Machine Learning for Technical Debt Identification

Dimitrios Tsoukalas, Nikolaos Mittas, Alexander Chatzigeorgiou, Dionysios Kehagias, Apostolos Ampatzoglou, Theodoros Amanatidis, and Lefteris Angelis

Abstract—Technical Debt (TD) is a successful metaphor in conveying the consequences of software inefficiencies and their elimination to both technical and non-technical stakeholders, primarily due to its monetary nature. The identification and quantification of TD rely heavily on the use of a small handful of sophisticated tools that check for violations of certain predefined rules, usually through static analysis. Different tools result in divergent TD estimates calling into question the reliability of findings derived by a single tool. To alleviate this issue we use 18 metrics pertaining to source code, repository activity, issue tracking, refactorings, duplication and commenting rates of each class as features for statistical and Machine Learning models, so as to classify them as High-TD or not. As a benchmark we exploit 18,857 classes obtained from 25 Java projects, whose high levels of TD has been confirmed by three leading tools. The findings indicate that it is feasible to identify TD issues with sufficient accuracy and reasonable effort: a subset of superior classifiers achieved an F_2-measure score of approximately 0.79 with an associated Module Inspection ratio of approximately 0.10. Based on the results a tool prototype for automatically assessing the TD of Java projects has been implemented.

Index Terms—Machine learning, Metrics/Measurement, Quality analysis and evaluation, Software maintenance

1 Introduction

Technical Debt (TD) is a metaphor facilitating the discussion among technical and non-technical stakeholders when it comes to investments on improving software quality [1]. As with financial debt, TD that is accumulated due to problematic design and implementation choices needs to be repaid early enough in the software development life cycle. If not done so, then it can generate interest payments in the form of increased future costs that would be difficult to be paid off. In this context, the development of software projects can be significantly impeded by the presence of TD: wasted effort due to TD in software companies can reach up to 23% of total developers' time [2] and in extreme situations may even lead to an unmaintainable software product and thus, to technical bankruptcy [3]. As a result, proper TD Management should be applied so as to identify, quantify, prioritize and repay TD issues. However, managing TD across the entire software development lifecycle is a challenging task that calls for appropriate tooling. A recent study [4] revealed 26 tools that can help towards the identification, quantification and repayment of TD. The strategies employed to identify TD issues differ, but the most frequently used approach relies on predefined rules that can be asserted by static source code analysis: any rule violation yields a TD issue while the estimated time to address the problem contributes to the overall TD principal. Due to their different rulesets, none of these tools results can be considered as an ultimate oracle [5], while using multiple tools to aggregate their results is not a feasible solution, since: (a) executing multiple tools for the same reason can be considered as a waste of resources; (b) there is no clear synthesis method; (c) acquiring multiple proprietary tools is not cost-effective.

The aforementioned shortcoming leads to important problems: On the one hand, academic endeavours face serious construct validity issues, because regardless of the tool used to identify Technical Debt Items (TDIs), there is always a doubt on if the results would be the same if a different tool infrastructure was used. On the other hand, regarding industry, usually the list of identified issues is very long, and the practitioners get lost in the numerous suggestions. On top of that, they face a decision-making problem: *"which tool shall I trust for TD identification/quantification?"*. As a first step to alleviate the aforementioned problems, in a previous study [5], we analyzed the TD assessment by three widely-adopted tools, namely *SonarQube* [6], *CAST* [7] and *Squore* [8] with the goal of evaluating the degree of agreement among them and building an empirical benchmark (TD Benchmarker[1]) of classes/files sharing similar levels of TD. The outcome of that study (apart from the benchmark per se) confirmed that different tools end-up in diverse assessments of TD, but to some extent they converge on the identification of classes that exhibit high-levels of TD.

Given the aforementioned result, in this paper, we propose the use of *statistical* and *Machine Learning* (ML) models

- *Dimitrios Tsoukalas is with the Department of Applied Informatics, University of Macedonia, Greece. E-mail: tsoukj@iti.gr*
- *Nikolaos Mittas is with the Department of Chemistry, International Hellenic University, Greece. E-mail: nmittas@chem.ihu.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 Centre for Research and Technology Hellas/Information Technologies Institute, Greece. E-mail: diok@iti.gr*
- *Apostolos Ampatzoglou is with the Department of Applied Informatics, University of Macedonia, Greece. E-mail: a.ampatzoglou@uom.edu.gr*
- *Theodoros Amanatidis is with the Department of Applied Informatics, University of Macedonia, Greece. E-mail: tamanatidis@uom.edu.gr*
- *Lefteris Angelis is with the Computer Science Department, Aristotle University of Thessaloniki, Greece. E-mail: lef@csd.auth.gr*

Manuscript received month day, year; revised month day, year.

1. http://195.251.210.147:3838/

for classifying TD classes (High/Not-High TD). As ground truth for the proposed classification framework, we use the dataset obtained from applying the TD-Benchmarker on 25 OSS Java projects, and we classify as high-TD the classes that all tools identify as TDIs. Then, supposing that the use of multiple sources of information will result in more accurate models, we built a set of independent variables based on a wide spectrum of software characteristics spanning from code-related metrics to metrics that capture aspects of the development process, retrieved by open-source tools. Finally, we apply various statistical and ML algorithms, so as to select the most fitting one for the given problem. Ultimately, the derived models subsume the collective knowledge that would be extracted by combining the results of three TD tools, exploiting a 'commonly agreed TD knowledge base' [5]. To facilitate their adoption in practice, the models have been implemented as a tool prototype that relies on other open-source tools to automatically retrieve all independent variables and identify high-TD classes for any software project.

The main contributions of this paper are summarized as follows: i) An empirical evaluation of various statistical and Machine Learning algorithms on their ability to detect high-TD software classes; ii) A set of factors to be considered while performing the classification, spanning from code metrics to repository activity, retrieved by open-source tools; iii) A tool prototype that yields the identified high-TD classes for any arbitrary Java project by pointing to its git repository.

2 Related Work

TD is an indicator of software quality with an emphasis on maintainability. During the past years, a plethora of studies have dealt with the aspect of software maintainability prediction, usually by employing ML regression techniques [9], [10], [11]. However, through our search in the related literature, we were unable to identify concrete contributions regarding the applicability of ML towards classifying a software module as high-TD or not. On the other hand, a multitude of studies have employed classification techniques to predict various quality aspects directly or indirectly related to the TD concept, such as code smells, change proneness, defect proneness, and software refactorings.

Code smells are considered one of the key indicators of TD [12]. Recently, a plethora of work related to code smell detection has been proposed in the literature [13], [14], [15], [16]. Most of these studies evaluate different ML classifiers on the task of detecting various types of code smells and conclude that with the right optimization, ML algorithms can achieve good performance. However, there is still room for improvement with respect to the selection of independent variables, ML algorithms, or training strategies [17]. Besides code smells, software change proneness and defect proneness are also popular TD indicators; a change-prone module tends to produce more defects and accumulate more TD [18]. To date, various studies have investigated the applicability of ML classifiers for the identification of change- and defect-prone modules [18], [19], [20], [21]. However, as these contributions are limited and suffer from external validity threats [22], further exploration should be performed by researchers towards these directions. Finally, TD is closely related to software refactoring, since the later constitutes the only effective way to reduce it on existing source code. However, researchers have only recently begun to explore how ML can be used to help in identifying refactoring opportunities [23].

From the above literature analysis, it is clear that ML classification methods, while not yet examined for their ability to identify the presence of TD per se, have been widely used in an attempt to accurately model various TD-related problems. More importantly, these studies present many similarities in terms of the classification frameworks that they adopt. To highlight these similarities, Table 1 presents an overview of the key decisions made in these studies, regarding their experimental setup. Regarding the employed classification algorithms, it can be seen that Logistic Regression (LR), Naïve Bayes (NB), and Random Forest (RF) are by far the most widely used classifiers, followed by Support Vector Machines (SVM) and Decision Trees (DT). Other studies also use J48, eXtreme Gradient Boosting (XGB), and K-Nearest Neighbor (KNN), among others. In terms of performance (highlighted in bold), Random Forest seems to be one of the most effective algorithms. With respect to the model train-and-validation strategy, 10-fold cross-validation is the de facto method used in every reported study. It is worth mentioning that, besides cross-validation, only one study [14] uses a hold-out set to test the produced models on completely unseen data. As regards the model parameter optimization, Grid Search is the most common method, followed by Random Search. Finally, as far as performance evaluation is concerned, Precision and Recall are used in almost every reported study, while F-Measure and ROC Curve are also very popular.

To design the experimental study presented within the context of this work, we adopted some of the most common practices regarding model selection, model configuration and performance evaluation. More details can be found in Section 3.3.

3 Methodology

In this section, we present the methodology followed throughout the study. The approach consists of three phases: (i) *data collection*, (ii) *data preparation (pre-processing and exploratory analysis)*, and (iii) *model building*. We note that throughout the rest of this document we will use the term *"modules"* for referring to software classes, to avoid confusing the reader by mixing software with classification classes.

