Local and Global Explainability for Technical Debt Identification

Dimitrios Tsoukalas, Nikolaos Mittas, Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Dionysios Kechagias

Abstract—In recent years, we have witnessed an important increase in research focusing on how machine learning (ML) techniques can be used for software quality assessment and improvement. However, the derived methodologies and tools lack transparency, due to the black-box nature of the employed machine learning models, leading to decreased trust in their results. To address this shortcoming, in this paper we extend the state-of-the-art and -practice by building explainable AI models on top of machine learning ones, to interpret the factors (i.e. software metrics) that constitute a module as in risk of having high technical debt (HIGH TD), to obtain thresholds for metric scores that are alerting for poor maintainability, and finally, we dig further to achieve local interpretation that explains the specific problems of each module, pinpointing to specific opportunities for improvement during TD management. To achieve this goal, we have developed project-specific classifiers (characterizing modules as HIGH and NOT-HIGH TD) for 21 open-source projects, and we explain their rationale using the SHapley Additive exPlanation (SHAP) analysis. Based on our analysis, complexity, comments ratio, cohesion, nesting of control flow statements, coupling, refactoring activity, and code churn are the most important reasons for characterizing classes as in HIGH TD risk. The analysis is complemented with global and local means of interpretation, such as metric thresholds and case-by-case reasoning for characterizing a class as in-risk of having HIGH TD. The results of the study are compared against the state-of-the-art and are interpreted from the point of view of both researchers and practitioners.

Index Terms—technical debt; technical debt identification; software quality; software metrics; explainable AI; SHAP

1 Introduction

Technical Debt (TD) identification[1] is considered as the first step of effective TD management and prioritization, in the sense that the complete technical debt of a system cannot be repaid [16]. TD is usually measured and identified with static analysis tools such as SonarQube, CAST Software, etc. [7]. Nevertheless, the use of such a tool leads to a numeric assessment of TD Principal that is questionable [9] (***challenge-1***); depends on the used tool's rationale, in the sense that different tools tend to lead to diverse TD quantification results [2] (***challenge-2***); and does not characterize if the specific measurement shall be perceived as a HIGH TD score (***challenge-3***).

To confront these challenges, in a series of previous works, we relied on a variety of statistical and *Machine Learning* (ML)-driven approaches. As a first step, we developed a "*commonly agreed TD knowledge base*" [2], i.e., an empirical benchmark of classes that exhibit high levels of TD (these classes are from now on termed as *"HIGH TD"* classes). The identification of HIGH TD classes has been performed based on archetypal analysis, pointing to classes for which three widely adopted TD assessment tools (namely SonarQube [12], CAST [15], and Squore [8]) converge, and indicate them as classes with a high chance of containing high levels of TD. Next, to decouple the application of the method from the need of retaining licenses and installations of all three tools, we have evaluated the ability of ML algorithms to classify software classes as HIGH TD and NOT-HIGH TD [40] [41]. As model features, we considered a wide range of software metrics spanning from code to process metrics. The findings revealed that a subset of superior classifiers (e.g., Random Forest) can identify HIGH TD classes with a sufficient accuracy and reasonable effort, achieving an F_2-measure of approximately 0.79 with an associated Class Inspection ratio of approximately 0.10.

Building on top of the benefits derived from the obtained TD identification ML models, in this work, we proceed one step further and apply *eXplainable AI* (XAI) techniques to shed light on the insights of the model. Such insights are expected to bring important benefits to ***quality assurance practice***, since: (a) explainability of the recommendations provided by automated tools can contribute to informed decision-making and data-driven discussions among technical stakeholders, improving trustworthiness and transparency; and (b) point to opportunities for improvement in the sense that a recommendation of a class as HIGH TD comes along with the reasons that render this class as problematic (*local interpretation*),

- *Dimitrios Tsoukalas is with the Information Technologies Institute, Centre for Research and Technology Hellas, Greece. E-mail: tsoukj@iti.gr*
- *Nikolaos Mittas is with the Hephaestus Laboratory, Department of Chemistry, School of Science, Democritus University of Thrace, Kavala, Greece. E-mail: nmittas@chem.duth.gr*
- *Elvira-Maria Arvanitou is with the Department of Applied Informatics, University of Macedonia, Greece. E-mail: earvanitoy@gmail.com*
- *Apostolos Ampatzoglou is with the Department of Applied Informatics, University of Macedonia, Greece. E-mail: a.ampatzoglou@uom.edu.gr*
- *Alexander Chatzigeorgiou is with the Department of Applied Informatics, University of Macedonia, Greece. E-mail: achat@uom.edu.gr*
- *Dionysios Kehagias is with the Information Technologies Institute, Centre for Research and Technology Hellas, Greece. E-mail: diok@iti.gr*

[1] TD Identification is the practice of understanding which modules of a software suffer from high levels of technical debt [25]

thereby fostering a culture of writing high quality code. For example, for a particular class, XAI could reveal that it is the excessive value of coupling that renders a class as HIGH TD. On the other hand, in terms of ***researchers***, the proposed analysis can provide synthesized knowledge (*global interpretation*) on the importance of certain metrics related to TD accumulation, contributing towards the body of knowledge on the root causes of Technical Debt. For instance, XAI could highlight metric thresholds beyond which a class or system would be classified as problematic, thereby addressing the challenging problem of domain-specific threshold extraction. Given the above, we plan to answer the following research questions:

[**RQ$_1$**] What are the most important metrics that can be used for TD identification (global explanation)?

[**RQ$_2$**] What are the thresholds that when surpassed a class has higher chances of being considered as HIGH TD (global explanation)?

[**RQ$_3$**] How can the analysis pinpoint specific opportunities for improvement (local explanation)?

We have preferred to build this work on top of a ML approach for TD identification [41], since: (a) it relies on three TD analysis tools—whereas most other approaches rely on individual tools (usually SonarQube); and (b) to the best of our knowledge it is the only approach that performs TD identification in a fully automated manner to enable a large-scale case study. To achieve this goal, we construct accurate project-specific classifiers for 21 software projects and exploit the *SHapley Additive exPlanation* (SHAP) analysis (for explainability) to extract feature importance ranks and interpret the effect that various software metrics (i.e., features in terms of a prediction model) have on classifying a software class as HIGH TD. Subsequently, given a list of ranked metrics per project, we investigate whether the most important ones (as extracted by SHAP analysis) overlap among projects. Moreover, through the metrics' global interpretation that SHAP analysis inherently supports, we extract thresholds (heuristic values) that may act as practical TD prevention guidelines (or rules of thumb) for developers. Finally, using local SHAP interpretation, we demonstrate how practitioners should deal with specific HIGH TD classes to reduce the levels of TD.

The rest of the paper is organized as follows: in Section 2, we present related work. Next, in Section 3 we present in detail the employed methodology for data collection and analysis. The experimental results are presented in Section 4 and discussed in Section 5. The study is wrapped up by reporting threats to validity (Section 6) and highlighting the important conclusions (Section 7).

2 Related Work

In this section, we present studies that are necessary for understanding the context of this study: We discuss the state-of-the-art on technical debt identification (related to the ***context*** of this work), a sample of studies[2] that attempt to identify metric thresholds (related to ***RQ$_2$***), and studies that apply AI / ML / DL for design-time software quality assessment (related to ***RQ$_1$*** and ***RQ$_3$***). Finally, we present studies that have applied XAI in software engineering (related to ***methodology***).

[2] We present only a few indicative studies in Sections 2.2 and 2.3, since the number of papers in these areas is enormous.

2.1 Technical Debt Identification

Alves et al. [1] performed a systematic mapping study for TD identification. The goal of this study was to identify: (a) the types of TD; (b) the strategies that can be used for TD identification; and (c) the TD management approaches. Regarding TD identification, Alves et al. [1] recorded the artifacts, the data sources, and the visualizations that have been proposed in the literature. The authors ended up exploring 100 studies. The results suggested that there are 16 most studied different types of TD (such as code, design, architecture, and defect) and various TD indicators (e.g., code smells, documentation issues) for each TD type. The validation of the TD identification approaches is most usually performed through case studies and controlled experiments. The most used artifact for TD analysis is source code followed by documentation; with respect to data sources, configuration management systems are the most common source of information when identifying TD. Finally, visualization seems to lag in this area of research, since only 6 primary studies employ visualization methods.

2.2 Quality Metric Thresholds

Mishra et al. [30] conducted a systematic mapping study for cataloguing the software product metrics' threshold that have been proposed in the literature. To achieve this, they catalogued: (a) the techniques for calculating metrics' thresholds—e.g., based on programmer experience or using statistical methods; and (b) the quality attributes and metrics for which thresholds have been studied. The search and filtering process concluded with 45 studies. Most of these studies apply statistical methods to derive thresholds for object-oriented metrics, through empirical analysis. Additionally, 16 studies focused on fault detection followed by design problems detection (10 studies). Regarding quality metrics, the Chidamber and Kemerer (CK) metric suite [13] is the most studied one.

