IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2024.0429000

A Large-Scale Empirical Investigation into Cross-Project Flaky Test Prediction

ANGELO AFELTRA[1], ALFONSO CANNAVALE[1], FABIANO PECORELLI[2], VALERIA PONTILLO[3], and FABIO PALOMBA[1]

[1]Software Engineering (SeSa) Lab — University of Salerno, Fisciano, Italy (e-mail: angelo.afeltra99@gmail.com, a.cannavale7@studenti.unisa.it, fpalomba@unisa.it)
[2]Pegaso Telematic University, Naples, Italy (e-mail: fabiano.pecorelli@unipegaso.it)
[3]Software Languages (Soft) Lab — Vrije Universiteit Brussel, Belgium (e-mail: valeria.pontillo@vub.be)

Corresponding author: Valeria Pontillo (e-mail: valeria.pontillo@vub.be).

ABSTRACT Test flakiness arises when a test case exhibits inconsistent behavior by alternating between passing and failing states when executed against the same code. Previous research showed the significance of the problem in practice, proposing empirical studies into the nature of flakiness and automated techniques for its detection. Machine learning models emerged as a promising approach for flaky test prediction. However, existing research has predominantly focused on within-project scenarios, where models are trained and tested using data from a single project. On the contrary, little is known about how flaky test prediction models may be adapted to software projects lacking sufficient historical data for effective prediction. In this paper, we address this gap by proposing a large-scale assessment of flaky test prediction in cross-project scenarios, i.e., in situations where predictive models are trained using data coming from external projects. Leveraging a dataset of 1,385 flaky tests from 29 open-source projects, we examine static test flakiness prediction models and evaluate feature- and instance-based filtering methods for cross-project predictions. Our study underscores the difficulties in utilizing cross-project flaky test data and underscores the significance of filtering methods in enhancing prediction accuracy. Notably, we find that the *TrAdaBoost* filtering method significantly reduces data heterogeneity, leading to an F-Measure of 70%.

INDEX TERMS Flaky Tests, Software Testing, Machine Learning, Empirical Software Engineering.

I. INTRODUCTION

Regression testing is a commonly used practice that consists in re-running existing test cases to ensure that newly committed code changes have not adversely affected previously functioning features or introduced new defects [41]. This practice is vital for developers' decisions [16] and productivity [7], [35], as it promptly identifies real faults [41].

Regrettably, tests themselves are not immune to defects and may occasionally experience *flakiness* [32]: this occurs when a test shows both passing and failing behaviors when executed against identical code, rendering it unreliable and generating non-deterministic results due to asynchronous calls, network capabilities, or environmental dependencies [13], [32].

Flaky tests hide real bugs and are hard to reproduce due to their non-deterministic nature [32]. They inflate testing costs, as developers spend time debugging false failures [27] and undermine developer confidence in testing activities, potentially leading developers to ignore defects [13]. Moreover, flaky tests can disrupt various collateral testing tools, e.g., mutation testing [41], where variability in mutation scores may arise from flakiness, biasing the test quality assessment [10].

For these reasons, researchers have expanded knowledge through empirical studies uncovering causes of flakiness [13], [29], [30], [32], [34], and by developing automated techniques to detect and mitigate its impact [4], [12], [50], [53]. Among the automated detection techniques, machine learning models have shown promising results in flaky test prediction, with researchers exploring various features like textual features [42], dynamic indicators [2], and static metrics [44], achieving high accuracy.

Such existing research has primarily focused on a *within-project* scenario, that is, training and testing machine learning models with data coming from a single project or dataset. As such, the findings from these studies may hold for software projects that maintain historical data on flaky tests and can utilize this data to develop predictive models tailored to their specific test code. However, this is not always the case: new software projects may lack sufficient historical data for effective flaky test prediction, or the systematic collection of flaky test data may prove prohibitively costly. In these cases, a *cross-project* approach may be more appropriate: in this scenario, prediction models are trained using external data, i.e., flaky tests from other projects. Nonetheless, cross-project approaches pose challenges due to the inherent heterogeneity of training data from external sources. Indeed, each software project has unique development practices, testing methods, and environmental factors, resulting in varying characteristics of flaky tests. This diversity may hinder the generalization of predictive models to new projects and introduce biases based on training data characteristics. The current state of the art lacks investigations into this matter and, for this reason, it is still unclear how the intrinsic challenges presented by the cross-project scenario affect flaky test prediction.

Hence, this paper proposes a large-scale assessment of this matter. First, we investigate the performance of a static test flakiness prediction model built in our previous study [44] in a cross-project setting, to verify how challenging this setting actually is. Second, we assess the effectiveness of feature- and instance-based filtering methods for improving the quality of cross-project predictions. We leverage the IDoFT dataset, encompassing 1,385 flaky tests from 29 open-source projects. Our results confirm the hardness of exploiting cross-project flaky test data, with many models struggling due to data heterogeneity. Yet, promising outcomes emerge with filtering methods, notably the *TrAdaBoost* approach, achieving an F-Measure of approximately 70%.

Structure of the paper. Section II overviews the most closely related work, positioning our work within the current body of knowledge. Section III elaborated on the research questions and context of the study, while Section IV reports on the research methods employed to address our research questions. In Section V, we analyze and discuss the results achieved from our work. The potential limitations of our study are discussed in Section VI. Finally, Section VII concludes the paper and outlines our future research agenda.

II. RELATED WORK

Test flakiness is widely recognized and discussed by both practitioners and researchers [14], [35]. Barboni et al. [3], Parry et al. [39], and Zheng et al. [54] have provided systematic analyses of the state of the art. At the same time, additional studies have explored the grey literature on the subject [13], [20], [45].

This paper centers on automating the detection of flaky tests. In recent years, various approaches have been proposed, varying in technique (such as data-flow analysis or machine learning) and objective (like analyzing specific root causes versus prediction).

For instance, Lampel et al. [31] devised an approach to classify failing jobs as software bugs or flaky tests. Rehman et al. [49] investigated test failures without production code defects at Ericsson. Bell et al. [4] introduced DeFlaker, analyzing code coverage differences to identify emerging flakiness. Lam et al. [28] created iDFlakies to detect flaky tests by rerunning them in different orders. Rahman et al. [47], [48] proposed FlakeSync and FlakeRake for automatic detection of async and timing-dependent flaky tests. With respect to these papers, ours is complementary: we indeed aim at advancing the current knowledge on machine learning solutions. These solutions, when combined with the approaches discussed above, may enhance the defense against test flakiness, providing developers with a mechanism to locate potentially flaky tests that can then be further diagnosed with additional instruments provided by researchers.