3.1 Data Collection

3.1.1 Project Selection and Dependent Variable

The dataset of this study is based on an empirical benchmark that was constructed within the context of a study by Amanatidis et al. [5]. The aim of that study was to evaluate the degree of agreement among leading TD assessment tools by proposing a framework to capture the diversity of the examined tools and thus, to identify profiles representing characteristic cases of modules with respect to their level of TD. For this purpose, they have constructed a benchmark dataset by using three popular TD assessment

TABLE 1
Related Studies and their Adopted Classification Framework

Study	Classification Algorithms	Model Optimization	Train and Validation Strategy	Evaluation metrics
[13]	SVM, NB, **RF**, J48, JRip	Grid-search	10-fold cross-validation	Accuracy, F-Measure, ROC Curve
[14]	LR, NB, KNN, DT, **RF**, XGB, MLP	Random-search	10-fold cross-validation, Hold-out set	Precision, Recall, F-Measure
[15]	LR, SVM, NB, **RF**, MLP	Grid-search	10-fold cross-validation	Precision, Recall, F-Measure, ROC Curve
[16]	SVM, NB, **RF**, J48, JRip	Grid-search	10-fold cross-validation	Precision, Recall, F-Measure, ROC Curve
[19]	LR, NB, DT, RF, **MLP**	-	10-fold cross-validation	Precision, Recall
[18]	LR, NB, DT, **RF**, J48, MLP	-	10-fold cross-validation	Accuracy, Precision, Recall, F-Measure, ROC Curve
[20]	LR, DT, RF, **XGB**, ExtraTrees	Grid-search	10-fold cross-validation	Precision, Recall, F-Measure, ROC Curve
[23]	LR, SVM, NB, DT, **RF**, MLP	Random Search	10-fold cross-validation	Accuracy, Precision, Recall,

tools, i.e., SonarQube (v7.9, 2019), CAST (v8.3, 2018), and Squore (v19.0, 2019), to analyse 25 Java and 25 JavaScript projects and subsequently extract sets of modules exhibiting similarity to a selected profile (e.g., that of high TD levels in all employed tools).

The primary goal of this study is to train effective statistical and ML techniques to identify high-TD level modules, that is, to produce models that predict modules belonging to the *Max-Ruler* class profile. The Max-Ruler class profile, as defined by Amanatidis et al. [5], refers to modules whose reference assessment type indicates a high amount of TD based on the results of the tree applied tools. We focus exclusively on projects developed using Java, since most of the tools that we used to extend the dataset can be applied only for Java applications. As in the case of the work by Amanatidis et al. [5], we have analyzed the same 25 projects, considering their modules as units of analysis. These projects are presented in detail in Table 2. According to the authors [5], the criteria for selecting them were the programming language (i.e., Java), their public accessibility in GitHub, their popularity (more than 3 K stars) and finally, their active maintenance till the time of the study.

The dataset containing TD assessment of the three tools for each module of the 25 Java projects is publicly available at Zenodo[2] as an Excel file. The Max-Ruler (i.e., high-TD) modules of each project can be obtained in the form of csv files from the TD Benchmarker[3]. To begin the construction of the dataset used throughout this study, we initially downloaded the Excel file containing all the modules of the 25 Java projects under examination and then, for each project, we downloaded the csv file containing only the high-TD modules. Subsequently, we merged high-TD module instances with the initial Excel file (containing all modules) and we labeled high-TD modules with "1". The rest of the modules were labeled as "0". This process resulted in a dataset containing 18,857 modules; out of which 1,283 belong to the Max-Ruler profile (i.e., high-TD modules).

TABLE 2
Selected Projects

Project	Description	LoC
arduino	Physical computing platform	27 K
arthas	Java Diagnostic tool	28 K
azkaban	Workflow manager	79 K
cayenne	Java object to relational mapping framework	348 K
deltaspike	CDI management	146 K
exoplayer	Android media player	155 K
fop	Print formatter using XSL objects	292 K
gson	Java library to convert Java Objects to JSON	25 K
javacv	Wrappers of commonly used libraries	23 K
jclouds	Toolkit for java cloud applications	482 K
joda-time	Date and time handling	86 K
libgdx	Game development framework	280 K
maven	Software project management tool	106 K
mina	Network application framework	35 K
nacos	Cloud application microservices build and management	60 K
opennlp	Natural Language Processing toolkit	93 K
openrefine	Data management	69 K
pdfbox	Library of processing pdf documents	213 K
redisson	Java Redis client and Netty framework	133 K
RxJava	Composing asynchronous and event-based programs with observable sequences	310 K
testng	Testing framework	85 K
vassonic	Performance framework for mobile websites	7 K
wss4j	Java implementation for security standards in web applications	136 K
xxl-job	Distributed task scheduling framework	9 K
zaproxy	Security tool	187 K

3.1.2 Analysis Tools and Independent Variables

For building efficient classification models, we need to investigate the ability of various metrics, ranging from refactoring operations to process metrics and from code issues to source code metrics, to effectively discriminate between high-and not-high-TD module instances. Therefore, to construct our dataset we have employed a set of tools that can be classified into two broad categories: those that compute evolutionary properties (i.e., across the whole project evolution) and those that compute metrics related to a single (i.e., the latest) commit.

Initially, evolutionary metrics for each module of the 25 selected Java projects were computed by employing two widely used OSS tools, namely *PyDriller* [24] and *RefactoringMiner* [25]. More specifically, PyDriller (v1.15.5, 2021), i.e., a Python framework meant for mining Git repositories, was used to compute module-level Git-related metrics, such as commits count, code churn, and contributors experience across the whole evolution of the projects. Subsequently, RefactoringMiner (v2.0.3. 2020), a Java-based tool able to detect 55 different types of refactorings, was employed to compute the total number of refactorings for each module across their whole evolution. Apart from the two aforementioned tools, *Jira* and *GitHub* issue tracker APIs (depending on the project) were also used to fetch the total number of issues related to each module of the selected projects, across its entire evolution.

2. https://zenodo.org/record/3951041#.X5ApmND7SUk
3. http://195.251.210.147:3838/

Besides the tools used to compute evolutionary metrics, three additional tools, namely *CK* [26], *PMD's Copy/Paste Detector (CPD)*[4], and *cloc*[5], were considered for computing metrics related to the latest commit of each module. More specifically, CK (v0.6.3, 2020), a tool that calculates class-level metrics in Java projects by means of static analysis was used to compute various OO metrics, such as CBO, DIT, and LCOM for each module. Subsequently, CPD (v6.30.0, 2020), a tool able to locate duplicate code in various programming languages, including Java, was employed to compute the total number of duplicated lines for each module. Finally, cloc (v1.88, 2020), an open-source tool able to count comment lines and source code lines in many programming languages, was used to compute the total number of code and comment lines for each module.

To train effective classification models able to identify high-TD modules, we need to investigate what kind of metrics (or sets of metrics) are closely related to whether a module is of high-TD or not. However, as stated in many studies [27], the growth of TD in a software system is related to the growth of the system itself in such a way that the larger a system is in size, the larger its TD value will be. Therefore, to exclude the possibility that the TD level of a module is correlated with a metric only because of the size of the module, we normalized two of the metrics to control for the effect of size. The first metric that went through this transformation is *duplicated_lines*, which after a division by the *ncloc* metric of each module, was renamed to *duplicated_lines_density*. In addition, the *comment_lines* metric was also normalised but instead of dividing by the lines of code, we divided by the sum of code and comment lines ($ncloc + comment\_lines$) of each module. Subsequently, *comment_lines* metric was renamed to *comment_lines_density*.

It should be noted that considering the number of detected refactoring operations individually for each of the 55 different refactoring types (identified by RefactoringMiner) could lead to an unnecessary increase in the number of variables and therefore to an increase in complexity in later stages (i.e., during model training). In addition, by inspecting the RefactoringMiner results we identified that for most of the modules, the number of refactoring operations for the majority of refactoring types were zeros and therefore could result in the addition of a relatively large sparse matrix into the dataset. Therefore, we decided to aggregate the number of refactoring operations per module into a new metric called *total_refactorings*. This metric counts the total number of refactoring operations (for all 55 refactoring types) performed in a module during the evolution period.

In Table 3, we present the metrics that were evaluated during the data collection step for each module of the 25 Java projects, along with a short description.

Therefore, the final dataset comprises a table with 18,857 rows (the number of analyzed modules) and 19 columns, where each one of the first 18 columns holds the value of a specific metric, while an extra column at the end of the table holds the value (class) of the Max-Ruler (i.e., whether a module is of high-TD or not). Since the goal of this study is to investigate the relationship of various metrics (code, people or processes-related) to whether a module is of high-TD or not and subsequently train effective ML models to identify high-TD modules, the columns that refer to metrics will play the role of independent variables, while the last column that refers to Max-Ruler class will play the role of the dependent variable, i.e., the module TD level that we try to predict. This format helped us during the classification model building phase described in Section 3.3.

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

3.2 Data Preparation

3.2.1 Data Pre-processing

After extracting the module-level metrics of each project (using the six tools) and merging them into a common dataset as described in Section 3.1, we proceed with appropriate data pre-processing tasks, which include missing values handling and outlier detection techniques.

Starting with missing values handling, we observed that there were some cases where the CK tool failed to run and therefore was unable to compute metrics for specific modules. More specifically, out of the total 18,857 modules, the CK tool had generated results for 18,609, meaning that 248 modules were skipped. Since this number is relatively small (1.3% of the dataset), we decided not to proceed with data imputation in order to substitute missing values but instead to remove the instances that contain missing values. This resulted in a new dataset containing 18,609 modules, a slightly smaller but equally representative dataset size.

