible, and well-managed, contributing to overall software quality.
- CI/continuous deployment (CD): Implementing CI/CD pipelines automates the building, testing, and deployment processes, facilitating faster and more reliable releases (Arachchi & Perera, 2018).
- Documentation: Maintaining comprehensive documentation, including user manuals and technical documentation, ensures that the software is well--understood and can be effectively maintained.
- Bug tracking and resolution: Efficient bug tracking systems help promptly identify, prioritize, and address issues. This contributes to a more stable and reliable software product.
- Customer feedback and iterative development: Gathering and incorporating customer feedback allows continuous improvement. Agile methodologies promote iterative development cycles, enhancing adaptability to changing requirements.
- Security measures: Implementing security practices, such as code reviews for security vulnerabilities and regular security audits, is crucial to ensuring the integrity and safety of the software.

Compliance with established standards such as ISO 9001: Quality Management Systems (ISO, 2015) and ISO/IEC 25010: Systems and Software Engineering (ISO, 2023). Adhering to these standards provides a robust framework for maintaining and enhancing software quality. ISO 9001 focuses on quality management systems, while ISO/IEC 25010 defines specific requirements and evaluations for software product quality. Together, they ensure that practices meet industry benchmarks, contributing significantly to the effectiveness of SQM strategies. This adherence helps achieve high-quality software products and ensures consistency and reliability in development processes (Goericke, 2020).

Apart from analyzing sources focused on SQM using AI techniques, our research method refers to ML experiments for regression test suite optimization. This highlights the transformative impact of AI/ML tools on software testing. It covers various areas such as defect analysis, automated test case generation, regression test suite optimization, defect prediction, GUI testing automation, requirements analysis, performance monitoring, and log analysis. These tools introduce intelligent automation, data-driven decision-making, and enhanced adaptability, ultimately leading to higher software quality and more efficient testing processes. Integrating AI/ML in testing significantly advances software quality assurance.

3. Software quality management quests

SQM is a systematic approach to the management and assurance of the quality of software products throughout their development and lifecycle. This management discipline encompasses a number of key practices and principles designed to ensure that software meets or exceeds customer expectations and adheres to predefined quality standards. This section provides a comprehensive overview of the key elements of SQM (Alexsoft Report, 2023; Laporte et al., 2018). SQA focuses on the processes used in software development to ensure quality. It involves applying standards, methods, and tools throughout the SDLC to prevent defects and ensure quality. The aim is to improve development and test processes so that defects do not arise when the software is being created.

Software quality planning (SQP), like quality planning, focuses on project management, proposing an individual plan for a specific project. During the software planning phase, a quality plan should be prepared with assumed standards, documentation, and other necessary details regarding responsible staff and means predicted in the process.

Software quality control (SQC) involves testing software to identify defects. Quality control (QC) is a more product-oriented process that is conducted during the software development process. It includes various forms of testing, such as unit testing, integration testing, system testing, and acceptance testing, to ensure that the software meets the required standards.

Software quality improvement (SQI) – after the software is released, ongoing improvements are crucial based on feedback and performance analysis. Quality improvement involves increasing the software's effectiveness and efficiency in providing added benefits to both the users and the organization.

Furthermore, SQM ensures compliance with standards like ISO 9001: Quality Management Systems (2015) and ISO/IEC 25010: Systems and Software Engineering (2023). Effective SQM significantly impacts product quality, lifecycle costs, customer satisfaction, and time to market. It helps organizations avoid rework costs, enhance customer satisfaction, and maintain market competitiveness.

For organizations seeking to implement or enhance their SQM practices, integrating these components seamlessly into the software development lifecycle is paramount. This integration ensures that quality processes are continuously monitored and improved (Alam et al., 2024). Implementing a comprehensive SQM process is of the utmost importance to effectively manage software quality. This process should include the establishment of general best practices in the field of software development and project management. The following key practices in SQM have been identified (Laporte & April, 2018; PMI, 2021):

- Risk management: Identifying and mitigating risks that could impact the quality of the software. This includes addressing potential issues related to requirements, design, development, and external factors.
- Configuration management: Managing changes to software artifacts (such as source code, documentation, and configuration files) to ensure consistency, traceability, and version control.
- Measurement and metrics: Collecting and analyzing data to assess the quality of the software and the effectiveness of quality management processes. Metrics are used to track progress and make informed decisions.
- Documentation: Creating and maintaining documentation that defines quality standards, processes, and procedures to ensure all team members are on the same page.
- Training and competence: Ensuring team members have the necessary skills and knowledge to adhere to quality management processes and produce high--quality software.
- Customer satisfaction: Focusing on meeting the needs and expectations of end-users and stakeholders to ensure that the software fulfills its intended purpose.

