Towards Refined Code Coverage: A New Predictive Problem in Software Testing

Carolin Brandt
Delft University of Technology
c.e.brandt@tudelft.nl

Aurora Ramírez
University of Córdoba
aramirez@uco.es

***Abstract*—To measure and improve the strength of test suites, software projects and their developers commonly use code coverage and aim for a threshold of around 80%. But what is the 80% of the source code that should be covered? To prepare for the development of new, more refined code coverage criteria, we introduce a novel predictive problem in software testing: whether a code line is, or should be, covered by the test suite. In this short paper, we propose the collection of coverage information, source code metrics, and abstract syntax tree data and explore whether they are relevant to predict whether a code line is exercised by the test suite or not. We present a preliminary experiment using four machine learning (ML) algorithms and an open source Java project. We observe that ML classifiers can achieve high accuracy (up to 90%) on this novel predictive problem. We also apply an explainable method to better understand the characteristics of code lines that make them more "appealing" to be covered. Our work opens a research line worth to investigate further, where the focus of the prediction is the code to be tested. Our innovative approach contrasts with most predictive problems in software testing, which aim to predict the test case failure probability.**

***Index Terms*—Code Coverage, Test Adequacy, Software Testing, Machine Learning, Explainable Artificial Intelligence**

I. Introduction

The extent to which tests cover the structure of source code—code coverage—has been used and debated for years as a test adequacy criterion. Studies show that coverage itself is not well correlated with post-release bug detection [1], and researchers propose stronger criteria such as mutation score [2], [3] or checked coverage [4], [5]. However, code coverage is widely used as a quality indicator for test suites [6]–[8], possibly due to its greater understandability [9], [10].

Coverage can serve as a necessary, but not a sufficient, quality criterion [11]: To have a strong test suite in terms of bug detection, you need high coverage, but high coverage is not sufficient to prove a test suite is strong. The basis for this is that in order to find a fault in code via testing, we need to at least execute that code during testing. While academically we claim that coverage alone is not sufficient, in practice projects are recommended to strive for a coverage threshold of 80 %. The rationale is that covering every bit of code might not be worth the engineering effort compared to the benefits gained from these tests [12]. Recent studies indicated that not all uncovered lines are worth covering according to developers [13], [14]. This leads us to the question of how we can effectively identify those lines of code that should be covered and distinguish them from lines that are less critical.

With this paper, we take the first steps towards a new adequacy criterion of *refined code coverage*, which takes into account the diversity of code lines to provide a more realistic—in the real-world project sense—judgment of how relevant it is to cover a given line of code. To start off, we propose learning from projects and their test suites, by studying the line-by-line code coverage. Our hypothesis is that if we can characterize the covered lines, then it would be possible to determine whether lines from other projects should be covered or not using machine learning (ML). As a first attempt, we propose the definition of characteristics of the source code at the line and method level, based on the common features such as size, complexity, function calls, etc. We also explore specific features related to the construct block and statements to identify whether certain functions (e.g., constructors) and control statements (conditional, loops, exceptions) are more frequently covered. After extracting the features for an example project, we analyze the performance of four ML algorithms. Our preliminary results are promising, as some algorithms achieve an accuracy of 90%. Inspection of the most relevant features suggests that predictions are based on a variety of properties, both related to size (line length), the method where the code line appears, and—less often—what the code line represents itself (type of statement).

II. Background and Related Work

In this section, we discuss relevant related work on code coverage and machine learning for software testing. For this study, "code coverage" refers to line coverage, sometimes also called statement coverage. Alternative structural coverage metrics are instruction or branch coverage [12]. We focus on line coverage, as it is widely used by developers, especially when visualizing code coverage [9], [15].