By quickly inspecting our data, we noticed that all independent variables presented skewed distributions, with a number of extreme values. Therefore, after performing missing values removal, we proceeded with outlier detection. Outliers are extreme values that may often lead to measurement error during the data exploration process or to poor predictive performance during ML model training. ML modeling and model skill in general can be improved by understanding and even removing these outlier values [28], [29]. In cases where the distribution of values in the sample is Gaussian (or Gaussian-like), the standard deviation of the sample can be used as a cut-off for identifying outliers. In our case however, the distributions of the metrics are skewed, meaning that none of the metrics follows a normal (Gaussian) distribution. Therefore, we used an automatic outlier detection technique known as the *Local Outlier Factor (LOF)* [30]. LOF is a technique that attempts to harness the idea of nearest neighbors for outlier detection. Each sample is assigned a scoring of how isolated it is or how likely it is to be an outlier based on the size of its local neighborhood. The samples with the largest score are more likely to be outliers. Applying LOF to the dataset resulted in a new dataset containing 17,797 modules, meaning that 812 modules were removed as the algorithm labeled them as outliers.

The descriptive statistics of the variables of the final dataset are presented in Table 3. It should be noted at this point that the performance evaluation of the selected classifiers, as will be presented later in Section 4, was also investigated without applying the aforementioned outlier detection and removal step. As expected, however, the non-removal of extreme values resulted in worse performance metrics and, therefore, we proceeded with the outlier removal step as described in this section.

TABLE 3
Selected Metrics and Descriptive Statistics

Metrics	Description	M	SD	min	Q_1	Mdn	Q_3	max
PyDriller								
commits_count	Total number of commits made to a file in the evolution period.	11.11	15.92	1	3	7	13	317
code_churn_avg	Average size of a code churn of a file in the evolution period.	27.31	44.89	-2	9	16	30	1543
contributors_count	Total number of contributors who modified a file in the evolution period.	3.50	2.70	1	2	3	4	45
contributors_experience	Percentage of the lines authored by the highest contributor of a file in the evolution period.	77.91	20.99	17.64	60.49	82.95	98.32	100
hunks_count	Median number of hunks made to a file in the evolution period. 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).	1.76	1.27	0	1	1.50	2	26.50
Jira/GitHub Issue Tracker								
issue_tracker_issues	Total number of times a file name has been reported in the project's Jira or GitHub issue tracker (mentioned within either the title or the body of the registered issue) in the evolution period.	10.08	53.05	0	0	0	3	1187
CK								
cbo	Coupling between objects. This metric counts the number of dependencies a file has.	7.44	7.91	0	2	5	10	109
wmc	Weight Method Class or McCabe's complexity. This metric counts the number of branch instructions in a file.	17.50	28.96	0	3	8	19	453
dit	Depth Inheritance Tree. This metric counts the number of "fathers" a file has. All classes have DIT at least 1.	2.04	1.84	1	1	1	2	52
rfc	Response for a Class. This metric counts the number of unique method invocations in a file.	14.15	22.58	0	1	7	17	293
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.	63.58	331.96	0	0	2	16	7503
max_nested_blocks	Highest number of code blocks nested together in a file.	1.31	1.49	0	0	1	2	21
total_methods	Total number of methods in a file.	8.68	12.04	0	2	5	10	256
total_variables	Total number of declared variables in a file.	9.54	18.30	0	1	4	10	305
RefactoringMiner								
total_refactorings	Total number of refactorings for a file in the evolution period.	15.66	52.23	0	1	1	9	1024
Copy-Paste Detector (PMD-CPD)								
duplicated_lines_density	Percentage of lines in a file involved in duplications (100 * duplicated_lines / lines of code). The minimum token length which should be reported as a duplicate is set to 100.	0.05	0.20	0	0	0	0	0.98
cloc								
comment_lines_density	Percentage of lines in a file containing either comment or commented-out code (100 * comment_lines / total number of physical lines).	0.39	0.22	0	0.22	0.36	0.54	0.94
ncloc	Total number of lines of code in a file, ignoring empty lines and comments.	97.53	142.38	2	23	51	110	1903

Note: M, SD, min, Q_1, Mdn, Q_3, max represent the mean, standard deviation, minimum, first quartile, median, third quartile, maximum values

3.2.2 Exploratory Analysis

An important step during exploratory analysis is to examine whether observable relationships exist between the selected metrics and the existence of high-TD and not-high-TD modules of the constructed dataset. For this purpose, we investigated both the discriminative and predictive power of the selected metrics using hypothesis testing and univariate logistic regression analysis, which are described in detail in the rest of this section.

To determine the ability of the selected 18 metrics to discriminate between high-TD and not-high-TD modules, we tested the null hypothesis that the distributions of each metric for high-TD and not-high-TD modules are equal. Since the metrics are skewed and have unequal variances (see Table 3), we used the non-parametric *Mann-Whitney U test* [31] and tested our hypothesis at the 95% confidence level ($a = 0.05$). In the first part of Table 4, we report the $p - values$ of the individual Mann-Whitney U tests. As can be seen, the $p - values$ of all metrics were found to be lower than the alpha level. Hence, in all cases the null hypothesis is rejected. This suggests that a statistically significant difference is observed between the values of the metrics of high-TD and not-high-TD modules, which indicates that all of these metrics can discriminate and potentially be used as predictors of high-TD software modules.

To complement the Mann-Whitney U test results, Table 4 also presents the median values of the computed metrics both for the high-TD and for the not-high-TD modules of

the dataset. We chose to present the medians instead of the mean values since normal distributions of the metrics cannot be assumed. As can be seen, the median values tend to be different in each metric. In all the cases, the metrics seem to receive a higher value at high-TD modules, with the only exception of the *contributors_experience* and *comment_lines_density* metrics.

By examining the discriminative power of the studied metrics we observed that their values demonstrate a statistically significant difference between high-TD and not-high-TD modules. In order to reach safer conclusions regarding the relationships between the selected metrics and the TD class of a module (high-TD / not-high-TD), we applied *univariate logistic regression* analysis. Univariate logistic regression focuses on determining the relationship between one independent variable (i.e., each metric) and the dependent variable (i.e., high-TD class) and has been widely used in software engineering studies to examine the effect of each metric separately [32], [33]. Thus, we used this method to help us with the process of identifying if underlying relationships between high-TD class and the selected metrics are statistically significant and in what magnitude.

Table 4 summarizes the results of the univariate logistic regression analysis for each metric, applied on our dataset. Column "Pseudo R^2" gives goodness-of-fit index pseudo R-squared (McFadden's R^2 index [34]), which measures improvement in model likelihood over the null model. Columns "$p-value$" and "SE" show the statistical significance and the standard error for the independent variables respectively. We set the significance level at $a = 0.05$. Metrics with $p-values$ lower than 0.05 are considered statistically significant to high-TD class. On the contrary, metrics with $p-values$ greater than 0.05 can be removed from further analysis, since they are not considered statistically significant. In our case, all 18 metrics are significantly related to the probability of high TD ($p-value \leq 0.05$).

In the last column of Table 4, we present the *Odds Ratio (OR)* of each metric. In short, the OR is the ratio of the probability of an event occurring to the event not occurring. Mathematically, it can be computed by taking the exponent of the estimated coefficients, as derived by applying Logistic Regression. The reason for considering the exponent of the coefficients is that, since the model is Logit, we cannot directly draw useful conclusions by inspecting the initial coefficient values (the effect will be exponential). Thus, converting to OR is more intuitive in the interpretation, as follows: $OR = 1$ means same odds, $OR < 1$ means fewer odds, and $OR > 1$ means greater odds. For example, by inspecting the "OR" column in Table 4 we observe that the metric *contributors_count* has an OR of 1.31, suggesting that for one unit increase in the number of contributors (i.e., an additional contributor) we expect the odds of a module being labeled as high-TD to be increased by a factor of 1.31 (i.e., a 31% increase). On the other hand, the metric *comment_lines_density* has an OR of 0.92 suggesting that for one unit increase in the percentage of the lines containing a comment we expect about 0.08 times decrease (i.e., a 8% decrease) in the odds of a module being labeled as high-TD.

To conclude the exploratory analysis, the Mann-Whitney U test revealed that all metrics can discriminate and potentially be used as predictors of high-TD software modules. Furthermore, the univariate logistic regression analysis suggested that all of the metrics were found to be significantly related to the probability of high TD ($p-value \leq 0.05$). Therefore, the entire set of metrics presented in Table 3 will be considered as input during the construction of the models described in Section 3.3. After all, some of the ML models used in this study are known to eliminate variables that they consider insignificant along their iterations based on embedded feature selection methods.

3.3 Model Building

This section provides details regarding the model building process that involves specific steps that are (i) *model selection*, (ii) *model configuration* and (iii) *performance evaluation*. Generally speaking, the design of our experimental study involves a candidate set of classification learning algorithms $A = A_1, ..., A_K$, where the objective of each candidate A_K is to learn a mapping function from input variables (or predictors) x_i to output variable (or response) y_i, where $y_i \in \{0, 1\}$ given a set of n input-output observations of the form $D = \{(x_i, y_i)\}_{(i=1)}^{n}$. In our case, the output variable is the characterization of TD into *not-High-TD* (denoted by 0) and *high-TD* (denoted by 1) and the output variables that are derived through the process described in Section 3.1.

3.3.1 Model Selection

Having in mind that there is a plethora of prediction candidates that can be used for classification purposes, we decided to explore a specific set of well-established statistical and ML algorithms that have been extensively applied in other similar experimental studies for code smell or bug prediction [17], [20]. More specifically, we used seven different classifiers that are summarized in Table 5: two are simple statistical/probabilistic models (*Logistic Regression* (LR), *Naïve Bayes* (NB)), and five are more sophisticated single (*Decision Trees* (DT), *K-Nearest Neighbor* (KNN), *Support Vector Machines* (SVM)) or ensemble (*Random Forest* (RF) and *eXtreme Gradient Boosting* (XGBoost)) ML models.

