UNIVERSITÉ DE MONTRÉAL

RECOMMENDING WHEN DESIGN TECHNICAL DEBT SHOULD BE SELF-ADMITTED

CÉDRIC NOISEUX
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AÔUT 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

RECOMMENDING WHEN DESIGN TECHNICAL DEBT SHOULD BE SELF-ADMITTED

présenté par: NOISEUX Cédric
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. ADAMS Bram, Doct., président
M. ANTONIOL Giulio, Ph. D., membre et directeur de recherche
M. KHOMH Foutse, Ph. D., membre

DEDICATION

À mes parents, Chantal Marceau et Noel Noiseux, pour leur appui tout au long de mes études, leur écoute, leurs sacrifices et leur patience face à mes ambitions.

À mes collègues de Polytechnique Montréal, Loic-Anthony Sarrazin-McCann, Cedrik Rochon, Vincent Couturier et Jean-Nicolas Dang, pour avoir été présent tout au long de mon cheminement universitaire.

À mon ami Marc-Antoine Beaupré pour avoir été un confidant loyal et une personne sur qui je peux toujours compter.

À ma compagne Yannick Dupéré pour son écoute active, sa présence, sa grande générosité et son soutien moral.

ACKNOWLEDGEMENTS

I would like to thank my research director, Giuliano Antoniol of the Software Engineering Department at Ecole Polytechnique de Montreal. You were always there when I needed support and made yourself available for me. You shared with me your experience and knowledge as well an ear in times of need. You manifested a lot of comprehension and patience , particularly considering my peculiar study background. I would also like to thank you for giving me the chance to pursue my studies in a domain so different from my previous ones, and for trusting me from the first to the last day.

I would also like to express my profound gratitude to my parents for always supporting me emotionally and financially during all those years of studying. It has been a long run since I started attending school, but your support and encouragement definitely made a huge positive difference in my success. Without you, I would not be where and who I am today. Thank you.

Cédric Noiseux

RÉSUMÉ

Les Technical Debts (TD) sont des solutions temporaires et peu optimales introduites dans le code source d'un logiciel informatique pour corriger un problème rapidement au détriment de la qualité logiciel. Cette pratique est répandue pour diverses raisons: rapidité d'implémentation, conception initiale des composantes, connaissances faibles du projet, inexpérience du développeur ou pression face aux dates limites. Les TD peuvent s'avérer utiles à court terme, mais excessivement dommageables pour un logiciel et au niveau du temps perdu. En effet, le temps requis pour régler des problèmes et concevoir du code de qualité n'est souvent pas compatible avec le cycle de développement d'un projet. C'est pourquoi le sujet des dettes techniques a déjà été analysé dans de nombreuses études, plus spécifiquement dans l'optique de les détecter et de les identifier.

Une approche populaire et récente est d'identifier les dettes techniques qui sont consciemment admises dans le code. La particularité de ces dettes, en comparaison aux TD, est qu'elles sont explicitement documentées par commentaires et intentionnellement introduites dans le code source. Les Self-Admitted Technical Debts (SATD) ne sont pas rares dans les projets logiciels et ont déjà été largement étudiées concernant leur diffusion, leur impact sur la qualité logiciel, leurr criticité, leur évolution et leurs acteurs. Diverses méthodes de détection sont présentement utilisées pour identifier les SATD mais toutes demeurent sujet à amélioration. Par exemple, la recherche de mots clés en lien avec les dettes techniques (*e.g.: hack, fixme, todo, ugly, etc.*) présents dans les commentaires ou l'utilisation du Natural Language Processing (NLP) combiné à l'apprentissage machine. Donc, cette thèse analyse dans quelle mesure des dettes techniques ayant déjà été consciemment admises (SATD) peuvent être utilisées pour fournir des recommandations aux développeurs lorsqu'ils écrivent du nouveau code. En d'autres termes, le but est d'être capable de suggérer quand admettre des dettes techniques ou quand améliorer du code en processus de rédaction.

Pour atteindre ce but, une approche d'apprentissage machine a été élaborée, nommée TEchnical Debt IdentificatiOn System (TEDIOUS), utilisant comme variables indépendantes divers types de métriques d'entrées au niveau des méthodes, de manière à pouvoir classifier des dettes techniques de conception avec comme oracle des SATD connus. En d'autres termes, notre approche vise à prédire précisément la présence de TD dans les projets logiciels. Le modèle a été entrainé et évalué sur neuf projets Java *open source* contenant des SATD précédemment étiquetés.

TEDIOUS fonctionne au niveau de granularité des méthodes, il détecte si une méthode

contient une dette de conception ou non. Il a été conçu ainsi car les développeurs ont d'avantage tendance à admettre des dettes techniques au niveau des méthodes ou des blocs de code. De plus, les TD peuvent être classifiés selon différents types: conception, requis, test, code et documentation. Les dettes de conception seulement ont été considérées car elles forment la majorité et analyser chaque type demanderait une analyse personnalisée.

TEDIOUS est entraîné avec des données étiquetées comme étant des SATD ou non et testé avec des données sans étiquettes. Les données étiquetées contiennent des méthodes marquées comme étant des SATD, obtenues à partir de neuf projets logiciels analysés par un autre groupe de recherche utilisant une approche NLP et validées manuellement. Les projets sont de différentes dimensions (*e.g.:* number of classes, methods, comments, etc.) et contiennent différentes proportions de dettes de conception. Des métriques sont extraits des données étiquetées: métriques de code source, métrique de lisibilité et alertes générées par des outils d'analyse statique. Neuf métriques de code source ont été retenues pour fournir un portrait de la dimension, du couplage, de la complexité et du nombre de composantes des méthodes. La métrique de lisibilité prend en considération, entre autres, les retraits, la longueur des lignes et des identifiants. Deux outils d'analyse statique ont été utilisés pour cerner de mauvaises pratiques de codage.

Un prétraitement des métriques est appliqué pour retirer celles étant superflues et garder celles étant les plus pertinentes par rapport à la variable dépendante. Certaines caractéristiques sont fortement corrélées entre elles et il serait redondant de toutes les conserver. D'autres subissent aucune ou trop de variations dans le contexte de notre ensemble de données, elles ne seraient pas utiles pour concevoir un prédicteur et sont donc retirées également. De plus, les métriques sont normalisées pour atteindre des valeurs de performance appréciables au niveau de la prédiction inter-projets. Cette normalisation est nécessaire car le code source des projets varie en termes de dimensions et complexité. Finalement, l'ensemble de données est déséquilibré, ce qui signifie que le nombre de méthodes étiqueté comme étant un SATD est faible. Un suréchantillonnage a été appliqué sur la classe en minorité pour générer de nouvelles instances artificielles à partir de celles existantes.

Les modèles d'apprentissage machine sont construits à partir de l'ensemble d'entrainement et les prédictions sont générées à partir de l'ensemble de test. Cinq types de *machine learners* ont été testés: Decision Trees (J48), Bayesian classifiers, Random Forests, Random Trees and Bagging with Decision Trees. Ces modèles ont été retenus pour obtenir une grande variété de résultats, provenant de différents algorithmes considérés comme étant les plus appropriés et précis dans le contexte de notre recherche.

Globalement, le but de cette thèse est d'évaluer la performance de prédiction des SATD

avec notre approche. La vision poursuivie est de favoriser une meilleure compréhension et maintenabilité du code source. La perspective est d'être capable de suggérer quand admettre un TD ayant été identifié précédemment. Trois questions de recherche sont abordées:

- **RQ1**: Comment TEDIOUS performe dans la recommandation de SATD intra-projet?
- **RQ2**: Comment TEDIOUS performe dans la recommandation de SATD inter-projet?
- **RQ3**: Comment un *smell detector* au niveau des méthodes se compare avec TEDIOUS?

Pour répondre à **RQ1**, une validation croisée de dix échantillons a été réalisée sur tous les projets, ce qui signifie que chaque modèle est entrainé sur 90% de toutes les méthodes d'un projet et testé sur 10% de ceux-ci. Le processus est répété dix fois pour réduire l'effet du hasard. Une approche similaire est suivie pour **RQ2** où un modèle est entraîné avec huit projets et testé avec un.

Pour évaluer la performance de TEDIOUS, des métriques standards telles que la précision, le rappel et la mesure F1 sont calculées sur la classe SATD. Ces métriques sont basées sur la quantité de vrais positifs, faux positifs et faux négatifs. De plus, le Matthews Correlation Coefficient (MCC) et le Receiving Operating Characteristics (ROC) Area Under the Curve (AUC) sont calculés, en partie pour tenir compte du nombre de vrais négatifs et car ils sont d'utiles indicateurs pour quantifier l'effet du hasard. Pour compléter cette évaluation, l'importance des métriques d'entrées dans la prédiction des dettes techniques est aussi considérée. Ce qui est visé au niveau de la performance des modèles d'apprentissage est un équilibre entre précision et rappel, donc, de suggérer *correctement* le plus grand nombre possible de TD à admettre.

Pour répondre à **RQ3**, la performance d'un *smell detector*, DEtection & CORrection (DECOR), a été évaluée selon sa capacité à classifier des méthodes étiquetées SATD comme étant des dettes techniques. Les odeurs au niveau des méthodes seulement ont été analysées, tout comme avec TEDIOUS. Finalement, quelques faux positifs et faux négatifs ont été analysés qualitativement pour exprimer les limites de notre approche.

Pour **RQ1**, les résultats ont démontré que le classificateur Random Forest a atteint les meilleures performances pour la recommandation de dettes de conception. La précision moyenne obtenue a été de 49.97% et le rappel 52.19%. Les valeurs de MCC et AUC pour chaque projet ont indiqué la présence de classificateurs de qualité. Équilibrer l'ensemble de données a permis d'accroître le rappel au détriment de la précision. La lisibilité, la complexité et la taille du code source ont joué un rôle significatif dans l'élaboration des prédicteurs.

Pour **RQ2**, la prédiction inter-projet a augmenté la performance des prédicteurs en comparaison à la validation croisée sur des projets singuliers grâce à un ensemble d'entraînement plus large et diversifié. La précision moyenne obtenue a été de 67.22% et le rappel 54.89%. Les valeurs de MCC et AUC ont encore une fois indiqué la présence de classificateurs de qualité. De manière similaire, la lisibilité, la taille et la complexité ont joué un rôle important dans l'élaboration des prédicteurs.

Pour **RQ3**, les odeurs Long Method (LM) et Long Parameter List (LP) ont été évalués par DECOR, très semblables aux métriques Lines Of Code (LOC) et nombre de paramètres qui ont joué un rôle important dans l'entraînement des machines d'apprentissage. Toutefois, les performances de DECOR ne se sont pas avérées aussi bonnes que pour TEDIOUS. Le score F_1 pour l'union de LM et LP n'a pu surpasser 22% et la valeur MCC a indiqué une faible corrélation de prédiction.

Suite à ces résultats, nous avons décidé de concevoir et tester une nouvelle approche pour améliorer la performance de TEDIOUS. Elle est similaire à la précédente car elle est basée sur l'apprentissage machine, elle fonctionne au niveau des méthodes et elle utilise des méthodes étiquetées comme SATD de conception. Toutefois, le modèle de système d'apprentissage favorisé est le Convolutionnal Neural Network (CNN), implémenté spécifiquement pour le contexte de notre recherche. Les variables indépendantes ne sont pas des caractéristiques du code source mais plutôt le code source *lui-même*. Comme pour les caractéristiques de l'approche précédente, le code source a aussi été prétraité, il a été transformé en jetons et un *word embedding* a été réalisé. Le CNN a été testé sur le même ensemble de données, intra-projet et inter-projet, mais aussi selon différentes variables indépendantes,:code source avec, sans et partiellement avec commentaires. Une méthode d'analyse similaire a aussi été suivie, utilisant la validation croisée et les métriques de performance standards. Pour le code source avec commentaires, la précision moyenne obtenue est 96.38% et le rappel 82.40%. Pour le code source sans commentaires, la précision moyenne obtenue est 88.18% et le rappel 33.68%. Pour le code source partiellement avec commentaires, la précision obtenue est 94.84% et le rappel 58.83%.

Pour conclure, ce mémoire décrit TEDIOUS, une approche d'apprentissage machine au niveau des méthodes conçu pour recommander quand un développeur devrait admettre un TD de conception, basé sur la taille, la complexité, la lisibilité et l'analyse statique du code source. Pour l'approche utilisant les caractéristiques du code source, les performances intra-projet basées sur 9 projets Java *open source* ont mené à des résultats prometteurs: environ 50% de précision, 52% de rappel et 93% de justesse. Les performances inter-projet se sont avérées supérieures: environ 67% de précision, 55% de rappel et 92% de justesse. L'ensemble

de données grandement déséquilibré a représenté le plus grand obstacle dans l'obtention de valeurs de performance élevées. Pour les projets les plus volumineux, une précision et un rappel supérieurs à 88% ont été obtenus. Pour l'approche utilisant le code source lui-même, les résultats obtenus se sont avérés meilleurs qu'avec TEDIOUS. Le CNN a été le plus performant en utilisant le code source avec commentaires, obtenant une précision de 96.38%, un rappel de 82.40% et une justesse de 99.44%. Il s'agit d'améliorations de +29.16% pour la précision, +27.51% pour le rappel et +7.55% pour la justesse, par rapport aux métriques de performance obtenus avec l'évaluation inter-projets de TEDIOUS.

