Recommending when Design Technical Debt Should be Self-Admitted

Anonymous[1]
[1]Affiliation

***Abstract*—Previous research has shown how developers "self-admit" technical debt introduced in the source code, commenting why such code represents a workaround or a temporary, incomplete solution. This paper investigates the extent to which previously self-admitted technical debt can be used to provide recommendations to developers when they write new source code, suggesting them when to "self-admit" design technical debt, or possibly when to improve the code being written. To achieve this goal, we have developed a machine learning approach named TEDIOUS (TEchnical Debt IdentificatiOn System), which leverages various kinds of method-level features as independent variables, including source code structural metrics, readability metrics and, last but not least, warnings raised by static analysis tools. We assessed TEDIOUS on data from nine open source projects for which there are available tagged self-admitted technical debt instances, also comparing the performances of different machine learners. Results of the study indicate that TEDIOUS achieves, when recommending self-admitted technical debts within a single project, an average precision of about 50% and a recall of 52%. When predicting cross-projects, TEDIOUS improves, achieving an average precision of 67% and a recall of 55%. Last, but not least, we noticed how TEDIOUS leverages readability, size and complexity metrics, as well as some warnings raised by static analysis tools.**

***Index Terms*—Self-Admitted Technical Debt; Recommender Systems; Static Analysis Tools.**

I. INTRODUCTION

Technical Debt (TD) refers to portions of a software system that are, from different perspectives, not in the most suitable shape yet. Cunningham defines TD as "not quite right code which we postpone making it right" [15]. In other words, TD could be code containing workaround, poorly structured or hard-to-read code, or even code that under some circumstances could turn out to be faulty. Technical debt has been related to various life cycle activities and various software artifacts, *e.g.,* requirements, design, code, test, or documentation [3], [22]. Previous studies have investigated developers perception of TD [17], [24]. Among other findings concerning TD introduction and management, such studies highlighted how very often TD introduction is intentional [24], but also that awareness constitutes an important problem in TD management [17].

Potdar *et al.* [16], [35] observed that very often, developers "self-admit" TD by commenting it with sentences explaining that the code is somewhat temporary and possibly requires more work in future. The presence of Self-Admitted Technical Debt (SATD) is not uncommon in software projects. Potdar *et al.* found that up to 31% of files contain SATD, and that such a SATD often remains in the source code for a long time; they also found that introducing SATD is something done more often by experienced developers than by non-experienced ones. This latter fact suggests that junior developers may possibly benefit from an appropriate support to decide when some code is not of good enough quality and should be documented as TD.

Previous work by Bavota and Russo [8] found that, in general, there is little correlation between SATD and code quality metrics such as Weighted Method Complexity (WMC), Coupling Between Objects (CBO), and Buse and Weimer Readability [10]. However, the scope of their study was to define a taxonomy of TD and the number of manually analyzed SATD was limited. Also, they computed metrics at class-level, where you could have methods with varying length, complexity, cohesion, coupling, or readability.

This paper proposes TEDIOUS (TEchnical Debt IdentificatiOn System), a machine learning approach that leverages various kinds of features extracted from source code as independent variables, and the knowledge of previous SATD (or SATD from other projects) as dependent variable to train machine learners able to recommend developers with "technical debt to be admitted". The purpose of such recommendations can be two-fold. First, it can be used to encourage developers to self-admit TD, especially since as Potdar *et al.* [35] have found, mainly experienced developers do that. Second, such recommendations could be used as an alternative (supervised, based on previous SATD) to smell detectors for identifying opportunities to improve source code.

TEDIOUS, differently from the study of Bavota and Russo [8], recommends SATD at method-level rather than at class-level. This is because we found that SATD-related comments are in most cases at method if not at block level. Second, we only consider certain types of TD, namely design debts. This is because design debts are the largest fraction [27]. Furthermore, requirement, documentation, defect or test debt would require a different analysis of further artifacts (which is in our future work agenda, but out of the scope of this paper).

We consider as features (i) a set of structural metrics extracted from methods, *e.g.,* LOC, McCabe cyclomatic complexity and number of parameters; (ii) the method's readability [10]; (iii) warnings produced by two static analysis tools, namely CheckStyle [1] and PMD [2]. As SATD knowledge, we use 9 Java open source projects from a previously-annotated dataset [27], from which we only consider design-related SATD. On such a dataset, we experimented with five different machine learners, performing SATD prediction within-project and cross-project.

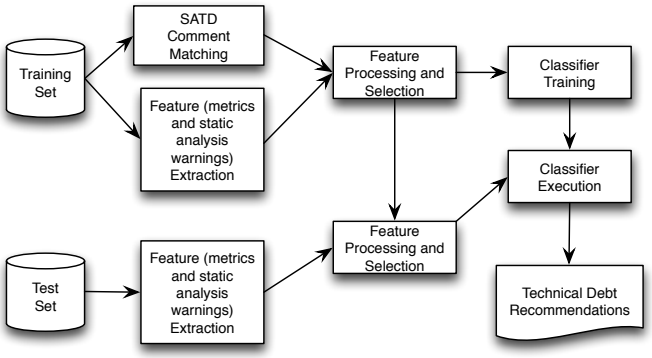

Fig. 1. Proposed approach for recommending SATD.

Our results indicate that TEDIOUS is able to recommend SATD with an overall within project prediction precision of (about) 50% and recall of 52%[1], where Random Forest is able to achieve the best performance. TEDIOUS is able to improve when working cross-projects, thanks to the availability of a larger training set. It achieves an average precision of 67%, recall 55%, and accuracy of 92%. This indicates its ability to be applied on new projects having little or no history of known SATD. We also provide evidence that SATD discovered by TEDIOUS are only weakly related to methods level anti patterns/code smells. Last, but not least, the analysis of feature importance highlights the major role played by different kinds of features, on the one hand readability, size, and complexity metrics, and on the other hand the output of static analysis tools.

II. The Approach

This section describes TEDIOUS, our proposed machine learning approach that classifies design TD using SATD as oracle. As mentioned in the introduction, the approach works at the granularity of methods, *i.e.,* it is able to determine whether a method could likely contain a design debt. This is because developers self-admit TD for methods or small blocks, rather than for entire classes [16], [35]. Overall, a class-level granularity would be too coarse to recommend problems similar to those that developers have previously self-admitted, plus it risks to mix different issues impairing accuracy and usefulness (*i.e.,* the TD could belong to any method).

TEDIOUS works as depicted in Fig. 1. It takes as input *labeled data* (training set), *i.e.,* programs in which TD has been self-admitted, and *unlabeled data* (test set) which could be for example source code under development, for which the approach should recommend whether or not to admit TD, or should point out source code worthwhile of being improved.

From the training set, on the one hand, we extract various kinds of metrics and static analysis warnings from source code. On the other hand, we identify SATD in order to build an oracle and train the model. After that, we preprocess the dataset, *i.e.,* we deal with multi-collinearity, perform feature selection and re-balancing. Finally, we use the preprocessed metrics and the oracle (*i.e.,*, the presence of a SATD in a method) to train our machine learners. At the same time, we extract the same features from the test set and perform the same filtering except re-balancing which is done only on the labeled data. Finally, we use the machine learning model previously built to classify the test set.