As for the predictive methods to preemptively alert developers about potential test flakiness, major efforts have been devoted to the analysis of the best features to use to predict flaky tests. In particular, Bertolino et al. [5] and Pinto et al. [42] demonstrated the usefulness of textual features, i.e., the vocabulary used in a test may be an indication of its flakiness. Alshammari et al. [2] proposed FlakeFlagger, a technique that integrates static and dynamic features for flakiness prediction. Among their findings, the authors reported that code coverage indicators may provide insights into the flakiness of test code. Pontillo et al. [43], [44] proposed a technique solely based on static metrics, in an effort of making flaky test prediction more scalable. They showed that a fully static approach may obtain comparable performance to more complex models. In the context of this study, we leverage the insights coming from Pontillo et al. [43], [44] and exploits their model as a baseline for our experimentation. Finally, Camara et al. [6] observed a correlation between design issues in test cases, known as test smells, and test flakiness.

All these papers assessed flaky test prediction in a within-project setting. On the contrary, our work aims at exploring the performance in a cross-project scenario, hence extending the current body of knowledge on flaky test prediction.

III. RESEARCH QUESTIONS AND CONTEXT SELECTION

The *goal* of the study was to evaluate the performance of a static machine learning-based approach for flaky test prediction trained in a cross-project scenario, with the *purpose* of assessing the feasibility of cross-project test flakiness prediction. The *perspective* is of both researchers and practitioners. The former are interested in understanding how a cross-project setting affects the performance of test flakiness prediction, possibly identifying further areas of improvement. The latter are interested in assessing the feasibility of using cross-project training in practice, possibly identifying strategies to adopt test flakiness prediction in real-world scenarios.

TABLE 1: List of metrics used as independent variables.

Name	Description	Computed on ...
Production and Test Code Metrics		
CBO	Coupling Between Object, i.e., the number of dependencies a class has with other classes [9].	Production Class
Halstead Length	The number of operator occurrences and the number of operand occurrences.	Production Class
Halstead Vocabulary	The total number of distinct operators and operands in a function.	Production Class
Halstead Volume	Proportional to program size, represents the space necessary for storing the program.	Production Class
LOC	Lines of Code, considering both source and comment lines.	Production Class
LCOM2	Lack of Cohesion of Methods version 2, i.e., the percentage of methods that do not access a specific attribute averaged over all attributes in the class.	Production Class
LCOM5	Lack of Cohesion of Methods version 5, i.e., the density of accesses to attributes by methods.	Production Class
McCabe	It indicates the number of linearly independent paths through a program's source code [33].	Test Class
MPC	Message Passing Coupling, measures the number of messages passing among class objects.	Production Class
RFC	Response For a Class, i.e., the number of methods (including inherited ones) that can potentially be called by other classes [9].	Production Class
TLOC	Number of lines of code of the Test Suite.	Test Class
WMC	Weighted Methods per Class, i.e., the sum of the complexities (i.e., McCabe's Cyclomatic Complexity) of all the methods in a class [9]. Note that Chidamber and Kemerer [9] did not define a predefined complexity metric to consider for the computation of WMC. In our case, we opted for the McCabe metric to account for the individual complexity of methods.	Production Class
Code Smells		
Class Data Should Be Private	A class that exposes its attributes, violating the information hiding principle.	Production Class
Complex Class	When a class has a high cyclomatic complexity.	Production Class
Functional Decomposition	When in a class inheritance and polymorphism are poorly used.	Production Class
God Class	When a class has a huge dimension and implements different responsibilities.	Production Class
Spaghetti Code	When a class has no structure and declares a long method without parameters.	Production Class
Test Smells		
Assertion Density	Percentage of assertion statements in the test code.	Test Class
Assertion Roulette	When a test method has multiple non-documented assertions.	Test Class
Conditional Test Logic	Conditional code within a test method negatively impacts the ease of comprehension by developers.	Test Class
Eager Test	When a test method invokes several methods of the production object.	Test Class
Fire and Forget	A test that is at risk of exiting prematurely because it does not properly wait for the results of external calls.	Test Class
Mystery Guest	When a test method utilizes external resources (e.g., files, database, etc.).	Test Class
Resource Optimism	When a test method makes an optimistic assumption that the external resource (e.g., File) exists.	Test Class
Sensitive Equal.	When the toString method is used within a test method.	Test Class

A. RESEARCH QUESTIONS

Our study was structured around two main research questions. Stemming from the peculiar challenges posed by training a flaky test prediction model in a cross-project setting, specifically concerned with the need to take heterogeneous data into account, we start our investigation by assessing the performance of a state-of-the-art flaky test prediction model trained in such a cross-project scenario. This preliminary investigation provided us with a baseline for understanding the current capabilities and limitations of cross-project flaky test prediction. Particularly, we asked:

> **RQ$_1$.** *What is the performance of a static flaky test prediction model trained in a cross-project setting?*

Upon setting a baseline, we furthered our analysis by addressing the role of feature- and instance-based filtering methods in effectively training cross-project models. In closely-related research fields, like defect prediction [23], [55], these methods proved to improve prediction capabilities. More specifically, feature-based filtering methods involve selecting or extracting relevant features from the data to improve model performance by reducing noise and irrelevant information. Conversely, instance-based filtering methods focus on selecting or filtering specific instances or samples from the dataset to enhance the quality of the training data, thus improving the model's ability to generalize to new, unseen instances. In this respect, our goal was to transfer the knowledge accumulated in the literature to the problem of test flakiness prediction, assessing how these methods may actually contribute to empowering its predictive capabilities. Hence, we asked:

> **RQ$_2$.** *How do feature- and instance-based filtering methods improve test flakiness prediction capabilities?*

The ultimate outcome of our research aimed at enlarging the current body of knowledge on flaky test prediction, providing insights into its feasibility in a cross-project scenario. In terms of reporting, we followed the *ACM/SIGSOFT Empirical Standards*[1], in particular, the *"General Standard"* and *"Data Science"* guidelines.

B. CONTEXT OF THE STUDY

The *context* of our study encompasses part of the Java open-source projects featured in the IDoFT (Illinois Dataset of Flaky Tests) dataset.[2] This dataset comprises data on 6,446 flaky tests across 423 open-source projects. In our study, we analyzed 29 open-source projects with a total of 29,839 test classes —more details about the demographics of our dataset are reported in Table 2. The motivation behind selecting this dataset stems from its availability, extensive coverage of projects with different characteristics, and popularity within

[1] Available at: https://github.com/acmsigsoft/EmpiricalStandards.
[2] Available at: http://mir.cs.illinois.edu/flakytests.

the flaky test research community. Regarding testing procedures, all projects employ a continuous integration pipeline to validate code changes against a comprehensive test suite.