Ferreira et al. [18] identified threshold values for six object-oriented software metrics. The authors selected 40 open-source software systems that were developed in Java and were of varying size (18 to 3500 classes) and application domains (11 distinct ones). For each quality metric, the authors proposed three ranges of reference values: good—refers to the most common values of the metric; regular—refers to an intermediate range of values with low frequency, but not irrelevant; and bad—refers to values with quite rare occurrences. For validation, the authors performed two experiments to: (a) explore if the proposed threshold values can help to identify classes with design problems, and (b) assess whether the thresholds can support identifying well-designed classes. The results suggest that the scores of five metrics (all except depth of inheritance tree) *can set useful thresholds for design evaluation.*

Boucher and Badri [11] performed an empirical study to identify metrics' thresholds that are useful for predicting fault-proneness. The main aim of that study was to investigate thresholds calculation techniques relying on CK metrics as predictors. To achieve this goal, the authors analyzed 12 datasets from eight software systems, and compared the performance of four ML and two clustering-based models. The results suggested that ROC Curves is the best performing technique among the examined ones. Shatnawi et al. [35] conducted an empirical study to provide a method that uses ROC Curves to identify project-specific metric thresholds. The examined metrics included the CK metric suite [13], Li metrics [25], and Lorenz and Kidd metrics [27]. The metrics were calculated for three projects, and the thresholds were related to fault proneness. The authors identified threshold scores for coupling, complexity, and size metrics that can be used to identify high-risk error-prone classes.

Finally, Beranic and Hericko [10] performed an empirical study to compare threshold values for nine software metrics, among four object-oriented programming languages. For each programming language, 100 software projects were analyzed. The results suggested that threshold values for the same software metric vary among different programming languages.

2.3 AI / ML / DL and Software Design Quality

Change Proneness Prediction: Kaur and Mishra [24] performed an experimental analysis to compare the efficiency of cognitive complexity (CogC) as a change-proneness predictor, against two complexity and six CK metrics. The analysis was made on multiple versions of JFreeChart and Heritrix. One statistical analysis and five ML techniques are used to build models with the motivation to draw inferences regarding the importance of the CogC metric as a change-proneness predictor. The results suggest that CogC could be an individual quantifier of version-to-version change-proneness of Java files.

Quality Classifiers: Herbold et al. [21] defined a data-driven methodology to classify classes as of good or bad quality. The authors analyzed 11 size, coupling, complexity, and inheritance metrics and proposed an algorithm called rectangle learning. To evaluate the approach, the authors used eight systems written in C, C++, C#, or Java. The results suggested that the methodology can improve the efficiency of existing metric sets.

Bad Smell Detection: Yang et al. [44] proposed a classification model that applies ML to identify code clones; and developed a web-based proof-of-concept system. For evaluation purposes, they performed an online survey with 32 participants. The results suggest that their classification model showed more than 70% accuracy on average and more than 90% accuracy for specific users and projects. Fontana et al. [19] compared 16 ML algorithms and 74 software systems to detect four code smell types (i.e., Data Class, Large Class, Feature Envy and Long Method). The authors selected 43 size, complexity, cohesion, coupling, encapsulation, and inheritance metrics as independent variables. The results suggest that J48 and Random Forest obtain the best performance.

Identifying Practices and Patterns: Zanoni et al. [45] developed MARPLE-DPD that uses ML for detecting design patterns. The methodology can identify five design patterns using nine ML algorithms. The testbed consisted of pattern instances extracted from 10 open-source projects (2,794 instances). The results suggest that the detection is successful for all patterns, except for Composite. Mirakhorli and Cleland-Huang [29] used ML approaches for detecting, tracing, and monitoring architectural tactics in code. Specifically, the authors used six ML algorithms to train classifiers for detecting the presence of architectural tactics in source code. The approach visualizes the architectural tactics in code by mapping those relevant code segments into Tactic Traceability Patterns and notifies the practitioners when those segments are modified. The training was performed on 50 open-source projects, and the results suggest that six classifiers performed equivalently in tactical detection tasks.

2.4 XAI in Software Engineering

Code Smells: Huang et al. [22] focused on code smell prioritization, using the SHAP approach. After analyzing developers' comments on the criticalities of code smells, the study assessed whether XAI explanations covering the top important model features could address developers' major concerns. Initial results revealed a noticeable gap between XAI explanations and developers' expectations in code smell prioritization. However, by employing feature selection adapted to developers' feedback, explanations could cover more than 70% of developers' concerns. Specifically, for simpler code smells, a basic explanation involving the inspection of a few top metrics (e.g., top-3 or top-5) sufficed. However, for more complex smells, human expertise was still required. Cruz et al. [14] used ML to detect bad smells and XAI (SHAP) to interpret models' decisions. The authors evaluated seven classifiers with various parameter settings for detecting four types of bad smells (applied on 20 systems). Random Forest and Gradient Boosting Machine demonstrated strong performance in identifying the "*God Class*" and "*Refused Parent Bequest*" bad smells. The authors employed SHAP to interpret the models' predictions and highlight the metrics that were most influential in detecting each smell.

Software Vulnerabilities: Sotgiu et al. [37] focused on employing XAI for software vulnerability discovery, using SHAP to analyze decisions made by a fine-tuned Transformer-based model. The authors performed the analysis on both a global and a local basis. Globally, the study revealed that the model often assigns importance to features (i.e., tokens) that are programming language-specific, raising questions about its effectiveness in identifying vulnerabilities. On the other hand, local analysis revealed how specific features contribute to individual meaningful decisions, aiding analysts in understanding the model decisions behind misclassified and correctly classified cases.

Defect Prediction: Rajbahadur et al. [32] evaluated XAI feature importance methods in the context of software defect prediction, aiming to determine the level of agreement between the rankings produced by different methods. The authors conducted an analysis on 18 commonly used software defect datasets using various classifiers. They found out that: (a) feature importance ranks obtained from classifier-agnostic (e.g., SHAP) and classifier-specific (e.g., Gini) methods do not always strongly agree with each other; (b) classifier-agnostic methods tend to exhibit a strong agreement for a given dataset, including the features ranked at the top positions; and (c) classifier-specific methods yield significantly different feature importance ranks even on the same dataset. Jiarpakdee et al. [23] performed an empirical study on defect prediction, focusing on XAI model-agnostic techniques for explaining predictions made by defect models. Specifically, the authors evaluated three model-agnostic XAI techniques (LIME, BreakDown, and their improved LIME version with Hyper Parameter Optimisation) on 32 publicly available defect datasets from open-source software systems. Their findings indicate that (a) local explanations generated by model-agnostic techniques are mostly overlapping with the global explanation of defect models; and (b) model-agnostic techniques are perceived by practitioners as necessary and useful to understand the predictions of defect models.

3 Methodology

3.1 Required Background

This section briefly presents the methodology that was followed within the context of our previous work to build the classification model that is responsible for identifying HIGH TD software classes (i.e., the black-box models on top of which we applied the proposed XAI approaches).

The dataset used in the current study for experimental and inferential purposes relies on an empirical benchmark that was constructed in the study by Amanatidis et al. [2]. The main objectives of that study were: (a) the investigation of the degree of agreement (or diversity) among three leading TD assessment tools (i.e., SonarQube, CAST, and Squore); and (b) the identification of profiles of classes/files sharing similar levels of TD (e.g., characterized as HIGH TD by all employed tools). The proposed multivariate statistical framework resulted into the discrimination of a set of 18,857 classes from 25 Java projects into classes belonging to either the HIGH TD (N = 1,283) or the NOT-HIGH TD profile (N = 17,574).

Subsequently, this dataset was reused by Tsoukalas et al. [40], as the basis for the evaluation of the discriminative power (i.e., classification based on the benchmark [2]) of seven well-established statistical and ML classifiers given a set of 18 features encompassing various code-related metrics (such as structural properties and size) and metrics that capture aspects of the development process (such as code churn, commits and contributors count). Regarding the set of metrics used as input features into the model building phase, a collection of well-known open-source tools was employed, namely: PyDriller [38], CKJM [3], PMD Copy/Paste Detector[3], and cloc[4].

[3] https://pmd.github.io/latest/pmd_userdocs_cpd.html
[4] https://github.com/AlDanial/cloc#quick-start

The final dataset of code and repository activity metrics along with the dichotomous response variable indicating, whether a class is characterized as a HIGH or NOT-HIGH TD artifact was subjected to necessary pre-processing and data analytics tasks. The pre-processing step includes missing values handling, outlier detection, feature selection, whereas a well-known oversampling technique, namely the *Synthetic Minority Oversampling Technique* (SMOTE) was adopted for mitigating the serious side effects of the class imbalance problem that deteriorates the prediction abilities of classification learners. The results suggested that a subset of four classifiers that exhibited superior performance (*Random Forest* (RF), *Logistic Regression* (LR), *Support Vector Machines* (SVR), and *eXtreme Gradient Boosting* (XGB)) can effectively identify HIGH technical debt software classes, with RF being the best-performing model among them achieving an F_2-measure score of approximately 0.79, with a recall close to 0.85.

3.2 Explainability for TD Identification

Even though the extraction of a subset of superior ML approaches for TD identification is expected to enable practitioners to identify candidate TD items in their own systems with a high degree of certainty, there is still skepticism, in the SE community, or even unwillingness for the adaptation of such solutions, due to the black-box nature of the derived outcomes.

Towards this direction, as we have already mentioned, the main scope of the current study is dedicated to the provision of explanations for ML models in TD identification with the aim of understanding the mechanisms behind the reasoning of actionable suggestions in TD Management activities. Typically, the lifecycle of providing explanations for ML algorithms is a two-step process that serves two general scopes related to the *global* and *local* interpretation of a model's behavior. At the higher level, global XAI approaches serve the identification of the subset of the most important (or key) features that heavily affect the expected behavior of the entirety of a ML model, whereas at the lower level, local XAI techniques focus on providing deeper insights related to the features that influence a particular decision, e.g., the expected value of the response, for a single instance under examination.

