Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 8, Nomor 4, October 2024
DOI : https://doi.org/10.33395/sinkron.v8i4.14201
e- ISSN : 2541-2019
p- ISSN : 2541-044X

Comparing BDD and TDD: Machine Learning Analysis of Software Quality with SHAP Interpretability

Gregorius Airlangga[1)*]
[1)]Information System Study Program, Atma Jaya Catholic University of Indonesia
[1)]gregorius.airlangga@atmajaya.ac.id

Submitted :Oct 20, 2024 | **Accepted** : Nov 18, 2024 | **Published** : Nov 28, 2024

Abstract: This study evaluates the impact of Behavior-Driven Development (BDD) and Test-Driven Development (TDD) on software quality using machine learning models, including Random Forest, XGBoost, and LightGBM. Key metrics such as bug detection, test coverage, and development time were analyzed using a dataset from multiple software projects. Polynomial feature expansion captured non-linear interactions, while SHapley Additive exPlanations (SHAP) enhanced interpretability. Results indicate that Random Forest achieved the best predictive accuracy, with an average RMSE of 7.64 and MAE of 6.39, outperforming XGBoost (average RMSE: 8.63, MAE: 7.37) and LightGBM (average RMSE: 6.89, MAE: 5.38). However, negative R^2 values across all models reveal challenges in generalization. SHAP analysis highlights the critical influence of higher-order interactions, particularly between test coverage and development time. These findings underscore the complexity of predicting software quality and suggest the need for additional features and advanced techniques to enhance model performance. This study provides a comprehensive, interpretable framework for assessing the comparative effectiveness of BDD and TDD in improving software quality.

Keywords: Behavior-Driven Development (BDD); Test-Driven Development (TDD); Software Quality; Machine Learning Models; SHAP Interpretability

INTRODUCTION

In the constantly evolving world of software development, methodologies like Behavior-Driven Development (BDD) and Test-Driven Development (TDD) have become pivotal in the pursuit of high-quality software systems (Österholm, 2021; Rahman & Nadia, 2024; Silva & Siriwardana, 2023). These approaches promise improvements in software maintainability, bug detection, and overall performance, yet their comparative effectiveness in real-world scenarios is still an ongoing topic of investigation (Latendresse, Abedu, Abdellatif, & Shihab, 2024; Motwani, Soto, Brun, Just, & Le Goues, 2020; Yang, Xia, Lo, & Grundy, 2022). As industries increasingly depend on software for critical operations, ensuring that development methodologies are optimized for producing robust, maintainable, and error-free systems is more important than ever (Nugroho, 2023). The urgency of this issue is heightened by the growing complexity of modern software systems, where even minor flaws can lead to significant operational disruptions, financial losses, or security breaches (Papastergiou, Kalogeraki, Polemi, & Douligeris, 2021). The demand for scalable, maintainable software is driving the need for research that rigorously evaluates how development practices like BDD and TDD impact key software quality metrics (BHATT, 2021). The urgency of investigating effective software development methodologies is clear from the widespread adoption of software in critical domains such as finance, healthcare, telecommunications, and autonomous systems (Ahonen, de Koning, Machado, Ghabcheloo, & Sievi-Korte, 2023). Failures in these sectors can lead to substantial adverse outcomes. BDD and TDD are two widely used methodologies that have gained traction because of their potential to reduce bugs and improve software reliability through rigorous automated testing practices (Zaeske & Durak, n.d.). However, while their theoretical benefits are well-documented, there is limited large-scale empirical evidence directly comparing these methodologies in real-world development settings. This gap in understanding presents an opportunity for more rigorous and comprehensive research (Myllynen Webb, 2023).

BDD emphasizes collaboration between developers and non-technical stakeholders by using natural language to define the desired behavior of a system before development begins (Smart & Molak, 2023). By structuring tests around these behavioral expectations, BDD aims to ensure that the software meets business requirements and is easy to maintain as it evolves (Nascimento, 2020). On the other hand, TDD focuses on creating test cases before writing code, ensuring that each feature is thoroughly tested from the outset. The cycle of writing tests,

*name of corresponding author

e- ISSN : 2541-2019
p- ISSN : 2541-044X

implementing code to pass the tests, and then refactoring the code is central to TDD's promise of creating highly reliable and clean software (Contieri, 2023). Although both methodologies promote high levels of automated test coverage and aim to improve software quality, the specific ways they impact long-term software maintainability, development time, and defect rates remain unclear (Colakoglu, Yazici, & Mishra, 2021; Robson, 2024; Saure, 2024). Prior research tends to focus on small-scale experiments or case studies, often in isolated environments, which may not be representative of larger, real-world projects (Osinga, Paudel, Mouzakitis, & Athanasiadis, 2022; Schäfer, Reis, & Stricker, 2022; Tibon, Geerligs, & Campbell, 2022). Moreover, while automated testing is a key feature of both BDD and TDD, few studies have explored how these methodologies compare across different dimensions of software quality such as bug detection, development efficiency, and overall maintainability (Pai, Joshi, & Rane, 2021).