SQM is integral to the software development lifecycle and essential for producing reliable, secure, high-quality software products. It helps reduce defects, improve productivity, increase customer satisfaction, and ultimately leads to a more successful software development process.

4. Experiment ML-based tests recommendation for continuous integration

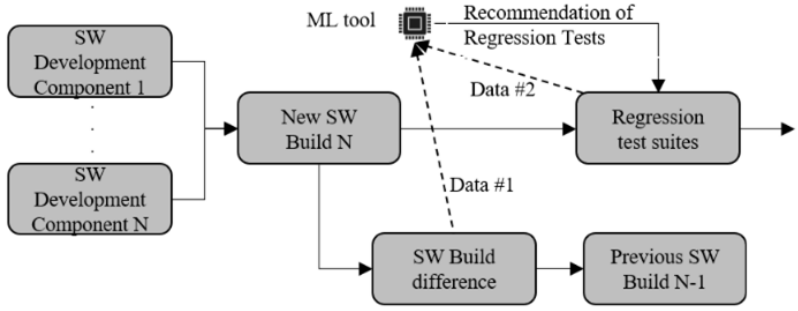
Machine learning can be pivotal in optimizing regression test suites, particularly in software testing. ML algorithms can analyze historical data on software changes, test executions, and outcomes to intelligently recommend which tests to include in the regression test suites. This process aids in enhancing efficiency, reducing redundancy, and ensuring that the most relevant tests are prioritized. The application of ML in this context contributes to a more streamlined and effective software testing workflow (ISTQB AI Testing, 2021; Guizzo et al., 2021). This experiment aimed to create an ML-based tool recommending regression test suites for new software build versions. The tool achieves this by analyzing data derived from the historical execution of test cases, the identification of defects, and software changes resulting from implementing new features. This approach is particularly valuable in the context of large-scale companies developing complex software products. In such environments, introducing new features frequently entails alterations in multiple software components (Yaraghi, 2022).

The challenge of mapping software code changes to corresponding tests can be effectively addressed by applying ML-based tools. By enhancing the association between code modifications and relevant tests, these tools significantly improve the accuracy of regression test recommendations and provide proper prioritization. Recent research underscores the impact of AI-based tools, revealing a potential 50% reduction in regression test scope while maintaining robust defect detection in the code (Rai & Seth, 2014). Furthermore, analysis indicates that a 40% reduction in test execution duration is achievable during CI testing without compromising error detection significantly. This highlights the transformative potential of leveraging AI/ML in optimizing testing processes and resource utilization (Marijan et al., 2018).

The proposed solution to the problem of increasing the scope of regression test suites during CI is using ML algorithms to recommend test cases to be performed on the new software build.

Figure 1 presents the workflow of an applied ML-based tool for a recommendation of regression test suites in a CI process.

Figure 1. Workflow with a recommendation of regression test suites by ML tool



Note: SW – software.

Source: Authors' own elaboration.

In the case of advanced software (SW) products, the new functionality is realized by code changes in software components (marked SW in Figure 1) or sometimes even in a few of them. The code is committed to the source control system, and the new software built for the system is created to run the unit and the component tests. After that, each new valid SW build is deployed on a target system to execute the regression test suites (subject of research). As part of the end-to-end system-level testing process, it ensures 100% test coverage for all supported features. This is done to validate whether all previously released functionalities are still supported.

In software development, system-level testing is crucial for validating a fully integrated software and hardware product, ensuring it functions as intended and meets all specified requirements. This testing phase, which follows unit and integration testing, evaluates software's operational capabilities and interaction among various system components. A subset of system-level testing is end-to--end testing, which simulates real-world usage scenarios to confirm that the application performs consistently from start to finish (Kobyliński, 2021; Mårtensson et al., 2019). It is of paramount importance to recognize that the incorporation of new functionality necessitates the addition of corresponding tests to the regression test suites. This presents a significant challenge, as the continuous growth of regression tests necessitates allocating time, resources, and ongoing optimization efforts to ensure the sustained effectiveness of the testing process.