TEDIOUS pourrait être utilisée pour diverses applications. Il pourrait être utilisé comme système de recommandation pour savoir quand documenter des TD nouvellement introduits. Deuxièmement, il pourrait aider à personnaliser les alertes relevées pour les outils d'analyse statique. Troisièmement, il pourrait complémenter des détecteurs d'odeurs préexistants pour améliorer leur performance, comme DECOR. Quant aux travaux futurs, un plus grand ensemble de données sera étudié pour savoir si ajouter d'avantage d'information est bénéfique aux performances de notre approche. De plus, nous planifions étendre TEDIOUS à la recommandation de d'avantage de types de dettes techniques.

ABSTRACT

Technical debts are temporary solutions, or workarounds, introduced in portions of software systems in order to fix a problem rapidly at the expense of quality. Such practices are widespread for various reasons: rapidity of implementation, initial conception of components, lack of system's knowledge, developer inexperience or deadline pressure. Even though technical debts can be useful on a short term basis, they can be excessively damaging and time consuming in the long run. Indeed, the time required to fix problems and design code is frequently not compatible with the development life cycle of a project. This is why the issue has been tackled in various studies, specifically in the aim of detecting these harmful debts.

One recent and popular approach is to identify technical debts which are self-admitted (SATD). The particularity of these debts, in comparison to TD, is that they are explicitly documented with comments and intentionally introduced in the source code. SATD are not uncommon in software projects and have already been extensively studied concerning their diffusion, their impact on software quality, their criticality, their evolution and the actors involved. Various detection methods are currently used to identify SATD but they are still subject to improvement. For example, searching for keywords (*e.g.: hack, fixme, todo, ugly, etc.*) in comments linking to a technical debt or using NLP in addition to machine learners. Therefore, this thesis investigates to what extent previously self-admitted technical debts can be used to provide recommendations to developers writing new source code. The goal is to be able to suggest when to "self-admit" technical debts or when to improve new code being written.

To achieve this goal, a machine learning approach was conceived, named TEDIOUS, using various types of method-level input features as independent variables, to classify design technical debts using known self-admitted technical debts as an oracle. In other words, our proposed machine learning approach aims to accurately predict technical debts in software projects. The model was trained and assessed on nine open source Java projects which contained previously tagged SATD.

TEDIOUS works at method-level granularity, in other words, it can detect whether a method contains a design debt or not. It was designed this way because developers are more likely to self-admit technical debt for methods or blocks of code. TD can be classified in different types: design, requirement, test, code or documentation. Only design debts were considered because they represent the largest fraction and other types would require their own analysis.

TEDIOUS is trained with *labeled data*, which are projects with labeled SATD, and tested with *unlabeled data*. The labeled data contain methods tagged as SATD which were obtained from nine projects analyzed by another research group using a NLP approach and manually validated. Projects are of various sizes (*e.g.:* number of classes, methods, comments, etc.) and contain different proportions of design debts. From the labeled data are extracted various kinds of metrics: source code metrics, readability metric and warnings raised by static analysis tools. Nine source code metrics were retained to capture the size, coupling, complexity and number of components in methods; the readability metric takes in consideration indents, lines and identifiers lengths, just to name a few; and two static analysis tools are used to check for poor coding practices.

Feature preprocessing is applied to remove unnecessary features and keep the ones most relevant to the dependent variable. Some features are strongly correlated between each others and keeping all of them would be redundant. Other features undergo important or no variations in our dataset, they would not be useful to build a predictor and thus are removed as well. Additionally, to achieve good cross-project predictions, metrics are normalized because the source code of different projects can differ in terms of size and complexity. Finally, the dataset is unbalanced, which means the amount of methods labeled as SATD is small. Consequently, over-sampling was applied on the minority class to generate artificial instances from the existing ones.

Machine learning models are built based on the training set and predictions are made from the test set. Five kinds of machine learners were tested: Decision Trees (J48), Bayesian classifiers, Random Forests, Random Trees and Bagging with Decision Trees. These models were retained to gather a wide variety of results, from different algorithms which were considered the most appropriate and accurate for the context of this research.

Globally, the goal of this thesis is to assess the SATD prediction performance of our approach. The quality focus is understandability and maintainability of the source code, achieved by tracking existing TD. The perspective is to be able to suggest when to admit those TD. To reach this goal, three Research Question (RQ) are aimed to be addressed:

- **RQ1**: How does TEDIOUS work for recommending SATD within-project?
- **RQ2**: How does TEDIOUS work for recommending SATD across-project?
- **RQ3**: How would a method-level smell detector compare with TEDIOUS?

To address **RQ1**, a 10-fold cross validation was performed on all projects, which means a machine learner is trained with 90% of a project's methods and tested with the other 10%

of them. The process is repeated 10 times to reduce the effect of randomness. A similar approach is used for **RQ2**, a machine learner is trained with eight projects and is tested with one project.

To assess the performance of TEDIOUS, standard metrics such as precision, recall and F1 score are computed for the SATD category. These metrics are based on the amount of True Positive (TP), False Positive (FP) and False Negative (FN). To complement the evaluation, accuracy, MCC and ROC AUC are computed, partly to take into account the amount of True Negative (TN) and because MCC and AUC are useful indicators to quantify the effect of chance. The importance of feature metrics is also taken into account to evaluate the models. What is aimed for in a machine learning model performance is a balance between precision and recall, to suggest as many *correct* TD to admit as possible.

To address **RQ3**, the performance of a smell detector, DECOR, was computed and evaluated in classifying as TD methods labeled as SATD. Only method-level smells were analyzed, similarly to TEDIOUS. Finally, some FP and FN were qualitatively discussed in order to explain the limits of our approach.

For **RQ1**, results showed that Random Forest classifiers achieved the best performance in recommending design debts. The average precision obtained is 49.97% and the recall 52.19%. The MCC and AUC values of each project generally indicated healthy classifiers. Balancing the dataset increased recall at the expense of precision and code readability, complexity and size played a significant role in building the predictors.

For **RQ2**, cross-project prediction increased the performance of predictors compared to the standard cross-validation on singular projects because of a larger and more diverse training set. The average precision obtained is 67.22% and the recall 54.89%. The MCC and AUC values still indicated healthy classifiers. Similarly to within project predictions, code readability, size and complexity played the most important role in recommending when to self-admit design TD.

For **RQ3**, LM and LP were the specific smells targeted and evaluated by DECOR, similar to LOC and number of parameters metrics, which played an important role in training machine learners in the context of our research. However, the detectors of DECOR were unable to achieve similar performance as TEDIOUS. The F_1 score for the union of LM and LP couldn't surpass 22% and the MCC value leaned towards a low prediction correlation.

Following these results, we decided to design and test a new approach in order to improve the performance of TEDIOUS. It is similar to the previous one because it is machine learning based, it works at method-level and it uses design SATD tagged methods. However, the

machine learning model favored is a CNN which was implemented for the context of our research. The independent variables are not source code features but rather the source code *itself*. Like features from the previous approach, the source code was also preprocessed, it was tokenized and a word embedding was performed. The CNN was tested using the same dataset, within-project and across-project, but also using different independent features: source code with comments, without comments or partially with comments. A similar analysis method was followed, using cross validation and standard performance metrics. For source code with comments, the average precision obtained is 96.38% and the recall 82.40%. For source code without comments, the average precision obtained is 88.18% and the recall 33.68%. For source code partially with comments, the average precision obtained is 94.84% and the recall is 58.83%.

To conclude, this paper describes TEDIOUS, a method-level machine learning approach designed to recommend when a developer should self-admit a design technical debt based on size, complexity, readability metrics, and checks from static analysis tools. For the approach using source code features, within-project performance values based on 9 open source Java projects lead to promising results: about 50% precision, 52% recall and 93% accuracy. Cross-project performance was even more promising: about 67% precision, 55% recall and 92% accuracy. Highly unbalanced data represented the biggest issue in obtaining higher performance values. For bigger projects, precision and recall above 88% were obtained. For the approach using the source code itself, results proved to be better than the ones with TEDIOUS. The best results for the CNN were obtained for source code with comments, obtaining a precision of 96.38%, a recall of 82.40% and an accuracy of 99.44%. These are improvements of +29.16% for precision, +27.51% for recall and +7.55% for accuracy, compared to performance metrics obtained with the cross-project evaluation of TEDIOUS.

Different applications could be made of TEDIOUS. It could be used as a recommendation system for developers to know when to document TD they introduced. Secondly, it could help customize warnings raised by static analysis tools, by learning from previously defined SATD. Thirdly, it could compliment existing smell detectors to improve their performance, like DECOR. As for our future work, a larger dataset will be studied to see if adding more information could be beneficial to our approach. Additionally, we plan to extend TEDIOUS to the recommendation of more types of technical debts.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATION

Acc	Accuracy
ASAT	Automated Static Analysis Tools
AUC	Area Under the Curve
CBO	Coupling Between Objects
CNN	Convolutionnal Neural Network
DECOR	DEtection & CORrection
FN	False Negative
FP	False Positive
HIST	Historical Information for Smell Detection
LM	Long Method
LOC	Lines Of Code
LP	Long Parameter List
MCC	Matthews Correlation Coefficient
MDI	Mean Decrease Impurity
ML	Machine Learner
NLP	Natural Language Processing
OO	Object-Oriented
OSS	Open Source Software
Pr	Precision
QMOOD	Quality Model for Object-Oriented Design
Rc	Recall
RNN	Recurrent Neural Network
ROC	Receiving Operating Characteristics
RQ	Research Question
SATD	Self-Admitted Technical Debt
SMOTE	Synthetic Minority Over-sampling Technique
SVM	Support Vector Machines
TD	Technical Debt
TEDIOUS	TEchnical Debt IdentificatiOn System
TN	True Negative
TP	True Positive
WMC	Weighted Method Complexity

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

In today's consumer society, products have to be designed and ready to hit the market as fast as possible, in order to stand out from other similar products and generate revenues. This pressure to produce can affect the quality, maintainability and functionality of the design. In software engineering, the repercussion of this mindset can be, in a certain way, measured with the amount of technical debts present in a project. The problem is that these TD can frequently go unnoticed if they are not admitted. In fact, studies have been conducted on technical debts that are "self-admitted", where developers comment why such code represent an issue or a temporary solution. In a similar vein, the subject of this thesis is to study how previously self-admitted technical debts can be used to recommend when to admit newly introduced technical debts.

1.1 Basic Concepts and Definitions

Technical debts are temporary and less than optimal solutions introduced in the code. They are portions of code that still need to be worked on even though they accomplish their purpose. Cunningham (1992) first described technical debts as "not quite right code which we postpone making it right". For example, TD can be workarounds which don't follow good coding practices, poorly structured code or hard-to-read code. By definition, technical debts don't typically cause errors or prevent the code from working but they can in some circumstances. However, various reasons can motivate the introduction of technical debts: to rapidly fix an issue, because the development team is at early stages of conception, because of a lack of comprehension, skills or experience (Suryanarayana et al., 2015).

TD are introduced throughout the whole conception timeline and under various forms, partly because writing quality code is not always compatible with the standard development life cycle (Brown et al., 2010). That is why an ontology and landscape was proposed by Alves et al. (2014) and Izurieta et al. (2012) to better define the subject. In this ontology, design, requirement, code, test and documentation debts represent the main branches of the classification tree, where each branch can be linked to a specific development stage and to specific criteria. For example, *design debt* "refers to debt that can be discovered by analyzing the source code by identifying the use of practices which violated the principles of good object-oriented design (e.g. very large or tightly coupled classes)" (Alves et al., 2014).

Other studies investigated the perception of developers on technical debts. Ernst et al.

(2015) found that the most important source of TD are architectural decisions, that recognizing the phenomenon is essential for communication and that there is a lack of tools to manage those debts. Additionally, software project teams recognize that this issue is unavoidable and necessary (Lim et al., 2012) and that they cause a lot of problems. For example, slower conception and execution of the software product (Allman, 2012), diminished software maintainability and quality (Wehaibi et al., 2016; Zazworka et al., 2011), and higher production cost (Guo et al., 2011).

Frequently, TD are introduced consciously and explicitly by developers. In those cases, they are "self-admitted" and explained in comments, describing what is wrong with the related block of code (Potdar and Shihab, 2014). Like technical debts, SATD are encountered in most software projects. It was found that 31% of files contain SATD, that they remain in the source code for a long period of time and that experienced developers are more prone to introducing them (Potdar and Shihab, 2014). This proves that a proper management tool is required to deal with this issue, and that unexperienced developers would greatly benefit from such support in order to decide when code should be reworked and documented as TD. The disparity between the experienced and unexperienced workers may also lie in the fact that the unexperienced ones don't want to admit their faults in order to maintain a positive image towards their superiors.