[1]Precision and recall computed on the positive instances, *i.e.,* those that should be recommended as SATD.

A. Features

To recognize TD, we employ three pieces of information, namely source code structural metrics, readability metrics, and warnings raised by static analysis tools. Overall, the rationale for combining source code structural metrics, readability metrics, and the output of static analysis tools is to capture (i) symptoms of complex, heavily coupled, poorly designed code (captured by structural metrics), (ii) symptoms of poorly documented code, likely, hard to read and understand (readability metrics), and (iii) more specific bad choices that could either influence the code maintainability and understandability, or that could potentially introduce defects (captured by static analysis tools). In the following, we describe in detail the metrics we extract and use.

1) Source Code Metrics: We use metrics easy to calculate but representative of size (*i.e.,* LOC, number of statements), coupling (*i.e.,* number of call sites), complexity (*i.e.,* McCabe cyclomatic complexity [31], number of defined variables, number of expressions, number of identifiers), and number of comments. Note: as it will be clearer in the study design, in the empirical evaluation we have removed the SATD-related comments, to avoid the approach becoming a self-prophecy. In total, we compute 9 source code metrics.

Source code metrics have been computed on an XML representation of the Java source code parse tree obtained through the *srcML* tool [13]. Concerning comments, we assign to a method any comment contained in the method body and any contiguous comment (block or line), *i.e.,* comment immediately preceding the method definition. We exclude from our analysis getter and setter methods, as they are irrelevant for the kind of TD we aim at detecting. This is done by (i) matching the method name prefix (*get* and *set*), and (ii) considering only as getters and setters methods having the desired prefix and made up at least of LOC long.

In addition to the aforementioned metrics, we consider a code readability metric (and related tool) proposed by Buse and Weimer [10]. This metric estimates code readability based on characteristics such as indentation, line length, identifier length, comment density, and the use of certain keywords and operators. We included the readability metric since, as reported in previous work by Bavota and Russo [8], code readability is one of the factors correlated with the introduction of a TD. A poor readability is likely to render the code difficult to understand and maintain [10].

2) Warnings Raised by Static Analysis Tools: Last, but not least, we consider warnings raised by static analysis tools. Indeed, static analysis tools check for a variety of poor practices. Each single, specific flagged practice, *e.g.,* unused

variables, is likely not related to a TD. However, one may wonder if too many flagged practices are justifiable or not, and if an excess of warnings may also be a symptom for a TD. To this aim, in this work we use two very popular static analysis tools able to identify warnings by analyzing source code, namely CheckStyle and PMD. CheckStyle [1] is a static analysis tool often used to check the adherence to coding standards, but also to identify pieces of code that are good candidates for code smells. CheckStyle performs its analyses using checks defined in a configuration file. In our study we used a default configuration file containing code styles defined by Oracle[2] and featuring 43 checks. PMD [2] is a source code analyzer able to find common programming flaws such as unused objects, unnecessary `catch` blocks, or incomprehensible naming. We use all 168 default PMD checks. We have chosen CheckStyle and PMD because of their adoption by OSS [9], because of their capability of providing a range of different kinds of warnings, related to code style and poor programming practices and, last but not least, because they can be executed on the source code without requiring compilation. Also for PMD and CheckStyle, we removed SATD comments when performing the analysis.

B. Identification of Self-Admitted Technical Debt

In order to train TEDIOUS, we need methods tagged as design debt. We use as tagging the presence of SATD [16], [35]. It is not the purpose of this paper to propose novel methods to identify SATD from information contained in comments, previous work has suggested approaches to identify SATD based on pattern matching [16], [35] or combining natural language processing (NLP) with machine learners [27]. Also, Maldonado *et al.* have published a corpus of annotated SATD from ten open source projects [27], which we use in our empirical evaluation.

C. Feature Preprocessing

First, we identify features that are strongly correlated, and for each group of such features we only keep one of them that better correlates with the dependent variable. To this aim, we use the R *varclus* function of the *Hmisc* package. Such a function produces a hierarchical clustering of features based on their correlation, in turn computed with a specified correlation measure (in this work we use the Spearman's ρ rank correlation measure). Then, we identify clusters by cutting the tree at a given level of ρ^2. In our case, we perform a cut for $\rho^2 = 0.64$, which corresponds to a strong correlation [12] (*i.e.,* $\rho = 0.8$).

After that, we remove features that in our dataset do not vary or vary too much, because they would not be useful to build a predictor. This is performed using the *RemoveUseless* filter implemented in Weka [21], which removes from a dataset, features that never vary as well as features with a percentage of variance above a threshold (we set to 99%). Also, we normalize metrics within each project dataset, especially because source code of different projects can be very different in terms of size and complexity, and we are interested to achieve a good cross-project prediction performance too.

Finally, we have to deal with the fact that in this work, the training set is unbalanced, *i.e.,* only a minority of the methods typically contain SATD. To balance the training set, we could either under sample the majority class (*i.e.,* methods not containing SATD), or over-sample the minority class by generating artificial instances from the existing ones. The latter is more suitable when the number of instances of the minority class is very small, as under sampling would result in a very small training set. In our work we apply the SMOTE (Synthetic Minority Over-sampling Technique) [11] implementation available in Weka to perform over-sampling.

D. Building and Applying Machine Learning Models

After preprocessing has been performed, we build the machine learning models on the training set and use them to perform predictions on the test set. In this work we experiment with five different kinds of machine learners implemented in Weka [21], namely Decision Trees (J48), Bayesian classifiers, Random Forests, Random Trees, and Bagging with Decision Trees. In this work, we used such machine learners with their default configuration provided by Weka.

III. Study Definition and Execution

The *goal* of this study is to assess the performance of the proposed approach in recommending SATD. The *quality focus* is source code quality, and in particular understandability and maintainability that could possibly be improved by keeping track of existing TD (by self-admitting it) in order to improve it afterwards. The *perspective* is that of researchers interested in developing and evaluating approaches able to suggest to developers when to admit TD. The study aims at addressing the following research questions:

- **RQ**$_1$: *How does* TEDIOUS *work for recommending SATD within-project?*
- **RQ**$_2$: *How does* TEDIOUS *work for recommending SATD across-project?*
- **RQ**$_3$: *How would a method-level smell detector compare with* TEDIOUS*?*

A. Dataset

To evaluate the proposed approach, we rely on a dataset containing validated SATD [27]. The dataset consists of ten open source projects, for which SATD has been identified using a NLP approach [27] and manually validated. From such dataset, we had to exclude the EMF project for which we were unable to download the considered source code release (2.4.1). Table I summarizes the characteristics of the dataset, namely project releases analyzed, their number of files, classes, methods, number of comment blocks attached to files, classes and methods, number and percentage of methods containing a design SATD. Some differences, *e.g.,* number of classes, methods or comments, could be observed when comparing figures in Table I and those reported by Maldonado *et al.* [27].

[2]www.oracle.com

TABLE I
CHARACTERISTICS OF THE STUDIED PROJECTS.