Due to the challenging task and practical implications of developing integrated solutions that increase the levels of understanding, transparency, and trustworthiness of complex ML algorithms, during the past years, there has been noted a rising shift towards the development of XAI approaches that fulfill the abovementioned scopes [5]. In this study, we leverage the SHAP approach that seems to gain a great amount of attention in both academic and industrial settings, since it satisfies three desirable properties (i.e., local accuracy, missing values, and consistency), and thus, it is theoretically guaranteed to produce optimal feature importance ranks [28]. Furthermore, various studies suggest that SHAP is being increasingly adopted also in the SE community to understand how software metrics contribute towards the examined phenomena (see Section 2.4).

SHAP is, in fact, a XAI method that takes advantage of both the merits of *Shapley values* [28] from coalitional game theory and local explanations like the *Local Interpretable Model-agnostic Explanations* (LIME) [43] into a unified approach. In brief, the contribution φ_i of each feature i on the estimated outcome (i.e., a single prediction for each instance of the training set) of the model is evaluated through the Shapley values. Afterwards, a feature explanation model g of features is defined as a linear function of binary values, as shown in the following equation:

$$g(z) = \varphi_0 + \sum_{i=1}^{m} \varphi_i z_i \qquad (1)$$

where $\varphi_i \in R$ for $i = 0,1,\dots,m$ are the Shapley values, m is the number of simplified input features, $z_i = \{z_1, z_2, \dots, z_m\}$ is a binary vector in simplified input space where $z \in \{0;1\}^m$. Note that $|\varphi_i|$ are feature importance scores that are guaranteed in theory to be locally, consistently, and additively accurate for each data point [28].

Apart from the solid theoretical foundation in game theory, the reasons for our preference on utilizing the SHAP approach instead of other XAI techniques are summarized into the following key points: (*i*) SHAP is a post-hoc and model-agnostic approach, which practically means that it can be used for explainability purposes of complex models (e.g. RF in our case) that are not interpretable by design, and can be applied on any model without any knowledge of its internal structure [5]; (*ii*) SHAP can be leveraged for fulfilling both global and local interpretation objectives, since the whole process is settled on a common basis of analysis that is the estimation of the Shapley values for each instance of the dataset; (*iii*) SHAP provides a suite of quantitative and visualization techniques that facilitate the inferential mechanisms of both the strength and the direction of the impact of features on the response variable.

3.3 Experimental Setup

This section describes the key elements of the experimental setup that was designed to provide answers to the posed RQs. More specifically, we provide details related to: (*a*) the dataset used for experimental purposes and (*b*) specific decisions concerning the fitting and evaluation of the project-specific classifiers for TD identification.

As mentioned in Section 3.1, the dataset used in this study has been created and used in our previous research efforts aiming at the building of ML classifiers that are able to identify classes with high level of TD accumulation [41]. The dataset comprises a plenty of information about 18 code-related metrics and metrics that capture aspects of the development process, that were used, in turn, as input features $\mathbf{X}$ for learning a mapping function f_C that labels each instance into NOT-HIGH or HIGH TD group of classes (dichotomous response I_{TD}). However, in contrast to our previous work where the classifiers were built, validated and tested on the whole aggregated dataset (a total of 25 Java open-source projects consisted of 18,857 classes), in this study, we built project-specific classifiers, which practically means that in each experimental run a single project along with its classes was used as the dataset D for learning the mapping function f_C. The selected projects, along with additional information regarding their descriptive statistics are presented in detail in Table 1. We must clarify that from the original set of projects we have removed *gson*, *javacv*, *vassonic*, and *xxl-job*, since we were unable to develop project-specific classifiers, due to the limited number of classes ($min = 64$, $max = 112$) and the highly skewed distribution of the response variable.

TABLE 1: SELECTED PROJECTS

Project	KLoC	Classes	HIGH TD Classes
arduino	~27	239	22
arthas	~28	295	24
azkaban	~79	526	38
cayenne	~348	1,579	117
deltaspike	~146	684	36
exoplayer	~155	674	53
fop	~292	1,586	109
jclouds	~482	2,971	125
joda-time	~86	169	10
libgdx	~280	1,967	143
maven	~106	646	41
mina	~35	457	27
nacos	~60	418	34
opennlp	~93	681	54
openrefine	~69	608	53
pdfbox	~213	1,005	72
redisson	~133	872	60
RxJava	~310	795	65
testng	~85	354	27
wss4j	~136	501	43
zaproxy	~187	1,137	90

Before proceeding to the model building phase, appropriate data pre-processing tasks need to be performed, which include missing and outlier values handling. With respect to missing values handling, similarly to our previous study [41], we analyzed the 21 project-specific datasets and removed a small number of specific cases (i.e., software classes) for which the analysis tools failed to run and therefore were unable to compute metrics (1.3% of the total dataset). Regarding the outlier detection, again similarly to the previous study, we used an automatic outlier detection technique known as the *Local Outlier Factor* (LOF) [46] to remove a small number of cases with extreme values. These two steps resulted in a slightly smaller but equally representative dataset, containing 17,797 software classes in total.

Regarding the model building phase, the first decision concerns the choice of the algorithm that would be adopted for the fitting of the project-specific classifiers. In this regard, we decided to investigate the subset of classifiers that showcased the best performance in our previous related study [41]. More specifically, the Scott-Knott multiple comparisons algorithm indicated that LR, SVR, RF and XGB models can be grouped into a homogenous cluster of classifiers that present superior performances in terms of F_2-measure score. In addition, extensive experimentation on the total set of the projects revealed that the RF algorithm can be considered as a rationale choice for developing the project-specific classifiers.

The second critical task during the model building phase is related to whether we must adopt a feature selection mechanism that would potentially reveal irrelevant and/or highly correlated metrics with the response variable. While the presence of multicollinearity in the data might not affect the predictive power or reliability of a model, it does affect calculations regarding individual features' impact on the response variable, and therefore its interpretability [23]. More specifically, multicollinearity creates a problem because some (or all) inputs of a model are influencing each other. Therefore, they are not actually independent, and it is difficult to test how much the combination of the input features affects the response variable, within a model. In a scenario where two features are correlated and their importance is compared, the model will still have access to the feature through its correlated feature. This will result in a lower importance value for both features, where they might be important. In other words, the presence of correlated features poses significant barriers to the interpretation of a classifier, resulting into unstable importance ranks. Jiarpakdee et al. [23] analysed the impact of correlated features on the feature importance ranks of a defect classifier, noting that including correlated features when building a classifier can result in generating inconsistent importance ranks.

In our previous study [41], a thorough statistical exploratory analysis was performed indicating that all metrics can discriminate and potentially be used as input features of HIGH TD software classes. However, multicollinearity among them was not considered within the context of that work. The main reason that led us to this decision is the fact that we, mainly, focused on maximizing the models' predictive performance and not on their inferential nature. In fact, we had repeated the experiments after removing the features that are responsible for multicollinearity, only to discover that removing any highly intercorrelated metrics did not improve the models' performance. On the contrary, it resulted in a slight performance drop. In the present study, we intend to potentially sacrifice a slight amount of the models' predictive performance in favour of explainability purposes.

To unveil threats related to multicollinearity issues, we first examined the intercorrelations among the entire set of metrics, by applying a Spearman's rank correlation analysis. We chose Spearman's rank correlation, as it is a nonparametric test that is not sensitive to outliers. Through the Spearman analysis, we identified that there are indeed a handful of metrics that are highly correlated with each other (see supplementary material[5]). For instance, *Non-Commented Lines of Code* (NCLOC) have a positive and high correlation with Coupling Between Objects (CBO), Weighted Method per Class (WMC), *Response for a Class* (RFC), *Total Methods* (TM), and *Total Variables* (TV). While Spearman analysis results could be used to manually remove any variables that show high intercorrelations, one of the most common ways to identify and quantify the severity of multicollinearity is the *Variance Inflation Factor* (VIF) [20]. The VIF is calculated by taking each predictor, regressing it against every other predictor in the model and then using the produced coefficient of determination (R^2). As a rule of thumb, a VIF between 1 and 5 indicates that a feature is moderately correlated with the others, while a value between 5 and 10 indicates that multicollinearity is likely present: the feature should be removed.

We iteratively computed the VIF factors for each one of the selected metrics and removed metrics with VIF values greater than 5, until there were no further features to remove. It should be noted that in cases where, during an iteration, two predictors had a similar VIF value and we needed to make a choice on which one to exclude (e.g., *WMC* vs *NCLoC*), we relied on manual selection based on expert knowledge, so as the remaining metric is more useful to a developer. As a result, six metrics (*Number of Commits*, *Experience of Contributors*, *Response for a Class*, *Non-Commented Lines of Code*, *Total Variables*, and *Total Methods*) were removed from the initial set, resulting in the final set of features to be used for building our models, as presented in Table 2. After removing correlated metrics, all VIF values were considerably less than 5, indicating that the final set of features does not suffer from multicollinearity anymore. A table showing our final predictor set along with the corresponding final VIF values is available as supplementary material[5].

TABLE 1: SELECTED FEATURES

Feature	**Acronym**	**Description**
AVG Code Churn	ACC	Average size of a code churn of a file along evolution.
Number of Contributors	NoC	Number of contributors who modified a file.
Number of Hunks	NH	Median number of hunks made to a file along evolution. A hunk is a continuous block of changes in a diff. This number assesses how fragmented the commit file is (i.e., lots of changes all over the file versus one big change).
Number of Issues in Issue Tracker	NoI	Number of times a file name has been reported in the project's Jira or GitHub issue tracker along the evolution of the project.
Coupling Between Objects	CBO	Coupling between objects. This metric counts the number of dependencies a file has.
Weighted Methods per Class	WMC	Weight Method Class or McCabe's complexity. This metric counts the number of branch instructions in a file.
Depth of Inheritance	DIT	Depth Inheritance Tree. This metric counts the number of "fathers" a file has. All classes have DIT at least 1.