C. VARIABLES OF THE STUDY

To address our research questions, we required the selection of a response variable and a set of independent variables. The former was concerned with the actual indication of the flakiness of test cases. In this respect, we exploited the information available within the IDoFT dataset: particularly, we label each test case as *"flaky"* or *"non-flaky"*. As such, our study focuses on a binary classification task, similar to previous studies in the field [2], [42], [44]. The latter were concerned with the source code metrics to use as features of a flaky test prediction model, i.e., the properties of test cases that suggest the presence of a flaky test. We relied on the set of metrics described in Table 1. More specifically, these are the 25 features employed by Pontillo et al. [44] to devise a static flaky test prediction model. The rationale behind the use of the work by Pontillo et al. [44] as reference is threefold.

In the first place, the set of features comprises multiple dimensions of test code quality that have been previously linked to test flakiness. These dimensions include *production and test code metrics*, which offer insights into the structural complexity and stability of the codebase [15]: on the one hand, production code quality may affect its testability and, consequently, the likelihood of a test to be poorly designed; on the other hand, test code quality is intrinsically related to flakiness, e.g., poorly constructed tests can inadvertently introduce instability and unpredictability into the testing process. Additionally, the feature set includes *code smells*, i.e., poor design or implementation choices in the code, and *test smells*, i.e., suboptimal development practices applied while designing test suites: code and test smells further assess the testability and quality of test cases, potentially providing further insights into the emergence of test flakiness [6].

In the second place, Pontillo et al. [44] reported that a flaky test prediction model relying on these static metrics achieves similar performance than those of more computationally-expensive models, like the models relying on dynamic and textual features. Hence, we opted for a cost-effective yet performing approach, optimizing our data collection process.

Finally, our choice to build upon the work of Pontillo et al. [44] mitigated potential threats to validity associated with re-implementing third-party research methods. Indeed, two of the authors of our paper were also involved in the original study (with the fourth author being the primary contributor to [44]). This ensured a faithful replication of the work, as we were familiar with the design choices made and could avoid errors or misinterpretations in the data collection procedures. This latter aspect allowed a fair comparison of our findings with respect to related literature.

To compute the independent variables of the study, we used the scripts made publicly available by Pontillo et al. [44]. In doing so, one technical matter is worth discussing. To associate production code metrics with test code, we had to establish explicit links between test cases and their corresponding production classes—otherwise, we could not compute the value of the production code metrics. For this purpose, we employed a pattern-matching strategy based on naming conventions, as proposed and utilized in prior studies [17], [21], [40], [44]. This method simply involves using the name of a production class (e.g., *'ClassName'*) to locate the corresponding test class, identified by the same name as the production class but with the prefix or postfix *'Test'* (e.g., *'TestClassName'* or *'ClassNameTest'*). If this pattern matching failed, indicating that the production class associated with the test class could not be identified, we had to exclude the test from our analysis. As a result of this linking process, we had to exclude 18,213 test classes from the dataset. In these instances, developers did not adhere to the aforementioned naming conventions, making it impossible for us to establish proper links between production and test classes. For the remaining test classes, the outcome of the linking process required the removal of certain test cases, notably those containing methods named *'setUp'* and *'tearDown'*. These methods serve as fixtures responsible for managing resource allocation and deallocation for tests but are not linked to any production class. Consequently, following these filtering actions, the final dataset comprised 87,875 test cases (including 1,385 flaky tests) from 29 projects.

Table 2 shows the amount of flaky and non-flaky tests in the projects. We can observe a significant disparity, with the number of non-flaky tests overwhelmingly surpassing the number of flaky tests. The only exceptions are the `Typescript-generator` and `Visualee` projects, in which flaky tests represent 25% and 32% of the total test cases. In addition, for two other projects, i.e., `Http-request` and `Adyen-java-api-library`, we note a percentage of flaky tests between 15% and 18%, while for all other projects, the number of flaky tests accounts for less than 8% of the total number of test cases analyzed.

IV. RESEARCH METHOD

Upon selection of the variables of the study, we proceeded with the definition of the research methods required to address our research questions. Figure 1 reports a methodical overview, which we further discuss in this section.

A. RQ_1 - RESEARCH METHOD

To comprehensively assess the capabilities of cross-project flaky test prediction and address $\mathbf{RQ}_1$, we (1) experimented with multiple machine learning pipelines and (2) conducted an *ablation* study to verify the contribution and importance of individual model components, features, and parameters to the model's overall performance. These two steps did not only allow us to measure the performance of a cross-project model under different configurations, but also the impact of each component on the model's predictive ability.

Specifically, we began with the dataset consisting of 87,875 test cases previously labeled with information on features and flakiness. We structured our analysis into four primary train-

TABLE 2: Overview of the projects in analysis.

Project	#Test Classes in the Project	#Test Classes linked to the Production Class	#Test Cases	#Flaky Tests
Achilles	185	43	401	24
Activiti	618	92	964	22
Admiral	796	262	2,390	55
Adyen-java-api-library	42	31	253	45
Aem-core-wcm-components	248	117	1,092	24
Chronicle-Wire	221	55	460	35
Commons-lang	175	111	2,711	25
Data Flow Template	156	123	694	39
Druid	3,655	1,420	10,729	30
Dubbo	673	248	1,639	139
Fastjson	2,496	834	1,860	25
Flink	3,173	1,261	9,254	23
Graylog2-server	474	374	2,477	21
Hadoop	3,928	1,322	13,662	177
Http-request	3	1	184	28
Ignite-3	558	268	2,173	118
Jackrabbit-oak	2,072	919	10,101	20
Java WebSocket	37	15	419	28
Mockserver	407	218	2,104	27
Nacos	429	365	1,850	26
Nifi	1,734	1,183	11,145	138
Ormlite-core	132	95	1,042	90
Ozone	837	364	2,480	24
Servicecomb-java-chassis	875	470	2,423	41
Typescript-generator	80	26	132	34
Visualee	26	22	134	43
Vpc-java-sdk	1,047	1,039	2,551	22
Wildfly	4,546	159	910	37
Wro4j	216	189	1,641	25