The state of the art in empirical software engineering has evolved significantly, with the use of data analytics, machine learning, and statistical methods offering new insights into software quality (Zhong et al., 2021). Researchers have increasingly begun applying these tools to evaluate the effectiveness of development methodologies (Vindrola-Padros & Johnson, 2020). For instance, advanced machine learning models, such as Random Forest and XGBoost, are now being used to predict software defects and assess code quality (Cheng, Liu, Guo, Xu, & Zhang, 2020). However, most studies still lack large-scale empirical datasets that span multiple projects, limiting their ability to provide conclusive evidence on the relative strengths and weaknesses of BDD and TDD. Additionally, the interpretability of machine learning models in the software development context remains an ongoing challenge, as it is crucial for developers to understand why certain development practices lead to better outcomes (Krauß, Boden, Oppermann, & Reiners, 2021). Our research seeks to bridge this gap by conducting an empirical comparison of BDD and TDD using a comprehensive dataset of quality metrics gathered from multiple real-world software projects. This dataset contains essential indicators of software quality, including test coverage percentage, development time, code complexity, and bugs detected during development (Romano, Zampetti, Baldassarre, Di Penta, & Scanniello, 2022). By applying a variety of machine learning models: Random Forest, XGBoost, and LightGBM, we aim to provide a quantitative analysis of how these methodologies impact key software quality attributes (Ferenc, Bán, Grósz, & Gyimóthy, 2020). Furthermore, the use of SHAP (SHapley Additive exPlanations) allows us to explain the importance of each feature in the models, thereby offering insights into the specific aspects of BDD and TDD that most significantly influence software outcomes (Mishra & Otaiwi, 2020).

The goal of our research is twofold. First, we aim to provide actionable insights into which methodology: BDD or TDD, it offers greater benefits for specific software development goals, such as reducing bugs or speeding up development time. By rigorously analyzing the performance of both methodologies, we hope to offer clear guidance for software teams on when to adopt BDD, TDD, or a combination of both, based on empirical evidence. Second, we aim to contribute to the growing body of knowledge in empirical software engineering by introducing a data-driven approach to evaluate the effectiveness of development methodologies. This approach leverages machine learning models not only to predict software quality outcomes but also to explain the underlying factors driving those outcomes, thus enhancing the transparency and applicability of the findings. The contribution of this study lies in its novel approach to combining machine learning with software engineering metrics, offering a robust empirical evaluation of BDD and TDD. By analyzing real-world data from multiple projects, we are able to provide a more comprehensive and nuanced understanding of how these methodologies perform across various software quality dimensions. Additionally, the use of SHAP values in our analysis ensures that the results are interpretable, allowing software engineers and project managers to make informed decisions about which methodology to use based on the specific needs of their projects. The remainder of this article is structured as follows: In the next section, we describe the literature survey to explain the position of this research, then the next section will explain about methodology used to conduct this study, including data collection, preprocessing, and the machine learning models applied. We also detail the evaluation metrics used to compare the performance of BDD and TDD. Following the methodology, we present the results of our empirical analysis, highlighting the differences in bug detection rates, development time, and maintainability between the two methodologies. We also provide SHAP analysis to explain the importance of various software quality features. In the discussion section, we interpret the results and suggest practical recommendations for software development teams. Finally, we conclude by summarizing the key findings of the study and outlining directions for future research.

LITERATURE REVIEW

The evolution of software development methodologies has been driven by the need to enhance software quality, reduce bugs, and streamline the development process. Behavior-Driven Development (BDD) and Test-Driven Development (TDD) have emerged as two prominent approaches that emphasize different aspects of development and testing (Alkadhim et al., 2022). While BDD focuses on collaboration between developers and stakeholders to create executable specifications, TDD revolves around writing tests before developing code to ensure that every functionality is covered by tests. Both methodologies claim to improve software quality, yet the empirical comparison of their effectiveness has remained an area of ongoing research. In recent years, machine

*name of corresponding author

e- ISSN : 2541-2019
p- ISSN : 2541-044X

learning models have been employed to predict software quality based on development of metrics. This literature review examines studies comparing BDD and TDD, as well as recent research on the application of machine learning models in predicting software quality.

Comparison of BDD and TDD

Behavior-Driven Development (BDD) is designed to encourage collaboration between all stakeholders involved in software development. Studies such as those by (Smart & Molak, 2023) and (Güncan & Onay Durdu, 2021) emphasize BDD's strength in reducing misunderstandings between developers and stakeholders by providing a common language for describing functionality. However, empirical studies comparing BDD to other methodologies, such as TDD, have shown mixed results. For instance, (Rocha Silva, Winckler, & Bach, 2020) found that while BDD facilitated better communication and clarity in requirements, its impact on reducing bugs was not significantly different from TDD. This suggests that while BDD improves stakeholder engagement, it may not directly translate into measurable improvements in software quality. Furthermore, Test-Driven Development (TDD) has been widely studied and promoted as a method that encourages developers to think through requirements and edge cases before writing code. Research by (Nascimento, 2020; Parsa, Zakeri-Nasrabadi, & Turhan, 2025) demonstrated that TDD leads to more testable code, better modularization, and higher test coverage. However, there have been conflicting findings regarding its effect on software quality. For example, (Parsa, n.d.) conducted a meta-analysis of TDD studies and found that while TDD improves code quality metrics like maintainability and test coverage, its impact on reducing the number of bugs remains inconclusive. This indicates that the advantages of TDD may lie in code organization and long-term maintainability rather than immediate quality gains in terms of fewer bugs.