Feature	Acronym	Description
Lack of Cohesion of Methods	LCOM	Lack of Cohesion in Methods. This metric counts the sets of methods in a file that are not related through the sharing of some of the file's fields.
MAX Nested Blocks	MNB	Highest number of code blocks nested together.
Total Refactorings	TR	Total number of refactorings for a file along evolution.
Duplicated Lines Density	DLD	Percentage of lines involved in duplications. The minimum token length is set to 100.
Comment Lines Density	CLD	Percentage of lines containing either comment or commented-out code.

Subsequently, the hyper-parameter tuning of the 21 project-specific RF classifiers was conducted via a stratified 10-fold cross-validation schema to deal with bias and overfitting threats due to the class imbalance problem. At this point, we must note that even though SMOTE (or any other resampling technique) may address the inherent limitation of classification algorithms to provide accurate predictions for the minority classes in the presence of imbalanced datasets, this resampling strategy also affects the explainability and interpretation of complex ML models [39]. To address this challenge and improve the performance of the project-specific models under the presence of the class imbalance problem, we adjusted the class weighting mechanism during the training phase of each model, to give more emphasis on the minority class. It should be also noted that during the training phase, we made a deliberate choice not to employ data normalization methods (e.g., Min-Max Scaling), since (*i*) such a transformation would affect SHAP analysis and the interpretability of our findings at a later stage (Section 4), and (*ii*) tree-based classifiers (such as the RF model used in our experiments) are scale-invariant, and therefore do not require feature scaling [47].
Finally, the performance evaluation of the 21 project-specific models was based on F_2-measure, since it takes into consideration both recall and precision, while giving more emphasis on the former. In other words, F_2-measure places more importance to False Negative (FN) compared to False Positive (FP) misclassified cases. The rationale behind choosing F_2-measure instead of F_1-measure is similar to our previous study [41]. We consider it riskier for a development team to ignore classes that might have high TD (i.e., many FNs which might lead to inappropriate decisions with respect to maintenance) than to go through many classes that are labelled as problematic whereas they are not (i.e., FPs).

3.4 Methodology Overview

Below, we present an overview (see Figure 1) of the used methodology to sum-up the information required to more easily follow-up the experimental results. The overview is based on the ML lifecycle, extended with extra steps for supporting the explainability that is offered by the current study setup:

- *Problem Understanding*. The understanding of the problem as initiated with the presented context on Section 1 and setting the 3 research questions.
- *Data Collection*. To answer the set RQs, we have analysed 21 OSS projects (Table 1) and recorded 18 variables (Table 2). The dependent variable has been assigned based on the agreement of 3 well-known TD measurement tools (see Section 1).
- *Data Preparation*. Data pre-processing included the handling of missing values and outliers' detection (see Section 3.3).
- *Model Engineering*. We have built individual models for each OSS project, using the RF classifier. To remove collinearity of features we have used VIF, and we have applied 10-fold cross validation for assessing the model (see Section 3.3).
- Model Evaluation. The performance of each model is presented in Table 3, using precision, recall, AUC-ROC, AUC PR, and the F_2-meassure.
- Model Explainability. Global explainability is performed using SHAP and Scott-Knott Effect Size Difference test for unifying the results calculated and reported for different projects. Local explainability is achieved with SHAP (see Sections 4.1 – 4.3).
- *Model Reliability Analysis*. To investigate the extent to which the results presented in Section 4 are threatened by the selection of the aforementioned XAI techniques, we replicated the analysis using impurity-based feature importance for global, and LIME for local interpretation. The process and the outcomes are presented in Section 6.
- *Model Deployment*. The models have been deployed and we were able to draw several implications for researchers and practitioners (see Sections 5.1 and 5.2) by: (a) interpreting the results, (b) contrasting them to existing studies, and (c) by identifying limitations in their adoption, under certain circumstances.
- *Performance Monitoring*. To explore the usefulness of the models in practice, we have applied the deployed models in an OSS project *TD Management* (TDM) process and evaluated the actionability of the suggestions through a pilot qualitative study (see Section 5.3).

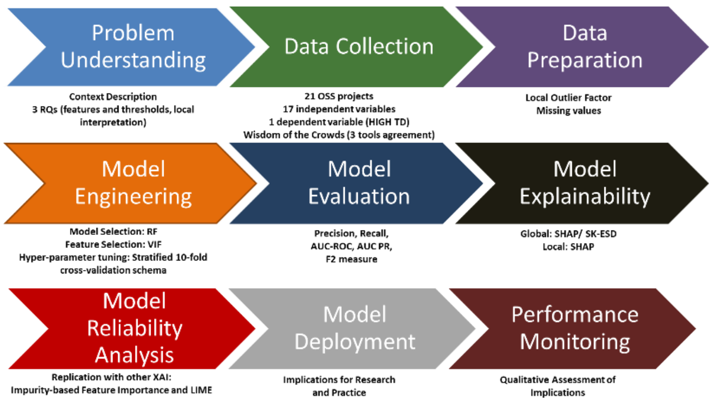

Problem Understanding
Context Description
3 RQs (features and thresholds, local interpretation)
Data Collection
21 OSS projects
17 independent variables
1 dependent variable (HIGH TD)
Wisdom of the Crowds (3 tools agreement)
Data Preparation
Local Outlier Factor
Missing values
Model Engineering
Model Selection: RF
Feature Selection: VIF
Hyper-parameter tuning: Stratified 10-fold cross-validation schema
Model Evaluation
Precision, Recall, AUC-ROC, AUC PR, F2 measure
Model Explainability
Global: SHAP/ SK-ESD
Local: SHAP
Model Reliability Analysis
Replication with other XAI: Impurity-based Feature Importance and LIME
Model Deployment
Implications for Research and Practice
Performance Monitoring
Qualitative Assessment of Implications

FIG. 1: EXPERIMENTAL METHODOLOGY OVERVIEW

4 Experimental Results

In this section, we report the findings of the experimental analysis, organized by research question. However, first we report the results of the performance evaluation for the modelling process of every project (see Table 3). The investigation of the fitting performance for the 21 project-specific models indicates satisfactory results, since they yield a F_2-measure ranging into the interval $[0.653, 0.897]$. The accurate modelling of the problem provides a solid basis for further analysis for explainability purposes.

TABLE 3: Performance Evaluation (Fitting) for the 21 Project-specific Models

Project	Precision	Recall	*AUC*		F_2
			ROC	*Precision Recall*	
Arduino	0.598	1.000	0.981	0.892	0.866
Arthas	0.757	0.900	0.969	0.828	0.840
Azkaban	0.702	0.783	0.958	0.781	0.758
Cayenne	0.472	0.862	0.968	0.764	0.735
deltaspike	0.424	0.917	0.965	0.670	0.721
exoplayer	0.779	0.873	0.978	0.889	0.846
fop	0.483	0.882	0.974	0.789	0.751
jclouds	0.730	0.768	0.979	0.815	0.755
joda-time	0.617	0.800	0.956	0.725	0.726
libgdx	0.518	0.861	0.972	0.780	0.756
maven	0.742	0.730	0.981	0.828	0.726
mina	0.540	0.717	0.942	0.718	0.653
nacos	0.752	0.917	0.986	0.937	0.862
opennlp	0.830	0.923	0.990	0.925	0.897
openrefine	0.664	0.903	0.978	0.831	0.837
pdfbox	0.429	0.900	0.952	0.653	0.734
redisson	0.678	0.933	0.990	0.910	0.864
RxJava	0.729	0.869	0.984	0.849	0.834
testng	0.487	0.883	0.957	0.722	0.724
wss4j	0.501	0.950	0.972	0.852	0.795
zaproxy	0.737	0.833	0.980	0.854	0.810

4.1 Metrics for HIGH TD Classes Identification (Global Explainability)

In RQ_1, we aim at globally investigating what are the metrics, whose scores are related to identifying HIGH TD classes. To achieve this goal, we focus on the results of the SHAP analysis and investigate if the same metrics are important for most projects. Initially, for each project, we develop a SHAP bee swarm plot, which summarizes insightful information concerning: (a) the contribution (or importance) of each feature on TD identification, and (b) its effect (positive/negative) on the response variable (NOT-HIGH / HIGH TD).

For illustrative purposes, we demonstrate the findings from the inspection of bee swarm plot for the JClouds project (Figure 2). In brief, the plot provides an overview of: (a) *Feature importance (y-axis)*: The metrics are ranked in descending order from top to bottom based on their absolute SHAP values as computed by the entire dataset (project in our case); (b) *Feature Impact*: The horizontal location of each dot shows, for each instance (class in our case) of the dataset, whether the value of the associated metric contributes towards a higher (HIGH TD) or lower (NOT-HIGH TD) predicted value. The further these dots extend on the x-axis (either positively or negatively), the higher their contribution to that prediction; (c) *Feature Value*: The colour of each dot shows, whether the metric score is high (in red) or low (in blue) for that observation. Based on the previous key points, in terms of global explainability for the case of JClouds project, we can infer that *WMC*, *CLD* and *LCOM* are the top three influential metrics for TD identification, whereas, *NH*, *CBO* and *DIT* contribute the least to the characterization of a class as NOT-HIGH / HIGH TD. The red dots extending far away (to the right) for the *WMC* metric imply that ***higher complexity*** values have a ***high positive contribution*** on the prediction of a class to be characterized as ***HIGH TD***. On the contrary, the blue dots extending to the right for the *CLD* metric imply that ***higher comment lines*** density has a ***high negative contribution*** on the characterization of a class as ***HIGH TD***.