The problem described was analyzed using a ML algorithm to recommend regression tests for a new software build. An analysis of the available data indicated that applying the unsupervised ML algorithm, K-means (scikit-learn, 2024) is the best approach. Trials with supervised ML algorithms did not pro-

vide promising results. This is mainly due to the lack of correlation between the code of functionality and validation tests. The K-Means algorithm, or the centroid algorithm, groups objects as input data without knowing their categories (decision classes), where clusters are defined deterministically. The algorithm requires determining the number of clusters and scales well to many data samples. It is used to solve problems in many different fields and areas of research.

The K-means algorithm can analyze and categorize data points, helping identify patterns and optimize the selection of relevant test cases. The algorithm works by partitioning data into k clusters, assigning each data point to the cluster with the nearest mean. This enables efficient grouping and recommendation of regression tests based on similarities in software changes and historical test data.

In this experimental design, the variable number of clusters (k), ranging from 2 to 10, is the independent variable impacting key performance measures, the fault detection rate, and profit (defined in the next section) of regression tests advised by the ML tool. The research seeks to identify the optimal cluster number to maximize these performance metrics.

Created the ML-based tool (Figure 1) gathered data from two sources: Data #1 and Data #2.

Data #1 includes the differences in code changes between the new software build "N" and the previous build "N–1". This data set comprises detailed information about modifications at the component level of the software, collected directly from the code repository.

The data set designated as Data #1 was extracted by a bespoke application specifically designed to retrieve code differences. The following section outlines the features and measurement methods employed to obtain Data #1:

- Added components: This feature lists the components present in the new build but not in the old one. It is calculated by comparing the two builds' components and identifying those unique to the new build.
- Removed components: This feature lists the components present in the old build but not in the new one. It is calculated by comparing the two builds' components and identifying those unique to the old build.
- Unchanged components: This feature lists the components present in both builds, which have the same version. It is calculated by comparing the components of the two builds and identifying those with matching versions.
- Unchecked components: This feature lists the components that could not be checked for differences due to missing repository URLs or other issues. It is calculated by identifying components with missing or mismatched repository information.

- Increase in components: This is the net increase in the number of components. It is calculated by subtracting the number of removed components from the number of added components.
- Increase in peg revisions: This characteristic represents the increase in the number of peg revisions. It is calculated by subtracting the number of peg revisions in the old build from the number of peg revisions in the new build.
- Compare components: This feature lists the components present in both builds but have different versions. It includes detailed diffs for each component, such as the number of lines added and removed in each file. The diffs are fetched using version control system fetchers (e.g., SVN, Git).
- Difference per component: These are the detailed diffs for each component, including the number of lines added and removed in each file and the difference in the number of authors. The diffs are calculated by comparing the repository revisions of the components in the two builds.

Data #2 comprises the results of regression test suites executed on previous software builds. The status of test cases after execution can be categorized as passed, blocked due to test automation issues, or failed due to code bugs. This regression test status data is collected from the test case repository system. Based on the specific system test level and experience of the test teams, additional exceptions are applied, such as keeping newly added test cases in the recommended TC by the ML tool for a certain period to train the model and ensure the relevant SW quality level.

The ML-based tool was developed and trained using a dataset that includes historical data on regression test execution outcomes from past software builds (Data #2) and the corresponding code changes between these builds (Data #1). The dataset, gathered over the last six months, provides a robust foundation for the tool's learning and predictive capabilities, enabling it to recommend regression tests effectively.

5. Defined KPI to measure model effectiveness

To assess the effectiveness of this research project, the regression test suites recommended by the ML model are compared against the actual results from the organization's comprehensive set of regression tests. The effectiveness of the ML model is quantified through metrics such as profit and fault detection rate, which align with the predefined KPIs for the ML model. This comparison plays a pivotal role in validating the utility and precision of the ML recommendations, particularly in the context of regression testing KPIs.