Machine Learning in Predicting Software Quality

The application of machine learning to predict software quality has gained traction in recent years, as traditional static analysis techniques struggle to capture the dynamic and contextual nature of software development. Studies by (Roman & Mnich, 2021) have employed decision trees, random forests, and support vector machines (SVMs) to predict software defects based on code metrics. These studies found that machine learning models could achieve high accuracy in predicting defect-prone modules by leveraging historical data. However, the challenge remains in generalizing these models to different projects and contexts. (Mehmood et al., 2023) pointed out that models trained on one project often fail to perform well on others, underscoring the need for cross-project learning or domain adaptation techniques.

While traditional machine learning models provide accurate predictions, they often lack transparency, making it difficult for developers to understand the reasoning behind the predictions. The introduction of SHapley Additive exPlanations (SHAP) by (Bi et al., 2020; Shivashankar, Orucevic, Kruke, & Martini, 2024) offers a solution to this problem by providing interpretability for complex models. SHAP assigns a contribution value to each feature, allowing developers to understand how individual features like test coverage and development time influence predictions. This level of transparency is crucial in software engineering, where decisions based on predictions need to be explained to stakeholders. Recent studies by [40]s have demonstrated the effectiveness of SHAP in enhancing the interpretability of software defect prediction models. By incorporating SHAP into the analysis, researchers can better understand the impact of various software metrics on quality outcomes, thus providing actionable insights for improving development practices.

Gaps in Existing Research

Despite the growing body of work comparing BDD and TDD, there remain several research gaps. First, most studies comparing BDD and TDD focus on qualitative metrics such as stakeholder satisfaction and test coverage rather than quantitative bug detection outcomes. While both methodologies claim to improve software quality, there is a lack of empirical data comparing their effectiveness in reducing software bugs. Furthermore, existing research often relies on small sample sizes or specific project contexts, limiting the generalizability of findings. Another gap lies in the application of machine learning to predict software quality. While many studies have used machine learning to predict software defects, few have integrated these predictions with specific software development methodologies like BDD and TDD. As a result, there is limited knowledge of how features related to BDD and TDD, such as test coverage or development time, interact with bug detection outcomes. Additionally, most machine learning models in this field lack interpretability, making it difficult for practitioners to understand the predictions and apply them in practice. Finally, the use of SHAP for model interpretability in the context of software quality prediction remains underexplored. Although SHAP has been used in other domains to explain machine learning models, its potential in providing insights into the interaction between software development practices and quality outcomes is still largely untapped.

Based on the literature reviewed, while both BDD and TDD have theoretical advantages in terms of improving communication and modularity, their impact on bug detection remains ambiguous. Machine learning models offer a promising approach for predicting software quality, but the lack of interpretability and cross-context

*name of corresponding author

e- ISSN : 2541-2019
p- ISSN : 2541-044X

generalizability are major challenges. SHAP provides a potential solution by making predictions more transparent and actionable, but its use in conjunction with development methodologies like BDD and TDD is still in its early stages. This research aims to fill these gaps by combining machine learning models with SHAP interpretability to compare the effectiveness of BDD and TDD in reducing software bugs. By leveraging polynomial feature expansion and SHAP, this study will provide a more detailed understanding of how test coverage, development time, and other metrics interact with software quality outcomes. Furthermore, this research will contribute to the growing field of interpretable machine learning by applying SHAP to a real-world software development context, offering actionable insights for both practitioners and researchers.

METHOD

This study applies to a data-driven approach to compare the effectiveness of Behavior-Driven Development (BDD) and Test-Driven Development (TDD) on software quality. The methodology covers several stages, including data collection, preprocessing, feature engineering, model training, evaluation, and model interpretation using SHAP values. Each step is supported by mathematical formulations to ensure rigor and clarity in understanding the effects of both methodologies.

Data Collection

The dataset used in this research was sourced from various software projects that employed either BDD or TDD as their development methodology and can be downloaded from (Pratama, 2024). The dataset includes several key variables for each project, such as *Test Coverage Percentage* (T_c), *Development Time* (T_d), *Code Complexity* (C_c), and *Bugs Detected* (B_d). Additionally, each project is labeled with the development methodology used, denoted by the categorical variable M, where M value is 1 if BDD is used and 0 if TDD is used. Thus, for each project i, the dataset provides a vector of features $X_i = [T_{c,i}, T_{d,i}, C_{c,i}, M_i]$ and an outcome $y_i = B_{d,i}$, representing the number of bugs detected during development.