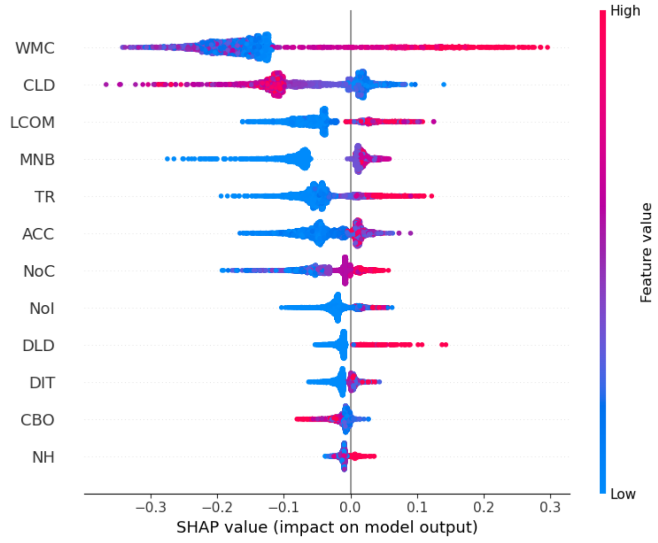


Fig. 2: SHAP Bee Swarm Plot for JClouds

For reasons of brevity and to reach more generalized findings, we do not go through each of the 21 projects in the paper[5], but we opted to follow a multiple hypothesis testing approach, namely the *Scott-Knott* (SK) test [48] to rank and cluster metrics according to their importance. Our preference on the utilization of the SK algorithm rather than other traditional inferential mechanisms (e.g. *Tukey's Honest Significant Difference*, *Scheffe's* tests etc. or their non-parametric analogue such as the *Nemenyi's* test) is due to its ability to identify non-overlapping homogenous clusters of metrics based on the mean differences of their importance scores [50]. More specifically, we made use of a variant of the original approach, namely the *Scott-Knott Effect Size Difference* (SK-ESD) test [51] that takes into consideration the effect size of an observed dif-

[5] The complete analysis is presented as supplementary material. Online: https://users.uom.gr/~a.ampatzoglou/aux_material/TD_XAI.pdf

ference that is related to the practical importance of the derived findings in the examined population.

The execution of the SK-ESD algorithm resulted in nine groups of homogenous clusters of metrics based on their pairwise average importance scores differences (i.e. the average of the mean absolute SHAP values). At this point, we have to note that the mean absolute SHAP values were square root transformed in order to meet the normality and homoscedasticity assumptions. The overall findings are graphically presented in Figure 3, in which the height of the bar indicates the average importance scores for each metric on the total set of the examined projects. In addition, the metrics are ranked in descending order starting from the most to the least important ones, while metrics that do not present statistically significant differences in accordance with their average importance scores are grouped into the same cluster (Cluster 1 to Cluster 9). The inspection of the graph suggests that when it comes to the metrics' contribution towards identifying HIGH TD classes, the *WMC* metric is the most significant feature, since it belongs to the 1st cluster presenting the highest average importance score. *CLD* is in 2nd place (Cluster 2), while *MNB* and *LCOM* are grouped into the 3rd position (Cluster 3). The group of top-7 metrics is completed with *CBO*, *TR*, and *ACC* metrics.

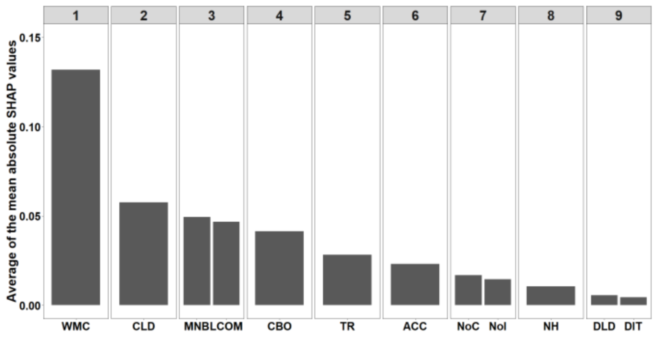


FIG. 3: RESULTS OF THE SK-ESD ALGORITHM ON SHAP

4.2 Thresholds for HIGH TD Classes Identification (Global Explainability)

To answer RQ_2, we focus on the top-3 clusters identified in RQ_1: *WMC*, *CLD*, *LCOM*, and *MNB*. To visualize and identify metric thresholds, per metric and per project, we have employed the collective SHAP stacked force plots—see Figure 4 for *WMC* with the data from the JClouds project. To this regard, the blue band shows how much a feature drags the final output value down (to NOT-HIGH TD class), and the red bands are those that increase it (up to HIGH TD class). We can observe that as *WMC* in a class increases beyond the value of ~20, the effect of this metric on labelling the class as HIGH TD increases significantly until the value of ~50, where it becomes constant but remains high. On the other hand, when *WMC* is below ~20, this metric contributes towards labelling a class as not HIGH TD. The rest of the force plots are presented as supplemental material[5].

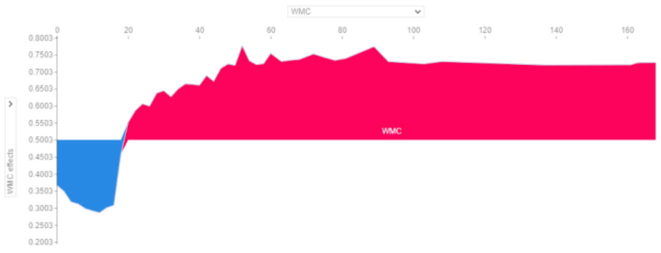


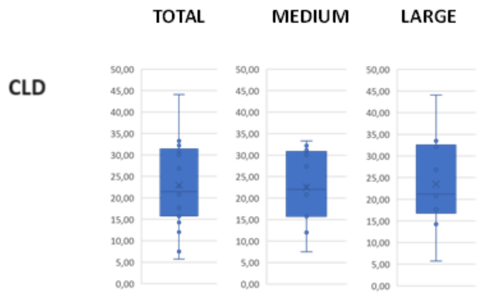
FIG. 4: METRIC THRESHOLD FOR WMC IN PROJECT JCLOUDS

Similarly to the answer for RQ_1, here we also aggregate and then present the results. The aggregation process can be described below: First, we retain the cut-off point for the metric score (the score in which the effect of the metric switches from contributing towards characterizing a class as NOT-HIGH TD to characterizing it as HIGH TD) for each project. Second, we report basic descriptive statistics (mean, min, max, standard deviation)—see Table 4. From Table 4, we can observe the mean threshold scores for each metric. However, due to quite large standard deviations of the threshold scores, we can conclude that a more fine-grained analysis might be required to reach a more reliable threshold.

TABLE 4: METRIC THRESHOLDS DESCRIPTIVE STATISTICS

Predictor	Mean	Min	Max	S. Dev.
CLD	22,94	5,71	44,05	9,48
LCOM	94,36	0,00	657,00	144,22
MNB	2,59	1,00	4,00	0,73
WMC	36,18	4,00	60,00	13,38

In this direction, in Figure 5, we present the distribution of the threshold scores, by considering size as a tentative parameter for getting more accurate thresholds. In that sense, we present the thresholds for the complete dataset, and for portions corresponding to medium-sized (<100K LoC) and large-sized systems (>100K LoC). The size categories have been extracted from SonarQube[6]. By inspecting Figure 5, we can observe that the large dispersion of thresholds is again evident, despite the splitting of the dataset. Therefore, it goes without saying that these global thresholds are only aiming an initial interpretation of metric scores, whereas project-based thresholds would be more accurate, and local interpretation (see Section 4.3) will lead to the most accurate possible understanding of the reasons for a class to be considered as in risk of being characterized as HIGH TD.



[6] https://docs.sonarsource.com/sonarqube/latest/

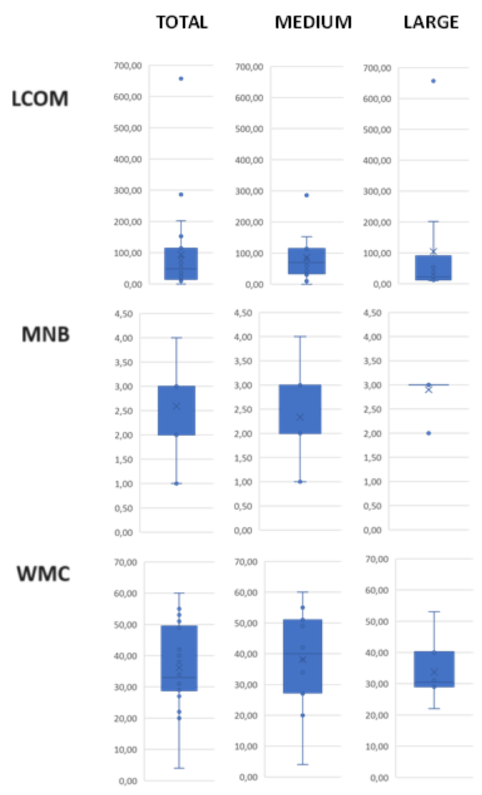


FIG. 5: METRIC THRESHOLDS VISUALIZATION

4.3 Identifying Opportunities for TD Repayment (Local Explainability)

In this section, we illustrate how the proposed methodology, enhanced by a local interpretation analysis, can lead to suggestions on how a HIGH TD class can be managed (answer to RQ3). To achieve this goal, we exploit SHAP force plots as case studies to find explanations for local prediction instances which can reflect the models' behaviour for concrete cases (classes in our case). Force plots demonstrate the following information: The $f(x)$ value is the predicted value for that observation; and the *colour* (red/blue) that showcases if the metric pushes the predicted value higher (to the right – towards HIGH TD—red colour), while those pushing the predicted value lower (to the left– towards NOT-HIGH TD—blue colour).