Bavota and Russo (2016) found that there is no clear correlation between code quality and SATD. Code quality metrics such as Weighted Method Complexity (WMC), Coupling Between Objects (CBO) and Buse and Weimer Readability (Buse and Weimer, 2010) were computed and analyzed to reach this conclusion. However, the primary purpose of this work was not to evaluate this relationship but rather to establish a taxonomy of TD. Some threats to the validity of their results could also be made concerning the number of manually analyzed SATD and the level at which the metrics were computed (class-level). A finer analysis would have been required because a single class can contain methods of different length, complexity, cohesion, coupling and readability. This same study found an average of 51 instances of SATD (0.3% of all comments) in the analyzed projects, that the developer who introduces a TD is generally the same that fixes it and that they aren't all fixed during the development life cycle.

1.2 Elements of the Problematic

It is pretty clear that technical debts account for a lot of issues in the development of software applications. They have been extensively analyzed and classified in order to have a better understanding of their impact. However, the identification, as much as the *correct*

identification of SATD, remains a struggle for researchers and developers. Current methods can obtain up to 25% of their total predictions as false positives (Bavota and Russo, 2016). This means that a quarter of all automatically identified TD are not really technical debts. Consequently, conclusions made by studies using these results could be erroneous considering the high level of false technical debts.

Additionally, many strategies can be employed to reduce the number of TD and fix them: take your time when implementing a solution, code refactoring, continuous tracking of TD, proactiveness in fixing debts, etc. (Ambler, 2013). However, they are not highly effective and frequently rely on the willingness of developers to fix the problem and their general knowledge.

To cope with these low accuracy values, various approaches have been proposed to improve the detection of TD. One of them is to identify comment patterns that relate to self-admitted technical debts (Potdar and Shihab, 2014). Potdar *et al.* manually went through 101 762 comments to determine them, which lead to the identification of 62 SATD patterns, for example, *hack, fixme, is problematic, this isn't very solid, probably a bug.* The main issue with this approach is the manual process behind it, which introduces human error and subjectivity. Another approach proposed by Maldonado et al. (2017a) is to use machine learning techniques combined with NLP to automatically identify SATD using source code comments. This idea is promising because it does not heavily depend on the manual classification of source code comments and in fact, it outperforms the previous approach. Manual classification is still required to build the training set for the NLP classifier but the model built from this dataset can then be used to automatically identify SATD in any project, making irrelevant any further manual analysis.

It is important to mention that our research does not revolve on proposing a new technical debts detection method using information contained in comments, but rather using these identified SATD as a base for our recommendation tool. Consequently, the proper classification of SATD methods used by TEDIOUS will directly affect its performance.

To properly establish the problematic of this thesis and prior to designing our approach, several research questions have to be addressed. The main one can be defined like this:

> How can we identify and detect technical debts in a software project using source code features and known self-admitted technical debts with a machine learning approach?

Consequently, in the quest of helping programmers, we designed and developed TEDIOUS, a machine learning inspired recommendation systems that uses manually labeled

training data to detect method-level technical debts. The goal of this thesis is then to assess the performance of TEDIOUS in recommending SATD. Three other high level research questions can be derived from the main one:

How does TEDIOUS work for recommending SATD within-project?
How does TEDIOUS work for recommending SATD across-project?
How would a method-level smell detector compare with TEDIOUS?

The evaluation goal can be addressed by these questions. Firstly, we want to evaluate the detection performance of a model trained with features from the source code of a specific system on himself. Secondly, we want to perform a similar performance evaluation on a model trained with features from several systems on another unrelated system. Finally, we want to compare the detection performance of TEDIOUS with other popular smell detectors.

Our approach is based on the hypothesis that current methods to detect technical debts are limited and inefficient, and that a new approach could be beneficial to the improvement of detection performance. We also think that manual analysis and human subjectivity is detrimental to the efficiency of current methods. Consequently, we believe that a well crafted machine learning approach could lead to better results and performance values in identifying technical debts and recommending when they should be self-admitted.

1.3 Research Objectives

The objective of this research is to design a machine learning approach that uses as independent variables various kinds of source code features, and as dependent variables the knowledge of previously self-admitted technical debts, to train machine learners in recommending to developers when a technical debt should be admitted.

As mentioned previously, the purpose of this thesis is not to propose a novel method to identify SATD from source code comments using patterns or NLP (Maldonado et al., 2017a; Potdar and Shihab, 2014). It is more about using these classified SATD comments to build our labeled dataset consisting of methods' source code information and metrics to identify possible technical debts to self-admit.

The main objective can be divided in two application scenarios. Firstly, tracking and managing technical debts is considered important but lacking in the industry (Ernst et al.,

2015). Consequently, TEDIOUS could be used to encourage developers to self-admit TD in order to easily track and fix the issues later. This is particularly true for junior developers, who are less prone to self-admitting than experienced ones (Potdar and Shihab, 2014). Secondly, our tool could be used as an alternative, or a complement, to existing smell detectors in proposing improvements to source code. In other words, TEDIOUS could act as a tracking, managing and improvement tool for software projects.

1.4 Design Objectives

The five research objectives can be achieved by following this design methodology. The first one aims at *defining and extracting relevant features from methods*. These features are characteristics that describe each method. Contrary to previous studies (Bavota and Russo, 2016), TEDIOUS works at method-level rather than class-level because we found that SATD comments are more frequently related to methods or blocks of code. We investigated a set of method structural metrics, a method readability metric and warning raised by static analysis tools.

The second specific objective aims at *identifying self-admitted technical debts*. Only one type of technical debt is considered for various reasons, the design debt. Firstly, it is the most common type of TD (Maldonado et al., 2017a). Secondly, the other types (requirement, code, test and documentation) would require a different analysis and other training features. However, these types are planned to be analyzed in our future work. To identify SATD methods, TEDIOUS reuses knowledge of manually labeled TD, metrics, and static analysis warnings. In fact, the training set was extracted from the manually labeled corpus of 9 open source Java projects provided by Maldonado et al. (2017a).

The third specific objective aims at *preprocessing the features*. Strongly correlated features are cleaned up to remove redundancy, metrics that don't vary or vary too much are removed, a normalization is applied to take into account the different nature of projects and the training set is balanced by oversampling the small number of SATD tagged methods.

The fourth specific objective aims at *building and applying machine learning models*. Five machine learners are trained and tested, performing SATD prediction within-project and across-project. The five retained ML are: Decision Trees (J48), Bayesian classifiers, Random Forests, Random Trees and Bagging with Decision Trees.

We could add another objective which aims at *improving the first TEDIOUS approach*. To do so, a Convolutionnal Neural Network is designed and implemented, using the source code itself as the independent variable. It is tested using the same dataset, within-project and across-project, and with different independent variable configurations: with, without or

partially with comments. Results show improvement compared to the previous approach, in terms of precision, recall, F_1 score and accuracy.

1.5 Thesis Overview

Chapter 2: Literature Review The literature review provides a current state-of-the-art overview of the knowledge on technical debts and other related topics. It summarizes relevant information extracted from previous studies concerning four main topics: relationship between technical debt and source code metrics, self-admitted technical debt, code smell detection and automated static analysis tools.

Chapter 3: The Approach and Study Definition The approach followed is thoroughly described in several steps. The types of features are described and the way they are extracted is explained. The provenance and identification of the SATD tagged comments is shared. The preprocessing that is performed on features is demystified and justified. The machine learning models chosen are revealed as well as their configuration. As for the study definition, the dataset characteristics (number of files, classes, comments, etc.) are shared for each project and the analysis method (cross validation, accuracy, precision, recall, F_1 score, MCC, ROC, AUC) explained.

Chapter 4: Analysis of Study Results and Threats to Validity The study results are analyzed based on each research question: performance for within-project prediction, performance for across-project prediction and comparison with a method-level smell detector. Results indicate that within-project prediction achieves at best 50% precision and 52% recall. Improvement is made for across-project prediction where prediction achieves at best 67% precision and 55% recall. The best machine learner turned out to be Random Forest. It was also found that SATD predictions made by TEDIOUS only weakly relate to method-level code smells. A qualitative discussion on false positives and negatives is also proposed. Following the results analysis, several threats to validity are shared: construct, internal, conclusion and external validity threats.

Chapter 5: Convolutionnal Neural Network with Comments and Source Code This chapter describes an updated approach to detect TD to self-admit and its preliminary results. First, the CNN characteristics and features are described. Secondly, the approach is explained: the features used, the identification of SATD, the use of word embeddings and the way the CNN is built and applied to the context of our research. Thirdly, the study definition

is described: the characteristics of the dataset and the analysis method. Finally, the study results are analyzed based on three prediction contexts: source code with comments, without comments and partially with comments. Various CNN configurations are also analyzed. Results indicate that source code with comments obtains the best performance values. It achieves 96.38% precision, 82.40% recall and 99.44% accuracy.

CHAPTER 2 LITERATURE REVIEW

The literature related to this thesis can be divided in four topics, which will be summarized in this chapter. The first one addresses the relationship between technical debt and source code metrics. The second defines what are self-admitted technical debts. The third topic describes code smell detection approaches and the last one covers the usage of automated static analysis tools.

2.1 Relationship Between Technical Debt and Source Code Metrics

Many researchers have tried to link technical debts, more specifically design and code types, to source code metrics. Marinescu (2012) proposed an approach based on a technique for detecting design flaws and built on top of a set of metric rules capturing coupling, cohesion and encapsulation. Griffith et al. (2014) empirically validated the relationship between TD and software quality models. Three TD detection methods were compared with Quality Model for Object-Oriented Design (QMOOD) (Bansiya and Davis, 2002) and only one of them had a strong correlation to quality attributes, namely reusability and understandability. A larger study was performed by Fontana et al. (2016a) where they analyzed how five different tools detect technical debts, their principal features, differences and missing aspects. They focused on the impact of design smells on code debt to give advices on which design debt should be prioritized for refactoring. These tools all took into account metrics, smells, coding rules and architecture violations. However, there was only a limited agreement among tools and they still ignored some important pieces of information.

2.2 Self-Admitted Technical Debt

Many studies have been conducted in order to describe and classify the nature of self-admitted technical debts. Potdar and Shihab (2014) investigated technical debts in the source code of open source projects and they found out that developers frequently self-admit TD they introduce, explaining in the form of comments why these particular blocks of code are temporary and need to be reworked. They are some of the first to acknowledge the existence of SATD and to propose a detection method using pattern matching in source code comments. da S. Maldonado and Shihab (2015) analyzed developers' comments in order to define and quantify different types of SATD. An approach similar to the one of Potdar and Shihab (2014) is followed, which is using pattern matching to classify SATD into five types: design,

defect, documentation, requirement and test. It was found that design debts are the most common, making up between 42% and 84% of all comments in software projects.

Bavota and Russo (2016) performed a large-scale empirical study on self-admitted technical debts in open source projects. They studied their diffusion and evolution, the actors involved in managing SATD and the relationship between SATD and software quality. They showed that there is on average 51 instances of SATD per system, that code debts are the most frequent, followed by defect and requirement debts, that the number of instances increases over time because they are not fixed by developers, and that they normally survive for a long time. Like Griffith et al. (2014), they found no real correlation between SATD and quality metrics (WMC, CBO, Buse and Weimer readability).

Wehaibi et al. (2016) also studied the relation between self-admitted technical debts and software quality. Their approach is based on investigating if more defects are present in files with more SATD, if SATD changes are more likely to cause the emergence of future defects and whether changes are more difficult to perform or not. They found that no real trend could be made between SATD and defects, that SATD changes did not introduce more future defects than no-SATD changes but that they are indeed more difficult to perform.

Maldonado et al. (2017a) recently proposed a new approach based on NLP techniques to detect self-admitted technical debts, more specifically design and requirement debts. They extracted comments from ten open source projects, cleaned them to remove irrelevant ones and manually classified them into the different types of SATD. This dataset was then used as the training set for a maximum entropy classifier. It turned out that the model could accurately identify SATD and outperform the pattern matching method of Potdar and Shihab (2014). Comments mentioning sloppy or mediocre source code were the best indicators of design debts and comments related to partially implemented requirement were the best for requirement debts.

Contrary to previous studies, the goal of TEDIOUS is to detect methods that are TD prone. This is to say differently from da S. Maldonado and Shihab (2015), our goal is not to classify comments but rather to categorize methods based on those classified comments.

2.3 Code Smell Detection

Several approaches to detect code smells have been proposed in today's literature: Travassos et al. (1999) developed reading techniques to guide developers in identifying defects in Object-Oriented (OO) designs, Marinescu (2004) formulated metrics-based rules as a detection strategy that can capture poor design practices and Munro (2005) used software metrics

to characterize bad smells. Others such as Moha et al. (2010) proposed DECOR, an approach using rules and thresholds on various metrics to detect smells. This smell detector is in fact used to compare its performance with TEDIOUS.