Project	Release	Number of				Number of Comments	Number of Design SATD		% of Methods
		Files	Classes	Methods	Comments	$\in$ Methods	$\notin$ Methods	$\in$ Methods	with design SATD
Ant	1.7.0	1,113	1,575	11,052	20,325	13,359	1	57	0.5%
ArgoUML	0.34	1,922	2,579	14,346	64,393	17,722	203	425	2%
Columba	1.4	1,549	1,884	7,035	33,415	10,305	8	418	5%
Hibernate	3.3.2 GA	2,129	2,529	17,405	15,901	9,073	21	377	2%
jEdit	4.2	394	889	4,785	15,468	10,894	6	77	2%
jFreeChart	1.0.19	1,017	1,091	10,343	22,827	15,412	4	1,881	18%
jMeter	2.1	1,048	1,328	8,680	19,721	12,672	95	424	5%
jRuby	1.4.0	970	2,063	14,163	10,599	7,809	16	275	2%
Squirrel	3.0.3	2,325	4,123	16,648	25,216	15,574	35	173	1%

We attribute these differences to the different tooling, tools characteristics and processing. In our processing we did not made a distinction between a class and its inner classes and the total number of classes includes the interfaces. However, this does not directly affect our work, as we were only interested to trace method-level SATD.

As it can be seen from Table I, and as explained in Section II, there is a clear prevalence of method-related SATD than of class-level SATD. Table I also clearly shows that the classification problem (SATD prone versus non SATD prone) is highly unbalanced. As it can be noticed, besides JFREECHART, where the percentage of design SATD methods is 18%, in all other cases it is 5% or below.

To build the oracle, we started from the replication package of Maldonado *et al.* [27]. Since the dataset reports SATD at file level, we attributed SATD to methods by matching the SATD string onto comments attached to methods. If the SATD is matched onto a comment contained in a class, but not attached to any method, then it is assigned to the class, while comments outside the class are attached to the file. In any case, both file and class-level SATD (a minority, as it can be noticed from Table I) are not considered in our study.

B. Analysis Method

To address $\mathbf{RQ}_1$ we performed a 10-fold cross validation for SATD of each project. Namely, we train every time the approach on 9/10 of the project methods, and we test on the remaining 1/10. To limit the effect of the randomness, the process is repeated 10 times, and performance indicators are averaged over the 10 iterations. For $\mathbf{RQ}_2$, instead, we train the approach every time on 8 projects and test it on the remaining one.

To assess the performance of TEDIOUS we use standard metrics in automated classification, namely, precision, recall, and F_1 score computed for the SATD category, *i.e.,* for methods classified as SATD with respect to the true SATD methods.

Precision (Pr) is defined as the percentage of methods predicted as having SATDs that are correct with respect to our oracle, *i.e.,* $Pr = TP/(TP + FP)$, where TP and FP are the number of true and false positives respectively. Recall (Rc) is the percentage of SATD methods in the oracle that the approach is able to retrieve, *i.e.,* $Rc = TP/(TP + FN)$. Finally, the F_1 score is the harmonic mean between precision and recall: $F_1 = 2 \cdot (Pr \cdot Rr)/(Pr + Rc)$.

Neither of the three metrics described above takes into account true negatives (because they are computed for the true SATD class only) therefore it is important to complement the results' evaluation using other metrics, namely, accuracy, Matthews Correlation Coefficient (MCC) [30] and the Area under the Receiving Operating Characteristics (ROC) Curve (AUC).

Accuracy (Acc) is the percentage of methods correctly predicted among the total number of methods analyzed: $Acc = (TP + TN)/(TP + TN + FP + FN)$.

The MCC is a measure used in machine learning to assess the quality of a two-class classifier especially when the classes are unbalanced [30]. It ranges between -1 an 1 (0 means that the approach performs like a random classifier). It is defined as:

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(FN + TN)(FP + TN)(TP + FN)}}$$

As for other correlation measures, $MCC < 0.2$ indicates a low correlation, $0.2 \leq MCC < 0.4$ a fair, $0.4 \leq MCC < 0.6$ a moderate, $0.6 \leq MCC < 0.8$ a strong, and MCC ≥ 0.8 a very strong correlation [12].

The ROC [36] curve represents the true positive rate against the false positive rate at various classifier thresholds. The higher the area below the ROC, the more the classifier will be better than a random classifier, which has an AUC=0.5.

Intuitively, we prefer a balancing between both precision and recall, as in practical settings we want an approach that is able to alert developers whenever in their code they omit to admit a TD. Moreover, we cannot use only the F_1 score as an indicator since we want to reduce the possibility of classifications that occurred by chance, and for this reason we also report and discuss MCC and AUC values.

In addition to the aforementioned performance indicators, we provide indications about the importance of the considered metrics. In our case we use a specific technique implemented by Weka for Random Forests (which in our study outperform the other classifiers) named Mean Decrease Impurity (MDI) [25], which measures the importance of variables on an ensemble of randomized trees.

To address $\mathbf{RQ}_3$ we compute and report the performance of a smell detector, namely DECOR [32], in classifying as TD methods labeled as SATD. Most of the smells defined by DECOR (and by most of the existing smell detectors) are at class level, therefore we limit the attention to method-level

TABLE II
AVERAGE PERFORMANCE OF DIFFERENT MACHINE LEARNERS FOR WITHIN-PROJECT PREDICTION.

Without Balancing						
ML	**Pr**	**Rc**	**F_1**	**Acc**	**MCC**	**AUC**
Random Forests	49.97	52.19	47.15	93.32	0.47	0.92
Bagging	51.91	48.45	45.97	93.35	0.45	0.92
Bayesian	24.29	78.77	34.18	89.01	0.38	0.93
j48	34.86	54.42	39.54	94.18	0.39	0.82
Random Trees	23.09	52.49	29.96	90.35	0.30	0.73
With Balancing						
ML	**Pr**	**Rc**	**F_1**	**Acc**	**MCC**	**AUC**
Random Forests	26.56	68.26	36.04	90.45	0.37	0.92
Bagging	18.4	75.12	28.24	85.58	0.31	0.90
Bayesian	4.00	94.07	7.55	15.66	0.04	0.72
j48	16.95	77.76	26.45	84.04	0.30	0.85
Random Trees	16.03	63.22	24.49	85.34	0.26	0.75

smells, namely *Long Method* and *Long Parameter List*. The rules DECOR uses for identifying a Long Method or a Long Parameter List are *LOC*$> th_1$ and *ParNbr*$> th_2$, where th_1 and th_2 are two thresholds on the LOC and on the number of parameters respectively. In our study, we consider th_1 and th_2 varying along different percentiles of LOC and number of parameters, belonging in the range $[0.5 - 0.95]$. Also, we consider DECOR's default thresholds, namely percentile 0.75 for LOC and outlier for long parameter list, *i.e.,* third quartile $+1.5 \cdot IQR$ (interquartile range).

Finally, we qualitatively discuss some examples of false positives and false negatives, explaining to what extent TEDIOUS is limited in its capability to recommend any kind of design TD.

IV. STUDY RESULTS

This section reports the study results, addressing the research questions formulated in Section III.

A. How does TEDIOUS *work for recommending SATD within-project?*

Table II highlights the average performance computed by executing 10 times the 10-fold cross validation for each system in our dataset, using different machine learners' algorithms, and with or without balancing the training set with SMOTE.