Although the candidate classifiers have shown to be effective in many application domains, their performances are significantly affected by the *class imbalance problem* [35]. This challenging issue occurs in classification tasks, when the class distributions of the response variable are highly imbalanced, which practically means that the one class is underrepresented (minority class) compared to the other (majority class). In this study, the response variable suffers from *intrinsic* imbalance [35] (ratio ∼1:15), due to the nature of the problem and thus, it presents a highly-skewed class distribution. This fact causes problems to the learning process, which leads to poor generalization ability of classifiers, since most of these algorithms assume balanced class distributions. Moreover, the majority of these learners aims to maximize the overall accuracy (or total error rate) of the fitted model, which results in turn, to classifiers that tend to be biased with models presenting an excellent prediction performance for the majority class but at the same time, extremely poor performance for the minority class.

The class imbalance problem is usually addressed by employing two widely-known techniques, namely *oversampling* and *undersampling* [36]. Oversampling achieves the desired

TABLE 4
Discriminative Power and Univariate Logistic Regression Results

Metrics	Discriminative Power			Univariate logistic regression			
	Median		Mann-Whitney U test (p-value)	Pseudo R^2	*p-value*	*SE*	*OR* (95% CI)
	not-high-TD	high-TD					
commits_count	6	21	<0.001	0.167	<0.001	0.002	1.056 (1.053 - 1.060)
code_churn_avg	15	24	<0.001	0.038	<0.001	0.001	1.010 (1.009 - 1.011)
contributors_count	3	5	<0.001	0.120	<0.001	0.009	1.315 (1.293 - 1.339)
contributors_experience	84	72	<0.001	0.016	<0.001	0.001	0.984 (0.981 - 0.987)
hunks_count	1	2	<0.001	0.035	<0.001	0.017	1.358 (1.313 - 1.405)
issue_tracker_issues	0	5	<0.001	0.006	<0.001	0.000	1.003 (1.002 - 1.003)
cbo	5	15	<0.001	0.201	<0.001	0.003	1.130 (1.123 - 1.137)
wmc	7	64	<0.001	0.389	<0.001	0.001	1.055 (1.053 - 1.058)
dit	1	2	<0.001	0.018	<0.001	0.012	1.168 (1.141 - 1.196)
rfc	6	49	<0.001	0.324	<0.001	0.001	1.059 (1.056 - 1.062)
lcom	1	92	<0.001	0.064	<0.001	0.001	1.002 (1.001 - 1.002)
max_nested_blocks	1	3	<0.001	0.233	<0.001	0.020	2.224 (2.139 - 2.312)
total_methods	4	21	<0.001	0.175	<0.001	0.002	1.073 (1.069 - 1.077)
total_variables	3	38	<0.001	0.371	<0.001	0.002	1.085 (1.081 - 1.089)
total_refactorings	1	22	<0.001	0.124	<0.001	0.000	1.014 (1.013 - 1.015)
duplicated_lines_density	5	14	<0.001	0.022	<0.001	0.001	1.016 (1.014 - 1.018)
comment_lines_density	38	17	<0.001	0.169	<0.001	0.002	0.929 (0.926 - 0.935)
ncloc	47	366	<0.001	0.531	<0.001	0.000	1.014 (1.014 - 1.015)

TABLE 5
Classification Models of the Experimental Setup

Classification Model	General Idea	Best Tuning Parameters
Logistic Regression (LR)	Employs a logit function to predict the probability of a categorical target variable belonging to a certain class.	penalty='none', solver='lbfgs'
Naïve Bayes Classifier (NB)	Applies Bayes' theorem to construct a probabilistic classifier based on the independence assumption between variables.	N/A
Decision Tree (DT)	Constructs hierarchical models composed of decision nodes and leaves to predict the class of the target variable.	criterion='gini', max_depth='none'
k-Nearest Neighbor (kNN)	Uses a distance function to predict the class of a new data point based on the majority label of the k data points closest to it.	n_neighbors=8
Support Vector Machine (SVM)	Tries to find the optimal N-dimensional hyperplane that maximises the margin between the data points to classify them into predefined classes.	kernel='rbf', C=1
Random Forest (RF)	A decision-tree-based ensemble algorithm that collects all the votes that are produced by its decision trees and provides a final classification result.	n_estimators=100, criterion='gini', max_depth='none'
XGBoost (XGB)	A decision-tree-based ensemble algorithm that uses multiple decision trees to predict an outcome based on a gradient boosting framework.	n_estimators=50, booster ='gbtree'

balance between classes by duplicating the minority class data, while undersampling does so by removing randomly chosen data from the majority class. There are cases where a combination of over-and under-sampling works better [36], i.e., by using specific ratios of both techniques until the desired trade-off between precision and recall is achieved.

In the current study, we opted for oversampling rather than undersampling, as we wanted to avoid removing valuable data from our dataset (and therefore avoid a possible drop in the models' performance). Furthermore, as regards the oversampling ratio, we chose to augment the minority class until its data instances become equal to those of the majority class (ratio 1:1). The reasoning behind this decision is twofold. First, a 1:1 class ratio guarantees the best possible representation of the minority class, which in this study we consider more important than the majority class. Second, during the evaluation process of ML models (described in Section 3.3.2), we noticed that having a 1:1 class ratio resulted in better performance metrics compared to the classifiers' performance when trying different ratios of oversampling, undersampling, or combinations of both.

To overcome this inherent limitation of classification learners to provide accurate predictions for both minority and majority classes in the presence of class imbalanced input variable, we made use of a well-known synthetic sampling method, namely the *Synthetic Minority Oversampling Technique (SMOTE)* [36]. *SMOTE* performs oversampling of the minority class by generating new synthetic instances based on the nearest neighbours belonging to that class. More details on where and how we integrated SMOTE into our experimental setup are presented in Section 3.3.2.

3.3.2 Model Configuration

To evaluate each classifier, we followed a *training-validation-test* approach. More specifically, the dataset was partitioned into two parts: 80% (14,238 samples) for training/validation and 20% (3,559 samples) for test. The first part was employed for model training, tuning and validation (using cross-validation), while the second one was used as a holdout set for the "final exam", i.e., the final evaluation of the models on completely unseen data. Data splitting was performed in a stratified fashion, meaning that we preserved the percentage of samples for each class (i.e., the initial high-TD – not-high-TD class ratio ∼1:15) among the

two parts. The latter is considered an important step, as it is of utmost necessity that the test set retains the initial class ratio, thus creating realistic conditions for final model evaluation.

Starting with the validation phase, we performed 3×10 *repeated stratified cross-validation* [37]. The dataset was randomly split into 10 folds, from which nine participate in training and the remaining one participates in testing, rotating each time the test fold until every fold serves as a test set. Moreover, each fold was initially stratified to properly preserve the initial high-TD – not-high-TD class ratio. Subsequently, oversampling (using SMOTE) was integrated into the cross-validation process. Despite the fact that the training folds undergo oversampling, the test fold always retains the initial class ratio for a proper model validation. The entire 10-fold cross-validation process was then repeated 3 times to account for possible sampling bias in random splits. Hence, the computed performance metrics of each classifier produced during the cross-validation are the averaged values of 30 ($3x10$) models trained and evaluated on the same dataset. In that way, we reduce the possibility of having bias introduced by the selection of non-representative subsets of the broader dataset.

As a common practice during the validation phase, we employed hyper-parameter tuning to determine the optimal parameters for each classifier and therefore increase its predictive power. To do so, we used the *Grid-search* method [38], which is commonly used to find the optimal hyper-parameters, by performing an exhaustive search over specified parameter values for an estimator. We chose the *F_2-measure* as the objective function of the estimator to evaluate a parameter setting (further details regarding the selection of *F_2-measure* are presented in Section 3.3.3). Hyper-parameter selection was performed using the stratified 3×10-fold cross-validation to avoid overfitting and ensure that the fine-tuned classifiers have a good degree of generalization before assessing their performance on the test set. In Table 5, we report the optimal hyper-parameters as they were adjusted during the tuning process.

During the validation phase we also performed a *MinMax* data transformation by scaling each feature individually in the range between zero and one. The reason behind this choice is that if a feature has a variance that is orders of magnitude larger than others, it might dominate during the learning process and thus make a classifier unable to learn from other features correctly. By applying the MinMax transformation we noticed that the predictive performance (and execution time) of most classifiers was improved, while for others there was no significant difference. Again, MinMax transformation was integrated into the stratified 3×10-fold cross-validation process where, during each iteration, it was fitted only to the training folds (to create realistic conditions) and then, used to transform both the training and test folds.

Finally, after fine-tuning and evaluating our classifiers through the validation phase described above, we proceeded with the final test phase. More specifically, we re-trained the fine-tuned classifiers using the training/validation set and applied them on the test set. Oversampling (using SMOTE) and MinMax transformation were used again, but only on the training/validation set, which at this point is used solely for model training. The data samples comprising the test set were never seen by the models during the previous phase (training/validation). Our goal was to create a hypothetical, but practical situation, where a new system or a set of systems is available, and the proposed classification framework must be applied to predict the presence of high-TD modules in the new (set of) system(s). In contrast to the cross-validation process where the performance metrics are averaged among the 30 ($3x10$) iterations, in this case, the computed metrics of each classifier comprise a single value, since each model was executed on the entire test set once. For our experiments, we used the Python language and more specifically the *scikit-learn*[6] ML library.