Many detection techniques rely on structural information, however, Palomba et al. (2015) exploited change history information to propose Historical Information for Smell Detection (HIST), a smell detector that identifies instances of five different code smells, with promising results. On the other hand, Fokaefs et al. (2011) used graph matching to propose JDeodorant, an Eclipse plugin that automatically applies refactoring on "God Classes". Using graph matching also, Tsantalis and Chatzigeorgiou (2009) proposed a methodology recommending "Move Method" refactoring opportunities for "Feature Envy" bad smells to reduce coupling and increase cohesion.

Machine learning techniques are also popular. Fontana et al. (2016b) compared and experimented with 16 different machine learning algorithms to detect code smells, Khomh et al. (2009) proposed a Bayesian approach to detect code and design smells, and Maiga et al. (2012) proposed SVMDetect, a new approach to detect anti-patterns using a Support Vector Machines (SVM) technique.

TEDIOUS is different from these previous approaches for two main reasons. Firstly, they use structural or historical information and metrics from the code to detect smells. However, in addition to these characteristics, we also use feedback provided by developers, in the form of SATD comments which leads to the identification of technical debts. Secondly, we also use warnings generated by Automated Static Analysis Tools (ASAT) to portray an even better representation of the source code quality.

2.4 Automated Static Analysis Tools

The subject of automated static analysis tools have already been widely covered to analyze its benefits on the development of software projects. To understand the actual gains provided by automated static analysis tools, Couto et al. (2013) studied the correlation and correspondence between post-release defects and warnings issued by the bug finding tool FindBugs. Only a moderate correlation and no correspondence were found between defects and raised warnings. On the other hand, three ASAT were evaluated by Wedyan et al. (2009) showing that they could successfully recommend refactoring opportunities to developers. Ayewah et al. (2007) also evaluated an FindBugs, its performance was measured to quantify its accuracy and the value of warnings raised. They found that warnings were mostly considered relevant by developers and that they were willing to make the appropriate modifications to

fix the issues. Beller et al. (2016) performed an evaluation of several ASAT on an even larger scale. They found that the use of ASAT is widespread but no strict usage policy is imposed in software projects. Generally, the automated static analysis tools are used with their default configuration, only a small amount is significantly changed. Also, ASAT configurations experience little to no modifications over time.

Many of the mentioned studies share common views and purposes with our research and TEDIOUS. However, as far as we know, TEDIOUS stands out because it is the first approach that attempts to predict technical debts at method-level with a wide variety of easy to use and to extract information.

CHAPTER 3 THE APPROACH AND STUDY DEFINITION

3.1 The Approach

This section will describe the steps followed to design TEDIOUS, our proposed machine learning detector to identify design technical debts to self-admit. It will also define its characteristics, how it works and how to use it. TEDIOUS works at method-level since it is typically the granularity at which developers introduce SATD (da S. Maldonado and Shihab, 2015; Potdar and Shihab, 2014). In other terms, it is able to detect whether a method contains a design technical debt or not. Class-level granularity would be to coarse because technical debts normally admitted by developers are related to specific and punctual issues in the source code. Additionally, a class could contain a TD but it would be impossible to precisely identify the source of the problem since a class contains several methods and LOC.

TEDIOUS works as shown in Figure 3.1. Two datasets are required as inputs: the training set and the test set. The training set contains labeled data, which is source code from a project where technical debts are known and have been self-admitted through comments. The test set contains unlabeled data, which can be any source code under development or already released where TEDIOUS can attempt to recommend where TD should be self-admitted or where source code should be improved.

For the training set, various kinds of metrics and static analysis warnings are extracted from methods in the source code as well as self-admitted technical debts in order to build an oracle to train the model. These labeled SATD methods are essential for the machine learner since supervised learning is performed, meaning each method is labeled as true (containing a TD) or false.

Once all the information is extracted, feature preprocessing and selection is applied. Multi-collinearity, a phenomenon occurring when two predictor variables are highly correlated, meaning that one can be linearly predicted by the other, is dealt with. Feature selection is applied to retain only the most relevant variables to train the predictors. Finally, re-balancing is performed to address the issue of the low amount of positive examples, *i.e.* SATD methods. With the preprocessed features and the oracle now defined (each method is labeled as SATD or not), the machine learners can be trained.

In parallel, the test set is also being prepared. The same features are extracted from the source code but no SATD matching is required since the data is unlabeled. SATD are only required for the oracle, which is used for training the models. A similar feature filtering is

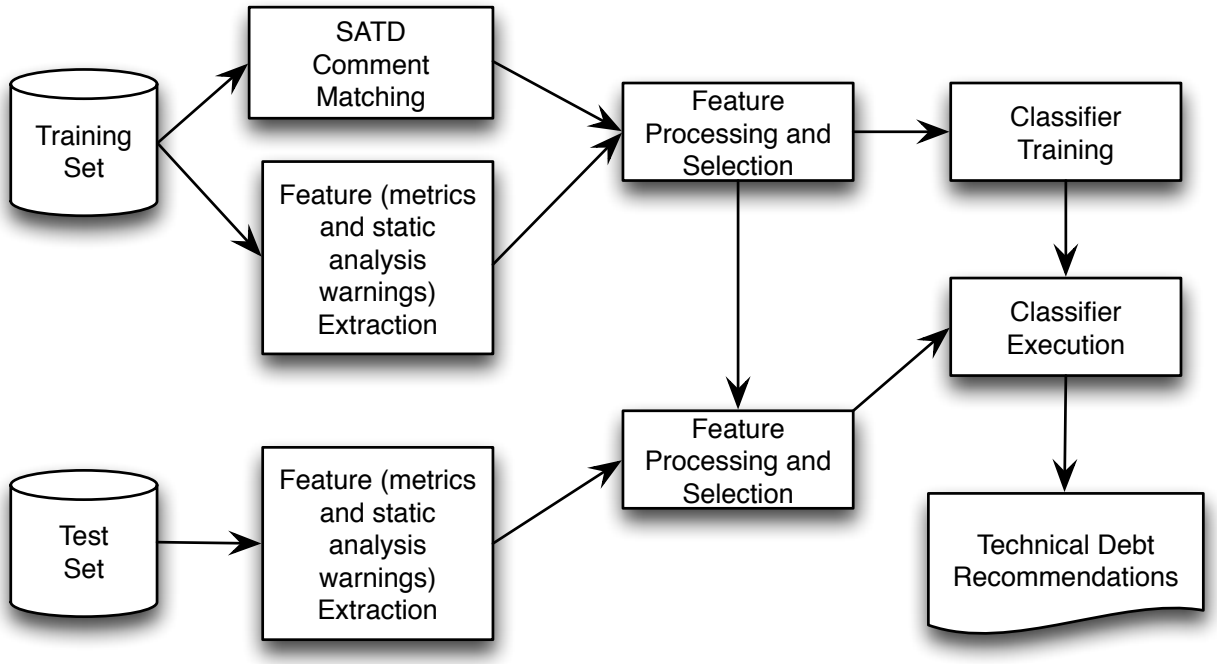

Figure 3.1 Proposed approach for recommending SATD with TEDIOUS.

applied, except for the re-balancing since it is only required on the labeled data. With both the test set and the previously trained classifiers, predictions can be made on the test set in order to recommend when to self-admit technical debts.

Each step of the process will be described in the following sections. Firstly, the features used will be detailed: source code metrics and warnings raised by static analysis tools. Secondly, the method employed to identify SATD will be explained. Thirdly, the feature preprocessing tools will be summarized: multi-collinearity, feature selection and re-balancing. And finally, the training and application of the machine learning models will be explained.

3.1.1 Features

Three pieces of information extracted from the source code are necessary to accurately describe it: structural metrics, readability metrics and warnings raised by static analysis tools. There are reasons these specific features were chosen to describe the source code. Structural metrics are essential to capture symptoms of complex, heavily coupled and poorly designed code. These metrics explain the quality of the implementation and the software's design. Readability metrics quantify symptoms of poorly documented, hard to read and difficult to understand code. Warnings from static analysis tools are related to more specific bad coding

choices rather than globally wrong code. They are issues which could lead to low maintainability and understandability of the code or which could potentially introduce defects in the future. In the following sections, these metrics and warnings are described more in depth.

Source Code Metrics

Nine source code metrics are extracted to characterize size, coupling and complexity. To define the size, metrics like LOC or number of statements are calculated. For coupling, a metric such as number of call sites is computed. For complexity, McCabe cyclomatic complexity (McCabe, 1990a), number of defined variables, number of expressions and number of identifiers are calculated. For readability, number of comments is an example of a metric used. It is important to know that not all comments are considered in the dataset. SATD related comments are ignored in the empirical evaluation to avoid TEDIOUS becoming a self-prophecy. Consequently, they are removed from the dataset. This issue will be covered later in the study design.

Firstly, to extract those source code metrics, an XML representation of the Java source code was generated using the rool srcML (Collard et al., 2003). The computation of metrics was performed on this representation. It was also required to link comments to their related methods. Therefore, the rule used was that any comments directly preceding a method was assigned to it, as well as comments inside it. Comments could be a blocks of code or just single lines. This step is essential to be able to classify which method is a SATD and which one is not. Some methods were excluded from our analysis because they did not fit the context of our research, namely getter and setter methods since they are irrelevant with design debts. To do so, we looked for method prefixes matching *get* or *set* and methods made up from more than a single line of code. When these two criteria were covered, the method was removed from the dataset.

Secondly, to compute the code readability metric, we use the metric proposed by Buse and Weimer (2010) and its related extraction tool, based on a machine learning approach. This metric is based on specific characteristics of the source code: indentation, line length, identifier length, comment density, and the use of keywords and operators. The tool was also designed using feedback from human annotators. They were asked to rate the readability of code snippets, that were then used to classify snippets as "less" or "more" readable. A value between 0 and 1 is then computed based on these characteristics, 1 being the highest readability. The readability metric was considered relevant because, as mentioned by Bavota and Russo (2016), code readability is strongly correlated with the introduction of technical debts. Source code that is difficult to read is consequently difficult to maintain and under-

stand (Buse and Weimer, 2010), and thus more likely to contain technical debts. Finally, here are the nine source-code metrics we extracted:

- *LOC*: Number of lines of code in the body of a method.
- *Number of statements*: Number of occurrences of expression statements in a method. In case of local class definitions, the number of statements in the enclosing method is increased by the total number of statements of the local class.
- *Number of comments*: Number of single-line and multi-line comments in a method.
- *McCabe cyclomatic complexity*: Number of linearly independent paths of a method (McCabe, 1990b).
- *Number of passed parameters*: Number of parameters of a method.
- *Number of identifiers*: Number of unique identifiers in a method.
- *Number of call sites*: Number of locations in the code where zero or more arguments are passed to a method, or zero or more return values are received from the method.
- *Number of declarations*: Number of variable and class declaration in a method.
- *Number of expressions*: Number of expressions contained in a method.

Warnings Raised by Static Analysis Tools

Warnings raised by static analysis tools are essential to detect poor coding practices, which are also related to the introduction of technical debts. We cannot directly relate a flagged practice raised by an ASAT to a technical debt. However, we can wonder if multiple close warnings are justified and if they can be caused by the presence of a TD in the source code. Having this hypothesis in mind, we used two popular static analysis tools, namely CheckStyle and PMD. Firstly, CheckStyle (che) is widely used to check the adherence of code to standard practices and to detect pieces of code that are potentially smells. It performs an analysis based on a default configuration file, which can be modified at the discretion of users. For our research, the default configuration was used, containing code styles defined by Oracle and 43 checks. PMD (pmd) main goal is to find common programming flaws such as unused objects, unnecessary catch blocks or incomprehensible naming. Similarly to CheckStyle, the default configuration was used, which contains 168 checks. Several reasons justify the choice of these two static analysis tools: they are commonly adopted by Open Source Software (OSS) (Beller

et al., 2016), they provide a wide range of warnings related to code styles and programming practices, and they can be executed on source code statically. It is important to know that SATD comments were removed when using these ASATs.

3.1.2 Identification of Self-Admitted Technical Debt

As mentioned previously, the purpose of this thesis is not to propose a novel approach to detect SATD using information from comments. Previous work has been completed aiming to address this issue by using pattern matching (da S. Maldonado and Shihab, 2015), (Potdar and Shihab, 2014) or NLP combined with machine learners (Maldonado et al., 2017b). However, we still needed a dataset with methods tagged as design debt to train our machine learning models. Maldonado et al. (2017b), which worked on the NLP approach, published a dataset of 10 open source projects annotated with methods tagged as technical debt or not. We used this dataset for our machine learning models where only the design debts were retained.

Some preprocessing had to be done on the dataset since it reported SATD at file-level and not method-level. To link SATD to methods, we matched the SATD comment string to comments attached to methods. We used pattern matching in order to achieve this result, making sure that method comment strings completely matched SATD comments before tagging the method as a technical debt.

3.1.3 Feature Preprocessing

Several features have been extracted from the source code, however, not all of them are relevant or necessary to train our models. Some clean up have to be done to reduce the size of the data input and improve the training phase. Multi-collinearity have to be dealt with, feature selection as to be applied and the training set has to be re-balanced.