The Random Forest classifiers executed on an unbalanced dataset achieve the best balancing between average precision (49.97%) and average recall (52.19%), with an average F_1 score of 47.15%. The accuracy (which keeps into account the classification of negatives) is 93.13%, and it is in all cases (except for Bayesian classifiers with balancing) above 80%. Moreover, MCC has an average value > 0.4 (moderate correlation), and the AUC in three cases (Random Forests, Bagging and Bayesian) is > 0.9 while for j48 and Random Trees it is > 0.7.

When using a balanced training set, as expected, we have a decrease in terms of precision and an increase of the recall. Random Forest classifiers are the ones that work better also using a balanced dataset (as shown in the bottom part of Table II). It exhibits an average F_1 score of 36.04% with a moderate correlation (MCC=0.47) regarding the predicted instances and AUC=0.92. The Bayesian classifiers are the ones which exhibit the worst performance if used on a completely balanced dataset. Indeed, the MCC value is $\simeq 0$ which means that the predictions are perfectly in-line with a random classifier. The other classifiers we considered have a precision that ranges in the interval $[16.03\% - 51.91\%]$, an Rc $\in [52.49\% - 78.77\%]$, with an F_1 score $\simeq 30\%$. Moreover, the MCC coefficient is always above 0.30, and AUC> 0.7, meaning that our predictions reach a fair to moderate correlation and perform better than a random classifier (which would have MCC=0 and AUC=0.5).

TABLE III
WITHIN-PROJECT PREDICTION: RESULTS OF RANDOM FORESTS FOR EACH SYSTEM, WITHOUT AND WITH SMOTE BALANCING.

Without Balancing						
System	**Pr**	**Rc**	**F_1**	**Acc**	**MCC**	**AUC**
Ant	0.91	16.39	1.73	84.59	0.00	0.77
ArgoUML	85.19	38.10	52.65	93.25	0.54	0.91
Columba	36.40	65.94	46.91	96.02	0.47	0.94
Hibernate	53.44	65.22	58.74	96.80	0.57	0.97
jEdit	5.24	25.71	8.71	85.51	0.06	0.81
jFreeChart	84.58	82.52	83.54	98.91	0.83	0.99
jMeter	53.38	47.37	52.30	96.69	0.51	0.94
jRuby	52.27	84.02	64.45	94.21	0.64	0.97
Squirrel	73.33	44.44	55.35	99.51	0.57	0.97
With Balancing						
System	**Pr**	**Rc**	**F_1**	**Acc**	**MCC**	**AUC**
Ant	2.46	44.26	4.67	85.02	0.08	0.83
ArgoUML	47.03	65.39	54.71	89.34	0.50	0.90
Columba	15.35	74.64	25.46	88.35	0.30	0.94
Hibernate	19.85	89.13	32.47	87.04	0.38	0.95
jEdit	7.74	34.29	12.63	87.25	0.11	0.86
jFreeChart	62.98	92.68	75.00	97.94	0.75	0.99
jMeter	32.03	64.47	42.79	93.4	0.42	0.92
jRuby	32.75	91.91	48.29	87.72	0.50	0.92
Squirrel	18.81	57.58	28.36	98.02	0.32	0.96

Table III reports detailed results per system, considering Random Forest classifiers only, *i.e.,* the best performing machine learner. The top part of the table reports results without balancing. First of all, it is important to highlight that, except for JFREECHART for which we have a percentage of methods with SATD in the oracle that is 18%, for the other eight systems we have very few methods containing SATD compared to the total number of methods analyzed ($\leq 5\%$). In particular, for ANT we have 0.5% of methods with SATD in the dataset. This explains the low prediction performances achieved for this system (without balancing we obtain a precision of 0.91% and a recall of 16.39%), and even balancing does not help so much (precision 2.46% and recall 44.26%), because data from the few SATDs is not even able to act as a seed for an artificial training set. To some extent, the same happens on JEDIT (precision 5.24% and recall 25.71%). For such systems, even if the AUC is > 0.7, we obtain a very low MCC, close to zero (random classifier). At the same time, TEDIOUS works quite well on the other systems. We can note that for JFREECHART, we obtain high precision and recall, (84.58% and 82.52% respectively) and a very high MCC (0.8) and AUC (0.99). Moreover, TEDIOUS is able to work well on the other five systems in our dataset,

TABLE IV
TOP 10 DISCRIMINANT FEATURES (WITHIN-PROJECT PREDICTION). (M): SOURCE CODE METRICS, (CS): CHECKSTYLE CHECKS, (P): PMD CHECKS.

Metric Name	Ant	ArgoUML	Columba	Hibernate	jEdit	jFreeChart	jMeter	jRuby	Squirrel
Readability (R)	5	1	2	1	1	1	1	1	1
LOC (M)	2	2	5	2	2	3	2	3	4
DeclNbr (M)	4	3	7	4	3	4	3	4	3
ParNbr (M)	8	5	9	7	7	7	7	7	7
ExprStmtNbr (M)	6	4	—	5	4	5	5	5	5
McCabe (M)	10	7	—	6	6	6	6	6	6
CommentNbr (M)	—	6	—	3	5	2	4	2	2
LineLength (CS)	—	—	—	9	—	—	9	8	9
LocalVariableCouldBeFinal (P)	—	—	—	10	9	—	—	9	10
DataflowAnomalyAnalysis (P)	—	10	—	—	10	—	—	—	—
FinalParameters (CS)	—	—	—	—	—	8	8	—	—
MissingSwitchDefault (CS)	—	8	4	—	—	—	—	—	—
AvoidReassigningParameters (P)	7	—	—	—	—	—	—	—	—
CollapsibleIfStatements (P)	9	—	—	—	—	—	—	—	—
EmptyIfStmt (P)	—	—	8	—	—	—	—	—	—
IfStmtsMustUseBraces (P)	—	—	—	—	8	—	—	—	—
LeftCurly (CS)	—	—	—	—	—	—	—	—	8
LocalVariableName (CS)	—	—	1	—	—	—	—	—	—
MethodArgumentCouldBeFinal (P)	—	—	—	—	—	—	—	10	—
MethodLength (CS)	—	—	—	—	—	—	10	—	—
OptimizableToArrayCall (P)	—	—	10	—	—	—	—	—	—
ParameterNumber (CS)	—	—	—	—	—	10	—	—	—
ParenPad (CS)	—	—	—	8	—	—	—	—	—
ShortVariable (P)	—	—	—	—	—	9	—	—	—
SimplifyBooleanReturns (CS)	—	9	—	—	—	—	—	—	—
SwitchStmtsShouldHaveDefault (P)	—	—	6	—	—	—	—	—	—
UselessParentheses (P)	3	—	—	—	—	—	—	—	—
UseLocaleWithCaseConversions (P)	—	—	3	—	—	—	—	—	—
UseStringBufferForStringAppends (P)	1	—	—	—	—	—	—	—	—

namely JRUBY, HIBERNATE, SQUIRREL, ARGOUML and JMETER with an $F_1 > 50\%$, an $MCC \in [0.51 - 0.64]$, and $AUC \in [0.91 - 0.97]$. Differently from ANT, and even if SQUIRREL has only 1% of SATD methods, TEDIOUS is able to correctly classify them with an average precision of 73.33% and an average recall of 44.44%.