TABLE 6
Confusion Matrix

		Predicted Class	
		Positive	Negative
Actual Class	Positive	TP (True Positives)	FN (False Negatives)
	Negative	FP (False Positives)	TN (True Negatives)

3.3.3 Performance Evaluation

In practice, the performance evaluation of a binary classifier is usually assessed through alternative metrics based on the construction of a *confusion matrix* (Table 6). Typically, in a class-imbalanced learning process, the minority and majority classes are considered to represent the positive ($+$) and negative ($-$) outputs, respectively [39].

Furthermore, the de facto evaluation metrics (*accuracy* and *error rate*) do not consider the cost and side effects of misclassifying cases that belong to the minority class. This is also the case in our study, since predicting accurately the modules belonging to the minority class, i.e., modules exhibiting high levels of TD, is of great importance from the practitioners' perspective. The rationale for the preference of lowering FN compared to FP stems from the belief that the effects of ignoring a high-TD module can be considered more risky and detrimental to software maintenance compared to the wasting of effort to examine a falsely reported low-TD module.

Based on the previous considerations, we made use of appropriate evaluation metrics that have been proposed for dealing with the class imbalanced problem [35]. The *F-measure* is a widely used metric combining *precision* and *recall*, which are defined as follows:

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

$$Recall = \frac{TP}{TP + FN} \quad (2)$$

$$F - measure = (1 + \beta^2) * \frac{Precision * Recall}{(\beta^2 * Precision) + Recall} \quad (3)$$

where β is a coefficient for adjusting the relative importance of precision with respect to recall. Each of the above metrics represents different aspects of prediction performance

6. https://scikit-learn.org/stable/

providing straightforward directions about the quality of a classifier. Both precision and recall focus on the minority (positive) class, while the former is known a measure of *exactness* and the latter as a measure of *completeness* [39]. These two metrics are combined in order to provide an overall evaluation metric related to the *effectiveness* of a classifier on predicting correctly the minority cases, that is, the class of great importance in our experimental setup. More importantly, the *F-measure* provides a straightforward manner to place more importance to FN compared to FP misclassified cases by setting $\beta = 2$ leading to a special case of F-measure, known as F_2*-measure*. In other words, we consider it more risky for a development team to ignore modules that might have high TD (i.e., to suffer from the presence of many FNs which might lead to inappropriate decisions with respect to maintenance) than to go through many modules which are marked as problematic whereas they aren't (i.e., FPs). Therefore, for evaluating the selected set of binary classification models, we chose the F_2*-measure* as one of our main performance indicators.

It should be noted that unlike other fields, such as vulnerability prediction, where the minimization of FNs is considered of utmost importance and therefore the absolute emphasis is given on recall, we also value the number of FPs as their presence increases the effort required to study the reported problematic modules. Therefore, the F_2*-measure* is ideal in our case, since it considers both recall and precision, while giving more emphasis on the former.

As already mentioned, apart from the ability of the produced models to accurately detect as many high-TD modules as possible, it is important to take into account the volume of the produced FPs. A large number of FPs is associated to the increased manual effort required by the developers to inspect a non-trivial number of modules, in order to detect an actual high-TD module. Therefore, even though we use the F_2*-measure* as the criterion to test the models' performance, we further investigate their practicality by measuring the required inspection effort.

Following a similar approach to other studies [33], [40], [41], we use the number of modules to inspect as an estimator of the inspection effort. We define the *Module Inspection (MI)* ratio as the ratio of modules (software classes) predicted as high-TD to the total number of modules:

$$MI = (TP + FP)/(TP + TN + FP + FN) \quad (4)$$

This performance metric essentially describes the percentage of modules that the developers have to inspect in order to find the TPs identified by the model. As an example, let's consider a model with recall equal to 80%. In practice, this means that the model is able to identify correctly the 80% of high-TD modules. Let's consider also that this model has *MI* ratio equal to 10%. This means that by using the model, we expect to find the 80% of true high-TD modules (TPs) by manually inspecting only the 10% of the total modules, i.e., only the modules that were predicted as high-TD by the model ($TPs + FPs$). Obviously, using such a model is unarguably far more cost-effective than randomly choosing modules to inspect, since identifying the 80% of high-TD modules without using the model would require us to inspect the 80% of total modules.

To conclude, the F_2*-measure* and *MI* performance metrics are used as the basis for the comparison and selection of the best model(s), as their combination provides a complete picture of the model performance in predicting high-TD modules. In fact, F_2*-measure* indicates how effectively the produced models detect TPs but without ignoring FPs (i.e., by weighting recall higher than precision), whereas the *MI* indicates how efficient the models are in predicting TPs, based on how many FPs have to be triaged by the developers until a TP is detected. For completeness, in the experimental results presented in Section 4 we report other indicators like *precision*, *recall* and *Precision-Recall curves*.

4 Experimental Results

4.1 Quantitative Analysis

Table 7 summarizes the performance metrics for the set of examined classifiers validated and tested through the *training-validation-test* approach introduced in Section 3.3.2. More specifically, "Validation" column presents the results obtained for each classifier through the repeated stratified cross-validation process (validation phase). To this regard, the results present an overall indicator computed by averaging the performance metrics of $k = 10$ folds over three repeated executions. Additionally, the standard deviation is also reported. On the other hand, "Test" column presents the performance metrics of each examined classifier on the test (holdout) set, i.e., the 20% of the dataset that has not been used during training/validation. In both cases, the best classifier in terms of each performance measure is denoted in bold font. As mentioned in Section 3.3.3, we considered F_2*-measure* and *MI* ratio as the main performance indicators, since they cover both accuracy and practicality of the produced models.

The findings of Table 7 indicate that there is a subset of superior classifiers presenting small divergences in terms of F_2*-measure*. More specifically, as regards the validation phase, LR achieves the highest F_2*-measure* score with a value of 0.764. However, XGB, RF and SVM classifiers are following closely with F_2*-measure* scores ranging between 0.758 and 0.761. On the other hand, KNN, NB and DT seem to have noticeably lower F_2*-measure* scores. In terms of results in the test (holdout) set, we notice that the performance of the classifiers is not only preserved compared to the validation phase results, but is also slightly higher. More specifically, RF achieves the highest F_2*-measure* score with a value of 0.790, followed closely by XGB, LR and SVM with F_2*-measure* scores ranging between 0.781 and 0.788. Certainly, the overall performance evaluation is totally based on statistical measures evaluated from samples and for this reason, they contain significant variability that could lead to erroneous decision-making regarding the superiority of a subset of classifiers against competing ones. Thus, identifying a subset of superior models should be based on statistical hypothesis testing [42].

To this regard, within the context of evaluating the examined classifiers through the validation phase, we made use of a multiple hypothesis testing procedure, namely the *Scott-Knott (SK)* algorithm [43] that takes into account the error inflation problem caused by the simultaneous comparison

TABLE 7
Evaluation Results for All Classifiers

	F2		Precision		Recall		MI		Cluster
	Validation	Test	Validation	Test	Validation	Test	Validation	Test	
RF	0.760 (0.025)	**0.790**	**0.586 (0.022)**	**0.601**	0.822 (0.035)	0.858	**0.090 (0.007)**	**0.092**	A
XGB	0.761 (0.021)	0.788	0.458 (0.018)	0.477	0.914 (0.031)	0.941	0.134 (0.006)	0.133	A
LR	**0.764 (0.020)**	0.787	0.462 (0.022)	0.479	0.914 (0.026)	0.937	0.133 (0.007)	0.131	A
SVM	0.758 (0.019)	0.781	0.441 (0.020)	0.452	**0.925 (0.027)**	**0.954**	0.141 (0.007)	0.142	A
KNN	0.731 (0.024)	0.743	0.456 (0.023)	0.464	0.861 (0.031)	0.874	0.127 (0.007)	0.126	B
NB	0.684 (0.027)	0.712	0.449 (0.021)	0.471	0.788 (0.038)	0.816	0.118 (0.007)	0.116	C
DT	0.663 (0.042)	0.694	0.527 (0.031)	0.546	0.709 (0.054)	0.745	0.094 (0.005)	0.096	D

of multiple prediction models [44]. A major advantage compared to other traditional hypothesis procedures is the fact that the algorithm results into mutually exclusive clusters of classifiers with similar performance and thus, the interpretation and decision-making is a straightforward process. Finally, the algorithm is totally based on well-established statistical concepts of *Design of Experiments (DoE)* taking into account both *treatment* and *blocking* factors. In our case, the validation phase consists of 30 repeated performance measurements (i.e., F_2*-measure*) evaluated from each classifier (*treatment effect*) on $k = 10$ folds after three repeated executions (*blocking effect*). Describing briefly, the treatment effect takes into account the differences between the set of candidate classifiers, whereas the blocking effect is an additional factor that should be taken into account but we are not directly interested in and it is related to the splitting of dataset into different test sets on each execution of the stratified cross-validation process.

The findings of the SK algorithm indicated a statistically significant treatment effect ($p < 0.001$) on F_2*-measure*, which means that the observed differences among the seven classifiers cannot have occurred just by chance and there is indeed, a subset of superior classifiers in terms of F_2*-measure*. To this regard, the SK algorithm resulted into four homogenous clusters of classifiers that can be found in the last column of Table 7. The classifiers are ranked starting from the best (denoted by A) to the worst (denoted by D) clusters, whereas classifiers that do not present statistically significant differences are grouped into the same cluster. The best cluster encompasses four classifiers (RF, XGB, LR, and SVM) that present similar prediction performances. On the other hand, DT can be considered as the worst choice for identifying if a module is a high-TD or not, based on metrics.