A. Studies of Code Coverage

Several studies explore what code is (not) covered by tests. Hora [16] found that popular Python projects exclude code from coverage that is non-runnable, debug-only, defensive, or platform-specific. The developers' motivation is that the code is low-level, complex or not tested. Zhai et al. [17] observe that for libraries, more deeply nested code is significantly less likely to be covered. Older code is more likely to be covered, while exception handling statements are covered much less frequently than other statements. Lima et al. [18] also find for

Java that exception-handling code is less covered than other code. In the JUnit project [19], van Deursen observes that deprecated code is less covered. The same applies to small blocks that are described as "too simple to test", "dead by design" or behaviors that are unlikely to happen, such as exception throwing and handling. We build our proposal on these findings, including features related to the type of code statement, e.g., whether a line belongs to a try/catch block.

Other work focuses on uncovered code and whether to nudge developers to cover it. Brandt et al. [13] observed that Mozilla developers find some coverage gaps not worth closing because they are small or concern early returns for invalid inputs. Gaps were also less important to close when they were unlikely reached or unlikely buggy because they are old and rarely changed. Ivanković et al. [14] identified uncovered code that should be tested by unit tests, by calculating the similarity to tested and frequently executed code.

A third angle to investigate is coverage evolution. Chen et al. [20] study test executions and coverage evolution for Java projects, concluding that coverage typically decreases in test-failure-introducing commits. Hilton et. al. [21] zoom in on how a change impacts the coverage of changed and not changed lines, concluding that aggregated code coverage can be misleading. Patches that appear to leave coverage unchanged can still change which lines are covered. They call for tools to also show why coverage changed. In contrast to these works, we take a snapshot look at code coverage at a given point in time and dive deeper into characterizing the lines of code that are (not) covered.

B. *Machine Learning for Software Testing*

Machine learning (ML) provides effective algorithms to solve complex predictive problems in software testing [22]. The goal is to predict a target variable as a combination of several input variables [23]. When the target variable is discrete, e.g., a boolean value, a classification problem is defined. If the variable is continuous, a regression problem is solved. In software testing, the input variables—features in ML terminology—are metrics extracted from the test code, such as length or coverage, and the testing process, such as number of previous executions and failure rate [24]. We focus on supervised learning [25], whose training strategy consists in using part of the collected data (train dataset) to build the predictive model. The train dataset includes labeled samples, i.e., both features and target values are known, so that the algorithm searches for the best fit between features and target variable. To assess the generalization prediction ability of the predictive model, the remaining part of the data is considered. The test dataset serves to compare the predicted result with the actual target values for previously unseen samples.

Supervised learning to assist in testing activities has been widely explored in recent years but has mainly focused on making predictions about different aspects of test cases. Some examples of regression problems are the estimation of test case readability [26], the estimation of branch coverage in search-based test case generation [27], the prediction of coverage in compiler testing [28] or software testing effort estimation expressed in person-hours [29]. As for classification, test case selection and prioritization are often supported by ML techniques [30]. Recent studies have also applied this approach to test smell detection [31], flaky test detection [32], and choosing CI configurations for automated testing [33].

ML models can exhibit high accuracy in their predictions, but at the cost of complex decision mechanisms that prevent software developers from understanding how decisions are made [34]. Providing explanations together with the predictions is highly relevant in many software testing scenarios [35]. Previous studies have applied explainable methods to know why a particular test case should be prioritized [35], [36] or which coverage metric is more decisive for estimating the mutation score of test cases [37].

III. Characterizing Covered vs. Not Covered Lines

A. *Data Collection*

We collect various types of data: (1) coverage, (2) metrics, and (3) code structure data. For this initial, exploratory study, we limit the scope to one open source Java project. We pick the project, `allegro/hermes`, from Khatami and Zaidman's set of projects [6], [38]. For this project, we were able to automatically build and calculate `Jacoco` coverage. After cloning the project[1], we run the following tools: (1) `Jacoco` [39] to calculate the detailed code coverage of the existing test suite, and (2) `ck` [40] to calculate a broad set of method-level metrics. Finally, (3) we perform a basic abstract syntax tree (AST) analysis with `tree-sitter` [41] to determine the AST node that spans a line of code.