When looking at results with balancing (bottom part of the table), we can notice that, in general, the performances are in-line with the ones obtained without balancing, but slightly lower. For example, the F_1 score varies in the range $[4.67\% - 54.71\%]$.

Table IV reports the top-10 features ranked, for each system in terms of MDI. For each system we indicate the ranking of each feature. The first four features, in the top-10 for all projects, are source code metrics. Interestingly, the most important metric is the Buse and Weimer's readability metric [10], which is the most important metric in 7 projects, the 2nd-most important in COLUMBA and ranked 5th for ANT. On the one hand, this contrasts with findings of previous work [8], but we recall that we have performed the analysis at method-level, since a class could contain very readable methods and other methods (tagged as SATD) with poor readability. Other important metrics are the number of declarations (*DeclNbr*), of parameters (*ParNbr*), and, as expected, the LOC. The last two metrics are particularly relevant since they are often used by smell detectors, and in $\mathbf{RQ}_3$ we will investigate the extent to which they are sufficient to perform a prediction without other indicators. Other features appearing in 7-9 systems are the number of expressions (*ExprStmtNbr*), the (again, expected) McCabe cyclomatic complexity, the number of comments (*CommentNbr*, which, again excludes the SATD comment blocks that would have biased the prediction), and the CheckStyle *LineLength* (*i.e.,* line too long) warning.

Other features, mostly CheckStyle and PMD warnings play a role in very specific systems, and many of them relate to poorly written code (*e.g., LocalVariableName* is related to single-character variables or local variables with the same name in different scopes; *LocalVariableCouldBeFinal* if assigned only once). Due to the lack of space, we refer to the CheckStyle[3] and PMD[4] documentation for further details on their meaning.

Among the top-10 features, we can also notice the *ParameterNumber* and *MethodLength* CheckStyle warnings. Intuitively, they could be correlated with two of our top features, *ParNbr* and LOC respectively. However, they are boolean features telling whether the method length or number of parameters are too high according to CheckStyle-defined thresholds (150 LOC and 7 parameters by default, respectively). Their outcome was not correlated with *ParNbr* and LOC, and therefore not removed by the Spearman's analysis. In any case, they are in the top-10 in one system each.

[3]http://checkstyle.sourceforge.net/checks.html

[4]https://pmd.github.io/pmd-5.5.5/pmd-java/rules/index.html

TABLE V
AVERAGE PERFORMANCE OF DIFFERENT MACHINE LEARNERS FOR CROSS-PROJECT PREDICTION.

Without Balancing						
ML	**Pr**	**Rc**	**F_1**	**Acc**	**MCC**	**AUC**
Random Forests	67.22	54.89	55.43	91.89	0.55	0.91
Bagging	58.85	58.50	52.46	91.27	0.52	0.88
Bayesian	49.25	64.35	48.18	89.11	0.47	0.85
j48	48.51	62.47	47.18	89.22	0.46	0.78
Random Trees	48.31	51.62	45.35	90.14	0.43	0.74
With Balancing						
ML	**Pr**	**Rc**	**F_1**	**Acc**	**MCC**	**AUC**
Random Forests	47.49	78.75	56.45	89.52	0.52	0.89
Bagging	28.42	83.17	38.91	75.25	0.31	0.86
Bayesian	15.68	98.04	23.84	21.70	0.06	0.83
j48	35.73	83.41	46.89	83.85	0.43	0.82
Random Trees	31.49	63.21	36.87	80.76	0.30	0.76

RQ_1 summary: Random Forest classifiers achieve the best performance in recommending design TD, with average precision of 49.97% and recall of 52.19%. Except for a couple of projects with a very limited percentage of SATD instances, in all other cases the high values of MCC and AUC indicate healthy classifiers. While balancing increases recall, it does not necessarily represent a convenient choice due to the substantial precision decrease. Code readability, complexity and size play a major role in the prediction, along with some system-specific static analysis checks.

B. How does TEDIOUS *work for recommending SATD across-project?*

To answer $\mathbf{RQ}_2$, we evaluate the performance of TEDIOUS in cross-project predictions. The average performances of each machine learner are reported in Table V without balancing (top part of the table) and with SMOTE balancing (bottom). By looking at the table, we can immediately notice that, on the one hand, results are consistent with those of $\mathbf{RQ}_1$, because (i) Random Forests outperform other machine learners, and (ii) there is no strong payoff in performing re-balancing with SMOTE. Specifically, the Random Forest classifiers achieve an average precision of 67.22% and an average recall of 54.89%, vs. within-project precision 49.97% and recall 52.19%. Also other classifiers exhibit good overall performance with a precision between 48.31% (Random Trees) and 58.85% (Bagging), and a recall between 51.62% (Random Trees) and 64.35% (Bayesian classifiers). MCC is always above 0.4 (moderate), and AUC above 0.7. Also in this case, balancing allows to increase recall at the cost of precision. Also in this case the machine learner achieving the best compromise between precision and recall is the Random Forests classifier, with a precision of 47.49%, a recall of 78.75%, and a F_1 score of 56.45% (slightly better than without balancing).

Table VI reports detailed results per system, considering the best performing machine learner, *i.e.,* Random Forests. The top part of the table reports results without SMOTE balancing. We have only 3 systems for which TEDIOUS is not able to reach an F_1 score greater than 37%, namely JRUBY, JEDIT and ANT. Again, these are systems with a low percentage of SATD methods ($< 2.15\%$). As for within-project prediction, an exception is represented by SQUIRREL, for which, despite the fact that only 1.42% of methods have SATD, TEDIOUS achieves a precision $\simeq 50\%$ and a recall of 70.62%. Finally, there are 2 systems, JFREECHART and ARGOUML, for which TEDIOUS achieves an F_1 score $> 91\%$ with MCC values of 0.94 and 0.85 respectively, *i.e.,* very strong correlation.

TABLE VI
CROSS-PROJECT PREDICTION: RESULTS OF RANDOM FORESTS FOR EACH SYSTEM, WITHOUT AND WITH SMOTE BALANCING.

Without Balancing						
System	**Pr**	**Rc**	**F_1**	**Acc**	**MCC**	**AUC**
Ant	27.94	53.52	36.71	98.23	0.38	0.97
ArgoUML	94.46	88.29	91.27	92.72	0.85	0.98
Columba	67.84	43.88	53.29	92.19	0.51	0.92
Hibernate	72.84	52.10	60.75	96.74	0.60	0.95
jEdit	35.90	24.78	29.32	96.55	0.28	0.91
jFreeChart	94.89	95.98	95.43	98.05	0.94	0.99
jMeter	70.51	59.76	64.69	95.55	0.63	0.91
jRuby	91.89	5.11	9.69	58.32	0.15	0.75
Squirrel	48.75	70.62	57.86	98.63	0.58	0.97
With Balancing						
System	**Pr**	**Rc**	**F_1**	**Acc**	**MCC**	**AUC**
Ant	13.56	71.83	22.82	95.34	0.30	0.96
ArgoUML	89.74	92.65	91.18	92.27	0.84	0.96
Columba	49.01	69.06	57.33	89.56	0.53	0.94
Hibernate	52.61	68.87	59.66	95.49	0.58	0.95
jEdit	20.70	57.52	30.44	92.42	0.31	0.72
jFreeChart	84.85	96.81	90.44	95.67	0.88	0.98
jMeter	46.05	79.05	58.19	92.25	0.57	0.91
jRuby	50.50	93.10	65.48	57.09	0.28	0.64
Squirrel	20.42	79.90	32.53	95.61	0.39	0.93

Therefore, results tell us that, except for some systems with a very limited percentage of SATD, cross-project prediction is surprisingly very beneficial. This can be explained because, when performing within-project prediction, TEDIOUS has to learn from a fairly limited number of SATD. SMOTE re-balancing does not help a lot because the very limited samples are not even enough to play a role as a seed for the generation of artificial samples, and, also, static analysis warnings have a sparse and often boolean nature for a method (either zero or one), making SMOTE not very appropriate for them.