Profit (KPI #1) evaluates the effectiveness of ML-recommended regression tests, expressing this as a percentage of the overall test suites. This KPI provides insights into the efficiency of machine-learning tools by indicating how well they help achieve desired test coverage while minimizing excess testing, ultimately contributing to better resource management:

$$Profit = 1 - \frac{Tr}{Rtc} \tag{1}$$

Tr – recommended regression tests by ML-based tool,
Rtc – full regression test suites.

Fault detection rate (KPI #2) in the predicted test evaluates the effectiveness of the ML tool by measuring the proportion of recommended failed test cases compared to the total failed test cases identified by the full regression suite. This KPI assesses the model's ability to select test cases that effectively target fault--prone code changes, ensuring critical defects are captured as they would be in a comprehensive test suite.

$$Fault\ Detection\ Rate\ (FDR) = \frac{Fr}{Rt} \tag{2}$$

Fr – number of test cases with failed status recommended by ML tool,
Rt – total number of test cases with failed status detected by the full regression test suite.

FDR compares the faults detected by the predicted test cases to those detected by running the complete test suite. This rate helps determine whether the model-suggested suite adequately covers critical fault-prone areas without missing significant defects.

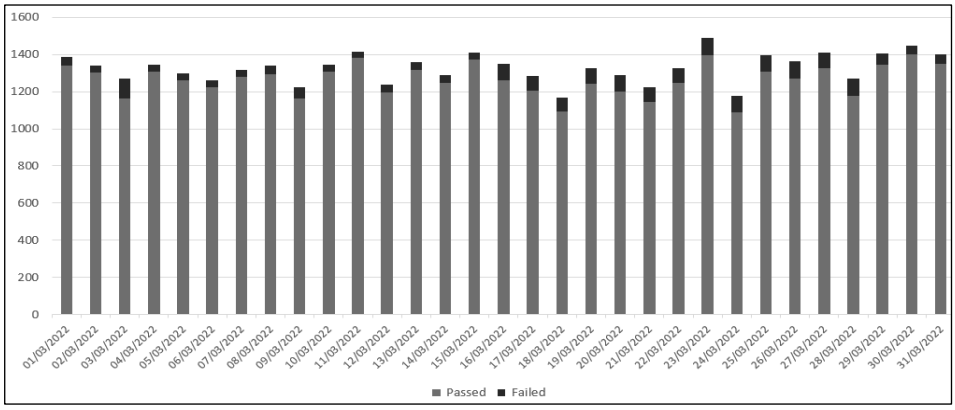
6. Results of recommendation

In a scientific study, a specific ML model was carefully implemented, trained, and thoroughly checked in a software development organization. This research took place during the regular CI process. Once the new software build got the green light from quick tests, it was installed on the end-to-end system for a complete round of system-level regression tests. The primary objective of this scientific investigation was to find solutions to the growing issue of an increasing number of regression test suites and to enhance their effectiveness (trunkbased development.com).

While researching each valid and approved software build, the ML-based tool recommended the regression tests suite. The research results were evaluated per the defined KPIs, namely KPIs #1 and #2.

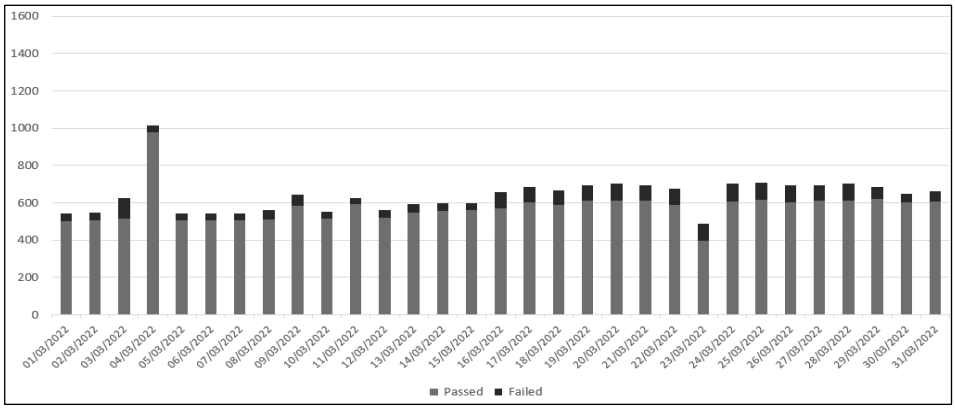
The comparison between the actual regression test suite execution results (Figure 2) and those recommended by the ML-based tool (Figure 3) provides a visual representation of the research and analysis conducted for each daily approved software build. This visual insight allows for a comprehensive assessment of how well the ML tool aligns with the real-world outcomes, offering valuable insights into the effectiveness and accuracy of the ML recommendations in the dynamic context of daily software builds. This comparative analysis serves as a tangible demonstration of the ML-based tool's impact on optimizing regression test execution and ensuring alignment with actual testing results.