B. *Feature Definition*

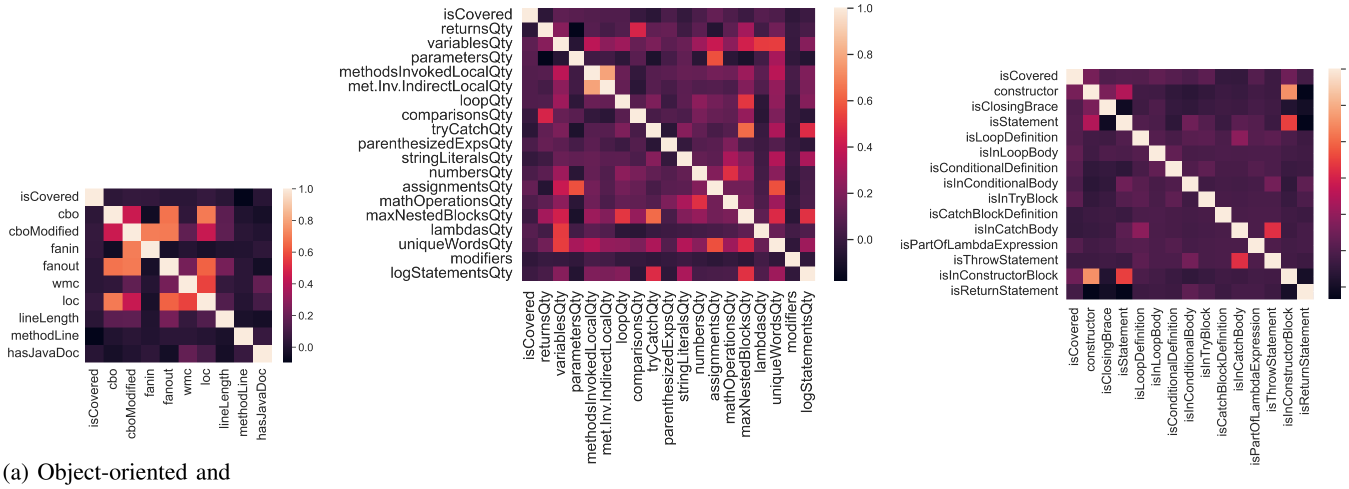
We assemble our features from the collected data. The target value **isCovered** is true if at least one instruction in the line is covered according to `Jacoco`.

The method-level metrics from `ck` are the same for all lines within a method. The first group of them mainly counts occurrences of language elements, and can be found in Figure 1b.

From *ck* we also take object-oriented and size metrics (Figure 1a): **methodLine**, the line number where the method starts; **cbo** ("coupling between objects"), number of dependencies excluding standard library; **cboModified**, cbo including references to the method by other types; **fanin**, number of input dependencies that reference this method; **fanout**, number of output dependencies referenced by this method; **wmc** ("weight method class"), McCabe's cyclomatic complexity, number of branching instructions; **loc**, source lines of code, excluding empty lines and comments.

Lastly, we determine the type of code statement, looking at the source code of the line and the AST node that spans it. The list of features can be found in Figure 1c.

[1] https://github.com/allegro/hermes, at commit d6cce21c



(a) Object-oriented and size metrics

(b) Frequency of code elements

(c) Type of code statements

Fig. 1: Correlation matrix between features (target feature: *isCovered*)

C. Data Preprocessing and Analysis

The initial dataset contains 20,741 samples (code lines) and 65 features from the allegro-hermes project. We exclude those features that serve to identify the line of code (project, source file, method, line number, line content) and some features used to derive others during code parsing. We then perform standard procedures to analyze the samples and obtain a dataset suitable for learning.

1) Data conversion: Features expressed as boolean values are converted to use binary values (0/1).
2) Missing values: For some lines of code, the AST analysis was not possible. Since the percentage of affected samples (5.82%) is low, we remove samples with missing values.
3) Data normalization: Some features, especially code metrics, have different ranges of numerical values. We normalized them using *MinMaxScaler* of sklearn library.