Even though we considered F_2*-measure* as one of the main performance indicators, the classifiers belonging to the best cluster (Cluster A) present similar prediction capabilities. Therefore, it is worth inspecting also other aspects of performance as expressed by alternative measures in order to decide upon the best choice. As regards the additional reported performance metrics, namely precision and recall, the SVM classifier shows the higher recall during validation phase, with a value of 0.925, followed by XGB and LR with a value of 0.914. SVM also shows the higher recall on the test phase, with a value of 0.954. These three classifiers however have a precision score lower than 0.5 in both validation and test settings, which may dramatically increase the number of predicted FPs. More specifically, a precision score lower than 0.5 would result in the number of predicted FPs being greater than the number of predicted TPs. On the other hand, we notice that while RF (the classifier with the higher F_2*-measure* testing score) has achieved a recall score of 0.822 during validation and 0.858 during the test phase, its precision score is approximately 0.6 in both cases, which can be considered satisfactory given that our training configuration places more emphasis on identifying FNs rather than FPs.

While we consider it riskier for a development team to ignore actual high-TD modules than inspect many modules which have been falsely reported as high-TD, a statement that a model showing high recall and low precision is better than a model showing high precision and low recall is bold and arguable. Developers may prefer to inspect a large amount of potentially problematic modules because they can afford a lot of resources for refactoring activities. On the other hand, another development team may prefer a model that reduces the waste of effort even at the cost of missing some cases of high-TD modules. After all, TD management is about reducing the wasted effort during development [2]. We can argue that the RF classifier strikes a balance between these two cases, since it provides a quite satisfactory recall score, while at the same it preserves a lower (but significantly higher compared to the other models) precision score.

To complement the comparison of classification performance metrics presented above, we also present *Precision-Recall curves* [45] of each classifier, averaging the results over each fold of the 3×10 repeated stratified cross-validation process followed within the validation phase. A Precision-Recall curve is a plot that shows the tradeoff between precision ($y-axis$) and recall ($x-axis$) for different probability cutoffs, thus providing a graphical representation of a classifier's performance across many thresholds, rather than a single value (e.g., F_2*-measure*). In contrast to the *ROC* curves that plot the FP vs TP rate, Precision-Recall curves are a better choice for imbalanced datasets, since the number of TNs is not taken into account [45]. We also provide values for the *area under curve* (AUC) for each classifier, summarizing their skill across different thresholds. A high AUC represents both high recall and high precision. As can be seen by inspecting Fig. 1, the curve of RF classifier is closer to the optimal top-right corner, which practically means that it can achieve similar recall scores by sacrificing less precision compared to the other classifiers. In addition, RF presents the highest AUC, with a value of 0.77, followed by LR ($AUC = 0.75$) and XGB ($AUC = 0.74$). The dashed horizontal line depicts a "no-skill" classifier, i.e., a classifier that predicts that all instances belong to the positive class. Its $y - value$ is 0.067, equal to the ratio of positive cases in the dataset.

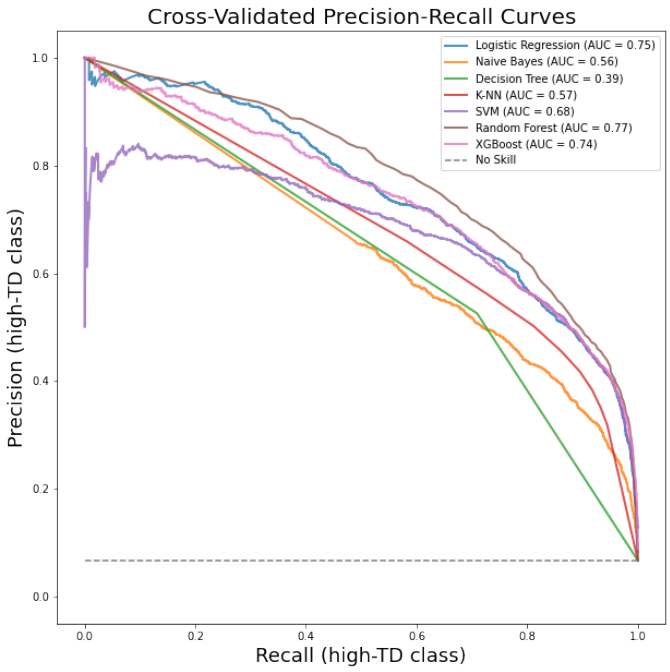

Fig. 1. Cross-validated Precision-Recall Curves

Regarding the second main performance indicator, i.e., the *MI* ratio, by inspecting the test phase metrics in Table 7 we can see that RF has the best (i.e., lowest) score with a value of 0.092. This can be seen as a logical outcome considering that RF achieved by far the best precision score among the examined classifiers. Possible tradeoffs between high recall and low precision have also implications in terms of cost effectiveness. As an example, let's consider a project coming from our dataset. The JClouds application has around 5,000 modules, among which 126 are labeled as high-TD. The SVM classifier provided the highest recall score (0.954) with a 14% *MI* ratio. This means that a development team would have to inspect $5,000 \times 14\% = 700$ potentially high-TD modules in order to identify the 95% of real high-TD cases, that is, 120 modules (and miss 6 high-TD modules). On the other hand, the RF classifier provided a recall score of 0.858 with a 9% *MI* ratio. This means that a development team would have to inspect $5,000 \times 9\% = 450$ potentially high-TD modules to identify the 86% of real high-TD cases, that is, 108 modules (and miss 18 high-TD modules). Similarly, if the additional 250 modules that need inspection if adopting the SVM over the RF model are worth the additional 12 high-TD modules identified by the SVM model is a company-specific decision. However, we believe that pursuing moderate precision and high recall might be a more cost-effective approach than pursing the highest recall at the cost of a very low precision.

4.2 Sensitivity Analysis

To investigate potential variations and ranking instabilities concerning the prediction capabilities of the examined classifiers per project, we decided to perform *sensitivity analysis*. Regarding the experimental setup, we performed a 3×10 repeated stratified cross-validation process twenty one times, using at each of the 21 iterations a single project as the dataset for both training and testing each examined classifier. To evaluate the classifiers' performance we considered again the F_2-*measure*. We have to clarify that we were not able to conduct the above analysis for four projects (i.e., gson, javacv, vassonic and xxl-job), due to the limited number of modules (min=64, max=112) that made the cross-validation process impossible to execute. The results of the sensitivity analysis are summarized in the Table 8.

Each row of the table presents the findings derived from the analysis conducted on each single dataset accompanied by the results of the SK algorithm regarding the statistical hypothesis testing procedure based on F_2-*measure*. Classifiers belonging to the best cluster are denoted in bold, whereas the last row provides an overview of classifiers' performance, that is, the percentage of experiments in which the classifiers were categorized into the set of superior models (denoted by the letter A). A first interesting finding concerns the ranking stability of classifiers characterized as superiors based on our previous cross-validation experimentation on the merged dataset (see Section 4.1), depicted also in the first row of Table 8. More specifically, XGB presents consistently outstanding prediction capabilities, since this classifier was grouped into the best cluster in 22 out of 23 experiments (95.45%). Furthermore, XGB exhibits even better F_2-*measure* scores on 13 datasets compared to the corresponding metric evaluated on the merged set of projects. RF and SVM can be also considered as a good alternative choice with noteworthy performances in the majority of the datasets. On the other hand, there is a strong indication of classifiers that consistently present moderate (KNN and NB) and poor (DT) predictive power, a finding that is aligned to the results obtained from the experimentation on the set of all projects.

Based on the results of both Section 4.1 and Section 4.2, it is of no surprise that more sophisticated algorithms, such as RF, XGB, and SVM, are performing better than other, more simple algorithms, such as DT or NB. This is a general remark, often encountered in various classification, or even regression tasks. In our case, the finding that XGB, RF and SVM are demonstrating a significantly higher performance compared to the rest of the examined classifiers might be attributed to the fact that these three algorithms stand out when non-linear underlying relationships exist in the data, which also applies in our case. Furthermore, the fact that XGB and RF, that is, two ensemble decision-tree-based algorithms, are ranked as the top 2 best-performing algorithms is also in line with the findings of similar studies (as presented in Section 2), where tree-based ensemble algorithms (e.g., RF) have been proved to be among the most popular and effective classifiers in related empirical SE studies.

4.3 Illustrative Examples of Identifying High-TD Classes

To complement the quantitative analysis, we also include indicative qualitative results. More specifically, in what follows, we present three distinct cases of modules from the dataset where not only there is an agreement between the three TD tools regarding their TD level (i.e., high/not-high TD), but the latter is also correctly predicted by the subset of our superior classifiers.