Figure 2. Regression tests in daily execution



Source: Authors' own elaboration based on internal reporting system.

Figure 3. Recommended by ML-based tools, the regression test suites



Source: Authors' own elaboration based on collected data.

The comparison between Figure 2, depicting the actual execution of total regression test suites daily across the organization, and Figure 3, illustrating the ML tool-recommended regression tests, offers an initial insight into the research findings. The bar charts reveal noticeable fluctuations in the daily recommended regression tests (Figure 3), influenced by software changes between builds and historical test execution results. The ML algorithm dynamically provides recommendations for daily CI regression tests juxtaposed with the actual scope of executed regression tests.

This comparison is subjected to evaluation using defined KPIs, specifically focusing on profit (1) as a measure of decreased test numbers and fault detection rate (2) for tests recommended with failed results. The research findings are consolidated and summarized for each testing team monthly, as presented in Table 1. This comprehensive overview leads to intriguing and valuable conclusions, providing a deeper understanding of the impact and effectiveness of the ML-based tool in the context of CI regression testing across diverse testing teams (cf. Section 5 of this article for a definition of fault detection rate and profit in Table 1).

Table 1. ML-based recommended regression test suites

Test team	Run reg. test cases	Recommended regression tests by ML tool	Fault detection rate (%)	Profit (%)	Reg. tests not executed	Time saved due to not executed reg. tests 1 TC, 15 min = 0,25h (h)
Test team 1	11159	1446	100.00	87.04	9713	2428.25
Test team 2	2207	1926	100.00	12.73	281	70.25
Test team 3	1897	1865	100.00	1.69	32	8
Test team 4	7026	5884	99.01	16.25	1142	285.5
Test team 5	951	935	99.47	1.68	16	4
Test team 6	328	296	100.00	9.76	32	8
Test team 7	7714	7521	100.00	2.50	193	48.25
Test team 8	2042	1413	100.00	30.80	629	157.25
Test team 9	1154	1126	100.00	2.43	28	7
Test team 10	762	621	100.00	18.50	141	35.25
Test team 11	6442	6319	100.00	1.91	123	30.75
Total	42179	29845	99.69	29.24	12330	3082.5

Source: Authors' own elaboration.

The findings from Figure 3 reveal that the majority of regression tests conducted in the regular official process yield positive results, surpassing an 85% success rate on a daily basis.

When considering the ML-based recommended number of regression tests, measured by Profit (KPI #1), variations emerge across different test teams. Calculated profits range between 1.68% and 87.04%, with an overall average profit of 29.24%. This diverse range prompts a deeper analysis of the actual tests within each team.

Compared to the actual regression test results with failed results, the fault detection rate in ML recommendations for regression test cases with failed status stands high at 99.69%. This underscores the precision and reliability of the ML tool in identifying tests associated with potential issues.

Regarding total monthly test numbers, the ML-based recommendation for regression test cases amounts to 29,845 tests. This contrasts with the 42,179 regression tests performed through the regular process. These figures offer a quantitative perspective on the optimization achieved through ML recommendations, indicating a potential reduction in test numbers while maintaining a high fault detection rate.

FDR measures the ML tool's ability to recommend regression tests with failed results compared to the actual outcomes indicating software bugs. On average, it achieved 99.69% across all test teams. However, two teams slightly fell below 100%, suggesting areas for potential improvement in ML recommendations to enhance bug detection.

In Table 1, the calculated difference reveals the regression tests that were not executed based on ML recommendations. Particularly noteworthy is Test Team 1, where the ML tool suggested running only 1,446 test cases out of 11,159, yielding a substantial profit of 87.04%. This underscores the efficiency gains and regression tests optimization potential of ML-based recommendations, prompting further analysis within each test team for deeper insights.