Data Preprocessing

The preprocessing phase is crucial to ensure data consistency and prepare it for analysis. It includes feature engineering, standardization, and addressing multicollinearity. First, in Feature Engineering phase, we create a non-linear model in order to capture relationships between variables, polynomial features were generated. For each feature vector $X_i = [T_{c,i}, T_{d,i}]$, we created a third-degree polynomial expansion to account for interaction and higher-order effects. The transformed feature set $X_{\text{poly},i}$ is defined as presented in equation 1. This transformation enables the model to capture complex non-linear relationships between test coverage, development time, and bug detection outcomes.

$$X_{\text{poly},i} = [T_{c,i}, T_{d,i}, T_{c,i}^2, T_{d,i}^2, T_{c,i} \cdot T_{d,i}, T_{c,i}^3, T_{d,i}^3, T_{c,i}^2 \cdot T_{d,i}, T_{c,i} \cdot T_{d,i}^2]$$

Secondly, we execute standardization techniques, to ensure that all features contribute equally to the model, we standardized the numerical variables T_c, T_d, and C_c to have a mean of zero and a variance of one. For a feature x_i, its standardized value $x_{\text{std},i}$ is given in equation 2.

$$x_{\text{std},i} = \frac{x_i - \mu_x}{\sigma_x}$$

where μ_x is the mean of the feature and σ_x is its standard deviation. This transformation ensures that each feature has the same scale, preventing features with larger magnitudes from dominating the learning process. Third, we try to handle multicollinearity, this problem arises when features are highly correlated, leading to unstable model estimates. To detect and address multicollinearity, we computed the *Variance Inflation Factor (VIF)* for each feature. For a feature X_j, its VIF is defined in equation 3.

$$\text{VIF}(X_j) = \frac{1}{1 - R_j^2}$$

where R_j^2 is the coefficient of determination obtained by regressing X_j against all other features in the dataset. Features with a VIF greater than a predefined threshold (typically 5) were removed iteratively to ensure the model's stability.

Machine Learning Models

The central goal of this study is to model the relationship between the development methodology (M) and the number of bugs detected during development (B_d) using advanced machine learning algorithms. We employed

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 8, Nomor 4, October 2024
DOI : https://doi.org/10.33395/sinkron.v8i4.14201
e- ISSN : 2541-2019
p- ISSN : 2541-044X

three different models: Random Forest (RF), XGBoost (XGB), and LightGBM (LGB). Random Forest is an ensemble learning technique that constructs multiple decision trees during training. Each tree $$T_k$$ outputs a prediction for the target variable B_d, and the final prediction is obtained by averaging the predictions of all trees as presented in equation 4.

$$\hat{y}_i = \frac{1}{K}\sum_{k=1}^{K} T_k(X_i)$$

where $\hat{y}_i$ is the predicted number of bugs for project i, X_i is the feature vector, and K is the number of trees in the forest. Different with random forest, the XGBoost is a gradient-boosting algorithm that builds decision trees sequentially. Each tree corrects the errors of the previous ones by minimizing the loss function. The objective function for XGBoost is given in equation 5.

$$\text{Objective} = \sum_{i=1}^{n} L(\hat{y}_i, y_i) + \sum_{k=1}^{K} \Omega(f_k)$$

where $L(\hat{y}_i, y_i)$ is the loss between the predicted $\hat{y}_i$ and actual y_i values, and $\Omega(f_k)$ is a regularization term to prevent overfitting by penalizing model complexity. Lastly, the LightGBM, is well known as similar to XGBoost, it is a gradient-boosting algorithm that optimizes a loss function. However, it uses a leaf-wise growth strategy to construct trees more efficiently. The prediction for project i in LightGBM is obtained by summing the outputs of all trees in the ensemble as presented in equation 6.

$$\hat{y}_i = \sum_{k=1}^{K} f_k(X_i)$$

where f_k is the k-th decision tree.

Evaluation Metrics

The performance of each model was evaluated using four regression metrics: Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), R-squared (R^2), and Mean Absolute Percentage Error (MAPE). RMSE measures the average magnitude of prediction errors and is defined as presented in equation 7. A lower RMSE indicates better predictive accuracy, as it penalizes larger errors more heavily.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

Besides, we use Mean Absolute Error (MAE) where MAE is the average absolute difference between predicted and actual values, calculated as presented in equation 8. MAE is useful for understanding the average magnitude of prediction errors.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$$

The R^2 score measures the proportion of variance in the target variable that is explained by the model. It is given by $R^2 = 1 - \frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$ where $\bar{y}$ is the meaning of the actual values. An R^2 value close to 1 indicates that the model explains a large portion of the variance in the data. Mean Absolute Percentage Error (MAPE) is a relative measure of prediction error, defined as $\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{\hat{y}_i - y_i}{y_i}\right|$. It expresses the prediction error as a percentage of the actual value. To explain the contributions of each feature to the model's predictions, we used SHAP (SHapley Additive exPlanations) values. SHAP values provide a fair attribution of each feature's contribution to the model's output. For feature i in the feature set F, the SHAP value ϕ_i is computed as $\phi_i = \sum_{S \subseteq F \setminus \{i\}} \frac{|S|!(|F|-|S|-1)!}{|F|!}[f(S \cup \{i\}) - f(S)]$ where S is a subset of features excluding i, and $f(S)$ is the model's prediction when only the features in $S are used. This formula ensures that each feature's contribution is evaluated in the context of all possible feature combinations. SHAP values allow us to understand how test coverage, development time, code complexity,