For example, consider the case of class *MapToDriveMetrics.java* (54 NCLoC), presented in Figure 6. The metric *MNB* has a positive impact on labelling the class as HIGH TD. The highest number of code blocks nested together in this class is 3, which is higher than the average mean threshold (i.e., 2.59). Therefore, due to its high score, this metric pushes the prediction to the right. On the other hand, *WMC*, *ACC*, *LCOM*, etc. all have a negative impact on labelling the class as HIGH TD (e.g., *WMC* = 16 << 36 the mean threshold from Section 4.2 and *ACC* = 7 << 61 the mean threshold from Section 4.2). Given the above, the class is labelled as NOT-HIGH TD with a probability of 0.01 (<0.5). Thus, no refactoring action is required for this case. However, out of these observations the team gets a "praise" on the good practices that they employ (i.e., keep complexity low, low code churn, high cohesion, and having performed some refactoring).

On the other hand, in Figure 7, we present the analysis for class *Metadata.java* (370 NCLoC), which is classified as HIGH TD from the model, with a probability of 0.91 (well above 0.5). In this case, the zero value of *DLD* metric pushes the predicted value to the left (i.e., towards characterizing the class as NOT-HIGH TD). However, the high score of *WMC* and the low score of *CLD*, among other metrics, have a strong positive impact on labelling this class as HIGH TD and push the predicted value far to the right. For the case of *WMC*, we can observe that the score of the class is 93 >> 36, while for comment lines density the score is 4.884 << 23 (the empirical mean thresholds defined in Section 4.2). From this analysis, the quality engineer gets an indication that the class suffers from high complexity and needs to be better documented. High complexity suggests that the class might have to be split into smaller, more focused classes. If complexity is not reduced by the split class refactoring, alternatives such as replacing conditionals with polymorphism can be explored.

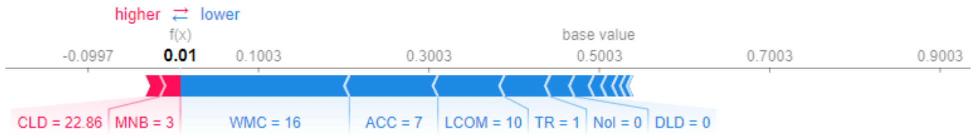


FIG. 6: LOCAL INTERPRETATION FOR NOT-HIGH TD CLASS

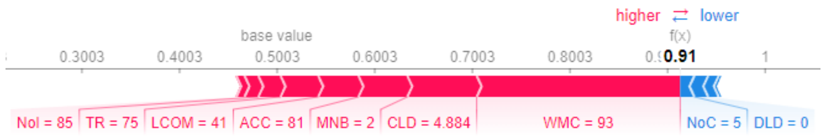


FIG. 7: LOCAL INTERPRETATION FOR HIGH TD CLASS

5. DISCUSSION

5.1 Interpretation of Results

In this section, we first summarize the answers to our RQs, and provide interpretations, based on the literature.

Important Metrics for TD Identification (RQ1): The answer to this research question revealed that *WMC*, *CLD*, *LCOM*, and *MNB* are consistently (across projects) the most important metrics for characterizing classes as HIGH TD. The interpretation of this finding can be performed as follows: (a) the extremely ***high consistency*** of the results confirms that the attempted global explainability makes sense; thus, this ***result is generalizable***; (b) the ***finding is intuitive.*** With respect to structural metrics, complexity (*WMC* and *MNB*), coupling (*CBO*) and cohesion (*LCOM*) have already been validated by previous research [33] as top maintainability predictors. On the other hand, with respect to process metrics, the existence of comments (as captured by *CLD*) has also been well-proven to help in understanding and maintaining code [4]; similarly, the frequent changes in code (high *ACC*) have been related to code quality deterioration [17]; (c) the extent to which a class undergoes refactoring (captured by *TR*) suggests that the specific class is either a design hotspot or code of low quality that needs to be improved [53], and (d) the mix of structure and process metrics in HIGH TD artifact identification confirms that ***TD is not a code-only phenomenon***, but it related to many other aspects of software engineering, such as architecture [26] [42] and technical management [31].

TABLE 5: RELAXED AND STRICT METRICS' THRESHOLDS

Metric	Relaxed	Strict
CLD	16,28	30,86
LCOM	106,25	16,25
MNB	3,00	2,00
WMC	47,25	29,00

Thresholds for TD Identification (RQ$_2$): Contrary to RQ$_1$, the analysis performed for RQ$_2$ has failed to produce project-agnostic metric thresholds for characterizing a class as HIGH TD[7] (because of high variation among projects). Subsequently, we deepened our analysis and explored if metric thresholds become less dispersed when treating medium- and large-scale projects separately. However, this extra analysis has not alleviated the problem—suggesting that ***global interpretation is not perfectly achievable through metrics***. This finding has been long supported by the literature, which suggests that generic metric thresholds are not applicable for software quality assessment and that domain-specific thresholds should be sought [18]. Nevertheless, by exploiting the box-plot analysis and relaxing the notion of thresholds from interquartile scores, we can claim that ***relaxed and strict thresholds can be identified*** (relying on the Q_1 and Q_3 quartiles threshold scores)—see Table 5. Although this finding cannot be blindly generalized to all projects, we believe that it provides a useful rule of thumb for practitioners. The identified strict and relaxed metrics agree with the thresholds derived in previous studies: e.g., see [10] for *MNB*, and [11] for *WMC* and *LCOM*.

Identification of Refactoring Suggestions (RQ$_3$): The finding obtained by answering RQ$_2$, further motivated the answer to RQ$_3$. In particular, the inability to safely provide a global interpretation strengthened our belief that a local (class-by-class) interpretation can play a significant role in Technical Debt Management and quality improvement. Our analysis provided a proof-of-concept that local interpretation can be useful for providing actionable suggestions to practitioners, by[8]: (a) "***praising***" and "***acknowledging***" the good practices that are identified in local cases; and (b) ***pointing to specific problems*** of a specific class, making the link to a ***specific refactoring more straightforward***. The usefulness of local interpretation has been acknowledged both in the field of software engineering [34], but also in other domains, e.g., precision medicine [6].

5.2 Implications for Researchers and Practitioners

Implications for Researchers: First, given the wealth of information that can be extracted from XAI analysis, we encourage software engineering researchers to make use of XAI on top of the ML/DL models. This analysis provides transparent models that are expected to be more applicable and acceptable from the industry. To verify this assumption, we aim to compare the acceptance of the results of this work against those of our previous study (black-box models) [41] in various software development industries. To achieve this goal, we intend to extend the tool of the black-box analysis [40] with XAI capabilities.

On top of this, we encourage researchers to use in their TD management endeavours all aspects of development, such as architecture and technical management, in the sense that they prove to be equally important and affect code TD. Finally, we believe that this work has advanced the domain of metric thresholds and deserves further exploration to identify project characteristics that might lead to less dispersed metrics' threshold scores. Nevertheless, we note that a full-fledged study that will validate the usefulness of these models in practice is required. Such a study would involve practitioners that would be provided with sets of HIGH TD classes (identified using strict and relaxed thresholds) and would ask them to validate (or invalidate) these classes as in need of special attention during TDM. Finally, the usefulness of local interpretations while performing refactoring of a specific class, needs to be qualitatively assessed.

Implications for Practitioners: In terms of practitioners, based on the findings of our study, and by exploiting the results of each research question, we can advise: (a) to focus their quality assessment on managing complexity, cohesion, commenting, and change frequency in the sense that these metrics seem to globally affect the probability of a class to be characterized as HIGH TD; (b) to use the relaxed and strict thresholds that we have identified in this study as a rule of thumb for their quality gates. We summarize these relaxed and strict thresholds in Table 5. We note that these thresholds can be safely perceived and used as follows (e.g., for Comment Line Density - CLD): "*In most projects a class does not need to be refactored in terms of comments density, if it has a score of CLD>>31%, or we need to increase the number of comments for classes with a score of CLD <<16%*"; and (c) for classes that are at HIGH TD risk, analyse the specific scores of metrics that drive in a positive and a negative direction. The "*good practices*" must be promoted by the company through training and

[7] The std. deviation of threshold scores among projects was quite high.

[8] Using the red and blue characterization of metrics in force plots.

by establishing quality gates in the CI/CD pipeline. At the same time special attention must be given to symptoms of poor quality that are recurring to all HIGH TD classes to identify their root causes and eliminate them. To enable the application of the complete methodology in practice we have developed a tool, named DEBTclock[9, 10]. For the special case that a software engineer wants to use DEBTclock early in the project history, when process metrics are still quite unstable, the model will rely mostly on the structural characteristics of classes. As the project evolves, the model will spot differentiations of process-based metrics and they will be treated as important features in the HIGH TD identification process.

5.3 Pilot Validation with Practitioners

To provide an early (or pilot) validation of the aforementioned implications to practitioners, we have used DEBTclock to manage the technical debt that has been accumulated along of the development of the ECLIPSE Open SmartCLIDE project[11]. In particular, we have analysed the components of ECLIPSE Open SmartCLIDE, and we have identified classes that pass the strict threshold of *CLD*, *WMC*, *LCOM*, and *MNB* (as defined in Table 5). Next, we have applied the prioritization approach of the "*Software Guidebook and Debt Calculator* [16]", as refined by Nikolaidis et al. [54] and identified the top-10 most HIGH TD classes. For these 10 classes, we have retrieved the local interpretation SHAP force plots. To validate these results, we have: (task-a) asked the 5 developers of the Eclipse community that worked on the project to validate that these classes are indeed HIGH TD; and (task-b) asked one developer per class to assess the usefulness of the local interpretation results. The validation has been performed in the form of a focus group.