TABLE 8
Sensitivity Analysis Results

Dataset	#modules	RF	LR	SVM	XGB	KNN	NB	DT
merged	17797	**A (0.760)**	**A (0.764)**	**A (0.758)**	**A (0.761)**	B (0.731)	C (0.684)	D (0.663)
arduino	231	**A (0.680)**	**A (0.702)**	**A (0.641)**	**A (0.728)**	**A (0.770)**	**A (0.774)**	**A (0.635)**
arthas	279	**A (0.641)**	B (0.554)	B (0.522)	**A (0.666)**	**A (0.707)**	**A (0.623)**	B (0.536)
azkaban	510	**A (0.793)**	B (0.697)	B (0.682)	**A (0.847)**	B (0.722)	**A (0.775)**	B (0.748)
cayenne	1508	**A (0.751)**	**A (0.758)**	B (0.699)	**A (0.768)**	B (0.710)	B (0.730)	C (0.623)
deltaspike	668	B (0.561)	**A (0.659)**	**A (0.672)**	**A (0.612)**	**A (0.743)**	**A (0.688)**	C (0.433)
exoplayer	636	**A (0.878)**	B (0.813)	**A (0.857)**	B (0.812)	B (0.824)	B (0.819)	B (0.780)
fop	1535	**A (0.792)**	**A (0.816)**	**A (0.824)**	**A (0.797)**	**A (0.821)**	**A (0.802)**	B (0.664)
jclouds	2889	**A (0.769)**	**A (0.791)**	**A (0.762)**	**A (0.805)**	B (0.699)	B (0.668)	B (0.692)
joda-time	165	B (0.640)	B (0.527)	B (0.557)	**A (0.840)**	B (0.674)	B (0.581)	**A (0.823)**
libgdx	1855	**A (0.805)**	**A (0.814)**	**A (0.812)**	**A (0.795)**	**A (0.772)**	**A (0.785)**	B (0.712)
maven	621	**A (0.716)**	**A (0.736)**	**A (0.723)**	**A (0.769)**	**A (0.777)**	**A (0.699)**	**A (0.670)**
mina	436	B (0.504)	**A (0.633)**	**A (0.545)**	**A (0.627)**	**A (0.576)**	**A (0.638)**	C (0.375)
nacos	390	**A (0.804)**	B (0.663)	**A (0.759)**	**A (0.758)**	**A (0.781)**	**A (0.747)**	B (0.667)
opennlp	670	**A (0.846)**	**A (0.826)**	**A (0.867)**	**A (0.829)**	**A (0.806)**	**A (0.844)**	**A (0.855)**
openrefine	593	**A (0.812)**	**A (0.784)**	**A (0.763)**	**A (0.818)**	B (0.734)	**A (0.805)**	B (0.674)
pdfbox	983	**A (0.662)**	**A (0.715)**	**A (0.692)**	**A (0.703)**	**A (0.711)**	**A (0.751)**	B (0.508)
redisson	823	**A (0.843)**	**A (0.830)**	**A (0.828)**	**A (0.835)**	**A (0.852)**	**A (0.817)**	**A (0.784)**
RxJava	771	**A (0.833)**	**A (0.850)**	**A (0.859)**	**A (0.864)**	B (0.787)	B (0.805)	B (0.782)
testng	343	**A (0.665)**	**A (0.653)**	**A (0.710)**	**A (0.700)**	**A (0.750)**	**A (0.670)**	**A (0.619)**
wss4j	486	**A (0.729)**	**A (0.757)**	**A (0.801)**	**A (0.721)**	**A (0.744)**	**A (0.752)**	B (0.653)
zaproxy	1121	**A (0.776)**	**A (0.776)**	**A (0.773)**	**A (0.809)**	**A (0.799)**	**A (0.769)**	B (0.701)
% of experiments belonging to best cluster A		86.36%	77.27%	81.82%	95.45%	63.64%	72.73%	27.27%

As a first example, the module *SslSocketFactory.java* of the Mina project, a module that belongs to the Max-Ruler profile (i.e., high-TD based on the results of all the three tools) of the TD Benchmarker, was identified as high-TD also by the subset of our superior classifiers. To understand why this module was labeled as high-TD in the first place, we need to take a closer look at the analysis results produced by the three TD assessment tools (i.e., SonarQube, CAST, and Squore). Regarding SonarQube, the tool has identified four code smell issues (among which one critical and one major), a low comment line density (3.9%), and a relatively high cyclomatic complexity (15), despite the module's small size (i.e., 73 ncloc). Regarding CAST, the tool has identified various issues, such as high coupling and low cohesion (i.e., "class with high lack of Cohesion", "class with high Coupling between objects"), which it marks as high-severity issues. In addition, similarly to SonarQube, CAST also points out the lack of comments (i.e., "Methods with a very low comment/code ratio") as a medium-severity issue. Finally, regarding Squore, the tool has identified eight medium-severity maintainability issues (e.g., "Multiple function exits are not allowed") and, similarly to the other tools, it points out the lack of comments (i.e., "The artifact is not documented or commented properly").

In another example, the module *IoUtils.java* of the Nacos project, also belonging to the Max-Ruler profile, was correctly identified as high-TD by our superior classifiers subset. SonarQube has identified 17 code smell issues and three bugs (among which two critical and eight major), a very low comment line density (0.7%), and a high cyclomatic complexity (36), despite the module's small size (i.e., 152 ncloc). SonarQube has also identified a 30% duplicate line density. Regarding CAST, the tool has identified various issues, such as high coupling and low cohesion (i.e., "class with high lack of Cohesion", "class with high Coupling between objects"), and a couple of bug occurrences (i.e., "Close the outermost stream ASAP", "Avoid method invocation in a loop termination expression", etc.), all of which it marks as high-severity issues. Similarly to SonarQube, CAST also points out the high cyclomatic complexity (i.e., "Artifacts with High Cyclomatic Complexity"), lack of comments (i.e., "Methods with a very low comment/code ratio"), as well as code duplication issues (i.e., "Too Many Copy Pasted Artifacts"). Finally, Squore has also identified various maintainability issues (e.g., "Consider class refactorization", "Multiple function exits are not allowed", etc.), among which three are labelled as major. Similarly to the previous tools, Squore also highlights the high cyclomatic complexity (i.e., "Cyclomatic Complexity shall not be too high"), the low comment ratio (i.e., "The artifact is not documented or commented properly"), and finally the code duplication (i.e., "Cloned Functions: There shall be no duplicated functions").

In a third example, the module *CharMirror.java* of the Fop project, a module that belongs to the Min-Ruler profile (i.e., not-high-TD based on the results of all the three tools) of the TD Benchmarker, was correctly classified as not-high-TD by the subset of our superior classifiers. Regarding SonarQube, the tool has identified no bugs, code smells, or other issues, while its cyclomatic complexity is very low (8) compared to its large size (732 ncloc). Regarding CAST, the tool has identified only one high-severity issue (i.e., "Numerical data corruption during incompatible mutation"), while there were no high coupling and low cohesion problems. Finally, Squore identified only three minor-severity issues (e.g., "Multiple function exits are not allowed") and no further problems.

5 Implications to Researchers and Practitioners

In this study we made an attempt to leverage the knowledge acquired by the application of leading TD assessment tools in the form of a benchmark of high-TD modules.

Considering 18 metrics as features, the benchmark allowed the construction of ML models that can accurately classify modules as high-TD or not. The relatively high performance of the best classifiers enables practitioners to identify candidate TD items in their own systems with a high degree of certainty that these items are indeed problematic. The same models provide the opportunity to researchers for further experimentation and analysis of high-TD modules, without having to resort to a multitude of commercial and open-source tools for establishing the ground truth. A prototype tool that is able to classify software modules as high/not-high TD for any arbitrary Java project is available online[7].

It is widely argued that ML models operate as black boxes limiting their interpretability. To address this limitation we attempted to shed light into the predictive power of the selected features. While all 18 examined metrics were found to be significantly related to the probability of high TD, the provided statistical results can drive further research into the factors which are more strongly associated to the presence of TD thereby leading to the specification of guidelines for its prevention.

6 Threats to Validity

Threats to *external validity* concern the generalizability of results. Such threats are inherently present in the study, since the applicability of ML models to classify a software module as high-TD/not-high-TD is examined on a sample set of 25 projects. While it is always possible that another set of projects might exhibit different phenomena, the fact that the selected projects are quite diverse with respect to application domains, size, etc. partially mitigates such threats. Another threat stems from the fact that the examined dataset consists of Java projects, thus limiting the ability to generalize the conclusions to software systems of a different programming language. However, the process of building classification models described in this paper primarily builds upon the output of the tools used to compute software-related metrics that can act as predictors for classifying TD modules (high-TD/not-high-TD). This means that the proposed models can be easily adapted to classify the modules of projects that are coded in a different OO programming language, as long as there are tools that support the extraction of such metrics. Finally, since the dataset does not include industry applications, we cannot make any speculation on closed-source applications. Commercial systems as well as other OO programming languages can be a topic of future work.

Threats to *internal validity* concern unanticipated relationships or external factors that could affect the variables being investigated. Regarding the selection of the independent variables, it is reasonable to assume that numerous other software-related metrics that affect TD might have not been taken into account. However, the fact that we constructed our TD predictor set based on metrics that have been widely used in the literature, limits this threat. Regarding our decision not to perform feature selection and consider the entire set of metrics (presented in Table 3) as input to the classification models, we avoided selection bias by using hypothesis testing and univariate logistic regression analysis to investigate their discriminative and predictive power. Subsequently, all metrics were found to be significant towards predicting high-TD software modules. Finally, as explained in Section 3.3, we employed oversampling to increase classifiers' effectiveness in predicting the minority class. Nevertheless, we acknowledge that balancing a dataset may lead to distributions far different from those expected in real life and may therefore result in less accurate models in practice. Investigating whether it is feasible to create equally accurate models trained on data reflecting real distributions will be the subject of future work.