The cleaned dataset contains 18,564 samples and 47 features. The dataset is fairly balanced, with 45.06% of the samples corresponding to covered lines and 54.94% of the samples corresponding to uncovered lines. None of the final features show a high correlation with the target variable, according to the Pearson index. This indicates that estimating whether a line of code will be covered is not a trivial classification problem, as it will depend on a combination of features. Also based on the Pearson index, we discard 5 features that exhibit strong correlation (≥ 0.8) with others or have a unique value for all samples. The final dataset has 42 features (including the target). Figure 1 shows the Pearson correlation for each group of features: *code element frequency* (18 features), *size and object-oriented metrics* (9), and *code statement type* (14). A notebook with the detailed analysis of the cleaned dataset can be found in the replication package [42].

D. Algorithm Configuration, Training and Evaluation

The dataset is split into train and test partitions with a ratio of 70:30 and ensuring the original distribution of the target variable (class stratification). We use the train partition to build four classifiers using the following sklearn algorithms: Decision Tree (DT), k-Nearest Neighbors (kNN), Naive Bayes (NB) and Random Forest (RF). We apply hyperparameter optimization with grid search, using the training set and a 5-fold cross-validation strategy. The configuration obtained for each algorithm[2] are shown in Table I. With the optimal configuration, we train the classifiers and evaluate the performance on the test set. These performance metrics are: accuracy, balanced accuracy, precision, recall, and f-measure. The f-measure metric is employed during hyperparameter tuning.

TABLE I: Configuration of classification algorithms

Algorithm	Configuration
Decision tree (DT)	class_weight: "balanced", criterion: "entropy", max_depth: None, min_samples_leaf: 1
k-Nearest Neighbors (kNN)	leaf_size: 5, n_neighbors: 4, p: 1, weights: "distance"
Random Forest (RF)	class_weight: "balanced", max_depth: None, min_samples_leaf: 1, min_samples_split: 3, n_estimators: 300

After building the classifiers, we apply the importance permutation method [43] to understand the influence of each feature on the predictions. This method provides an average performance degradation score for each feature, such that the most relevant features are those that would cause the classifier to reduce its accuracy the most. For tree-based classifiers (DT and RF), we also inspect the internal decision structures to understand the relevance of the features during training.

E. Performance Evaluation and Explainability

Table II shows the results obtained on the test partition for the four applied algorithms. DT and kNN achieve higher accuracy (0.9975) than RF (0.9944) during training, but RF exhibits better generalization capabilities. NB performs poorly on both the train and test partitions, suggesting that its assumption about the independence of input features does not hold

[2] Naive Bayes is excluded as it does not have hyperparameters to optimize.

TABLE II: Classification results

Alg.	Accuracy	Bal. Accuracy	Precision	Recall	F-measure
DT	0.8930	0.8933	0.8701	0.8964	0.8830
kNN	0.7923	0.7897	0.7729	0.7633	0.7681
NB	0.5966	0.5779	0.5779	0.3888	0.4649
RF	0.9023	0.9018	0.8876	0.8968	0.8922

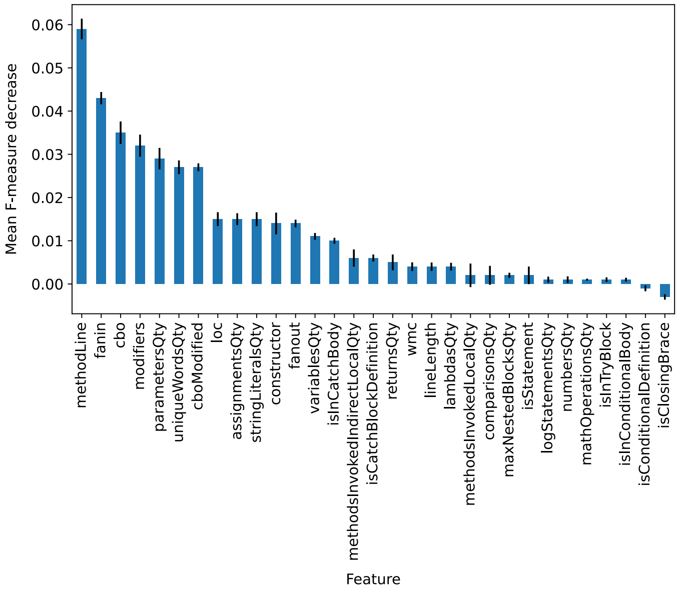


Fig. 2: Feature importance in the RF classifier.