Regarding task-a, 42 (out of 50) responses that we have obtained were positive. For 6 (out of 10) classes, all 5 developers agreed that the class seems difficult to maintain. For 2 classes, there were 2 disagreeing developers; whereas for the other 2 classes, there were 1 and 3 disagreeing developers. Although these results are preliminary, since the developers where not asked to select the top-10 most HIGH TD classes and contrast them, we believe that this finding demonstrates that the TD identification relies on a correct and practically intuitive basis.

Regarding the local interpretation and the ability to explain why a class is considered of HIGH TD, the results were also encouraging. First, all developers (5 out of 5) that participated in the focus group agreed that the visualization through force plots was very useful, since it unveiled "*reasons of poor quality*" that are not evident by any other tool that exists for TDM. Additionally, the 3 (out of 5) participants praised the fact that explainability is local, since in "*different projects different metrics scores might be problematic, or OK*". Finally, 3 (out of 5) participants claimed that "*knowing which metric is the root cause of HIGH TD can lead to refactoring*", whereas only 1 (out of 5) suggested that "*further automation for refactoring support would be welcome*".

6. Threats to Validity

For the current study focusing on the identification of metrics that contribute to the characterization of a software module (class) as HIGH TD or not, we analysed 21 Java open-source projects comprising 17,797 classes. As a result, the findings on the metrics that are important for the characterization of a class as TD prone depend on the context of the study and may not be generalizable to projects of a different domain or programming language. To this end, since the goal of this work was to provide project-specific thresholds, we have not performed any cross-project validation. Considering the ease with which ML models can be trained on any new dataset, the proposed methodology for interpreting classification models and deriving metric thresholds can be extended to any new context.

TD as a concept is not directly measurable but is captured at the operational level through measurements by static analysis tools, and these measurements constitute a construct. Construct validity is defined by how adequate these measurements [36] (which in our study formed the basis for labelling classes as HIGH TD or not) represent the concept of TD. While the assessment of TD through tools and especially the focus on code-level TD has been the subject of criticism, such construct threats are mitigated because: (a) the three employed platforms for building the benchmark of labelled classes are leading tools which are widely adopted by software industries and researchers [34] and (b) archetypal analysis was employed to synthesize their findings thereby increasing the trust in the commonly agreed findings [2].

As for the use of SHAP values for interpreting the RF classifiers, other methods in the literature may have provided different explanations on the importance of the considered metrics and the corresponding thresholds. Further research could indicate whether different approaches converge or not and to what extent the identified important metrics can be replaced by other factors. For this reason, we decided to perform sensitivity analysis with the aim of investigating possible ranking instability problems with respect to the metrics' importance on the predicted outcome for the set of the examined projects. More specifically, the *impurity-based feature importance* approach [49] was selected for evaluating the *Mean Decrease in Impurity* (MDI) measure that can be used, in turn, for acquiring an understanding on the relative contribution of each metric on the predicted outcome. In this regard, after the computation of the MDI score for each metric within the set of the examined projects, we made use of: (a) the SK-ESD test for investigating the overall ranking and clustering of the metrics based on the new criterion (MDI measures) and (b) the *Kendall's W coefficient of concordance* [52] for evaluating the level of agreement between the two XAI approaches evaluated on the rankings of the metrics' importance (SHAP vs. MDI) scores within each project.

[9] http://195.251.210.147:3006 (uname: uom@gr / pass: uom)
[10] DEBTclock Demonstration Video (url)
[11] https://projects.eclipse.org/projects/ecd.opensmartclide

Regarding the overall metric scores, the SK-ESD algorithm resulted into 9 homogenous clusters from which we can infer a generally high consistency of feature importance values between the two "*evaluators*" for all except one pair of metrics (*LCOM-MNB*). Additionally, both XAI approaches advocate that the *WMC* (Cluster 1) and *CLD* (Cluster 2) are the first and second most informative metrics, respectively. Finally, *MNB*, *LCOM* and *CBO* are highly ranked and grouped into the top-rated clusters. After the overall evaluation of the metrics' importance, the interest focuses on the investigation of the inter-rater agreement analysis via the computation of the Kendall's W coefficient of concordance for the set of the examined projects. In general, the values of the coefficients range from 0.745 to 1.000 with a mean value of 0.917 (95% CI [0.886, 0.949]) within the set of 21 projects indicating an almost perfect agreement between the two XAI global interpretation approaches. Therefore, we consider this threat as mitigated.

The interpretation of the metrics that render a module susceptible to having HIGH TD and the derivation of thresholds for metric scores that can be viewed as indicators of poor maintainability was based on previously constructed models [41]. Regarding the choice of the input features for these classifiers, threats to internal validity emerge, as various other metrics that can affect TD might have not been considered. Nevertheless, the employed TD features comprise widely studied metrics and reflect both code- and process-related measures.

Finally, to mitigate reliability threats we extensively describe the experimental setup and provide all results in the supplementary material. Researcher bias does not apply since the dataset of analysed classes has been retrieved from a previous study with no subjective interpretation by the researchers. We encourage the independent replication of the study in the same or other contexts to assess the validity of the derived metrics and thresholds.

7 Conclusion

Software quality assurance entails the assessment of internal characteristics and within each of them the quantification of sub characteristics using metrics. In a similar manner, TD management assumes the identification, measurement, and mitigation of individual TD issues, which are primarily found through static analysis tools. However, focusing on all software metrics or all identified TD issues is impractical and/or infeasible. Machine Learning techniques have opened news ways of assessing software quality by considering a plethora of features at once and classifying a module as 'good' or 'bad'. However, the black-box nature of the underlying models often decreases the trust in their findings and does not inform developers on what should be praised or blamed.

Building upon a previously constructed benchmark, we have developed project-specific classifiers for 21 open-source projects, characterizing classes as HIGH TD or not. Through SHAP analysis we found that complexity, comments ratio, cohesion, coupling, nesting of control flow statements, refactoring activity, and code churn are consistently the most important metrics that render a class susceptible to having high TD. The global interpretation of the results revealed metric threshold ranges which can serve as rules of thumb for class design, despite the high variability across projects. For example, when the *WMC* metric exhibits a value below 20, then this metric contributes towards labelling a class as NOT-HIGH TD. Through local interpretation, concrete recommendations can be obtained on which quality aspects are to be praised and which should be improved through refactoring.

Considering that the use of ML models for assessing all aspects of software development will increase further, we urge practitioners and researchers to take advantage of XAI approaches such as SHAP analysis to obtain insights on trained models. The interpretability of AI-based recommendations not only increases trust, but also acts as a means of informing and educating the stakeholders.

References

[1] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, ''Identification and management of technical debt: A systematic mapping study,'' *Information and Software Technology*, 70, pp. 100–121, Feb. 2016.

[2] T. Amanatidis, A. Moschou, N. Mittas, A. Chatzigeorgiou, A. Ampatzoglou, and L. Angelis, "Evaluating the Agreement among Technical Debt Measurement Tools: Building an Empirical Benchmark of Technical Debt Liabilities", *Empirical Software Engineering*, Springer, 2020.

[3] M. Aniche. "Java code metrics calculator (CK)", 2015 (url).

[4] O. Arafat and D. Riehle, "The comment density of open-source software code," *31st International Conference on Software Engineering (ICSE '09)*, Canada, 2009.

[5] A. B. Arrieta, N. Díaz-Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, and F. Herrera, F., "Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI", *Information fusion*, 2020.

[6] E. A. Ashley, "Towards precision medicine", *Nature Reviews Genetics*, 17, pp. 507–522, 2016.

[7] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering", *Dagstuhl Reports*, 2016.

[8] B. Baldassari, "SQuORE: a new approach to software project assessment.", *International Conference on Software & Systems Engineering and their Applications*, vol. 6, 2013.

[9] M. T. Baldassarre, V. Lenarduzzi, S. Romano, N. Saarimäki, "On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube", *Information and Software Technology*, Elsevier, 128, 2020

[10] T. Beranic & M. Hericko, "Comparison of systematically derived software metrics thresholds for object-oriented programming languages", *Computer Science and Information Systems*, 2020.

[11] A. Boucher & M. Badri, "Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison", *Information and Software Technology*, 96, 2018.

[12] G. A. Campbell & P. P. Papapetrou, "SonarQube in action, 2013.

[13] S. Chidamber, and C. Kemerer, "A metrics suite for object-oriented design", *Transactions on Software Engineering*, 20, 1994.

[14] D. Cruz, A. Santana & E. Figueiredo. "Detecting bad smells with machine learning algorithms: an empirical study", *3rd International Conference on Technical Debt (TechDebt '20)*, 2020.

[15] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the

principal of an application's technical debt," *IEEE Software*, 2012.

[16] R. J. Eisenberg, "A threshold-based approach to technical debt", *SIGSOFT Software Engineering Notes*, 37 (2), pp. 1–6, March 2012.

[17] C. Faragó, P. Hegedűs and R. Ferenc, "Cumulative code churn: Impact on maintainability," *15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015.

[18] K. Ferreira, M. Bigonha, R. Bigonha, L. Mendes & H. Almeida, "Identifying thresholds for object-oriented software metrics", *Journal of Systems and Software*, 2012.

[19] F.A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting with machine learning techniques for code smell detection", *Empirical Software Engineering*, 21 (3), pp.1143-1191, 2016.

[20] J. F.Hair, R. Anderson, R. L. Tatham, and W. C. Black, W. C., "Multivariate Data Analysis", *Upper Saddle River*, 2006.