Leveraging the organization's extensive practice experience, the effort estimation for a single test case executed in the regression test suite is assessed at 15 minutes. This includes test preparation, execution, post-analysis, and investigation in case of failed results. Notably, the ML-based recommendation of regression test cases results in significant time savings. The calculated savings amount to 3,082.5 hours, representing the effort saved by executing only the recommended regression tests, with an average FDR level of 99.69%. This conclusion underscores the tangible benefits of resource efficiency and time optimization achieved through applying ML-based recommendations in regression testing.

7. Conclusions

The study highlights the important role of ML in optimizing regression testing for software development. By correlating software changes with historical regression test results, the ML-based model recommends a balanced set of tests, achieving a 29.24% reduction in test cases while maintaining a high fault detection rate of 99.69%. This optimization aligns with the organization's rigorous quality standards and results in a significant effort saving of 3,082.5 hours, underscoring its efficiency and cost-effectiveness.

Moreover, the emphasis on financial benefits, reduced testing effort, and alignment with quality assurance goals demonstrates the broader value of ML-driven recommendations. This research also highlights the importance of linking test cases to code changes, paving the way for algorithmic improvements that enhance profit and fault detection rate as key quality indicators.

However, it is important to note the limitations of the study. The model's effectiveness depends on the quality and availability of historical test data, which may not always be comprehensive or representative across diverse software projects. Additionally, the approach may not effectively apply to all software domains or highly dynamic development environments where code bases and requirements change rapidly. Another limitation lies in the potential computational overhead of implementing and maintaining ML models, which could pose challenges for smaller organizations with limited resources.

Future studies should explore advanced ML; techniques, evaluate broader software contexts, and diversify testing approaches to deepen insights. Strengthening the connection between software code and specific regression tests could further enhance the effectiveness of ML-driven recommendations toward a higher optimization level.

The study also anticipates future advancements, including adopting transfer learning and large language models (LLMs) (Fan, 2023). These technologies promise to revolutionize software testing through real-time code analysis and automated test case generation, enabling higher efficiency and faster product delivery. Transfer learning will facilitate precise testing with pre-trained models, while LLMs will dynamically generate test cases and detect defects, offering actionable insights for developers (Hoffmann & Frister, 2024; Wang, 2024).

In conclusion, the application of ML in regression testing represents a substantial advancement in quality management, offering a more efficient, resource-effective, and robust approach that aligns seamlessly with continuous improvement principles. By addressing the noted limitations through further research and refinement, these technologies hold significant potential to transform software development processes, ensuring exceptional product quality and cost efficiency.

Author contributions

Mieczyslaw Lech Owoc – 30% (research concept and design, data analysis and interpretation, writing the article, critical revision of the article, final approval of the article).
Adam Stambulski – 70% (research concept and design, collection and/or assembly of data, data analysis and interpretation, writing the article, critical revision of the article, final approval of the article).

Disclosure statement

No potential conflict of interest was reported by the author(s).

References

Alam, M. M., Priti, S. I., Fatema, K., Hasan, M., & Alam, S. (2024). Ensuring excellence: A review of software quality assurance and continuous improvement in software product development. In A. Hamdan (Eds.), *Achieving sustainable business through AI, technology education and computer science* (Studies in Big Data, Vol. 163, pp. 331-346). Springer. https://doi.org/10.1007/978-3-031-73632-2_28

Alexsoft report. (2023). *Quality assurance, quality control, and testing – the basics of software quality management*. https://www.altexsoft.com/whitepapers/quality-assurance-quality-control-and-testing-the-basics-of-software-quality-management/

Arachchi, S. A. I. B. S., & Perera, I. (2018). Continuous integration and continuous delivery pipeline automation for agile software project management. In *2018 Moratuwa Engineering Research Conference (MERCon)* (pp. 156-161). IEEE. https://doi.org/10.1109/MERCon.2018.8421965

Da Roza, E. A., Lima, J. A. P., Silva, R. C., & Vergilio, S. R. (2022). Machine learning regression techniques for test case prioritization in continuous integration environment. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 196-206). IEEE. https://doi.org/10.1109/SANER53432.2022.00034