for this predictive problem. The precision and recall values are well balanced for the DT and RF classifiers. Both classifiers correctly predict almost the same number of covered lines of code (true positives), but RF shows better precision due to a lower false positive rate. Based on the metrics reported for RF, we consider that predicting whether a line of code will be covered or not based on the proposed features seems feasible.

Inspection of the DT and RF classifiers shows that four features (*methodLine*, *uniqueWordsQty*, *lineLength* and *cbo*) appear among the top-5 most relevant features. For DT, *loc* is also important, while *cboModified* is slightly more relevant than *loc* for RF. These features were the most relevant during training, but they might not explain the influence on the prediction of unseen data. To explain this, Figure 2 shows the result of applying importance permutation on the RF classifier on the test partition. While *methodLine* and *cbo* are still important features, we observe that *fanin*, *modifiers* and *parametersQty* have more impact on the predictions than *uniqueWordsQty*, *cboModified* and *loc*. Equivalent plots are available for the rest of classifiers in the replication package [42].

IV. Discussion

While the good performance of the classifiers shows the feasibility of the prediction problem: "Should this line be covered by the test suite?", the features that emerge as relevant for the prediction allow for interesting observations and discussion.

Both coupling between objects (*cbo*/*cboModified*) and *fanin* characterize how many dependencies a method has, either by the types it uses or by the methods that call it. One explanation for the high importance could be that being called by many other methods makes it more likely the method is executed when one of those invoker methods is tested. Modifiers of a method determine whether a method can be directly testable by unit tests (e.g., "yes" for public and "no" for private methods). The number of parameters, unique words, *loc* and line length can all indicate the complexity of a method or line. More complex methods with potentially complicated logic may warrant more tests to ensure their behavior remains correct, while simple standard methods may be considered too trivial to test [10], [19]. Particularly interesting is the feature *methodLine*, which indicates the line in the file where a method starts. This implies that the location of a method within the source file can affect the likelihood of whether a line is tested. One possible explanation is that key functionalities of a class are implemented early and in the first methods, while edge cases are added later and at the end of the class file. Later methods may be tested less due to exceptional behavior, and newer code is known to be less tested [17], [18].

What stands out is that features related to the method as a whole (metrics, modifiers and quantities) are relevant to the classifiers, while line-individual features like the AST information are not as relevant. Potentially, this comes from little variation in the coverage of one method, i.e., if the method is covered, the tests run all lines. This interaction between coverage and features of neighboring lines, and its impact on coverage prediction, should be investigated further.

V. Conclusion and Outlook

Our investigation in this paper motivates deeper studies of code coverage in Java projects, supported by explainable predictive models. Our first steps involving the problem definition, feature extraction, and analysis of feature relevance shows that this topic is interesting to be further explored from both the ML and software testing perspectives.

As future work, the experimental study should be extended with more software projects and ML algorithms to generalize our findings. This would help identify broader aspects influencing whether a line is covered, but also to understand how coverage differs across projects on a fine-grained level. A second angle is considering which tests cover code, how directly they test it (i.e., how many method calls are between test and method) [44] and how often a line is executed by the test suite. This would indicate how intentionally a line is tested: Direct coverage by a few tests may indicate that the line is deliberately tested, while indirect coverage by many tests might be accidental coverage [9]. Finally, generating contrastive explanations could provide testers with additional knowledge to understand why a particular line of code should be prioritized for testing compared to others.