Threats to *construct validity* concern the relation between the theory and the observation. In this work, construct validity threats mainly stem from possible inaccuracies in the measurements performed for collecting the software-related metrics (i.e., the independent variables), but also from inaccuracies in the measurements performed for the construction of the TD Benchmarker [5] (i.e., the dependent variable). To mitigate the risks related to the data collection process, we decided to use five well-known tools, which have been widely used in similar software engineering studies to extract software-related metrics [14], [15], [23], [46], [47]. As regards the dependent variable, i.e., the TD Benchmarker created within the context of Amanatidis et al. [5], construct validity threats are mainly due to possible inaccuracies in the static analysis measurements performed by the three employed tools, namely SonarQube, CAST and Squore. However, all three platforms are major TD tools, widely adopted by software industries and researchers [4]. Regarding our decision to consider as high-TD only the modules that have been identified as such by all three TD tools (i.e., belonging to the Max-Ruler profile), we believe that while modules recognized as high-TD by multiple tools warrant more attention by the development team, ML models can be trained on any provided dataset. If a development team wishes to tag as high-TD modules even the ones identified as such by only a fraction of the three tools (e.g., two out of three), a different archetype can be selected (e.g., the Partner profile [5]) for extracting the training dataset. In an industrial context, the training dataset could also result from actual labeling of problematic classes so as to reflect the developers' perception of high TD. As for the experimented classification models, we exploited the ML algorithms implementation provided by the scikit-learn library, which is widely considered as a reliable tool.

Reliability validity threats concern the possibility of replicating this study. To facilitate replication, we provide an experimental package containing the dataset that was constructed, as well as the scripts used for data collection, data preparation and classification model construction. This material can be found online[8]. Moreover, the source code of the 25 examined projects is publicly available on GitHub to obtain the same data. Finally, as a means of practically validating our approach, we provide in the supporting material a prototype tool able to classify software modules as high-TD/not-high-TD for any arbitrary Java project. We believe that this tool will enable further feature experimentation through its use in academic or industrial setting and will pave the way for more data-driven TD management tools.

7. https://sites.google.com/view/ml-td-identification/home

8. https://sites.google.com/view/ml-td-identification/home

7 Conclusion and Future Work

In this study we investigated the ability of well-known Machine Learning algorithms to classify software modules as high-TD or not. As ground truth we considered a benchmark of high-TD classes extracted from the application of three leading TD tools on 25 Java open-source projects. As model features we considered 18 metrics covering a wide spectrum of software characteristics, spanning from code metrics to refactorings and repository activity.

The findings revealed that a subset of superior classifiers can effectively identify high-TD software classes, achieving an F_2-*measure* score of approximately 0.79 on the test set. The associated *Module Inspection ratio* was found to be approximately 0.10 while the recall is close to 0.86, implying that one tenth of the system classes would have to be inspected by the development team for identifying 86% of all true high-TD classes. Through the application of the Scott-Knott algorithm it was found that Random Forest, Logistic Regression, Support Vector Machines and XGBoost presented similar prediction performance. Such models encompass the aggregate knowledge of multiple TD identification tools thereby increasing the certainty that the identified classes suffer indeed from high-TD. Further research can focus on the common characteristics shared by the problematic classes aiming at the establishment of efficient TD prevention guidelines.

Acknowledgments

Work reported in this paper has received funding from the European Union's Horizon 2020 Research and Innovation Programme through EXA2PRO project under Grant Agreement No. 801015 and SmartCLIDE project under Grant Agreement No. 871177.

References

[1] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (dagstuhl seminar 16162)," in *Dagstuhl Reports*, vol. 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[2] T. Besker, A. Martini, and J. Bosch, "Software developer productivity loss due to technical debt—A replication and extension study examining developers' development work," *Journal of Systems and Software*, vol. 156, pp. 41–61, 2019.

[3] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[4] P. Avgeriou, D. Taibi, A. Ampatzoglou, F. Arcelli Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, A. Moschou, I. Pigazzini, N. Saarimäki, D. Sas, S. Toledo, and A. Tsintzira, "An Overview and Comparison of Technical Debt Measurement Tools," *IEEE Software, accepted for publication*, 2021.

[5] T. Amanatidis, N. Mittas, A. Moschou, 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*, vol. 25, no. 5, pp. 4161–4204, 2020.

[6] G. A. Campbell and P. P. Papapetrou, *SonarQube in action*, 1st ed. Manning Publications Co., 2013.

[7] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the principal of an application's technical debt," *IEEE software*, vol. 29, no. 6, pp. 34–42, 2012.

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

[9] M. O. Elish and K. O. Elish, "Application of TreeNet in Predicting Object-Oriented Software Maintainability: A Comparative Study," in *2009 13th European Conference on Software Maintenance and Reengineering (CSMR)*, 2009, pp. 69–78.

[10] A. Chug and R. Malhotra, "Benchmarking framework for maintainability prediction of open source software using object oriented metrics," *International Journal of Innovative Computing, Information and Control*, vol. 12, no. 2, pp. 615–634, 2016.

[11] R. Malhotra and K. Lata, "On the Application of Cross-Project Validation for Predicting Maintainability of Open Source Software using Machine Learning Techniques," in *2018 7th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)*. IEEE, 2018, pp. 175–181.

[12] N. S. R. Alves, T. S. Mendes, M. G. d. Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology*, vol. 70, pp. 100 – 121, 2016.

[13] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[14] D. Cruz, A. Santana, and E. Figueiredo, "Detecting Bad Smells with Machine Learning Algorithms: An Empirical Study," in *Proceedings of the 3rd International Conference on Technical Debt (TechDebt '20)*. ACM, 2020, pp. 31–40.

[15] F. Pecorelli, F. Palomba, F. Khomh, and A. De Lucia, "Developer-Driven Code Smell Prioritization," in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*. ACM, 2020, pp. 220–231.

[16] S. Lujan, F. Pecorelli, F. Palomba, A. De Lucia, and V. Lenarduzzi, "A Preliminary Study on the Adequacy of Static Analysis Warnings with Respect to Code Smell Prediction," in *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation (MaLTeSQuE 2020)*. ACM, 2020, pp. 1–6.

[17] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.

[18] R. Abbas, F. A. Albalooshi, and M. Hammad, "Software Change Proneness Prediction Using Machine Learning," in *2020 International Conference on Innovation and Intelligence for Informatics, Computing and Technologies (3ICT)*, 2020, pp. 1–7.

[19] N. Pritam, M. Khari, R. Kumar, S. Jha, I. Priyadarshini, M. Abdel-Basset, H. V. Long, and others, "Assessment of Code Smell for Predicting Class Change Proneness Using Machine Learning," *IEEE Access*, vol. 7, pp. 37 414–37 425, 2019.

[20] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are sonarqube rules inducing bugs?" in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 501–511.

[21] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, 2007.

[22] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4, pp. 531–577, 2012.

[23] M. Aniche, E. Maziero, R. Durelli, and V. Durelli, "The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring," *IEEE Transactions on Software Engineering, accepted for publication*, 2020.

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

[25] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Transactions on Software Engineering*, 2020.

[26] M. Aniche, *Java code metrics calculator (CK)*, 2015.

[27] G. Digkas, A. N. Chatzigeorgiou, A. Ampatzoglou, and P. C. Avgeriou, "Can Clean New Code reduce Technical Debt Density," *IEEE Transactions on Software Engineering, accepted for publication*, 2020.

[28] M. R. Smith and T. Martinez, "Improving classification accuracy by identifying and removing instances that should be misclassified," in *International Joint Conference on Neural Networks (IJCNN 2011)*, 2011, pp. 2690–2697.

[29] M. Kuhn and K. Johnson, *Applied predictive modeling*, 1st ed. Springer, 2013.
[30] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "LOF: Identifying Density-Based Local Outliers," *SIGMOD Rec.*, vol. 29, no. 2, pp. 93–104, 2000.
[31] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.
[32] Y. Zhou and B. Xu, "Predicting the maintainability of open source software using design metrics," *Wuhan University Journal of Natural Sciences*, vol. 13, no. 1, pp. 14–20, 2008.
[33] E. Arisholm and L. C. Briand, "Predicting fault-prone components in a java legacy system," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering and measurement (ESEM)*. ACM, 2006, pp. 8–17.
[34] D. McFadden and others, "Conditional logit analysis of qualitative choice behavior," *Institute of Urban and Regional Development, University of California*, 1973.
[35] H. He and E. A. Garcia, "Learning from Imbalanced Data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263–1284, 2009.
[36] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.
[37] H. He and Y. Ma, *Imbalanced learning: foundations, algorithms, and applications*. John Wiley & Sons, 2013.
[38] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS)*, vol. 2, 2015, pp. 2962–2970.
[39] P. Branco, L. Torgo, and R. P. Ribeiro, "A Survey of Predictive Modeling on Imbalanced Domains," *ACM Comput. Surv.*, vol. 49, no. 2, 2016.
[40] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.
[41] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," *International Symposium on Software Reliability Engineering (ISSRE)*, pp. 23–33, 2014.
[42] J. Demšar, "Statistical Comparisons of Classifiers over Multiple Data Sets," *J. Mach. Learn. Res.*, vol. 7, pp. 1–30, 2006.
[43] A. J. Scott and M. Knott, "A Cluster Analysis Method for Grouping Means in the Analysis of Variance," *Biometrics*, vol. 30, no. 3, pp. 507–512, 1974.
[44] N. Mittas and L. Angelis, "Ranking and Clustering Software Cost Estimation Models through a Multiple Comparisons Algorithm," *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 537–551, 2013.
[45] J. Davis and M. Goadrich, "The Relationship between Precision-Recall and ROC Curves," in *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*. ACM, 2006, pp. 233–240.
[46] Z. Codabux and C. Dutchyn, "Profiling Developers Through the Lens of Technical Debt," in *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '20)*. ACM, 2020.
[47] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "The technical debt dataset," in *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, 2019, pp. 2–11.