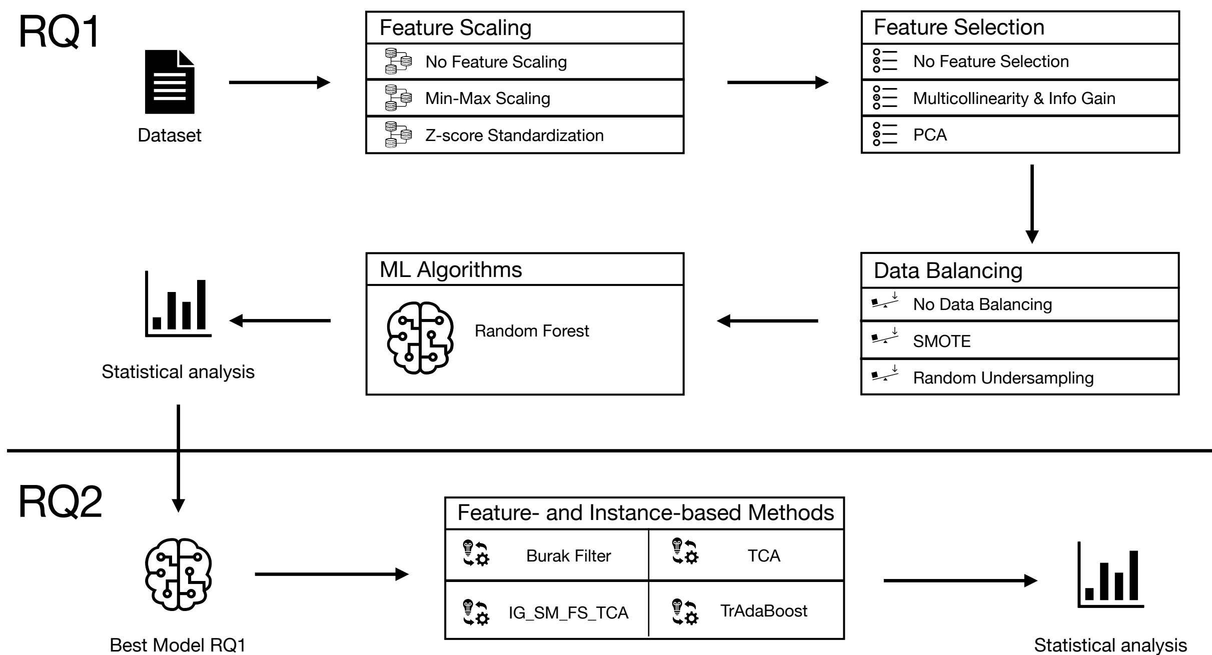


FIGURE 1: Overview of the research method employed.

ing stages: feature scaling, feature selection, data balancing, and machine learning algorithm selection. Within each stage, we explored various techniques, e.g., multiple algorithms for feature scaling. We systematically disabled individual stages to assess their impact on predictive performance. Furthermore, we examined the application of each technique within a specific stage, along with any combination of techniques from other stages. This approach resulted in the creation of a total

of 27 different pipelines, representing the cartesian product of all possible configurations. Finally, we statistically analyzed the performance of each experimented pipeline to identify the best one and provide a final answer to $\mathbf{RQ}_1$. As for individual training stages:

Feature Scaling. This stage could be required because the domains of the features under consideration varied widely from one another. When features have different scales, it can lead to issues during the training process, such as one feature dominating the others or the model converging slowly. Hence, we scaled the features experimenting with *min-max* and *z-score* scaling algorithms.

Feature Selection. We filtered out features with low predictive power or high correlation with other features to mitigate the risks of model overfitting. We experimented with three approaches. First, the `vif` (Variance Inflation Factors) function to discard highly correlated variables, putting a threshold value equal to 5 [37]. Second, we measured the information gain [46] of each feature, discarding those that did not provide any expected beneficial effect on the performance (info gain equals to 0), as done in previous work [2], [7]. Third, we applied Principal Component Analysis (PCA) [18] to identify the most informative features based on their variance in the data, consequently filtering out those with low information content.

Data Balancing. As flaky tests were underrepresented in the dataset (1.6%), we assessed the role played by data balancing. In this respect, we experimented with *Synthetic Minority Oversampling Technique*, a.k.a. SMOTE [8], an oversampling algorithm that generates synthetic instances of flaky tests based on the information available in the dataset. We also experimented with a *Random Undersampling* approach that explored the distribution of majority instances in a random fashion and under-samples them.

Machine Learning Algorithm. We assessed the capabilities of Random Forest [22] as algorithm. We relied on this algorithm since previous studies on flaky test prediction consistently identified Random Forest as the best algorithm in both dynamic and static contexts [2], [42], [44]. To implement it, we employed the *Scikit-Learn* library [26] in *Python*, which provides public APIs that let us configure, execute, and validate the above-mentioned classifier.

As for the validation technique, we adopted the *leave-one-out cross-validation* [52]. This approach iteratively designates one project as the test set, with the rest serving as the training set. Notably, preprocessing steps were exclusively applied to the training sets to prevent any bias in the evaluation of model performance and to ensure the integrity of the testing process.

We finally evaluate the performance metrics of each experimented pipeline, computing *precision*, *recall*, *F-Measure*, and *AUC-PR*. To determine the most effective pipeline in addressing $\mathbf{RQ}_1$, we employed the Nemenyi test [36] for statistical significance and present the results through mean on MCM (Multiple Comparison with the best) plots [25].

B. RQ_2 - RESEARCH METHOD

To address $\mathbf{RQ}_2$, we assess the impact of transfer learning techniques on the performance of the best pipeline resulting from $\mathbf{RQ}_1$. We evaluated two prominent approaches: the feature-based and the instance-based methods [19], [51]. The former aims to identify commonalities in features across different projects to create a unified representation, thereby mitigating distribution differences between the source and target domains. The latter addresses distribution disparities by recalibrating the weights of training instances, assigning greater significance to source domain data to adapt the model to the target domain. More specifically, we applied two different feature-based methods and two instance-based methods:

Burak Filter. Initially proposed by Turhan et al. [51], this instance-based method uses a filtering technique that selects training data based on its proximity to the test data, measured through Euclidean distance. We employed the Burak filter with $k = 10$, as this configuration showcased a potential decrease in the false alarm rate within the closely-related field of defect prediction.

Transfer Component Analysis (TCA). Originally designed to learn a shared feature subspace between the source and target domains, TCA [38] operates within a Reproducing Kernel Hilbert Space (RKHS). Its objective is to align data distributions across distinct domains while safeguarding critical feature information.

IG_SM_FS_TCA. Inspired by Khatri et al.'s research [24], this adaptation technique initiates by minimizing disparities in the feature sets across domains, before implementing TCA. Employing a multi-phase approach, it systematically identifies crucial features for optimal domain adaptation, ultimately striving to enhance the performance of machine learning models in cross-domain scenarios.

TrAdaBoost. Extending AdaBoost, this instance-based method [11] fine-tunes the weights of source domain instances during training, diminishing the impact of those least resembling the target domain data. Through iterative refinement, it prioritizes error reduction in instances from the target domain, thereby aligning the model more closely with the characteristics of the target domain. We applied TrAdaBoost techniques by splitting the dataset into 25%, 50%, and 75% segments to evaluate its effectiveness across different data proportions. This approach allows us to systematically analyze the performance in varying conditions, from minimal data availability to substantial data, providing insights into its robustness, efficiency, and maximum potential in aligning the source and target domains.