Acknowledgments

Work partially supported by Grant PID2023-148396NB-I00 funded by MICIU/AEI/10.13039/501100011033 and FEDER, EU; and AI4Software research network (RED2022-134647-T) funded by MICIU/AEI/10.13039/501100011033.

References

[1] P. S. Kochhar, D. Lo, J. Lawall, and N. Nagappan, "Code coverage and postrelease defects: A large-scale study on open source projects," *IEEE Trans. Reliab.*, vol. 66, no. 4, pp. 1213–1228, 2017.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *International Conference on Software Engineering (ICSE)*. ACM, 2005, pp. 402–411.

[3] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *International Conference on Software Testing Verification and Validation (ICST) Workshops*. IEEE CS, 2009, pp. 220–229.

[4] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE CS, 2011, pp. 90–99.

[5] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 214–224.

[6] A. Khatami and A. Zaidman, "State-of-the-practice in quality assurance in java-based open source software development," *Softw. Pract. Exp.*, vol. 54, no. 8, pp. 1408–1446, 2024.

[7] M. Ivankovic, G. Petrovic, R. Just, and G. Fraser, "Code coverage at google," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, pp. 955–963.

[8] S. Berner, R. Weber, and R. K. Keller, "Enhancing software testing by judicious use of code coverage information," in *IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE CS, 2007, pp. 612–620.

[9] C. Brandt and A. Zaidman, "How does this new developer test fit in? A visualization to understand amplified test cases," in *Working Conference on Software Visualization (VISSOFT)*. IEEE, 2022, pp. 17–28.

[10] ——, "Developer-centric test amplification," *Empir. Softw. Eng.*, vol. 27, no. 4, p. 96, 2022.

[11] C. R. Prause, J. Werner, K. Hornig, S. Bosecker, and M. Kuhrmann, "Is 100% test coverage a reasonable requirement? lessons learned from a space software project," in *International Conference on Product-Focused Software Process Improvement (PROFES)*, ser. LNCS, vol. 10611. Springer, 2017, pp. 351–367.

[12] M. Aniche, *Effective Software Testing: A Developer's Guide*. Simon and Schuster, 2022.

[13] C. Brandt, M. Castelluccio, C. Holler, J. Kratzer, A. Zaidman, and A. Bacchelli, "Mind the gap: What working with developers on fuzz tests taught us about coverage gaps," in *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM, 2024, pp. 157–167.

[14] M. Ivankovic, G. Petrovic, Y. Kulizhskaya, M. Lewko, L. Kalinovcic, R. Just, and G. Fraser, "Productive coverage: Improving the actionability of code coverage," in *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM, 2024, pp. 58–68.

[15] J. Lawrance, S. Clarke, M. Burnett, and G. Rothermel, "How well do professional developers test with code coverage visualizations? An empirical study," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2005, pp. 53–60.

[16] A. Hora, "Excluding code from test coverage: Practices, motivations, and impact," *Empirical Software Engineering*, vol. 28, no. 1, p. 16, 2023.

[17] H. Zhai, C. Casalnuovo, and P. T. Devanbu, "Test coverage in python programs," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*. IEEE / ACM, 2019, pp. 116–120.

[18] L. P. Lima, L. S. Rocha, C. I. M. Bezerra, and M. Paixão, "Assessing exception handling testing practices in open-source libraries," *Empir. Softw. Eng.*, vol. 26, no. 4, p. 85, 2021.

[19] (2012, 12) Line coverage: Lessons from junit. [Online]. Available: https://avandeursen.com/2012/12/21/line-coverage-lessons-from-junit/

[20] A. R. Chen, T. P. Chen, and S. Wang, "T-evos: A large-scale longitudinal study on CI test execution and failure," *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 2352–2365, 2023.

[21] M. Hilton, J. Bell, and D. Marinov, "A large-scale study of test coverage evolution," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2018, pp. 53–63.

[22] V. Durelli, R. Durelli, S. Borges, A. Endo, M. Eler, D. Dias, and M. G. aes, "Machine Learning Applied to Software Testing: A Systematic Mapping Study," *IEEE Trans. Rel.*, vol. 68, no. 3, pp. 1189–1212, 2019.