Fan, A. G., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. M. (2023). *Large language models for software engineering: Survey and open problems*. Cornell University. https://doi.org/10.48550/arXiv.2310.03533

Forgács, I., & Kovács, A. (2024). *Modern software testing techniques. A practical guide for developers and testers*. Apress-Springer. https://doi.org/10.1007/978-1-4842-9893-0

Gill, S. S., Xu, M., Ottaviani, C., Patros, P., Bahsoon, R., Shaghaghi, A., Golec, M., Stankovski, V., Wu, H., Abraham, A., Singh, M., Mehta, H., Ghosh, S. K., Baker, T., Parlikad, A. K., Lutfiyya, H., Kanhere, S. S., Sakellariou, R. ..., & Uhlig, S. (2022). AI for next generation computing: Emerging trends and future directions. *Internet of Things, 19*, 100514. https://doi.org/10.1016/j.iot.2022.100514

Goericke, S. (Ed.). (2020). *The future of software quality assurance.* Springer. https://doi.org/10.1007/978-3-030-29509-7

Guizzo, G., Petke, J., Sarro, F., & Harman, H. (2021). Enhancing genetic improvement of software with regression test selection. In *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (pp. 1323-1333). IEEE. https://doi.org/10.1109/ICSE43902.2021.00120

Hoffmann, J., & Frister, D. (2024). Generating software tests for mobile applications using fine-tuned large language models. In *2024 IEEE/ACM International Conference on Automation of Software Test (AST)* (pp. 76-77). IEEE. http://doi.org/10.1145/3644032.3644454

ISO. (2015). *ISO 9001: Quality management systems*. https://www.iso.org/standard/62085.html

ISO. (2023). *ISO/IEC 25010: Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Product quality model*. https://www.iso.org/standard/78176.html

ISTQB AI Testing. (2021). *Certified tester AI testing (CT-AI) syllabus*. https://www.istqb.org/certifications/

ISTQB Testing. (2018). *Certified tester foundation level syllabus*. International Software Testing Qualifications Board. https://www.istqb.org/sdm_downloads/release-notes-certified-tester-foundation-level-syllabus/

Kalech, M., Abreu, R., & Last, M. (2021). *Artificial intelligence methods for software engineering*. World Scientific Connect. https://doi.org/10.1142/12360

Khaliq, Z. F. (2022). *Artificial intelligence in software testing: Impact, problems, challenges and prospect*. https://doi.org/10.48550/arXiv.2201.05371

Kim, D., Wang, X., Kim, S., Zeller, A., Cheung, S. C., & Park, S. (2021). Which crashes should I fix first? Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Transactions on Software Engineering, 37*(3), 430-447. https://ieeexplore.ieee.org/document/5711013

Kobyliński, A. (2021). *Modele cyklu życia oprogramowania. Modele tradycyjne* [Software development life cycle models. Traditional models] (1 ed.). SGH.

Laporte, C. Y., & April, A. (2018). *Software quality assurance*. IEEE Computer Society. John Wiley & Sons. https://doi.org/10.1002/9781119312451

Liu, K., Reddivari, S., & Reddivari, K. (2022). Artificial intelligence in software requirements engineering: State-of-the-art. In *IEEE 23rd International Conference on Information Reuse and Integration for Data Science* (IRI) (pp. 106-111). IEEE. https://doi.org/10.1109/IRI54793.2022.00034

Marijan, D., Gotlieb, A., & Liaaen, M. (2018). A learning algorithm for optimizing continuous integration development and testing practice. *Journal of Software: Practice and Experience, 49*(2), 192-213. https://doi.org/10.1002/spe.2661

Marijan, D. (2023). Comparative study of machine learning test case prioritization for continuous integration testing. *Software Quality Journal, 31*, 1415-1438. https://doi.org/10.1007/s11219-023-09646-0

Mårtensson, T., Ståhl, D., & Bosch, J. (2019). Test activities in the continuous integration and delivery pipeline. *Software: Evolution and Process, 31*(4), e2153. https://doi.org/10.1002/smr.2153

Nama, P. (2024). Integrating AI in testing automation: Enhancing test coverage and predictive. *World Journal of Advanced Engineering Technology and Sciences, 13*(01), 769-782. https://doi.org/10.30574/wjaets.2024.13.1.0486