Similarly to $\mathbf{RQ}_1$, we evaluated the performance in terms of *precision*, *recall*, *F-Measure*, and *AUC-PR*. We also statistically compared the models trained using different transfer learning approaches using the Nemenyi test [36].

V. ANALYSIS OF THE RESULTS

This section illustrates the results achieved by our study.

A. RESULTS FOR RQ$_1$

In the context of **RQ**$_1$, we evaluated 27 different pipelines using Random Forest as machine learning algorithm.

Once we had collected the performance, we statistically evaluated our findings — Figure 2 reports the results of the Nemenyi Test. First, we can observe that we were able to collect data only for 25 of the 27 analyzed pipelines. Specifically, using SMOTE and PCA combined with SMOTE did not produce any result.

The best-performing pipeline (i.e., the one with the largest difference) is the one with the *PCA* as feature scaling and *Random Undersampling* as a balancing technique. However, this pipeline has statistically comparable performance with more than half of the other experimented configurations.

An interesting point to consider relates to the significance of data balancing techniques. All the best-performing pipelines use *Random Undersampling* as data balancer, clearly indicating that undersampling is the best choice for such kind of analysis.

Conversely, it seems that the choice of different feature scaling techniques does not significantly impact the performance (e.g., PCA appears both in the best- and worst-performing configurations).

Once we identified the best configuration, we experimented with it on our dataset to measure the performance achieved. Table 3 reports the results in terms of *precision, recall, AUC-PR*, and *F1 score*.

As we can observe, performance are overall poor, with an average *F1 score* of 0.03 and a max *F1 score* of 0.18 for `Wildfly`. The main reason for the low performance seems to be related to the high rate of *false positives*. Indeed, in almost all cases, the results for *Precision* are below 0.1.

The overall poor performance highlights the inherent challenges in cross-project flaky test prediction. Differences in development practices, codebases, and testing environments across projects contribute to the difficulty in generalizing a model trained on one project to another.

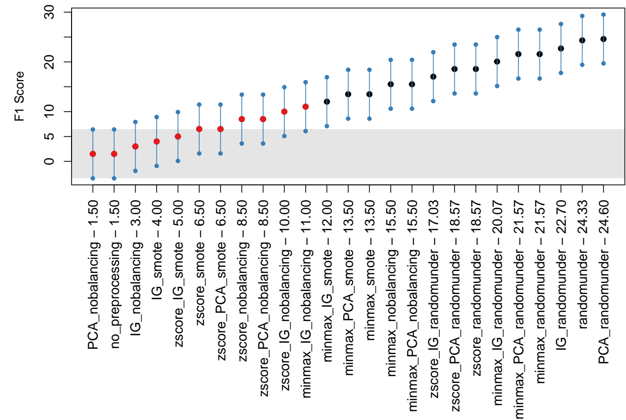


FIGURE 2: Overview of the Nemenyi Test employed on the pipeline evaluated in terms of F1 score. Circle dots are the median likelihood, while the error bars indicate the 95% confidence interval. 60% of likelihood means that a pipeline appears at the top-rank for 60% of the experiments.

TABLE 3: Performance of the best algorithm for **RQ**$_1$.

Project	Precision	Recall	AUC-PR	F1
Achilles	0.04	0.04	0.19	0.04
Activiti	0.05	0.54	0.13	0.1
Admiral	0.05	0.2	0.12	0.08
Adyen-java-api-library	0.07	0.02	0.21	0.03
Aem-core-wcm-components	0.0	0.0	0.07	NaN
Chronicle-Wire	0.09	0.06	0.10	0.07
Commons-lang	0.0	0.0	0.01	NaN
Data Flow Template	0.03	0.02	0.05	0.02
Druid	0.01	0.16	0.02	0.01
Dubbo	0.13	0.12	0.12	0.13
Fastjson	0.0	0.0	0.01	NaN
Flink	0.01	0.3	0.01	0.02
Graylog2-server	0.01	0.1	0.17	0.02
Hadoop	0.005	0.04	0.01	0.01
Http-request	0.0	0.0	0.17	NaN
Ignite-3	0.11	0.13	0.11	0.12
Jackrabbit-oak	0.0	0.0	0.10	NaN
Java WebSocket	0.0	0.0	0.16	NaN
Mockserver	0.08	0.04	0.02	0.01
Nacos	0.05	0.19	0.12	0.07
Nifi	0.02	0.16	0.02	0.04
Ormlite-core	0.0	0.0	0.07	NaN
Ozone	0.04	0.3	0.1	0.08
Servicecomb-java-chassis	0.0	0.0	0.01	NaN
Typescript-generator	0.0	0.0	0.41	NaN
Visualee	1.0	0.1	0.54	0.17
Vpc-java-sdk	0.004	0.04	0.06	0.01
Wildfly	0.11	0.46	0.1	0.18
Wro4j	0.03	0.08	0.1	0.04
Aggregate Results	0.02	0.1	0.1	0.03

B. RESULTS FOR RQ$_2$

The goal of **RQ**$_2$ was to evaluate the impact of feature- and instance-based filtering methods on the performance of flaky test predictors in a cross-project scenario. To this aim, we compared the performance of six different strategies, as described in Section IV. Figures 3 and 4 report the results of the statistical test and the boxplots comparing the techniques' performance respectively.

As we can observe from Figure 3, the best performance is provided by the approaches based on *AdaBoost*, which significantly outperforms all the other alternatives. Differently, the other feature- and instance-based filtering techniques do not provide any significant improvement to the traditional algorithm performance.

The boxplots in Figure 4 reinforce these observations, illustrating the high improvement in all performance metrics when *TrAdaBoost* is applied. The reason behind such an improvement is likely due to the intrinsic nature of *TrAdaBoost*. Indeed, the technique effectively manages the heterogeneity of cross-project data by iteratively adjusting the weights of the training instances, decreasing the influence of less relevant source domain instances and increasing the influence of more relevant target domain instances. This emphasis on target-relevant data allows TrAdaBoost to better align the model with the specific characteristics of the target project, thus improving prediction accuracy and reducing false positives.