Multi-Collinearity

We computed several features to characterize the source code of our software projects, however, some of them can be strongly correlated and can vary in the same way. Keeping these pairs of features would cause redundancy since they can be mutually and linearly predicted. That is why, when facing a pair of such features, we only keep the one that better correlates with the dependent variable (SATD methods). The *varclus* function in R, from the *Hmisc* package, was used to help us achieve this preprocessing. This function performs a hierarchical cluster analysis of features to detect when two variable are positive based on similarity

measures. It is mainly used to assess collinearity and redundancy, consequently resulting in data reduction. Hoeffding D statistic, squared Pearson or Spearman measures can be applied with *varclus* to evaluate the correlation between variables. In this research, the Spearman's ρ rank correlation measure was retained. To identify the problematic pairs using this coefficient, the cluster tree generated has to be cut at a particular level of ρ^2, which represents the correlation value. In our case, a value of $\rho^2 = 0.64$ was used since it corresponds to a strong correlation (Cohen, 1988).

Feature Selection and Normalization

Some features will vary greatly or not at all between methods. These features are not useful to build a predictor because of their high degree of variance and they won't be determinant compared to all the others available. The process of selecting only a relevant subset of all possible features is called *feature selection*. Several reasons can justify going through this selection: to improve the prediction performance of machine learning models, to improve the speed and cost-effectiveness of predictors, and to simplify models to better understand and interpret the underlying process behind the dataset generation (Guyon and Elisseeff, 2003).

The *RemoveUseless* filter implemented in Weka (Hall et al., 2009) was used to perform feature selection. It looks for features that never vary or features that vary too much. For the latter, it looked for features that had a percentage of variance above a specific treshold, which was set to 99%.

In addition to feature selection, the metrics were also normalized. Several projects were analyzed, all having significant differences in complexity and size characteristics. Therefore, normalization is necessary to reduce the effect of those differences during the training phase and to achieve good cross-project prediction performance. We will further discuss those characteristics of the dataset in the study definition section.

Re-Balancing

To build performing predictors, a training set of quality must be fed to the machine learners. One important aspect is to have a balanced dataset, which means having as much as possible an equal number of positive and negative examples. In our context, this means as many SATD tagged methods as correct methods. This is a serious problem for us since the vast majority of methods are not technical debts (this will be discussed more in depth in the next section), only a minority contains SATD.

There are two ways to address this issue: under-sample the majority class (methods with

no SATD) or over-sample the minority class (SATD tagged methods). Since the training set is so highly unbalanced, which means the number of instances of the minority class is excessively small, the second option was favored. In fact, under-sampling would result in a very small training set.

To apply over-sampling, artificial instances of SATD methods must be generated from the existing ones. To do so, Weka provides a tool to perform over-sampling, called Synthetic Minority Over-sampling Technique (SMOTE) (Chawla et al., 2002), which combines under-sampling the majority class and over-sampling the minority class to achieve improved classifier performance.

3.1.4 Building and Applying Machine Learning Models

We extracted various metrics, we identified SATD methods and we performed the preprocessing of features, the only remaining step is building and applying the machine learning models. Two sets are required: the training set and testing set. The training set contains the methods with their corresponding features and an additional variable to tag them as positive or negative (SATD or not). This set is used to build the models. The testing set contains the same methods with their corresponding features, but not the variable tagging the methods because we want to test the prediction performance on a blank dataset. We experimented with five types of machine learners in Weka (Hall et al., 2009): Decision Trees (J48), Bayesian classifiers, Random Forests, Random Trees, and Bagging with Decision Trees. Only default configurations were used, however, further work could be done by trying to optimize these configurations. Here is a little overview of each algorithms:

- *Decision trees*, namely J48, implement the standard C4.5 algorithm using the concept of information entropy (Quinlan, 1993).
- *Bayesian classifiers* apply the Bayes' theorem to classify observations, assuming strong independence between features. More specifically, we use BayesNet in Weka, which is the base class for Bayes Network classifiers. It provides datastructures and facilities common to learning algorithms K2 and B.
- *Random forests* average multiple decision trees trained on different parts of a same training set. The goal is to reduce the variance of classifications and the risk of overfitting the training set (Breiman, 2001).
- *Random trees* build decision trees considering a number of randomly chosen attributes at each node.

- *Bagging* combines multiple classifiers (decision trees in our case) built on random samples of the training set (Breiman, 1996). It was designed to improve the stability and accuracy of machine learning algorithms, to reduce the variance and to avoid overfitting.

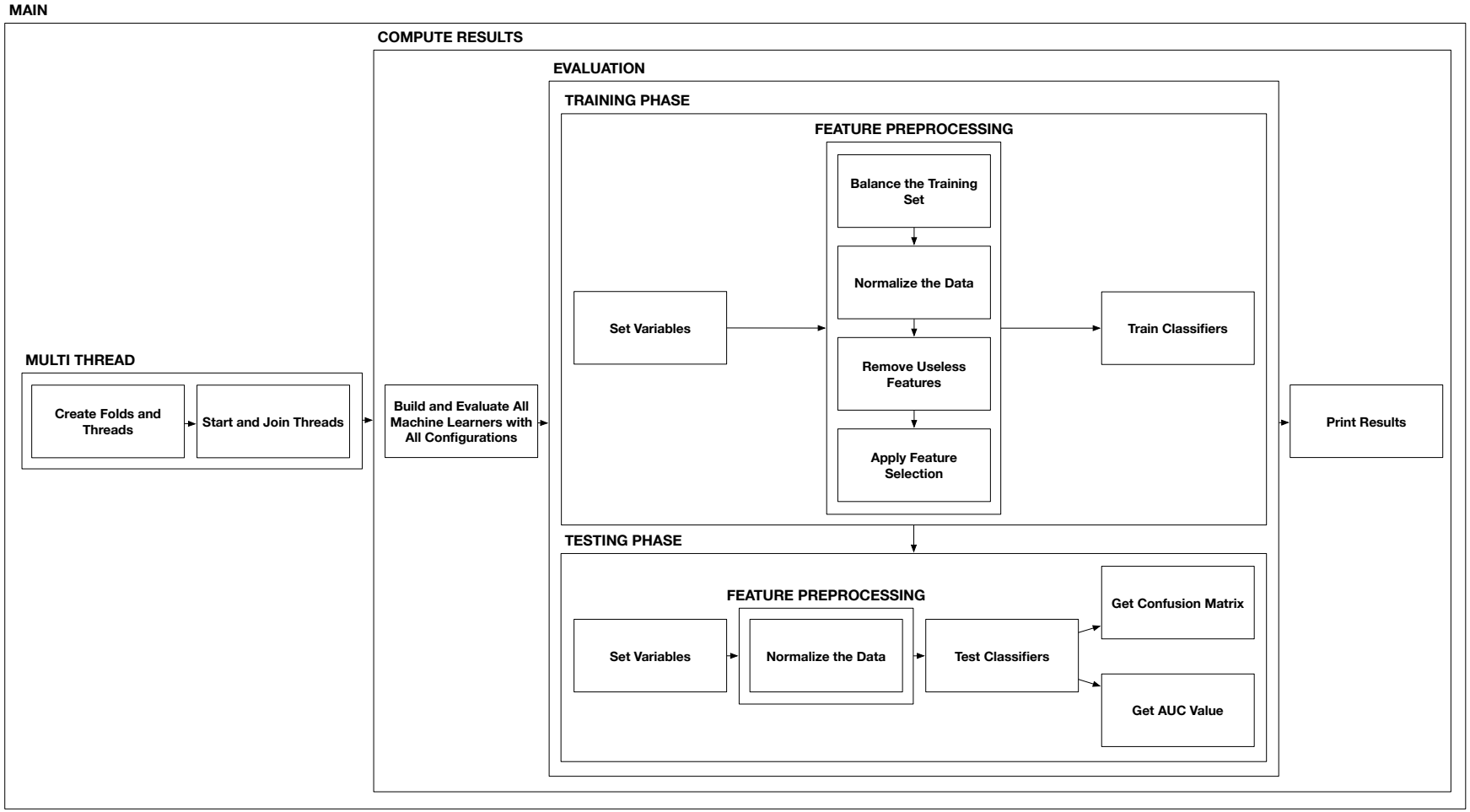

Figure 3.2 Process for building and applying machine learners.

Figure 3.2 provides an overview of the machine learning process. To increase the training speed, several threads are created. For the 10 fold cross-validation inter-project analysis, one thread is started for each fold, which were generated beforehand for each software system. Actions in the evaluation phase are performed simultaneously on each thread and the computation is finished once all of them are finished. All possible combinations of machine learners and configurations are used for training and testing on each fold. We have 5 different types of machine learners, the possibility of balancing the training set, and the possibility of applying feature selection, for a total of 20 possible predictors per fold.

The evaluation phase consists of a training and testing phase. In the training phase, the variables to predict are initialized, *i.e* SATD methods, and feature preprocessing is applied. This means that the training set can be balanced or not, data is normalized, useless features are removed and feature selection can be applied or not. Afterwards, the classifiers are trained. In the testing phase, the variables to predict are initialized again. As for the feature preprocessing, only data normalization is performed. The classifiers are tested and then the confusion matrix and the AUC value are computed. Finally, results are printed to perform a deeper analysis.

3.2 Study Definition

The goal of this thesis is to assess the prediction performance of our machine learning based approach in recommending technical debts to self-admit. The focus is to enhance source code quality, more specifically its maintainability and understandability, by keeping track of technical debts which can be corrected in the future. The perspective of this thesis is to be able to suggest to developers when to admit technical debts. Globally, we aim at addressing three research questions:

- **RQ1**: How does TEDIOUS work for recommending SATD within-project?
- **RQ2**: How does TEDIOUS work for recommending SATD across-project?
- **RQ3**: How would a method-level smell detector compare with TEDIOUS?

3.2.1 Dataset

Table 3.1 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%

To evaluate our approach, we used a dataset that was already analyzed to find SATD methods (Maldonado et al., 2017b). Those methods were detected using a NLP approach, and then manually validated and classified. The dataset contains 10 open source projects, however, we only used 9 of them since we could not download the specific version of EMF. Table 5.1 summarizes the characteristics of all studied projects. It provides information on project releases; number of files, classes, methods and comments in projects; number of comments in methods; number of design SATD in methods and in classes; and percentage of methods with design SATD.

Some differences were observed with the characteristics extracted from the original paper (Maldonado et al., 2017b), concerning the number of classes, methods and comments. Several reasons can explain these disparities: usage of different extraction tooling, tools characteristics and processing. For example, we considered each comment as a single entity, whereas Maldonado et al. (2017b) regrouped successive line comments. Additionally, we did not establish a separation between classes and their inner classes, and we considered interfaces as classes. Methods related to inner classes were associated to its container. However, these differences are not an issue for our research since they concern classes and our approach is method-level based. Additionally, some files from Maldonado et al. (2017b) analysis could have been left aside because of their absence of comments.

We observe some interesting facts when looking at Table 5.1. As explained previously, we clearly see a prevalence of method-related SATD compared to class-level SATD. They are at least 2 times more common for ArgoUML, which contains about half of all the class-level TD in the dataset, and can be up to 470 times more common for jFreeChart where only 4 of the 1,885 design SATD are at class-level. Globally, we are around 10 times more likely to encounter a method-level design technical debt in our dataset than class-level. We also observe that the dataset is highly unbalanced between SATD prone and non-SATD prone methods. jFreeChart provides a decent ratio with 18% of methods containing a design technical debt, but all other projects have 5% or less of their methods containing design debts. To put this into perspective, out of the 11,052 methods in Ant, only 57 are SATD prone. For jEdit, only 77 instances out of 4,785 are proned to contain a SATD. Unsurprisingly, as we will discuss in the analysis of study results, these two projects achieved the lowest performance values.

The replication package provided by Maldonado et al. (2017b) contains information on SATD at class-level and not method-level. The issue is that we need to assign SATD at method-level to build our oracle. To do so, we performed pattern matching between known SATD comments from the replication package and comments attached to methods in the 9

software projects. The other cases are: if a comment is matched inside a class but not a method, it is attached to the class, and if it is matched outside of a class, it is attached to the file. These class-level and file-level technical debts are not considered in our research, which is not a big issue since they represent a minority of all design debts.

3.2.2 Analysis Method

For **RQ1**, we want to know how TEDIOUS works for recommending SATD within-project. A 10-fold cross validation was performed on each project. In other terms, the dataset of a single project is divided into 10 folds, the machine learner is trained on 9 of them and tested on the remaining one, until all 10 configurations are processed in order to limit the effect of randomness. The performance values are averaged over the 10 iterations to obtain the most representative picture. For **RQ2**, we want to know how our approach works for recommending SATD across-project. The process is similar to **RQ1**, but instead we train on 8 projects and test on the remaining one, until all 9 possible combinations are executed.

Standard performance metrics to evaluate automated classification were computed to analyze our approach: precision, recall and F_1 score. These metrics were computed for the SATD category.