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 8, Nomor 4, October 2024
DOI : https://doi.org/10.33395/sinkron.v8i4.14201
e- ISSN : 2541-2019
p- ISSN : 2541-044X

and the choice of development methodology (BDD vs. TDD) influence the number of bugs detected. This provides valuable interpretability, making it easier for software engineers to understand the impact of each feature on software quality. This study employs a rigorous methodology to compare BDD and TDD using machine learning models. By applying polynomial feature expansion, standardization, and advanced machine learning techniques, we gain insights into how these development methodologies affect software quality. The use of SHAP values enhances interpretability, providing a deeper understanding of the factors driving model predictions and offering practical recommendations for optimizing software development practices.

RESULT

As presented in the table 1, the performance of three machine learning models: Random Forest (RF), XGBoost (XGB), and LightGBM (LGB) was evaluated using K-fold cross-validation. Metrics such as RMSE, R^2, MAE, and MAPE were used to assess the predictive power of each model for detecting bugs based on test coverage and development time. For the Random Forest model, the RMSE ranged from 6.5645 to 8.2446 across five folds, reflecting moderate predictive power. The R^2 values ranged from 0.0393 to -9.2451, suggesting limited variance explanation. MAE values were between 5.3250 and 7.6050, and MAPE showed an average error percentage of 22.24% to 32.75%.

The XGBoost model exhibited slightly worse performance, with RMSE ranging from 6.5869 to 11.5728. Negative R^2 values dominated, with the worst being -22.5482 and the best reaching only 0.0077. MAE values spanned 5.3254 to 10.2551, while MAPE varied between 21.01% and 40.92%. The LightGBM model faced notable challenges, with RMSE values ranging from 2.4820 to 8.4410 and consistently negative R^2 scores, the lowest being -0.8227. The MAE ranged from 2.2500 to 7.2500, and MAPE was between 9.01% and 30.17%. SHAP analysis provided insights into feature importance for all models. The aggregated feature importance (Figure 7) highlighted $Test\_Coverage\_Percentage^2 \times Development\_Time\_Hours$ as consistently influential. SHAP dependence plots (Figures 1–6) revealed non-linear interactions, particularly between test coverage and development time, which varied across models.

DISCUSSION

The results indicate significant challenges in predicting bug detection outcomes using machine learning models based solely on test coverage and development time. Despite moderate RMSE values, the consistently negative R^2 scores highlight the models' poor ability to generalize and explain variance in unseen data. The Random Forest model performed moderately better than XGBoost and LightGBM. Its lower RMSE and MAPE suggest a better fit to the training data, likely due to its robustness in capturing non-linear interactions. However, the negative R^2 scores indicate overfitting and limited generalization to unseen data. The SHAP analysis (Figures 1 and 2) demonstrated the model's reliance on higher-order polynomial feature interactions, such as the interplay between test coverage and development time.

The XGBoost model struggled more with generalization, as seen in its higher RMSE and MAPE, along with the extremely poor R^2 values. The SHAP analysis (Figures 3 and 4) showed similar feature interactions to Random Forest but suggested greater sensitivity to dataset structure, which may have impacted performance. The LightGBM model faced the most difficulty, as indicated by warnings during training and consistently poor metrics. The inability to split data meaningfully suggests a need for improved hyperparameter tuning or additional feature engineering. The SHAP analysis (Figures 5 and 6) confirmed limited feature importance and inconsistent relationships between inputs and predictions. The aggregated feature importance (Figure 7) emphasized the significance of non-linear feature interactions, particularly between $Test\_Coverage\_Percentage^2 \times Development\_Time\_Hours$. However, the inability of all models to achieve positive R^2 scores underscores the complexity of the problem. These findings suggest that additional features, such as code complexity or historical bug data, might be needed to improve performance. Moreover, enhanced feature engineering techniques or alternative algorithms better suited for small, non-linear datasets may provide more robust predictions.