[21] S. Herbold and S. W. Jens Grabowski, "Calculation and optimization of thresholds for sets of software metrics", *Empirical Software Engineering*, 16, pp. 812–841, 2011.

[22] Z. Huang, H. Yu, G. Fan, Z. Shao, M. Li, and Y. Liang, "Aligning XAI explanations with software developers' expectations: A case study with code smell prioritization", *Expert Systems with Applications*, 238, part A, March 2024.

[23] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The impact of correlated metrics on defect models", *IEEE Transactions on Software Engineering*, 2022.

[24] L. Kaur and A. Mishra, "Cognitive complexity as a quantifier of version-to-version Java-based source code change: An empirical probe", *Information and Software Technology*, 106, pp. 31-48, 2019.

[25] W. Li, "Another metrics suite for object-oriented programming", *Journal of Systems and Software*; 44(2), pp: 155–162, 1998.

[26] Z. Li, P. Liang, and P. Avgeriou, "Architectural Technical Debt Identification Based on Architecture Decisions and Change Scenarios", *12th Working Conference on Software Architecture*, 2015.

[27] M. Lorenz & J. Kidd, "Object-oriented Software Metrics", 1994.

[28] S. Lundberg & S. Lee, "A unified approach to interpreting model predictions", *Advances in neural information processing systems*, 2017.

[29] M. Mirakhorli and J. Cleland-Huang, "Detecting, tracing, and monitoring architectural tactics in code", *Transactions on Software Engineering*, 42 (3), pp. 205–220, 2015.

[30] A. Mishra, R. Shatnawi, C. Catal, and A. Akbulut, "Techniques for Calculating Software Product Metrics Threshold Values: A Systematic Mapping Study", *Applied Sciences*, 11 (23), 2021.

[31] N. Nikolaidis, N. Mittas, A. Ampatzoglou, E. M. Arvanitou, and A. Chatzigeorgiou, "Assessing TD Macro-Management: A Nested Modelling Statistical Approach", *IEEE Transactions on Software Engineering*, 2023.

[32] G. K. Rajbahadur, S. Wang, G. A. Oliva, Y. Kamei, and A. E. Hassan, "The Impact of Feature Importance Methods on the Interpretation of Defect Classifiers", *IEEE Transactions on Software Engineering*, 48 (7), pp. 2245-2261, 1 July 2022.

[33] M. Riaz, E. Mendes, and E. Tempero, "A systematic review on software maintainability prediction and metrics", *3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'09)*, IEEE Computer Society, Florida, USA, 2009.

[34] D. Sas and P. Avgeriou, "An Architectural Technical Debt Index Based on Machine Learning and Architectural Smells", *Transactions on Software Engineering*, pp. 4169-4195, 2023.

[35] R. Shatnawi, W. Li, J. Swain, and T. Newman, "Finding software metrics threshold values using ROC curves", *Journal of Software Maintenance and Evolution: Research and Practice*, 22(1), 2010.

[36] D. I. K. Sjøberg and G. R. Bergersen, "Construct Validity in Software Engineering", *IEEE Transactions on Software Engineering*, 49 (3), pp. 1374-1396, 1 March 2023.

[37] A. Sotgiu, M. Pintor, and B. Biggio, "Explainability-based Debugging of Machine Learning for Vulnerability Discovery", *17th International Conference on Availability, Reliability and Security (ARES '22)*, Vienna, Austria 23 – 26 August 2022.

[38] D. Spadini, M. Aniche, and A. Bacchelli. "PyDriller: Python Framework for Mining Software Repositories", *26th European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018.

[39] C. Tantithamthavorn, A. E. Hassan and K. Matsumoto, "The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models," *IEEE Transactions on Software Engineering*, 2020.

[40] D. Tsoukalas, A. Chatzigeorgiou, A. Ampatzoglou, N. Mittas, and D. Kechagias, "TD Classifier: Automatic Identification of Java Classes with High Technical Debt", *5th International Conference on Technical Debt (TechDEBT' 22)*, 2022.

[41] D. Tsoukalas, N. Mittas, A. Chatzigeorgiou, D. Kehagias, A. Ampatzoglou, T. Amanatidis, and L. Angelis, "Machine Learning for Technical Debt Identification", *IEEE Transactions on Software Engineering*, IEEE Computer Society, 2022.

[42] R. Verdecchia, I. Malavolta, and P. Lago, "Architectural technical debt identification: the research landscape", *Int. Conference on Technical Debt (TechDebt '18)*, Sweden, 2018.

[43] M. Wang, K. Zheng, Y. Yang, and X. Wang, "An Explainable Machine Learning Framework for Intrusion Detection Systems", *IEEE Access*, 8, pp. 73127–73141, 2020.

[44] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Classification model for code clones based on machine learning," *Empirical Software Engineering*, 20 (4), 2015.

[45] M. Zanoni, F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection", *Journal of Systems and Software*, 103, pp. 102–117, 2015.

[46] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "LOF: Identifying Density-Based Local Outliers," *SIGMOD Rec.*, 2000.

[47] P. Duboue, The Art of Feature Engineering: Essentials for Machine Learning, *Cambridge University Press*, 2020

[48] A. J. Scott and M. Knott, "A cluster analysis method for grouping means in the analysis of variance", *Biometrics*, 1974.

[49] G. Louppe, G., "Understanding random forests: From theory to practice", *ArXiv preprint* arXiv:1407.7502, 2014.

[50] N. Mittas and L. Angelis, "Ranking and clustering software cost estimation models through a multiple comparisons' algorithm", *IEEE Transactions on Software Engineering*, vol. 39, 537-551, 2013.

[51] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models", *IEEE Transactions on Software Engineering*, 43 (1), pp. 1-18, 2016.

[52] M. G. Kendall, "Rank Correlation Methods", *Oxford*, 1948.

[53] A. Eposhi, W. Oizumi, A. Garcia, L. Sousa, R. Oliveira, and A. Oliveira, "Removal of Design Problems through Refactorings: Are We Looking at the Right Symptoms?", *27th International Conference on Program Comprehension (ICPC)*, Canada, 2019.

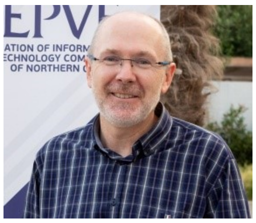
[54] N. Nikolaidis, N. Mittas, A. Ampatzoglou, D. Feitosa, and A. Chatzigeorgiou, "A metrics-based approach for selecting among various refactoring candidates", *Empirical Software Engineering*, Springer, 2024.

Dr. Dimitrios Tsoukalas is a Post-Doctoral Researcher at the Information Technologies Institute of the Centre for Research and Technology Hellas (CERTH). He holds a Ph.D. in "Machine Learning Techniques for Technical Debt Estimation and Forecasting" from the University of Macedonia (UoM), Greece. He also holds a B.Sc. and a M.Sc. from the University of Macedonia (UoM), Greece, as well as a M.Sc. from the Aristotle University of Thessaloniki (AUTH), Greece. His main research interests lie in the areas of Software Engineering and Intelligent Systems.

Dr. Nikolaos Mittas is an Associate Professor in the Department of Chemistry at the Democritus University of Thrace. He received the BSc degree in Mathematics from the University of Crete and the MSc and PhD degrees in Informatics from the Aristotle University of Thessaloniki (AUTH). His current research interests are focused on the application of statistics and data analytics in Software Engineering.

Dr. Elvira-Maria Arvanitou is a Post-Doctoral Researcher at the Department of Applied Informatics, in the University of Macedonia, Greece. She holds a PhD degree in Software Engineering from the University of Groningen (Netherlands, 2018), an MSc degree in Information Systems from the Aristotle University of Thessaloniki, Greece (2013), and a BSc degree in Information Technology from the Technological Institute of Thessaloniki, Greece (2011). Her PhD thesis has been awarded as being part of the top-3 ICT-related in Netherlands for 2018. Her research interests include technical debt management, software quality metrics, and software maintainability.

Dr. Apostolos Ampatzoglou is an Associate Professor in the Department of Applied Informatics in University of Macedonia (Greece), where he carries out research and teaching in software engineering. Before joining the University of Macedonia, he was an Assistant Professor at the University of Groningen (Netherlands). He holds a BSc in Information Systems (2003), an MSc on Computer Systems (2005) and a PhD in Software Engineering by the Aristotle University of Thessaloniki (2012). He has published more than 100 articles in international journals and conferences and is / was involved in over 15 R&D ICT projects, with funding from national and international organizations. His current research interests are focused on technical debt management, software maintainability, reverse engineering software quality management, open-source software, and software design.



Dr. Alexander Chatzigeorgiou is a Professor of Software Engineering in the Department of Applied Informatics and Vice Rector of Extroversion and International Relations at the University of Macedonia, Thessaloniki, Greece. He received the Diploma in Electrical Engineering and the PhD degree in Computer Science from the Aristotle University of Thessaloniki, Greece, in 1996 and 2000, respectively. His research interests include software maintenance, technical debt, and software evolution analysis. He has published more than 150 articles in international journals and conferences and participated in several European and national research programs. He is a Senior Associate Editor of the Journal of Systems and Software and an Associate Editor of the ACM Transactions on Software Engineering and Methodology.

Dr. Dionysios Kechagias is a Principal Researcher (grade B) with the Information Technologies Institute of the Centre for Research and Technology Hellas (CERTH). He received the Diploma and Ph.D. degrees in Electrical and Computer Engineering from the Aristotle University of Thessaloniki (AUTH), Greece, in 1999 and 2006, respectively. His research interests include software technologies, algorithms, data mining, machine learning, time-series analysis, big data analytics, service-oriented architectures, and ontology-based knowledge engineering.