Precision (Pr) is the percentage of relevant instances of methods predicted as SATD among all retrieved instances. TP and FP are respectively the number of true positives, correct methods predicted as SATD, and false positives, incorrect methods predicted as SATD.

$$Pr = \frac{TP}{TP + FP}$$

Recall (Rc) is the percentage of relevant instances of SATD methods that have been retrieved over all relevant instances. FN is the number of false negatives, incorrect methods predicted as non-SATD.

$$Rc = \frac{TP}{TP + FN}$$

The F_1 score is the harmonic mean between precision and recall, which provides a single measurement to evaluate a system.

$$F_1 = 2 \cdot \frac{PR \cdot RC}{PR + RC}$$

The previous metrics are specific to the SATD class, which means true negatives TN, correct methods predicted as non-SATD, are not considered yet in the performance evaluation. Consequently, other metrics are required to complement the analysis: accuracy, Matthews Correlation Coefficient (MCC) (Matthews, 1975), and the Area Under the Curve (AUC) of the Receiving Operating Characteristics (ROC).

Accuracy (Acc) is the total number of methods correctly predicted, whether it is containing a SATD or not, among all the methods analyzed.

$$Acc = \frac{TP + TN}{TP + TN + FP + FN}$$

The MCC is a metric used in machine learning to evaluate the quality of a two-class classifier. It is especially useful when the dataset is unbalanced (Matthews, 1975). Values vary between -1 for a completely wrong classifier and 1 for a perfect classifier.

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

The ROC curve is created by plotting the true positive rate against the false positive rate at various classifier thresholds. The AUC is the area under the ROC curve, it provides a value to evaluate the quality of the classifier. A value of AUC=0.5 refers to a random classifier and the higher the value, the better the classifier.

To have a good idea of the performance of each machine learner, each of the previous metrics have to be considered in the evaluation. We want a balance between precision and recall because we want as much as possible to detect real technical debts and all of them. We cannot only use F_1 score because we want to take into account the effect of chance on predictions made. Thats is why we also computed the MCC and AUC values.

In addition to these performance indicators, we also consider the importance of each features during the training of the predictors. We used a technique implemented in Weka for Random Forests named Mean Decrease Impurity (MDI) (Louppe et al., 2013), which measures the importance of variables on randomized decision trees.

For **RQ3**, we want to compare TEDIOUS with a popular method-level smell detector (DECOR) (Moha et al., 2010) in classifying as technical debt methods labeled as SATD. DECOR can detect a large amount of smells, but most of them are at class-level, which are not relevant with the level at which TEDIOUS works. Instead of using all of them, we narrowed our analysis to two method-level smells, *Long Method* and *Long Parameter List*.

To identify a Long Method smell, DECOR follows the rule $LOC > th_1$ where $> th_1$ is a threshold for the LOC. To identify a Long Parameter List smell, DECOR follows the rule $ParNbr > th_2$ where $> th_2$ is a threshold for the number of parameters. Various thresholds for LOC and ParNbr were considered in our research, between percentile 0.5 and 0.95, as well as the default thresholds, more specifically percentile 0.75 for LOC and outlier (third quartile $+1.5 \cdot IQR$ (interquartile range)) for ParNbr.

To finish, we performed a qualitative analysis on false positives and false negatives examples we obtained when evaluating our predictors. Its purpose is to complement our quantitative analysis and discuss the limitations of our approach in recommending TD with real examples.

CHAPTER 4 ANALYSIS OF STUDY RESULTS AND THREATS TO VALIDITY

4.1 Study Results

This section reports study results in the context of each research question. Tables provide visual representations and summarize the main results. More in depth analysis is discussed for the three research questions and the qualitative analysis.

4.1.1 How does TEDIOUS work for recommending SATD within-project?

Table 4.1 presents the average performance results of a 10-fold cross validation within-project executed 10 times and with five different machine learners. The average was computed for the 9 studied projects. The 10-fold cross validation was performed with balancing using SMOTE and without balancing.

On the unbalanced dataset, the best classifier is the one using the Random Forests algorithm. It achieves the best balance between precision (49.97%) and recall (52.19%), obtaining a F_1 score of 47.15%, the highest of all machine learners. We also notice that the Bagging algorithm is performing almost as well as Random Forests, even obtaining a slightly better precision but a weaker F_1 score. The accuracy of Random Forests, which includes the correct classification of negatives (the vast majority of the data), is 93.32% and almost all the other machine learners obtain an accuracy higher than 90%, between $[89.01\% - 93.35\%]$. MCC is on average > 0.4, which is translated into a moderate correlation, and AUC is > 0.9 (close to a perfect classifier) for Random Forests, Bagging and Bayesian, and > 0.7 for j48 and Random Trees.

On the balanced dataset, the best classifier is still Random Forests, with a precision of 26.56%, recall of 68.26% and F_1 score of 36.04%. Its MCC value is the highest at 0.37, which translate to a moderate correlation, and the same goes for its AUC value which is the same as previously, 0.92. The purpose of balancing is achieved since the recall of each machine learners is higher than previously but at the expense of precision. There is a clear gap between the Bayesian classifier and the others, it is definitely performing more poorly. It achieves the worst performance values, except for the recall which is excellent but not enough to compensate. In fact, it performs like a random classifier if we look at the MCC value which is close to 0. The other classifiers all performed similarly, having a precision between $[16.03\% - 18.40\%]$, a recall between $[63.22\% - 75.12\%]$ and a F_1 score around 25%.

Table 4.1 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

Their MCC value is around 0.3, which translates to a fair correlation and the AUC value is decent at 0.7 or more. However, the results are globally weaker with balancing than without it.

Table 4.2 highlights the within-project prediction results for each system, using Random Forests, and using balancing or not. Random Forests only was used since it was the best classifier based on Table 4.1. If we look at the unbalanced dataset, two systems are performing way worse than the others, namely Ant and jEdit. There is a reason behind this if we look back at Table 3.1. The analyzed projects, other than jFreeChart with 18%, all have a percentage of their methods containing SATD below or equal to 5%. Ant only has 0.5% of its methods containing SATD and jEdit only 2%. This explains the low performance values of Ant (precision 0.91%, recall 16.39% and F_1 score 1.73%) and jEdit (precision 5.24%, recall 25.71% and F_1 score 8.71%). The AUC values are still decent, respectively with 0.77 and 0.81, but the MCC values, respectively with 0 and 0.06, clearly prove us that the classifier is as good as a random one.

jFreeChart is the project containing the most number of SATD and is consequently the project where the classifier performs the best. It obtains high precision and recall (84.58% and 82.52%) as well as high MCC and AUC (0.83 and 0.99). Performance values on the other 6 projects are also decent, the F_1 score is almost always $> 50\%$, the MCC is between $[0.47 - 0.64]$ which translates to a moderate to strong correlation, and the AUC is in the interval $[0.91 - 0.97]$. We notice that the prediction performance of TEDIOUS is dependent on the system and not only the number of SATD it is trained on. Squirrel has a slightly higher percentage of SATD methods than Ant and slightly lower than jEdit, but it still performs significantly better than these two projects (73.33% precision and 44.44% recall).

Table 4.2 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	93.40	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

If we look at the balanced dataset, the same trend is observed as in the balanced dataset but with lower performance results. Precision is generally lower except for Ant and jEdit which obtain a small improvement. These two systems should normally benefit from balancing but the data from the few SATDs is not even enough to build a decent artificial training set, leading to a negligible gain in precision. However, as expected with balancing, we see a decent increase in recall and the same goes for the other systems. Generally speaking, the accuracy, F_1 score, MCC and AUC is better for Ant and jEdit only, the other systems did not benefit from the balancing.

Table 4.3 reports the top 10 features in within-project prediction according to the MDI technique. The importance of each feature is ranked for each system. Four features are in the top 10 of all projects, and they are all source code metrics. The most important one is the readability metric of Buse and Weimer (2008), which is the top feature for 7 systems. This observation is contrasting with the work of Bavota and Russo (2016) where they found that there was little correlation between SATD and code quality metrics as well as readability. The difference may lie in the fact that TEDIOUS works at method-level and not class-level

Table 4.3 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	—	—	—	—	—	—	—	—

like in the study of Bavota and Russo (2016). A class contains several methods, some can be readable and others not really. This previous study may not have been able to work at granularity fine enough to detect these potential SATDs. The other three top features are the number of declarations (*DeclNbr*), number of of parameters (*ParNbr*) and the number of lines of code (*LOC*). Having *ParNbr* and *LOC* in the top features is interesting because they are typical metrics used in smell detectors. In fact, for $\mathbf{RQ}_3$, we studied how a smell detector compares with TEDIOUS by relying solely on *Long Method* and *Long Parameter List* smell detection.

Other important features, which appears in the top 10 of over half of the systems, are the number of expressions (*ExprStmtNbr*), the McCabe cyclomatic complexity (*McCabe*) and the number of comments (*CommentNbr*). For *CommentNbr*, the SATD comments were excluded in order to keep the prediction unbiased. All these metrics are also source code metrics.

The other features are all warning checks from CheckStyle and PMD. The length of lines (*LineLength*) and *LocalVariableCouldBeFinal* warnings are the most common, with the others being relevant for specific systems. Most of these features relate to poorly written code. For example, *LineLength* checks for long lines, which are hard to read in printouts or if the coding screen space is limited. *LocalVariableCouldBeFinal* refers to a variable that is

assigned only once. *LocalVariableName* is the most important feature for COLUMBA and it checks for single-character variables or local variables with same name in different scopes. *UseStringBufferForStringAppends* is the most important feature in ANT and it checks if their is a non-trivial amount of the operator += for appending strings in the source code. For further details on the meaning of each warning, you can refer to the documentation of CheckStyle[1] and PMD[2].

We notice that two Checkstyle warnings, *ParameterNumber* and *MethodLength*, are very similar to two source code metrics, namely *ParNbr* and *LOC*. Intuitively, they should have been removed by the Spearman's analysis since they seem strongly correlated to these source code metrics. However, differently from the metrics, the warnings are boolean features. CheckStyle looks if there are over 7 parameters for *ParameterNumber* and over 150 LOC for *LOC*, returning *TRUE* or *FALSE* accordingly. This is why these warnings were not remove by Spearman's analysis.

RQ_1 summary: Random Forests classifiers achieve the best average performance for within-project prediction of design technical debts to recommend. Precision of 49.97%, recall of 52.19% and F_1 score of 47.15% are achieved for an unbalanced dataset. When using Random Forests on each system, high MCC and AUC values indicate healthy classifiers except for the ones with a small number of SATD instances. Balancing does improve recall but it does not result in better classifiers because of a substantial decrease in precision. Code readability, complexity and size are the most useful features in building the predictors, for all systems, in addition to some system-specific analysis checks.

4.1.2 How does TEDIOUS work for recommending SATD across-project?

Table 4.4 highlights the average performance of different machine learners for cross-project prediction. The process is similar to **RQ1**, instead of performing a cross-validation on each system individually, the classifier is trained on 8 systems and tested on 1 system. The classifiers are trained with a balanced dataset (top section of the table) and with an unbalanced one (bottom section). The same trend is observed for within-project and cross-project predictions: Random Forests outperforms the other machine learners and rebalancing does not provide a significant payoff.

On the unbalanced dataset, the best classifier is the one using the Random Forests algorithm. It achieves the best precision (67.22%), a good recall (54.89%), and the best F_1 score (55.43%). Compared to the within-project results, the improvement in precision

[1] `http://checkstyle.sourceforge.net/checks.html`
[2] `https://pmd.github.io/pmd-5.5.5/pmd-java/rules/index.html`

Table 4.4 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

is +17.25%, in recall +2.70% and in F_1 score +8.28%. For the other machine learners, the precision vary between $[48.31\% - 58.85\%]$ and the recall between $[51.62\% - 64.25\%]$. MCC is always > 0.4 (moderate correlation) and AUC > 0.7.

On the balanced dataset, the best classifier is still Random Forests, with a precision of 47.49%, recall of 78.75% and F_1 score of 56.45%. The F_1 score is slightly better than without balancing, but as expected precision suffers a large loss in order to obtain higher recall. We also notice that MCC and AUC values are slightly lower than without balancing. A similar trend is observed for the other machine learners, with Bayesian suffering the most from rebalancing, performing almost like a random classifier (MCC near 0). It is important to notice that, other than Random Forests, none of the other algorithms obtain a higher F_1 score with balancing.

Table 4.5 reports the cross-project prediction results for each system, using the best classifier, Random Forests, and using balancing or not. The top part of the table is without balancing results and the bottom part is with rebalancing using SMOTE. For the balanced dataset, Random Forests machine learner performs the best on the same systems as in within-project prediction. Systems with the lowest percentage of SATD methods are also the ones with the weakest performance results, namely jRuby, jEdit and Ant. These systems have a low percentage of SATD methods ($< 2.15\%$) and can't achieve a F_1 score $> 37\%$, the other 6 systems all have a F_1 score $> 53\%$. jRuby's performance metrics are significantly worse than in within-project prediction but it is the only system that experiences this decrease, the 8 others are almost all improving. We notice that, even though Squirrel also have a small amount of SATD methods (1.42%), it still achieves a precision of 48.75% and a recall of 70.62%. It shows that not only the number of SATDs but also the features and context of

Table 4.5 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

these SATDs is important for the prediction quality of a classifier.