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 8, Nomor 4, October 2024
DOI : https://doi.org/10.33395/sinkron.v8i4.14201
e- ISSN : 2541-2019
p- ISSN : 2541-044X

Figure 1. SHAP Summary Plot Random Forest

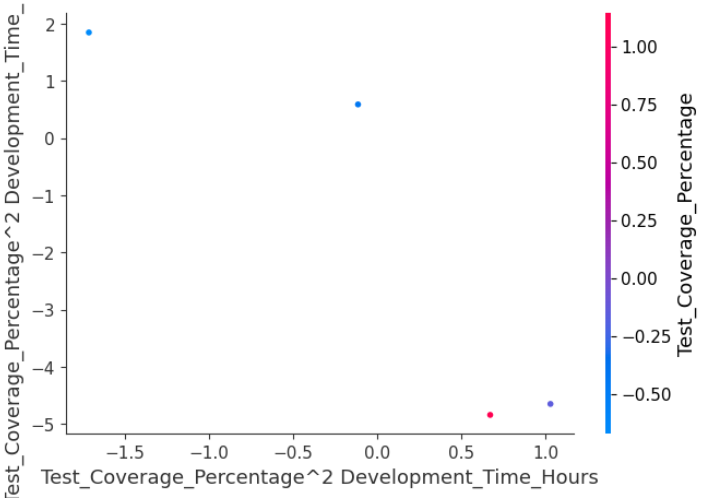

Figure 2. SHAP Dependence Plot Random Forest

Figure 3. SHAP Summary Plot XGBoost

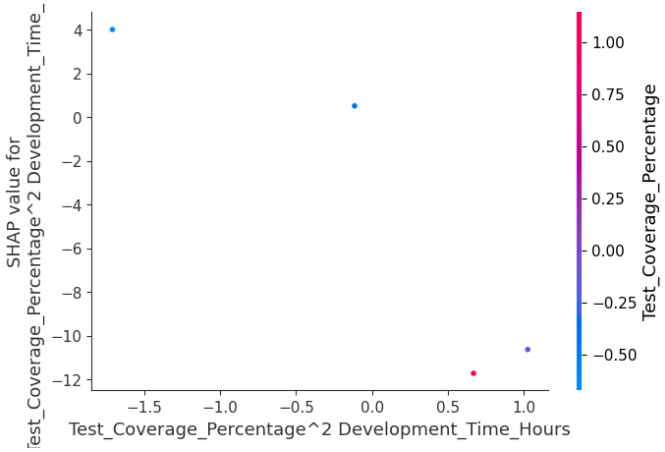

Figure 4. SHAP Dependence Plot XGBoost

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 8, Nomor 4, October 2024
DOI : https://doi.org/10.33395/sinkron.v8i4.14201
e- ISSN : 2541-2019
p- ISSN : 2541-044X

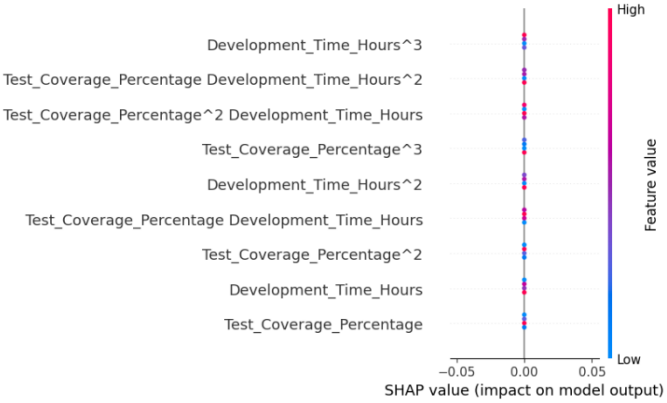

Figure 5. SHAP Summary Plot LightGBM

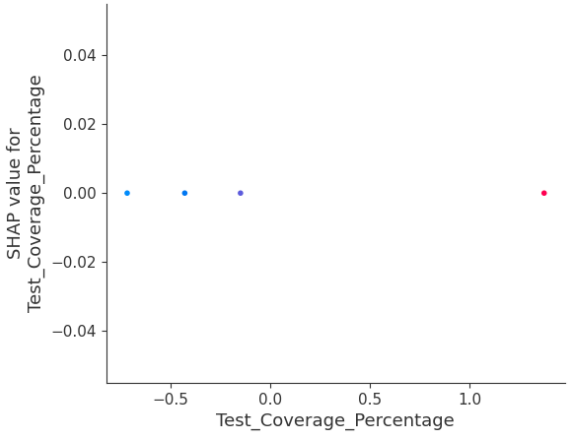

Figure 6. SHAP Dependence Plot LightGBM

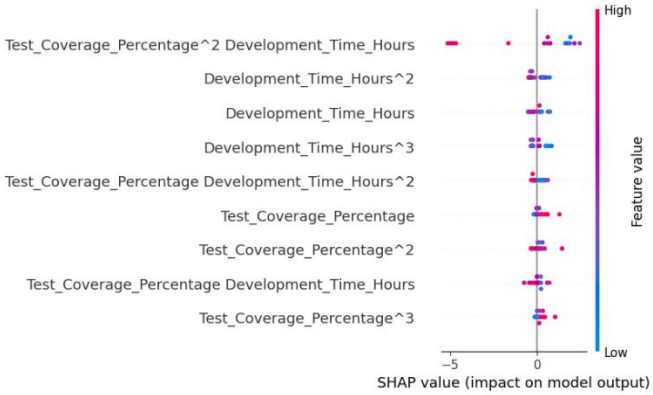

Figure 7. Feature Importance

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 8, Nomor 4, October 2024
DOI : https://doi.org/10.33395/sinkron.v8i4.14201
e- ISSN : 2541-2019
p- ISSN : 2541-044X