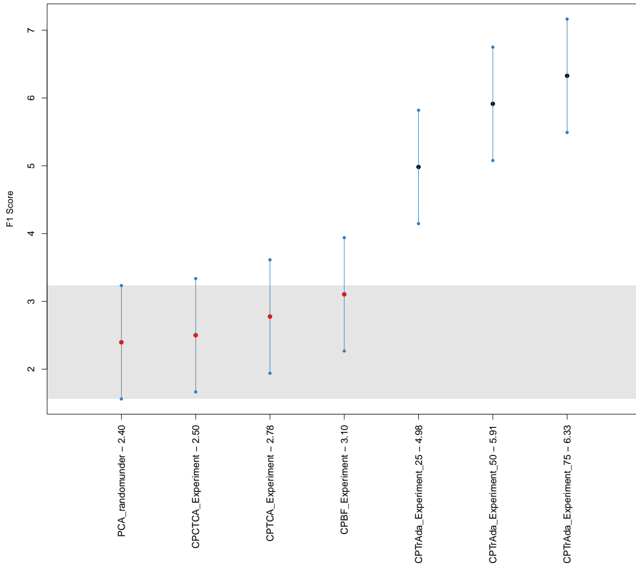


FIGURE 3: Overview of the Nemenyi Test employed for $\mathbf{RQ}_2$ in terms of F1 score. Circle dots are the median likelihood, while the error bars indicate the 95% confidence interval. 60% of likelihood means that a pipeline appears at the top-rank for 60% of the experiments.

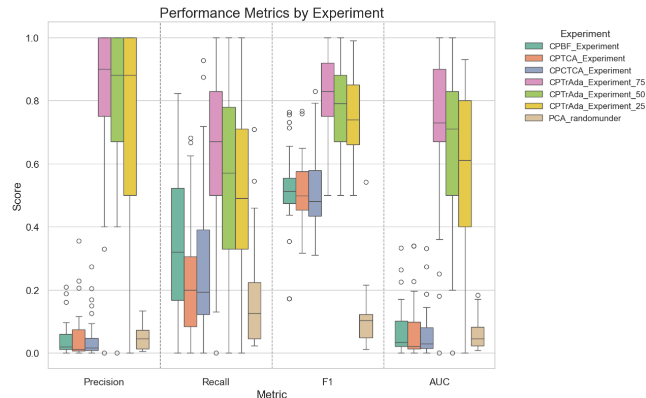


FIGURE 4: Overview of the performance metrics employed for $\mathbf{RQ}_2$ and compared with the best pipeline for $\mathbf{RQ}_1$.

Consequently, this iterative refinement process ensures that the model is robust and better generalized to the target domain, making it particularly suitable for the cross-project prediction of flaky tests. In other terms, instance-based fine-tuning represents the real advantage that allows to improve the classification of flaky tests significantly.

Table 4 reports the detailed performance of the best configuration (i.e., *TrAdaBoost_75*) on all systems in our dataset —the detailed performance of all techniques are in the online appendix [1]. Overall, the usage of such an instance-based filtering approach led to a huge performance enhancement, with the aggregate F1 score increasing from 0.03 to 0.7, registering a boost of +2300%.

An interesting consideration to make concerns the way the instance-based technique mitigates the main limitation of the standard approach (i.e., the high number of false positives). Indeed, in more than 70% of the analyzed systems (21 out of 29) we can observe a *Precision* higher than 0.80. Additionally, in 11 of these cases the experimented approach achieves a perfect precision (i.e., 1.0).

TABLE 4: Performance of the best algorithm for $\mathbf{RQ}_2$.

Project	Precision	Recall	AUC-PR	F1
Achilles	1.0	0.83	0.92	0.91
Activiti	1.0	1.0	1.0	1.0
Admiral	0.86	0.43	0.71	0.57
Adyen-java-api-library	1.0	0.73	0.86	0.84
Aem-core-wcm-components	1.0	0.83	0.92	0.91
Chronicle-Wire	0.0	0.0	0.5	0.0
Commons-lang	1.0	0.5	0.75	0.67
Data Flow Template	0.89	0.8	0.9	0.84
Druid	0.33	0.13	0.56	0.18
Dubbo	0.9	0.54	0.77	0.68
Fastjson	0.5	0.33	0.66	0.4
Flink	0.6	0.5	0.75	0.55
Graylog2-server	1.0	0.6	0.8	0.75
Hadoop	0.93	0.57	0.78	0.7
Http-request	1.0	0.71	0.86	0.83
Ignite-3	0.87	0.87	0.93	0.87
Jackrabbit-oak	0.67	0.8	0.9	0.73
Java WebSocket	0.88	1.0	0.99	0.93
Mockserver	1.0	0.71	0.86	0.83
Nacos	0.8	0.57	0.78	0.67
Nifi	0.95	0.57	0.79	0.71
Ormlite-core	1.0	0.83	0.91	0.9
Ozone	0.5	0.17	0.58	0.25
Servicecomb-java-chassis	1.0	0.9	0.95	0.95
Typescript-generator	0.89	1.0	0.98	0.94
Visualee	0.92	1.0	0.98	0.96
Vpc-java-sdk	1.0	0.5	0.75	0.67
Wildfly	0.75	0.67	0.83	0.71
Wro4j	0.4	0.33	0.66	0.36
Aggregate Results	0.81	0.63	0.81	0.70

VI. THREATS TO VALIDITY

This section discusses the potential limitations of our work and how we mitigated them.

A. CONSTRUCT VALIDITY

Our study is primarily threatened by potential imprecision in the data, derived from public datasets used in previous research [2], [28]. Although these datasets have been validated, some flaky tests might not have exposed their unreliability during the data collection procedures conducted by the authors of the dataset. As such, replications using different datasets could strengthen our findings. The automated tools used to compute independent variables might introduce noise, e.g., false positive test and code smells. However, manual detection was unfeasible in our case, so we used well-established tools known for their accuracy. Additionally, we linked test classes to production classes using pattern matching based on naming conventions. Although this approach balances accuracy and scalability, it may produce false positives, particularly in systems with identical class names but different paths. No such cases were present in our study, but future replications should consider this potential issue.

B. CONCLUSION VALIDITY

In terms of conclusion validity, we assessed the suitability of cross-project flakiness prediction by performing a large-scale ablation study, hence experimenting with multiple techniques and training stages. Afterwards, we engaged with a number of instance- and feature-based transfer learning methods with the aim of broadening the scope of our study. In both research questions, our analyses were supported by statistical tests. The process conducted makes us confident about the reliability of the conclusions drawn. However, more extensive analyses might still be beneficial to corroborate our findings.

C. EXTERNAL VALIDITY

Our study's external validity may be limited to the Java open-source projects belonging to the IDoFT dataset. Despite the variety of projects, additional datasets may reveal different patterns and, therefore, different results. While our approach is likely applicable to other object-oriented languages, its effectiveness with procedural languages requires further investigation. In addition, the practical adoption of our findings may be limited by the need for naming conventions in the linking process. However, our methodology can be adapted to project-specific standards, potentially improving model performance with larger, more tailored datasets.

VII. CONCLUSIONS

In this paper, we explored the challenges and potential of cross-project flaky test prediction. By leveraging the IDoFT dataset, we evaluated a static flaky test prediction model and investigated various filtering methods to improve its efficacy.