Further analysis of methods' characteristics labeled as SATD in jRuby, jEdit and Ant would be necessary to try and understand their low performance results. One possible explanation is that SATDs from the testing set could greatly differ from the ones in the training set, making the training phase not optimal for proper classification of unlabeled data. Finally, jFreeChart still have the best performance values (precision of 94.89%, recall of 95.98% and F_1 score of 95.43%) with ArgoUML being pretty close (precision of 94.46%, recall of 88.29% and F_1 score of 91.27%). They both have MCC values > 0.85, which translates to very strong correlation.

For the balanced dataset, results do not seem to be improving. Some systems benefit from the rebalancing on certain levels and others do not. In other words, the results are in line or lower than what is obtained without balancing. As expected, recall increases at the expense of precision. Accuracy is generally lower and no real benefits are observed on MCC and AUC values. ArgoUML achieves the highest F_1 score (91.18%) with jFreeChart

being really close to it (90.44%). The only system that really benefits from rebalancing is JRUBY with the following improvements: recall +87.99% and F_1 score +55.79%. Precision obviously decreased but is still at a reasonable value of 50.50%.

To summarize, we can conclude that SMOTE rebalancing does not help a lot because (i) the very limited samples of positive examples is not enough to act as a seed for the generation of articifial instances and (ii) static analysis warnings are sparse and have a boolean nature, which is not appropriate for a proper usage of SMOTE. We can also conclude that cross-project prediction can be very beneficial, except for systems with a small amount of SATDs. The main reason is that, for cross-project prediction, the number of SATD methods in the training set is larger than for within-project prediction.

Table 4.6 reports the top 10 features in cross-project prediction according to the MDI technique. The same features as in within-project predictions are at the top, the most important being the source code metrics. *Readability* is still the feature playing the biggest role, for all the systems, followed by *LOC* once again and *CommentNbr* which moves up 4 ranks. *ParNbr* drops some ranks and becomes less important than other metrics capturing the code size and complexity, namely *DeclNbr*, *ExprStmtNbr* and *McCabe*. In within-project prediction, 22 warning checks were in the top 10 of at least one system, in cross-project prediction, there are only 4 of them (*LocalVariableCouldBeFinal*, *MethodArgumentCouldBeFinal*, *FinalParameters* and *LineLength*). However, these checks are in the top 10 of more systems ($> 4 systems$). These checks are related to declaring final variables, parameters not being reassigned and the length of lines. Again, two checks seem correlated, namely *FinalParameters* and *MethodArgumentCouldBeFinal*. Both were kept after the Spearman's analysis because the latter recommends the argument to be FINAL only if it is not reassigned, *FinalParameters* does not.

RQ_2 summary: Results from the cross-project prediction are similar to within-project predictions: Random Forests is the machine learner which performs the best, rebalancing does not provide significant performance improvements and the same systems achieve the best results. However, cross-project prediction globally achieves better performance values in recommending technical debts to self admit because of a larger and more diverse dataset. Code readability, size and complexity are the most important characteristics used to recommend design SATD.

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

Table 4.6 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

Table 4.7 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

Table 4.7 reports the overall DECOR performances in predicting SATD. They rely in the detection of *Long Method* and *Long Parameter List* smells by DECOR, and the union of both. In other terms, we want to know how well this smell detector can recommend technical debts based on the detection of these smells. DECOR was tested with thresholds at different percentiles of LOC and number of parameters. The unions of the two smells are done for same threshold values.

As we see in the table, DECOR's performances are never as good as TEDIOUS's performances. Precision is always $< 25\%$, recall is always $< 70\%$, F_1 score is at most 23.05% and MCC values are all < 0.17, which translates to low correlation. *Long Parameter List* gives a better balance than *Long Method* between precision and recall, and slightly better results. The union of both gives decent recall values but generally low precision and it does not seem beneficial to the predictions' performances.

RQ_3 summary: *LOC* and *number of parameters* metrics play an important role in within-project and cross-project predictions using TEDIOUS. However, Long Method and Long Parameter List smell detectors of DECOR are not able to achieve performances comparable to TEDIOUS.

4.1.4 Qualitative discussion of false positive and false negatives

In this section, we discuss some examples out of 100 reported SATD methods that we manually inspected. The purpose behind this analysis is to explain cases where TEDIOUS correctly or incorrectly classified SATDs.

As examples of **true positives**, we have in ArgoUML two methods labeled as SATD with different source code metrics values: *createFlow* in class CoreFactoryEUMLImpl and *invokeFeature* in class ModelAccessModelInterpreter. The first method has a $Readability \approx 1$, $LOC = 2$ and $McCabe = 1$. The second method has a $Readability = 0$, $LOC = 755$ and $McCabe = 178$. Even though both methods have obvious differences in the values of features defining them, TEDIOUS was still able to detect them as being SATD prone. It proves that our approach can detect a wide variety of methods containing a technical debt based on their characteristics.

As an example of a **false positive**, we have in jEdit the method *initialize* in class RE with the following characteristics: $LOC = 511$, $NbParameters = 5$, $NbCalls = 197$, $NbDeclarations = 32$, $NbExpressions = 618$, $NbComments = 97$, $McCabe = 102$ and $Readability = 0$. The readability is obviously null considering the size of the method. TEDIOUS clearly classified this method has a SATD while it is not. In fact, this method plays

a major role in the class and is intrinsically complex. It may not contain a technical debt, but it may require some improvements to make it more understandable. Recommendations on such long and complex methods should not be worthless or annoying for developers. It should be taken as a hint that this kind of method has to be carefully implemented because of their complex nature, making sure that it is as readable and understandable as it can be.

As an example of a **false negative**, we have in COLUMBA the method *start* in class COLUMBASERVER that is labeled as a design SATD but that was classified as a non-SATD method by TEDIOUS. The SATD comment linked to the method is positioned immediately after an IF statement and it says *"something is very wrong here"*. The developer intentionally mentions that the block of code is problematic, consequently leading external viewers to believing that a technical debt may be present. However, if such viewers analyze the structure of the method, nothing could justify the presence of a technical debt. This means that there is a deeper level to the characteristics defining a technical debt, other than structural and source code metrics. The TD in this case could be justified if we look at the bigger picture of the class or its context, not only its metrics. In other words, there are cases where TEDIOUS could not properly identify the presence of technical debts only using source code and structural metrics, which limits it applicability.

4.2 Threats to Validity

Here are the threats to validity of our research, based on the guidelines for case study research (Yin, 2013). We identified 5 threats: construct, internal, conclusion, reliability and external validity threat.

4.2.1 Construct validity

Construct validity threats concern the relationship between theory and observation. These threats are mainly due to measurement errors of metrics and labeled design SATD. Different calculation processes can result in different values of source code metrics, and the same goes for warnings obtained from static analysis tools. We used CheckStyle and PMD, but several other tools are available.

As for the SATD, we used the dataset from Maldonado et al. (2017b), where they annotated comments, labeling them as SATD or not. We used this information to build our oracle and some preprocessing had to be done because the information was gathered at file-level, and not method-level. Pattern matching was performed to link SATD comments to their respective methods, which may have introduced some imprecisions. We established

a strict threshold to match SATD comments, which could be revised. Indeed, we only accepted the exact identity of comments from the dataset of Maldonado et al. (2017b) with the ones from the source code. Not all comments were matched using this approach because of a different processing chain, consequently, some methods which are in fact containing a technical debt were not tagged as such. In the future, we plan to revise those matches which are close to being perfect but are not, to draw a more accurate picture of the systems.

However, this aspect can be detrimental and beneficial for TEDIOUS. On one side, the learning phase is more difficult since the process creates more false negatives (methods tagged as non-SATD but that are SATD). On the other side, these false negatives would make a more balanced dataset, if they can be traced back by our approach, leading to improved performance. Additionally, any errors made by Maldonado et al. (2017b) when analyzing the systems would have an impact on the accuracy of our approach. Finally, some comments in the dataset exactly matched more than one comment in the source code. When we encountered such cases, we tagged these comments as *Maybe*, since we could not exactly classify them.

4.2.2 Internal validity

Internal Validity threats concern internal factors that could have influenced our results. Several of them can be identified, (i) machine learners have been applied only with default settings, (ii) CheckStyle and PMD were used only with default configurations, (iii) source code metrics were computed with srcML and (iv) EMF project was not used in our dataset.

The fact that machine learners were built with default configurations only means that better results could have been obtained with a proper parameter optimization. In the worst case, this only means that our results are the lower-bound of what could be achieved. For Checkstyle and PMD, only default rules provided by the tools were used for smell detection. Similarly to machine learners, a proper calibration of rules could have resulted in a different set of warnings which would have depicted a different view of the systems. Consequently, the usefulness of checks could have been improved for the prediction. In fact, we plan to use SATD to help customize CheckStyle and PMD rules.

Source code metrics were computed using srcML (Collard et al., 2013) (except for the readability metric which was computed using the tool of Buse and Weimer (2008)), therefore, other metrics extractors could have provided different results. However, the purpose of this thesis was not to evaluate our approach for specific static analysis tools and metrics extractors. These tools were selected because of their ease of use and their popularity. They were complementary to the realization of TEDIOUS. The main purpose of our work was to

highlight the potential of learning technical debts to self-admit from source code features.

We did not include EMF project to our dataset because we were unable to download the archive release 2.4.1. Since we already have 9 systems, some with similar amounts of classes (ArgoUML, Hibernate and Columba) or a smaller amount (Squirrel), we believe omitting EMF will not bias our results, even though it is the largest in terms of LOC.

4.2.3 Conclusion validity

Conclusion Validity threats concern the relationship between treatments and outcomes. To avoid these threats, we use appropriate metrics to quantify the performance of machine learners (AUC and MCC) and tools to compute the importance of learning features (MDI). We use these diagnostics in addition to thresholds to define the acceptability of our outcomes ($AUC > 0.5$ and $MCC > 0$, the closer to 1.0 the better).

4.2.4 Reliability validity

Reliability validity threats concern the replicability of our research. We attempt to provide, as far as we can, all the necessary information to replicate our approach. We plan to share a replication package containing: source code, raw data and scripts.

4.2.5 External validity

External validity threats concern the generalization of our results. We cannot guarantee that our results can be generalized to all Java programs even though we used the same systems from a previous study (Maldonado et al., 2017b) and even if the systems cover different domains. Additionally, our dataset is somewhat limited since we only have 9 systems. As future work, more studies will be required to verify the extent at which TEDIOUS can be employed and how our findings can be generalized to other projects, domains and programming languages.

CHAPTER 5 CONVOLUTIONAL NEURAL NETWORK WITH COMMENTS AND SOURCE CODE

5.1 Convolutional Neural Network

Several machine learners were tested with TEDIOUS, using source code metrics as training features. Results were promising but the approach asked for a lot of preparation work: building the XML representation of the Java code, pattern matching between comments from the dataset and the original source code, extracting source code metrics, extracting warnings raised by automated static analysis tools and feature preprocessing. These preparation steps take time and require the knowledge of the whole process. Consequently, we wanted to experiment with a novel approach, easier and faster to set up. We decided to test a Convolutional Neural Network (CNN) with Natural Language Processing (NLP) directly on the Java source code of software projects.

The idea of using a CNN was inspired by a paper written by Kim (2014). He used a CNN for sentence-level classification tasks and showed that you can achieve excellent performance results with little parameter calibration. To prove his point, he tested his model on a wide variety of benchmarks. Dos Santos and Gatti (2014) performed a similar work concerning sentiment analysis of short texts. More specifically, they analyzed Twitter messages and movie reviews, and tried to classify them as being of positive or negative sentiment. Our idea is similar to these studies, we plan to use a CNN to classify comments and/or methods using the source code directly instead of features, labeling them as technical debts or not.

Typically, convolutional neural networks were employed for image classification (Krizhevsky et al., 2012), however, this type of neural network has also been used recently combined with NLP. To describe CNN in further details, we can think of a convolution as a window sliding across a whole matrix. This window is in fact acting like a filter. For images, this matrix contains pixels, for words and sentences, it contains word vectors (word embeddings). In image classification, filters slide over local batches, in NLP, they slide over entire rows since a row is typically a single word embedded into a row matrix of the size of the embedding dimension. To implement a CNN, you just have to add several layers of these convolutions, where each of these layers have a specific task and act as different filters. CNNs are very fast and efficient in terms of representation, which means they give good representations without needing the whole vocabulary of a dataset. Figure 5.1 presents an example of a CNN architecture.