Looking at the bottom part of Table VI, reporting results with SMOTE balancing, we can notice that performances are in-line with the ones obtained when not applying the balancing, even if they are generally lower. Indeed, in this case the maximum F_1 score is 91.18% using ARGOUML instances as testing set. The only project for which re-balancing turns out to be very beneficial is JRUBY. The balancing helped to improve the performances: the F_1 score increased up to 65.48% with a precision of 50.50% and a recall of 93.10%. This is because without re-balancing JRUBY exhibited a very high precision (91.89%) but a very poor recall (5.11%). In this case, re-balancing brought the recall very high while keeping the precision still acceptable.

Table VII reports the top-10 features for cross-project prediction, ranked by MDI. Results are in-line with respect

TABLE VII
TOP 10 DISCRIMINANT FEATURES (CROSS-PROJECT PREDICTION). (M): SOURCE CODE METRICS, (CS): CHECKSTYLE CHECKS, (P): PMD CHECKS.

Metric	Ant	ArgoUML	Columba	Hibernate	jEdit	jFreeChart	jMeter	jRuby	Squirrel
Readability (M)	1	1	1	1	1	1	1	1	1
LOC (M)	2	2	3	2	2	2	2	2	2
CommentNbr (M)	7	3	4	3	3	4	4	3	3
DeclNbr (M)	4	4	2	4	4	3	3	4	4
ExprStmtNbr (M)	5	5	5	5	5	5	5	5	5
McCabe (M)	6	6	6	6	6	6	6	6	6
ParNbr (M)	3	7	7	7	7	7	7	7	7
LocalVariableCouldBeFinal (P)	10	9	9	8	10	8	10	10	8
MethodArgumentCouldBeFinal (P)	—	10	10	10	8	9	8	8	7
FinalParameters (CS)	8	—	8	9	9	—	8	8	8
LineLength (CS)	9	8	—	—	—	10	—	—	10

to within-project prediction, and the contribution of the same features we discussed before is becoming clearer. Readability plays again (and always) the most important role, followed by LOC and *CommentNbr*. The number of parameters (ParNbr) becomes less important than other metrics capturing the code size and complexity, such as *DeclNbr* and *ExprStmptNbr*. The four important checks are the PMD and CheckStyle ones related to declaring final variables, parameters not being reassigned, and the *LineLength*. Also in this case we notice the presence of potentially related variables, *i.e.,* both CheckStyle's *FinalParameters* and PMD's *MethodArgumentCouldBeFinal*. They are different because the latter recommends `final` only if the parameter is never re-assigned, while the former does not. For such a reason, the variables did not correlate enough and they were both retained in the model.

RQ$_2$ summary: When recommending SATD, cross-project prediction is able to increase the performance of predictors because of the larger and diverse training set available. Features related to code readability, size and complexity play a major role in the recommendation of design SATD.

C. How would a method-level smell detector compare with TEDIOUS?

Table VIII reports the performance of a prediction performed, instead of using our approach, by relying on DECOR Long Method and Long Parameter List smell detectors, and by considering the union of both, *i.e.,* by recommending a TD if either a Long Method or a Long Parameter List has been detected. As explained in Section III-B, we report results for thresholds at different percentiles of LOC and number of parameters, as well as the DECOR's default thresholds. When combining the smells, for the sake of simplicity we consider the same threshold for both.

As the table shows, even with different thresholds, smell detection is never able to achieve performances comparable to TEDIOUS. That is, both precision and recall are low, the F_1 score is always below 22% and the MCC below 0.20 (low correlation). Not surprising, the most suitable "trivial" predictor is the Long Method, which achieves a precision $\in [7.76\% - 24.50\%]$ and a recall $\in [55.18\% - 18.48\%]$.

RQ$_3$ summary: While LOC and number of parameters play an important role for machine learners, DECOR Long Method and Long Parameter List smell detectors are not able to achieve performances comparable to TEDIOUS.

D. Discussion of false positives and false negatives

In the following we discuss some examples, among 100 we manually inspected, to explain cases in which TEDIOUS correctly classified or misclassified SATD.

As an example of true positives of different nature, in ARGOUML there are two methods *createFlow* in class `CoreFactoryEUMLImpl` and *invokeFeature* in class `ModelAccessModelInterpreter` both having a labeled design SATD, characterized by different source code metrics. More in detail, the first one has a $Readability \approx 1$, only two lines of code and $McCabe = 1$, while the second one has a $Readability = 0$, is made up of 755 lines of code with a complexity of 178. TEDIOUS is able to correctly identify both of them, possibly because of multiple trees produced by the Random Forests ensemble classifier.

As an example of false positive, the method *initialize* of the class `RE` (regular expression) of JEDIT has 511 LOC, 5 parameters, 197 calls, 32 declarations, 618 expressions, 97 comments, $McCabe = 102$ and, unsurprisingly, $Readability = 0$. It is clearly classified as SATD while it is not, because it is a piece of intrinsically complex code, which may or may not be improved by developers. At the same time, SATD recommendations for such unreadable, complex and long methods should not be totally annoying and worthless.

As an example of false negative, the method *start* in class `ColumbaServer` of COLUMBA has a design-SATD comment, immediately after an "if" condition, saying that *"something is very wrong here"*. However, if one looks at the method structure, there is nothing that could be considered as a symptom of the TD presence. In summary, there are cases for which the SATD could not be properly detected using only structural and source code metrics. This is a limitation for TEDIOUS' applicability.

TABLE VIII
OVERALL DECOR PERFORMANCES IN PREDICTING SATD (THE LAST LINE REPORTS RESULTS FOR DEFAULT THRESHOLDS).