Our study confirms that while cross-project flaky test prediction is challenging due to the heterogeneity of the data, we demonstrated that feature- and instance-based filtering methods, especially the *TrAdaBoost* approach, significantly enhance prediction performance, achieving an average F1 score of 70%. This highlights the potential for these methods to be applied in practical scenarios, providing a robust framework for identifying flaky tests across diverse projects.

To sum up, our paper provided the following contributions:

1) A large-scale empirical investigation on cross-project flaky test prediction using static metrics;
2) Evidence and analysis of the improvement provided by feature- and instance-based filtering methods, especially *TrAdaBoost*;
3) An online appendix [1] in which we provide all material and scripts employed to address the goals of the study.

Further research is needed to refine these methods and explore their applicability to other programming languages and development environments. We also plan to further investigate machine learning models for flaky test prediction based on their root causes, aiming to improve the predictions. Nonetheless, our findings contribute valuable insights into the feasibility and effectiveness of cross-project flaky test prediction, paving the way for more reliable and maintainable software testing practices.

ACKNOWLEDGEMENT

Valeria is partially supported by the FWO SBO BaseCamp Zero project (Code: S000323N).

REFERENCES

[1] Afeltra, A., Cannavale, A., Pecorelli, F., Pontillo, V., Palomba, F.: A large-scale empirical investigation into cross-project flaky test prediction (2024), https://github.com/alfcan/crossproject-flaky-test-prediction.git

[2] Alshammari, A., Morris, C., Hilton, M., Bell, J.: Flakeflagger: Predicting flakiness without rerunning tests. In: ICSE 2021. pp. 1572–1584. IEEE (2021)

[3] Barboni, M., Bertolino, A., De Angelis, G.: What we talk about when we talk about software test flakiness. In: International Conference on the Quality of Information and Communications Technology. pp. 29–39. Springer (2021)

[4] Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., Marinov, D.: Deflaker: Automatically detecting flaky tests. In: ICSE 2018. pp. 433–444. IEEE (2018)

[5] Bertolino, A., Cruciani, E., Miranda, B., Verdecchia, R.: Know your neighbor: Fast static prediction of test flakiness

[6] Camara, B., Silva, M., Endo, A., Vergilio, S.: On the use of test smells for prediction of flaky tests. In: Brazilian Symposium on Systematic and Automated Software Testing. pp. 46–54 (2021)

[7] Catolino, G., Palomba, F., Zaidman, A., Ferrucci, F.: How the experience of development teams relates to assertion density of test classes. In: ICSME 2019. pp. 223–234. IEEE (2019)

[8] Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: synthetic minority over-sampling technique. Journal of artificial intelligence research **16**, 321–357 (2002)

[9] Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. IEEE TSE **20**(6), 476–493 (1994).

[10] Cordy, M., Rwemalika, R., Franci, A., Papadakis, M., Harman, M.: Flakime: laboratory-controlled test flakiness impact assessment. In: Proceedings of the 44th International Conference on Software Engineering. pp. 982–994 (2022)

[11] Dai, W., Yang, Q., Xue, G.R., Yu, Y.: Boosting for transfer learning. In: Proceedings of the 24th international conference on Machine learning. pp. 193–200 (2007)

[12] Daniel, B., Jagannath, V., Dig, D., Marinov, D.: Reassert: Suggesting repairs for broken unit tests. In: ASE 2009. pp. 433–444. IEEE (2009)

[13] Eck, M., Palomba, F., Castelluccio, M., Bacchelli, A.: Understanding flaky tests: The developer's perspective. In: ESEC/FSE 2019. pp. 830–840 (2019)

[14] Fowler, M.: Eradicating non-determinism in tests. Martin Fowler Personal Blog (2011), https://martinfowler.com/articles/nonDeterminism.html

[15] Garousi, V., Felderer, M., Kılıçaslan, F.N.: A survey on software testability. Information and Software Technology **108**, 35–64 (2019)

[16] Grano, G., De Iaco, C., Palomba, F., Gall, H.: Pizza versus pinsa: On the perception and measurability of unit test code quality. In: ICSME 2020. pp. 336–347. IEEE (2020)

[17] Grano, G., Palomba, F., Gall, H.: Lightweight assessment of test-case effectiveness using source-code-quality indicators. IEEE TSE (2019)

[18] Greenacre, M., Groenen, P.J., Hastie, T., d'Enza, A.I., Markos, A., Tuzhilina, E.: Principal component analysis. Nature Reviews Methods Primers **2**(1), 100 (2022)

[19] Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. Journal of machine learning research **3**(Mar), 1157–1182 (2003)

[20] Habchi, S., Haben, G., Papadakis, M., Cordy, M., Traon, Y.L.: A qualitative study on the sources, impacts, and mitigation strategies of flaky tests. arXiv preprint arXiv:2112.04919 (2021)

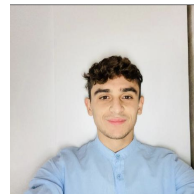
[21] Haben, G., Habchi, S., Papadakis, M., Cordy, M., Le Traon, Y.: A replication study on the usability of code vocabulary in predicting flaky tests. In: MSR 2021 (2021)

[22] Ho, T.K.: Random decision forests. In: Proceedings of 3rd international conference on document analysis and recognition. vol. 1, pp. 278–282. IEEE (1995)

[23] Hosseini, S., Turhan, B.: A comparison of similarity based instance selection methods for cross project defect prediction. In: Proceedings of the 36th annual ACM symposium on applied computing. pp. 1455–1464 (2021)