Pachouly, J., Ahirrao, S., Kotecha, K., Selvachandran, G., & Abraham, A. (2022). A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools. *Engineering Applications of Artificial Intelligence, 111*, 104773. https://doi.org/10.1016/j.engappai.2022.104773

Pan, R., Bagherzadeh, M., Ghaleb, T. A., & Briand, L. (2022). Test case selection and prioritization using machine learning: A systematic literature review. *Empirical Software Engineering, 27*(2), 29. https://doi.org/10.1007/s10664-021-10066-6

PMI. (2021). *PMI PMBOOK: A guide to the project management body of knowledge (PMBOK Guide)* (7th ed.). PMI Standard. https://www.pmi.org/pmbok-guide-standards/foundational/pmbok

Rai, D., & Seth, K. (2014). Regression test case optimization using honey bee mating optimization algorithm with fuzzy rule base. *World Applied Sciences Journal, 31*(4), 654-662. https://www.researchgate.net/publication/336133351_Regression_Test_Case_Optimization_Using_Honey_Bee_Mating_Optimization_Algorithm_with_Fuzzy_Rule_Base

Russell, S. J., & Norvig, P. (2020). *Artificial intelligence: A modern approach* (4 ed.). Pearson.

Sutar, S., Kumar, R., Pai, S., & Shwetha, B. S. (2020). Regression test cases selection using natural language processing. In *2020 International Conference on Intelligent Engineering and Management* (ICIEM) (pp. 301-305). IEEE. https://doi.org/10.1109/ICIEM48762.2020.9160225

scikit-learn.org. (2024). *Clustering* (User guide). https://scikit-learn.org/stable/modules/clustering.html#clustering

Shash, M. (2021). *Using machine learning for log analysis and anomaly detection: A practical approach to finding the root cause*. DZone. https://dzone.com/articles/using-machine-learning-for-log-analysis-and-anomal

Soares, E., Sizilio, G., Santos, J., da Costa, D., & Kulesza, U. (2022). The effects of continuous integration on software development: A systematic literature review. *Empirical Software Engineering, 27*(3), 78. https://doi.org/10.1007/s10664-021-10114-1

Stige, Å., Zamani, E. D., Mikalef, P., & Zhu, Y. (2023). Artificial intelligence (AI) for user experience (UX) design: A systematic literature review and future research agenda. *Information Technology & People, 37*(6), 2324-2352. https://doi.org/10.1108/ITP-07-2022-0519

Theissler, A., Pérez-Velázquez, J., Kettelgerdes, M., & Elger, G. (2021). Predictive maintenance enabled by machine learning: Use cases and challenges in the automotive industry. *Reliability Engineering & System Safety, 215*, 107864. https://doi.org/10.1016/j.ress.2021.107864

Tuncali, C. E., Fainekos, G., Prokhorov, D., Ito, H., & Kapinski, J. (2019). Requirements-driven test generation for autonomous vehicles with machine learning components. *IEEE Transactions on Intelligent Vehicles, 5*(2), 265-280. https://doi.org/10.1109/TIV.2019.2955903

Virvou, M., Tsihrintzis, G. A., Bourbakis, N. G., & Jain, L. C. (2022). *Handbook on artificial intelligence-empowered applied software engineering*. Vol. 1: *Novel methodologies to engineering smart software systems*. Springer. https://doi.org/10.1007/978-3-031-08202-3

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software testing with large language model: Survey, landscape, and vision. *IEEE Transactions on Software Engineering, 50*(4), 911-936. https://doi.org/10.1109/TSE.2024.3368208

Yang, L., Chen, J., Wang, Z., Wang, W., Jiang, J., Dong, X., & Zhang, W. (2021). Semi-supervised log-based anomaly detection via probabilistic label estimation. In *EEE/ACM 43rd International Conference on Software Engineering (ICSE)* (pp. 1448-1460). IEEE. https://doi.org/10.1109/ICSE43902.2021.00130

Yaraghi, A. S., Bagherzadeh, M., Kahani, N., & Briand, L. C. (2022). Scalable and accurate test case prioritization in continuous integration contexts. *IEEE Transactions on Software Engineering, 49*(4), 1615-1639. https://doi.org/10.1109/TSE.2022.3184842