Percentile	Long Method (LM)					Long Parameter List (LPL)					LM ∪ LPL				
	Prec.	Rec.	F_1	Acc.	MCC	Prec.	Rec.	F_1	Acc.	MCC	Prec.	Rec.	F_1	Acc.	MCC
0.50	7.76	55.18	13.60	54.01	0.05	11.93	43.91	18.76	75.06	0.12	7.93	68.28	14.21	45.91	0.06
0.55	8.31	53.53	14.38	58.19	0.06	11.93	43.91	18.76	75.06	0.12	8.35	67.80	14.87	49.09	0.08
0.60	8.47	49.26	14.46	61.77	0.06	11.93	43.91	18.76	75.06	0.12	8.75	67.14	15.48	51.89	0.09
0.65	8.88	46.86	14.93	64.98	0.07	11.93	43.91	18.76	75.06	0.12	9.07	65.97	15.94	54.36	0.10
0.70	9.83	43.70	16.05	70.01	0.08	11.93	43.91	18.76	75.06	0.12	9.56	63.71	16.62	58.07	0.11
0.75	11.36	40.41	17.74	75.41	0.11	11.93	43.91	18.76	75.06	0.12	10.27	61.88	17.61	62.02	0.12
0.80	12.59	36.66	18.74	79.15	0.12	17.62	33.30	23.05	85.41	0.17	12.74	53.53	20.58	72.89	0.15
0.85	14.55	31.72	19.95	83.30	0.13	17.62	33.30	23.05	85.41	0.17	14.14	50.77	22.11	76.54	0.17
0.90	15.74	23.62	18.89	86.69	0.12	13.52	12.58	13.03	88.99	0.07	14.16	31.76	19.58	82.89	0.13
0.95	24.50	18.48	21.07	90.92	0.17	14.91	7.09	9.61	91.25	0.06	19.58	22.59	20.98	88.83	0.15
Default	11.36	40.41	17.73	75.41	0.11	17.62	33.29	23.04	85.41	0.17	11.58	54.69	19.12	69.64	0.13

V. THREATS TO VALIDITY

In this section, we discuss the threats to validity of our study following the guidelines for case study research [41].

Construct validity threats concern the relationship between theory and observation. In this study construct validity threats are mainly due to measurement errors of labeled design-SATD and of metrics. As for the SATD, we use the annotated dataset of Maldonado *et al.* [27] to build our oracle. However, as explained in Section III-A, we had to match the SATD pattern onto methods, as method information was not available in the dataset. This introduced the possibility of imprecision. We are aware that not all comments in the dataset [27] were matched in the code. We attribute this to the different processing chain. However, this only implies we are making the task more difficult as the process creates false negatives. Indeed, methods that should be tagged as SATD prone, but that are not, if properly traced, would make the dataset more balanced, likely leading to improved results. As future work we plan to manually revise those cases where a non perfect match exists.

Internal validity threats concern factors internal to our study that could have influenced our results. As explained in Section II-D, machine learners have been applied with default Weka settings, therefore it is possible that better results could be obtained through an appropriate calibration. However, this simply means that in the worst case our results represent a lower-bound. Also, for CheckStyle we used default configuration rules provided by Oracle (available with the CheckStyle tool), and default rules for PMD as well. Developers have the capability of customizing such rules, and an appropriate customization could have produced a different set of warnings and, possibly, increased the usefulness of the checks for the prediction. In future work we plan to use SATD to help customizing CheckStyle and PMD rules as well.

Source code metrics (except for readability, for which we used the tool by Buse and Weimer [10]) have been computed using *srcML* [13], and therefore it is possible that alternative metric extractors could produce different results. Besides that, our aim is not to show how TEDIOUS works with specific metric detectors and with specific static analysis tools, but to highlight the general potential of learning SATD recommendations from source code features.

As explained in Section III-A, we skipped the EMF project from the Maldonado *et al.* [27] dataset as we could not download the archive of release 2.4.1. However, we believe this would not bias our results because, despite EMF is the largest system of the Maldonado *et al.* [27] in term of LOC, it is comparable to ARGOUML, HIBERNATE and COLUMBA in terms of number of classes, and smaller than SQUIRREL.

Conclusion validity threats concern the relationship between the treatments and the outcome. We report results using appropriate diagnostics for the machine learner performances (AUC and MCC) and for evaluating the importance of features (MDI). Then, when discussing findings we keep into account ranges of acceptability of AUC and MCC (*i.e.,* AUC should be larger than 0.5 and MCC should be in the positive range).

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details needed to replicate our study. We plan to share our full replication package comprising source code, raw data and scripts.

External validity threats concern the possibility of generalizing our results. Despite the fact we used the same systems of the previous study [27], and even though the 9 systems cover different domains, we cannot be guaranteed that the finding generalize to the universe of Java programs. We report a consistency evaluation across the 9 systems, however more studies are needed to verify to what extent our findings generalize to other projects, domains, or programming languages.

VI. RELATED WORK

The main subjects related to this paper can be divided in four sections: relationship between technical debt and source code metrics, self-admitted technical debt, code smell detection, and the use of static analysis tools in software projects.

A. Relationship Between Technical Debt and Source Code Metrics

Several authors have tried to relate source code metrics to technical debt, and specifically to design and code debt. Marinescu proposed to identify design debt using a rule-based approach leveraging source code metrics capturing coupling, cohesion and encapsulation [29]. Griffith *et al.* [20] empirically compared the detection of technical debt performed by five different methods and tools, finding that while some of such methods produced correlated results, they do not exhibit a strong relationship with software quality attributes of the

QMOOD quality model [7]. Also, Arcelli Fontana *et al.* [5] studied how different tools detect technical debt, finding that there is limited agreement among tools, and also that tools ignore some pieces of information, *e.g.,* the change history. Arcelli Fontana *et al.* [4] focused on the impact of design smells on code debt, in order to give advices about which design debt should be prioritized for refactoring actions.

B. Self-Admitted Technical Debt

Potdar and Shihab [35] investigated TD in the source code of open source projects and observed that developers often "self-admit" technical debt by commenting it with sentences explaining that the code is somewhat temporary and may require rework in future. Maldonado and Shihab [16] developed an approach that leverages developers' comments to identify instances of SATD in the code. The proposed approach is based on pattern matching and classifies SATD into five types: design, defect, documentation, requirement and test. Bavota and Russo [8] conducted a large-scale empirical study of self-admitted technical debt in open source projects, investigating the diffusion and evolution of SATD, the actors involved in SATD management, and the relation between SATD and code quality metrics. They observed that SATD are prevalent in software projects and survive for long time. Similarly to previous work by Griffith *et al.* [20], they found no clear relation between SATD and some software quality metrics, namely WMC, CBO, and Buse and Weimer Readability metrics. Wehaibi *et al.* [40] examined the relation between SATD and fault-proneness, finding that there is no strong relation between the occurrence of SATD and the fault-proneness of a system. More recently, Maldonado *et al.* [27] used NLP techniques to detect SATD. Comments were classified into different types of technical debt and were used to train a maximum entropy classifier, which turned out to outperform the previous pattern matching approach by Potdar and Shihab [35]. In this paper we do not propose a new approach to identify SATD using information from developers' comments. Instead, we leverage structural information about the code to recommend developers with "technical debt to be admitted" acting at method-level.

C. Code Smell Detection and Automated Static Analysis Tools

Several code smells detection approaches exist in the literature. They range from manual inspection techniques [37], to metric-based heuristics [28], [33], using rules and thresholds on various metrics [32], historical information [34], graph matching [18], [38] or machine learning techniques [19], [23], [26]. Differently from these previous approaches that do not take into account developers' feedback during the detection of code smells, our approach leverages instances of SATD provided by developers to identify technical debt in the code. Also, our approach uses warnings generated by automated static analysis tools in addition to source code metrics.