[24] Khatri, Y., Singh, S.K.: An effective feature selection based cross-project defect prediction model for software quality improvement. International Journal of System Assurance Engineering and Management pp. 1–19 (2023)
[25] Koning, A.J., Franses, P.H., Hibon, M., Stekler, H.O.: The m3 competition: Statistical tests of the results. International Journal of Forecasting **21**(3), 397–409 (2005)
[26] Kramer, O.: Scikit-learn. In: Machine learning for evolution strategies, pp. 45–53. Springer (2016)
[27] Lacoste, F.: Killing the gatekeeper: Introducing a continuous integration system. In: 2009 agile conference. pp. 387–392. IEEE (2009)
[28] Lam, W., Oei, R., Shi, A., Marinov, D., Xie, T.: idflakies: A framework for detecting and partially classifying flaky tests. In: ICST 2019. pp. 312–322. IEEE (2019)
[29] Lam, W., Winter, S., Wei, A., Xie, T., Marinov, D., Bell, J.: A large-scale longitudinal study of flaky tests. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–29 (2020)
[30] Lam, W., Winter, S., Astorga, A., Stodden, V., Marinov, D.: Understanding reproducibility and characteristics of flaky tests through test reruns in java projects. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE). pp. 403–413. IEEE (2020)
[31] Lampel, J., Just, S., Apel, S., Zeller, A.: When life gives you oranges: detecting and diagnosing intermittent job failures at mozilla. In: 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1381–1392 (2021)
[32] Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: ESEC/FSE 2014. pp. 643–653 (2014)
[33] McCabe, T.: A complexity measure. IEEE TSE **SE-2**(4), 308–320 (1976).
[34] Memon, A., Cohen, M.: Automated testing of gui applications: models, tools, and controlling flakiness. In: ICSE 2013. pp. 1479–1480. IEEE (2013)
[35] Micco, J.: The state of continuous integration testing@ google. In: ICST (2017)
[36] Nemenyi, P.B.: Distribution-free multiple comparisons. Princeton University (1963)
[37] O'brien, R.: A caution regarding rules of thumb for variance inflation factors. Quality & quantity **41**(5), 673–690 (2007)
[38] Pan, S.J., Tsang, I.W., Kwok, J.T., Yang, Q.: Domain adaptation via transfer component analysis. IEEE transactions on neural networks **22**(2), 199–210 (2010)
[39] Parry, O., Kapfhammer, G.M., Hilton, M., McMinn, P.: A survey of flaky tests. ACM Transactions on Software Engineering and Methodology (TOSEM) **31**(1), 1–74 (2021)
[40] Pecorelli, F., Palomba, F., De Lucia, A.: The relation of test-related factors to software quality: A case study on apache systems. Empirical Software Engineering **26**(2) (2021)
[41] Pezzè, M., Young, M.: Software testing and analysis: process, principles, and techniques. John Wiley & Sons (2008)
[42] Pinto, G., Miranda, B., Dissanayake, S., D'Amorim, M., Treude, C., Bertolino, A.: What is the vocabulary of flaky tests? In: MSR 2020. pp. 492–502 (2020)
[43] Pontillo, V., Palomba, F., Ferrucci, F.: Toward static test flakiness prediction: a feasibility study. In: Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution. pp. 19–24 (2021)
[44] Pontillo, V., Palomba, F., Ferrucci, F.: Static test flakiness prediction: How far can we go? Empirical Software Engineering **27**(7), 187 (2022)
[45] Pontillo, V., Palomba, F., Ferrucci, F.: Test code flakiness in mobile apps: The developer's perspective. Information and Software Technology **168**, 107394 (2024)
[46] Quinlan, J.R.: Induction of decision trees. Machine learning **1**(1), 81–106 (1986)
[47] Rahman, S., Massey, A., Lam, W., Shi, A., Bell, J.: Automatically reproducing timing-dependent flaky-test failures. In: International Conference on Software Testing, Verification, and Validation (2024)
[48] Rahman, S., Shi, A.: Flakesync: Automatically repairing async flaky tests. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 1–12 (2024)
[49] Rehman, M.H.U., Rigby, P.C.: Quantifying no-fault-found test failures to prioritize inspection of flaky tests at ericsson. In: 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1371–1380 (2021)
[50] Terragni, V., Salza, P., Ferrucci, F.: A container-based infrastructure for fuzzy-driven root causing of flaky tests. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). pp. 69–72. IEEE (2020)
[51] Turhan, B., Menzies, T., Bener, A.B., Di Stefano, J.: On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering **14**, 540–578 (2009)
[52] Vehtari, A., Gelman, A., Gabry, J.: Practical bayesian model evaluation using leave-one-out cross-validation and waic. Statistics and computing **27**(5), 1413–1432 (2017)
[53] Zhang, S., Jalali, D., Wuttke, J., Muşlu, K., Lam, W., Ernst, M., Notkin, D.: Empirically revisiting the test independence assumption. In: ISSTA 2014. pp. 385–396 (2014)
[54] Zheng, W., Liu, G., Zhang, M., Chen, X., Zhao, W.: Research progress of flaky tests. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 639–646. IEEE (2021)
[55] Zhu, X., Qiu, T., Wang, J., Lai, X.: A novel instance-based method for cross-project just-in-time defect prediction. Software: Practice and Experience **54**(6), 1087–1117 (2024)

ANGELO AFELTRA received the M.Sc. degree at the University of Salerno, Italy, in 2023. His interests include software maintenance and evolution, AI & ML engineering, and robotics. He currently works as a consultant at Cluster Reply.



ALFONSO CANNAVALE received the M.Sc. degree at the University of Salerno, Italy, in 2024. His interests include software maintenance and evolution, empirical software engineering, and AI & ML engineering.

FABIANO PECORELLI is an Associate Professor at the Pegaso Telematic University. Formerly, he was a postdoctoral researcher at the University of Salerno (Italy), Eindhoven University of Technology (the Netherlands), and Tampere University (Finland). He received his Ph.D. in Computer Science in 2022 from the University of Salerno, Italy. He has more than 30 publications in international journals and conferences. His research interests include software maintenance and evolution, empirical software engineering, AI & ML engineering, and quantum software engineering. He serves, and has served, as a referee for various international journals in the field of software engineering (e.g., IEEE Transactions on Software Engineering, ACM Transactions on Software Engineering and Methodology, Empirical Software Engineering).

VALERIA PONTILLO is a post-doctoral researcher at the Software Languages Lab of the Vrije Universiteit Brussel. She received her Ph.D in Computer Science in 2024 from the University of Salerno, Italy. Her research activities are in Software Engineering, particularly Software Testing, Software Quality, and Software Maintenance and Evolution. Her research interests also include the development of novel software engineering for artificial intelligence tools and techniques. She served as a reviewer for international conferences (e.g., ICSE 2024 Artifact Evaluation, SANER 2024/2025, CHASE 2025, SCAM NIER, ICSME NIER, and ASE NIER) and journals in the software engineering field (e.g., IEEE Transactions on Software Engineering, ACM Transactions on Software Engineering and Methodology, Empirical Software Engineering).

FABIO PALOMBA is an Assistant Professor at the Software Engineering Lab, University of Salerno. He earned his European Ph.D. in Management & Information Technology in 2017 and received the 2017 IEEE Computer Society Best Ph.D. Thesis Award. In 2023, he was honored with the IEEE Computer Society Technical Council of Software Engineering Rising Star Award for his contributions to code refactoring and code smells.

His research focuses on software maintenance and evolution, empirical software engineering, source code quality, and mining software repositories. He has won several prestigious paper awards, including two ACM/SIGSOFT and one IEEE/TCSE Distinguished Paper Awards and Best Paper Awards at CSCW'18 and SANER'18. In 2019, he received the SNSF Ambizione grant.

He served on the ICPC steering Committee and has held various chair positions, including program co-chair of SANER 2024 and ICPC 2021. He is an Editorial Board Member of several journals, including IST, EMSE, EISEJ, IEEE Transactions on Software Engineering, TOSEM, JSS, and SCICO. He has received multiple awards for his reviewing activities.

• • •