Table 1. Model results for each fold

Model	Fold	RMSE	R2	MAE	MAPE (%)
Random Forest	1	8.2153	0.0393	7.0175	27.72
Random Forest	2	6.5645	-0.3158	5.325	26.95
Random Forest	3	8.1045	-0.5756	7.605	32.75
Random Forest	4	7.6334	-9.2451	7.0775	27.49
Random Forest	5	8.2446	-1.6144	6.93	22.24
XGBoost	1	8.3491	0.0077	6.0028	21.01
XGBoost	2	6.5869	-0.3248	5.3254	26.97
XGBoost	3	8.274	-0.6422	6.4994	26.72
XGBoost	4	11.5728	-22.5482	10.2551	40.92
XGBoost	5	10.3544	-3.1236	8.7811	28.97
LightGBM	1	8.441	-0.0142	7.25	30.17
LightGBM	2	5.8095	-0.0305	4.5	22.72
LightGBM	3	6.741	-0.09	5.5938	26.75
LightGBM	4	2.482	-0.0831	2.25	9.01
LightGBM	5	6.8841	-0.8227	5.3125	15.94

CONCLUSION

This study has examined the existing body of literature on Behavior-Driven Development (BDD), Test-Driven Development (TDD), and their impact on software quality, with a particular focus on the use of machine learning models to predict bug detection. A key takeaway from the review is that while both BDD and TDD have their merits in enhancing software quality, their comparative impact on bug detection remains underexplored in empirical, data-driven studies. Current research tends to focus on qualitative assessments or anecdotal evidence rather than systematic, quantitative comparisons. Moreover, while machine learning models such as Random Forest, XGBoost, and LightGBM have proven effective in predicting software defects based on historical data, their application in evaluating the effectiveness of software development methodologies like BDD and TDD is still limited. Existing machine learning studies typically focus on variables such as code complexity and commit frequency, rather than directly comparing development methodologies. Furthermore, the lack of model interpretability is a significant limitation in the practical use of machine learning for software quality prediction. The introduction of SHAP values to improve model interpretability presents an exciting opportunity to bridge this gap. SHAP allows for a deeper understanding of how factors such as test coverage and development time influence software defects, providing actionable insights for developers. However, the application of SHAP to compare BDD and TDD, and more broadly, to software quality prediction, has not been extensively researched. In conclusion, this literature review underscores the need for further empirical research that combines machine learning with interpretability tools like SHAP to provide a more nuanced, data-driven understanding of how BDD and TDD influence software quality. By addressing these gaps, future studies can contribute significantly to both the academic field and the practical application of these methodologies in real-world software development.

REFERENCES

Ahonen, A., de Koning, M., Machado, T., Ghabcheloo, R., & Sievi-Korte, O. (2023). An exploratory study of software engineering in heavy-duty mobile machine automation. *Robotics and Autonomous Systems*, *165*, 104424.

Alkadhim, H. A., Amin, M. N., Ahmad, W., Khan, K., Nazar, S., Faraz, M. I., & Imran, M. (2022). Evaluating the strength and impact of raw ingredients of cement mortar incorporating waste glass powder using machine learning and SHapley additive ExPlanations (SHAP) methods. *Materials*, *15*(20), 7344.

BHATT, P. C. P. (2021). *Software design, architecture and engineering: Concepts and practice*. PHI Learning Pvt. Ltd.

Bi, Y., Xiang, D., Ge, Z., Li, F., Jia, C., & Song, J. (2020). An interpretable prediction model for identifying N7-methylguanosine sites based on XGBoost and SHAP. *Molecular Therapy-Nucleic Acids*, *22*, 362–372.

Cheng, X., Liu, N., Guo, L., Xu, Z., & Zhang, T. (2020). Blocking bug prediction based on XGBoost with enhanced features. *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, 902–911.

Colakoglu, F. N., Yazici, A., & Mishra, A. (2021). Software product quality metrics: A systematic mapping study. *IEEE Access*, *9*, 44647–44670.

Contieri, M. (2023). *Clean Code Cookbook: Recipes to Improve the Design and Quality of Your Code*. " O'Reilly