Prior to this work, several authors have also investigated the benefits that automated static analysis tools (ASATs) can bring to software development activities. Couto *et al.* [14] examined the role that ASATs can play in the context of fault detection and found no relationship between warnings generated by FindBugs and fault occurence. However, Wedyan *et al.* [39] were able to recommend refactoring opportunities to developers successfully using information provided by ASATs. Ayewah *et al.* [6] evaluated the relevance of warnings generated by FindBugs, a static analysis tools for Java, and found that developers consider them to be relevant and they are willing to fix the issues raised in these warnings. Recently, Beller *et al.* [9] conducted a large-scale study on the use of ASATs in OSS and reported that ASATs are not adopted by many popular software systems, because of the limited amount of programming languages supported by these tools.

Overall, although commonalities can be found with several works referred above, to the best of authors' knowledge, TEDIOUS is the first attempt to predict TD at the method level using a variety of easy to obtain information.

VII. Conclusion

This paper describes TEDIOUS (TEchnical Debt IdentificatiOn System), a machine learning recommender to suggest when developers should admit method-level design TD. It is based on size, complexity, readability metrics, and checks produced by static analysis tools.

We have evaluated TEDIOUS on 9 projects from a publicly available dataset from Maldonado *et al.* [27]. Results indicate for within-project prediction an average precision of about 50%, recall of 52%, and accuracy of 93% when recommending true instances. Because of the highly unbalanced data (few methods contain labeled design-SATD), and despite the fact that oversampling introduces limited benefits with respect to that, cross-project prediction substantially improves the performance of TEDIOUS (average precision of 67%, recall of 55% and accuracy of 92%). For some projects, JFREECHART and ARGOUML, TEDIOUS achieved values above 88% for both precision and recall. TEDIOUS leverages, in particular, source code readability, as well as size and complexity metrics, but also some static analysis tool checks.

Given the findings, we envision different applicability scenarios for TEDIOUS. In the first, TEDIOUS acts as a recommendation system suggesting to developers when documenting TDs. In the second scenario, TEDIOUS could help customizing — for example in a Continuous Integration pipeline — the warnings raised by static analysis tools, by learning rules upon them from previous SATD. In the third scenario, TEDIOUS could complement existing smell and anti pattern detection tools such as DECOR [32], as our study suggested that a simple detector of Long Methods and Long Parameter List is not able to achieve performance comparable to TEDIOUS.

Our future works will be geared to extend the pool of studied programs and to verify if any benefit could be achieved by adding more information. Last, but not least, we plan to extend TEDIOUS to the recommendation of further families of SATD.

REFERENCES

[1] "CheckStyle. http://checkstyle.sourceforge.net/ (last access: 03/30/2017)."

[2] "PMD. https://pmd.github.io/ (last access: 03/30/2017)."

[3] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an ontology of terms on technical debt," in *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*. IEEE, 2014, pp. 1–7.

[4] F. Arcelli Fontana, V. Ferme, and S. Spinelli, "Investigating the impact of code smells debt on quality code evaluation," in *Proceedings of the Third International Workshop on Managing Technical Debt, MTD 2012, Zurich, Switzerland, June 5, 2012*, 2012, pp. 15–22.

[5] F. Arcelli Fontana, R. Roveda, and M. Zanoni, "Technical debt indexes provided by tools: A preliminary discussion," in *8th IEEE International Workshop on Managing Technical Debt, MTD 2016, Raleigh, NC, USA, October 4, 2016*, 2016, pp. 28–31.

[6] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2007, pp. 1–8.

[7] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Software Eng.*, vol. 28, no. 1, pp. 4–17, 2002.

[8] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 315–326.

[9] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 470–481.

[10] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.

[11] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, pp. 321–357, 2002.

[12] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Lawrence Earlbaum Associates, 1988.

[13] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An xml-based lightweight C++ fact extractor," in *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, 2003, pp. 134–143.

[14] C. Couto, J. E. Montandon, C. Silva, and M. T. Valente, "Static correspondence and correlation between field defects and warnings reported by a bug finding tool," *Software Quality Journal*, vol. 21, pp. 241–257, 2011.

[15] W. Cunningham, "The WyCash portfolio management system," *OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[16] E. da S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *7th IEEE International Workshop on Managing Technical Debt, MTD@ICSME 2015, Bremen, Germany, October 2, 2015*, 2015, pp. 9–15.

[17] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 50–60.

[18] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "JDeodorant: identification and application of extract class refactorings," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. ACM, 2011, pp. 1037–1039.

[19] 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.

[20] I. Griffith, D. Reimanis, C. Izurieta, Z. Codabux, A. Deo, and B. Williams, "The correspondence between software quality models and technical debt estimation approaches," in *Proceedings of the 2014 Sixth International Workshop on Managing Technical Debt*, ser. MTD '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 19–26.

[21] M. A. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *SIGKDD Explorations*, vol. 11, no. 1, pp. 10–18, 2009.

[22] C. Izurieta, A. Vetro, N. Zazworka, Y. Cai, C. B. Seaman, and F. Shull, "Organizing the technical debt landscape," in *Proceedings of the Third International Workshop on Managing Technical Debt, MTD 2012, Zurich, Switzerland, June 5, 2012*, 2012, pp. 23–26.

[23] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Quality Software, 2009. QSIC'09. 9th International Conference on*. IEEE, 2009, pp. 305–314.

[24] E. Lim, N. Taksande, and C. Seaman, "A balancing act: what software practitioners have to say about technical debt," *IEEE software*, vol. 29, no. 6, pp. 22–27, 2012.

[25] G. Louppe, L. Wehenkel, A. Sutera, and P. Geurts, "Understanding variable importances in forests of randomized trees," in *Proceedings of the 26th International Conference on Neural Information Processing Systems*, ser. NIPS'13. USA: Curran Associates Inc., 2013, pp. 431–439.

[26] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support vector machines for anti-pattern detection," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 2012, pp. 278–281.

[27] E. Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Transactions on Software Engineering*, 2017 (to appear).

[28] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the 20^{th} International Conference on Software Maintenance*. IEEE CS Press, 2004, pp. 350–359.

[29] ——, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, p. 9, 2012.

[30] B. W. Matthews, "Comparison of the predicted and observed secondary structure of T4 phage lysozyme," *Biochimica et Biophysica Acta (BBA*, vol. 2, no. 405, pp. 442–451, 1975.

[31] T. J. McCabe, "Reverse engineering reusability redundancy: the connection," *American Programmer*, vol. 3, pp. 8–13, October 1990.

[32] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[33] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Proceedings of the 11^{th} International Software Metrics Symposium*. IEEE Computer Society Press, September 2005.

[34] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Mining version histories for detecting code smells," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 462–489, 2015.

[35] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 91–100.

[36] D. M. W. Powers, "Evaluation: From precision, recall and F-measure to ROC, informedness, markedness & correlation," *Journal of Machine Learning Technologie*, vol. 2, no. 1, pp. 37–63, 2007.

[37] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality," in *Proceedings of the 14^{th} Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 1999, pp. 47–56.

[38] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[39] F. Wedyan, D. Alrmuny, and J. M. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *2009 International Conference on Software Testing Verification and Validation*. IEEE, 2009, pp. 141–150.

[40] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 179–188.

[41] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*. London: SAGE Publications, 2002.