[23] A. Ramírez and B. Miranda, "Foundations of machine learning for software engineering," in *Optimising the Software Development Process with Artificial Intelligence*, J. Romero, I. Medina-Bulo, and F. Chicano, Eds. Singapore: Springer Nature Singapore, 2023, pp. 309–344.

[24] A. Ramírez, R. Feldt, and J. R. Romero, "A taxonomy of information attributes for test case prioritisation: Applicability, machine learning," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 21:1–21:42, 2023.

[25] S. Ajorloo, A. Jamarani, M. Kashfi, M. Haghi Kashani, and A. Najafizadeh, "A systematic review of machine learning methods in software testing," *Applied Soft Computing*, vol. 162, p. 111805, 2024.

[26] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 107–118.

[27] G. Grano, T. V. Titov, S. Panichella, and H. C. Gall, "Branch coverage prediction in automated testing," *J. Softw. Evol. Process.*, vol. 31, no. 9, 2019.

[28] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Coverage Prediction for Accelerating Compiler Testing," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 261–278, 2021.

[29] C. López-Martín, "Machine learning techniques for software testing effort prediction," *Software Quality Journal*, vol. 30, pp. 65–100, 2022.

[30] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: a systematic literature review," *Empirical Software Engineering*, vol. 27, 2021.

[31] V. Pontillo, D. Amoroso d'Aragona, F. Pecorelli, D. Di Nucci, F. Ferrucci, and F. Palomba, "Machine learning-based test smell detection," *Empirical Software Engineering*, vol. 29, 2024.

[32] J. Wang, Y. Lei, M. Li, G. Ren, H. Xie, S. Jin, J. Li, and J. Hu, "Flakyrank: Predicting Flaky Tests Using Augmented Learning to Rank," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2024, pp. 872–883.

[33] A. Lönnfält, V. Tu, G. Gay, A. Singh, and S. Tahvili, "An intelligent test management system for optimizing decision making during software testing," *Journal of Systems and Software*, vol. 219, p. 112202, 2025.

[34] C. Tantithamthavorn, J. Cito, H. Hemmati, and S. Chandra, "Explainable AI for SE: Challenges and Future Directions," *IEEE Software*, vol. 40, no. 3, pp. 29–33, 2023.

[35] A. Ramírez, M. Berrios, J. Romero, and R. Feldt, "Towards Explainable Test Case Prioritisation with Learning-to-Rank Models," in *AIST@ICST*. Dublin, Ireland: IEEE, 2023, pp. 66–69.

[36] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. C. Briand, "Scalable and accurate test case prioritization in continuous integration contexts," *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 1615–1639, 2023.

[37] G. Grano, F. Palomba, and H. Gall, "Lightweight Assessment of Test-Case Effectiveness Using Source-Code-Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 47, no. 4, pp. 758–774, 2021.

[38] A. Khatami and A. Zaidman, "State-of-the-practice in quality assurance in open source software development—replication package," 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6563549

[39] "Jacoco website," https://www.eclemma.org/jacoco/, accessed: 2024-11-25.

[40] "Ck tool on github," https://github.com/mauricioaniche/ck, accessed: 2024-11-25.

[41] "tree-sitter website," https://tree-sitter.github.io/tree-sitter/, accessed: 2024-11-25.

[42] Anonymous. (2024) Cleaned data and notebooks for the paper "Towards refined code coverage: A new predictive problem in software testing"'. [Online]. Available: https://anonymous.4open.science/r/understanding-covered-code-4950

[43] P. Biecek and T. Burzykowski, *Explanatory Model Analysis: Explore, Explain, and Examine Predictive Models*. Chapman and Hall/CRC, 2021.

[44] Q. Zhu, A. Zaidman, and A. Panichella, "How to kill them all: An exploratory study on the impact of code observability on mutation testing," *J. Syst. Softw.*, vol. 173, p. 110864, 2021.