*name of corresponding author

Media, Inc."
Ferenc, R., Bán, D., Grósz, T., & Gyimóthy, T. (2020). Deep learning in static, metric-based bug prediction. *Array*, *6*, 100021.
Güncan, D., & Onay Durdu, P. (2021). A user-centered behavioral software development model. *Journal of Software: Evolution and Process*, *33*(2), e2274.
Krauß, V., Boden, A., Oppermann, L., & Reiners, R. (2021). Current practices, challenges, and design implications for collaborative AR/VR application development. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 1–15.
Latendresse, J., Abedu, S., Abdellatif, A., & Shihab, E. (2024). An Exploratory Study on Machine Learning Model Management. *ACM Transactions on Software Engineering and Methodology*.
Mehmood, I., Shahid, S., Hussain, H., Khan, I., Ahmad, S., Rahman, S., … Huda, S. (2023). A novel approach to improve software defect prediction accuracy using machine learning. *IEEE Access*, *11*, 63579–63597.
Mishra, A., & Otaiwi, Z. (2020). DevOps and software quality: A systematic mapping. *Computer Science Review*, *38*, 100308.
Motwani, M., Soto, M., Brun, Y., Just, R., & Le Goues, C. (2020). Quality of automated program repair on real-world defects. *IEEE Transactions on Software Engineering*, *48*(2), 637–661.
Myllynen Webb, K. (2023). *Mixed-method research approaches within non-governmental programmes to improve maternal and child health in Zimbabwe*. London School of Hygiene \& Tropical Medicine.
Nascimento, N. P. do. (2020). *A study of teaching BDD in active learning environments*. Pontif{\'\i}cia Universidade Católica do Rio Grande do Sul.
Nugroho, I. (2023). Elevating Software Development Frameworks: Harnessing Automation Testing to Drive Continuous Improvement and Quality Assurance. *Sage Science Review of Educational Technology*, *6*(1), 101–136.
Osinga, S. A., Paudel, D., Mouzakitis, S. A., & Athanasiadis, I. N. (2022). Big data in agriculture: Between opportunity and solution. *Agricultural Systems*, *195*, 103298.
Österholm, V. (2021). *Overview of Behaviour-Driven Development tools for web applications*.
Pai, A. R., Joshi, G., & Rane, S. (2021). Quality and reliability studies in software defect management: a literature review. *International Journal of Quality \& Reliability Management*, *38*(10), 2007–2033.
Papastergiou, S., Kalogeraki, E.-M., Polemi, N., & Douligeris, C. (2021). Challenges and issues in risk assessment in modern maritime systems. *Advances in Core Computer Science-Based Technologies: Papers in Honor of Professor Nikolaos Alexandris*, 129–156.
Parsa, S. (n.d.). *Software Testing Automation*.
Parsa, S., Zakeri-Nasrabadi, M., & Turhan, B. (2025). Testability-driven development: An improvement to the TDD efficiency. *Computer Standards \& Interfaces*, *91*, 103877.
Pratama, Y. (2024). *TDD and BDD Comparison*. Retrieved from https://www.kaggle.com/datasets/yogi2727/tdd-and-bdd-comparison/data
Rahman, S., & Nadia, F. (2024). Pioneering Testing Technologies: Advancing Software Quality Through Innovative Methodologies and Frameworks. *Journal of Artificial Intelligence and Machine Learning in Management*, *8*(2), 44–70.
Robson, C. (2024). *Real world research*. John Wiley \& Sons.
Rocha Silva, T., Winckler, M., & Bach, C. (2020). Evaluating the usage of predefined interactive behaviors for writing user stories: an empirical study with potential product owners. *Cognition, Technology \& Work*, *22*(3), 437–457.
Roman, A., & Mnich, M. (2021). Test-driven development with mutation testing--an experimental study. *Software Quality Journal*, *29*, 1–38.
Romano, S., Zampetti, F., Baldassarre, M. T., Di Penta, M., & Scanniello, G. (2022). Do static analysis tools affect software quality when using test-driven development? *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 80–91.
Saure, W. (2024). *Do tests really enable change? On the relationship between unit test coverage and maintainablity of production code*.
Schäfer, A., Reis, G., & Stricker, D. (2022). A survey on synchronous augmented, virtual, andmixed reality remote collaboration systems. *ACM Computing Surveys*, *55*(6), 1–27.
Shivashankar, K., Orucevic, M., Kruke, M. M., & Martini, A. (2024). Identifying Technical Debt and Its Types Across Diverse Software Projects Issues. *ArXiv Preprint ArXiv:2408.09128*.
Silva, D. I., & Siriwardana, L. K. B. (2023). *Comparative Analysis of Software Quality Assurance Approaches in Development Models*.
Smart, J. F., & Molak, J. (2023). *BDD in Action: Behavior-driven development for the whole software lifecycle*. Simon and Schuster.
Tibon, R., Geerligs, L., & Campbell, K. (2022). Bridging the big (data) gap: levels of control in small-and large-scale cognitive neuroscience research. *Trends in Neurosciences*, *45*(7), 507–516.

*name of corresponding author

e- ISSN : 2541-2019
p- ISSN : 2541-044X

Vindrola-Padros, C., & Johnson, G. A. (2020). Rapid techniques in qualitative research: a critical review of the literature. *Qualitative Health Research*, *30*(10), 1596–1604.

Yang, Y., Xia, X., Lo, D., & Grundy, J. (2022). A survey on deep learning for software engineering. *ACM Computing Surveys (CSUR)*, *54*(10s), 1–73.

Zaeske, W., & Durak, U. (n.d.). *DevOps for Airborne Software*.

Zhong, S., Zhang, K., Bagheri, M., Burken, J. G., Gu, A., Li, B., … others. (2021). Machine learning: new ideas and tools in environmental science and engineering. *Environmental Science \& Technology*, *55*(19), 12741–12754.

*name of corresponding author