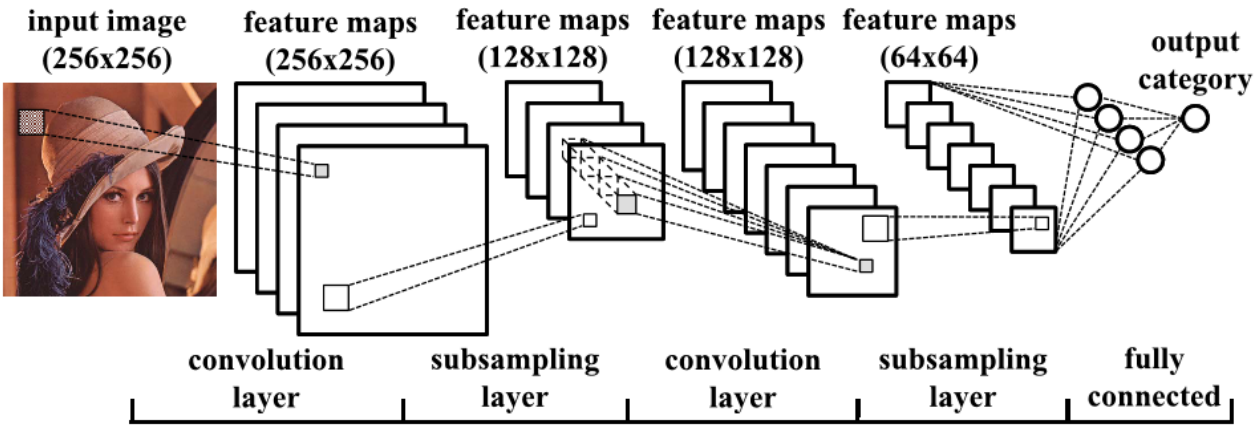

Figure 5.1 An example of a convolutional neural network (Cong and Xiao, 2014).

The main purpose of this thesis was to explain, test and analyze TEDIOUS, our machine learning approach using source code features. The next sections will describe the other approach using a CNN, but less in depth than for TEDIOUS since it is still at its preliminary stages. However, we still judged interesting to present the initial results of this new promising method. Since some steps from Chapter 3 are replicated in our CNN approach, they will be reviewed rapidly in order to avoid redundancy.

5.2 The Approach

This section will describe the steps followed to design this new approach, a convolutional neural network combined with natural language processing to identify technical debts to self-admit. Additional information will be shared on the characteristics of this machine learner and how it works. Like TEDIOUS, this approach works at method-level since it is the granularity at which we are most likely to detect TD (Potdar and Shihab, 2014). In other terms, this approach is able to detect if a technical debt is contained in a method or not.

As shown in Figure 5.2, two datasets are required to build our model: the training and test set. The training set contains labeled data, which is source code where SATD are known. The test set contains unlabeled data, which can be any source code where we want to recommend where technical debts should be admitted. The presence and location of TDs are unknown in the test set.

For the training set, various combinations of source code and comments, labeled as containing a SATD or not, are extracted: source code comments only, source code with comments, source code without comments and source code partially with comments. It is

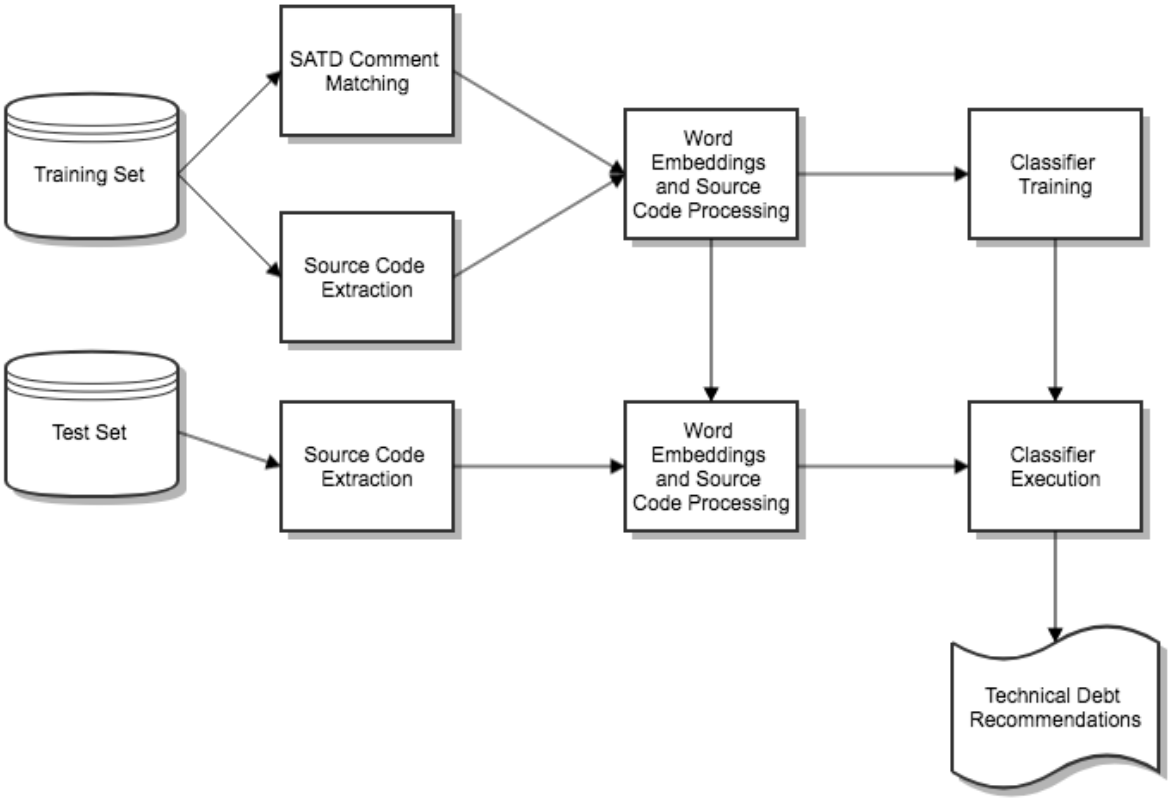

Figure 5.2 Proposed approach for recommending SATD with a CNN.

essential to have this classification because the CNN is based on supervised learning. Pattern matching using comments from the dataset of Maldonado et al. (2017b) is employed to classify methods and comments.

Once all the source code is extracted and classified, some preprocessing has to be done. The source code is tokenized, the purpose being to demarcate and transform the source code into a string of word tokens. The comments specifically are cleaned up to remove extra spaces, non-ASCI characters, upper case letters, etc. Once the source code is preprocessed, word embeddings is performed, which means strings of tokens are transformed into vectors of numerical values using word2vec tool (wor). With the source code preprocessed and the oracle built, the model can be trained with the CNN.

But before, in parallel, the test set is prepared. The same combinations of source code and comments are extracted, but SATD matching is not required because the data is unlabeled and we want to predict the presence of technical debts. The matching of SATDs is

only required for the oracle to train models. The same preprocessing and word embeddings are applied on the test dataset. With the previously trained model and the test set, predictions can be made in order to recommend when to self-admit technical debts.

An overview of the process was described in this section, but more details will be shared in the next ones. We will discuss the source code and its nature, how we identified the SATD, how we performed word embeddings and the process to train and apply the models generated by the CNN.

5.2.1 Source Code

The main difference between TEDIOUS and this new approach is the nature of the training features. For TEDIOUS, source code metrics and warnings were used. For the CNN, we use the source code itself, transformed into word vectors. The same process as before was performed to extract the source code, where an XML representation of the Java source code was generated using the srcML tool (Collard et al., 2013). To build the oracle, SATD comments from the dataset of Maldonado et al. (2017a) were linked to their respective methods in the studied projects in order to classify them as containing a technical debt or not. The same rules were followed, as explained in section 3.1.1 Features for TEDIOUS. Once the XML representation is obtained, the different dataset combinations can be generated.

The first feature combination is *source code comments only*. They are extracted from the dataset of comments provided by Maldonado et al. (2017b). Comments can be encountered under different forms: single line, multiple lines or block. Single line comments are comments written on a single line of code. They use the (// ...) commenting method. Multiple line comments are several single line comments grouped together. Block comments are comments written over several lines of code but that are considered as a single entity. They use the (/* ... */) commenting method.

Contrary to TEDIOUS, it is important to specify that not solely design debts were retained but all of them: defect, design, documentation, implementation and test. We decided to use as many SATD-methods as possible in order to have the best chances of getting good prediction results. By using different types of technical debts, we decrease the unbalance of the dataset by adding more positive examples.

The second combination is *source code with comments* where the complete XML representation of the source code is used. The third combination is *source code without comments* where the XML representation is parsed to remove comments. The fourth combination is *source code partially with comments*, which means only comments related to SATD are re-

moved. The reason behind this removal is to be sure to avoid the CNN model being a self-prophecy. For these three datasets, only design and implementation TDs are retained, to be more in line with the dataset used by TEDIOUS. The details of the process behind the extraction of each combination will be explained in the Source Code Preprocessing section.

5.2.2 Identification of Self-Admitted Technical Debt

Like TEDIOUS, the purpose of this approach is not to propose a new way to detect SATD using information from comments. However, will still needed a classified dataset of SATD comments in order to train our CNN model. We used the dataset of Maldonado et al. (2017b), which contains a classification of 10 open source projects, where comments are tagged as relating to a technical debt or not. Various types of TDs are considered, depending on the source code combination. The dataset reports SATD at file-level instead of method-level, consequently, some preprocessing had to be performed using pattern matching to tag SATD comments to their related methods.

5.2.3 Source Code Preprocessing and Word Embeddings

The source code is extracted in a XML format, which is not quite compatible for the machine learner to train on. To make it compatible, XML files have to be tokenized. Instead of using standard coding lexicon (conditional statements, variables types and names, parameters declaration) and separators (brackets, parentheses, spaces) directly in our dataset, demarcations are added (*i.e. begin_type, end_type*) to transform the structure into series of word tokens. For comments, strings are normalized: extra spaces are removed, upper cases are transformed to lower cases, non-ASCI are removed as well as new lines. Also, if a comment is matched with a SATD pattern, it will be printed as:

{COMMENT_BEGIN_SATD COMMENT-NORMALIZED-TEXT COMMENT_END_SATD}

This step is essential to build the oracle and the dataset partially with comments. By explicitly defining comments tagged as SATD in this format, it is easy to parse the tokenized dataset in order to remove them. Comments are linked to their respective method, so a method linked to a SATD comment will also be tagged as SATD. By tokenizing the source code, it is also easier to remove the comments entirely, if necessary. Transforming the source code into tokens also acts as a normalization process, which will make the word embedding process more efficient.

Word embeddings is the process of transforming words and phrases from the dataset into vectors of real number. Word2vec models were used to generate word embeddings.

The vector dimension we used is 100 because we tried to have a balance between a proper representation of words and processing time. A new word embedding was generated for each source code combination since they are all different in some ways.

Finally, the methods extracted are all tagged as positive or negative examples. In order for the CNN to be trained and tested, these methods have to be divided in 4 standard files: POSITIVE-TRAIN, NEGATIVE-TRAIN, POSITIVE-TEST and NEGATIVE-TRAIN. Each fold of the cross validation contains these 4 files, consequently, for a 10-fold cross validation, 40 files are required. Instead of using an additional feature to classify each method in a single file, the files act as classifier entities. Here is a detailed definition of each file, considering a 10-fold cross validation:

- *Positive-Train*: Training set containing SATD methods (90% of all positive examples)
- *Negative-Train*: Training set containing non-SATD methods (90% of all negative examples)
- *Positive-Test*: Testing set containing SATD methods (10% of all positive examples)
- *Negative-Test*: Testing set containing non-SATD methods (10% of all negative examples)

5.2.4 Building and Applying CNN

So far, we extracted source code from various projects, with or without comments, we identified SATD methods, we preprocessed the dataset and performed a word embedding. The only step remaining is building and applying the convolutional neural network. Four datasets are required, as described in the previous section: two sets for training, one containing SATD methods and the other not, and two sets for testing, one of positive and the other of negative examples. The sets contain the tokenized source code of the 9 studied projects, divided between them.

Figure 5.3 provides an overview of the CNN process. To increase the training speed and to benefit as much as possible from the processing power available, several threads are created. One thread is started for each fold, feeding it with a pair of previously generated training files. Up to five threads can be processed at the same time. The next actions are all performed simultaneously on each thread, until we have models trained for all folds.

The evaluation process consists of two main phases: the training and the testing phase. In the former, some data preparation is required: load the two training sets, build the

vocabulary, shuffle the datasets, and split the dataset in a training and development set. The development set is used to tune parameters of the CNN and to prevent it from over-fitting during the training process. Afterwards, the CNN is built using user-defined parameters and default values. In our case, mainly the default configuration is considered. The training procedure is defined before executing it and summaries are generated during the process for: loss, accuracy, train, development and model. Then, variables are initialized, such as embedding vectors, and training batches are generated. Finally, a training loop is executed where training and evaluation steps are repeated. The trained model is saved multiple times during the loop and the last one is used for the testing.

We can start the testing phase once the training is finished. Data preparation is also required for this step: load the two testing sets and map them into the vocabulary. Afterwards, the meta graph and the variables from the model previously trained are loaded. Testing batches are generated and tensors we want to evaluate are created. Then, we can start the testing loop where predictions are made. Once done, performance metrics can be computed: accuracy, recall, precision, specificity and F_1. These metrics are saved in a different file from the file containing predictions made on each method. The final step consists of combining performance metrics of all models for further analysis.

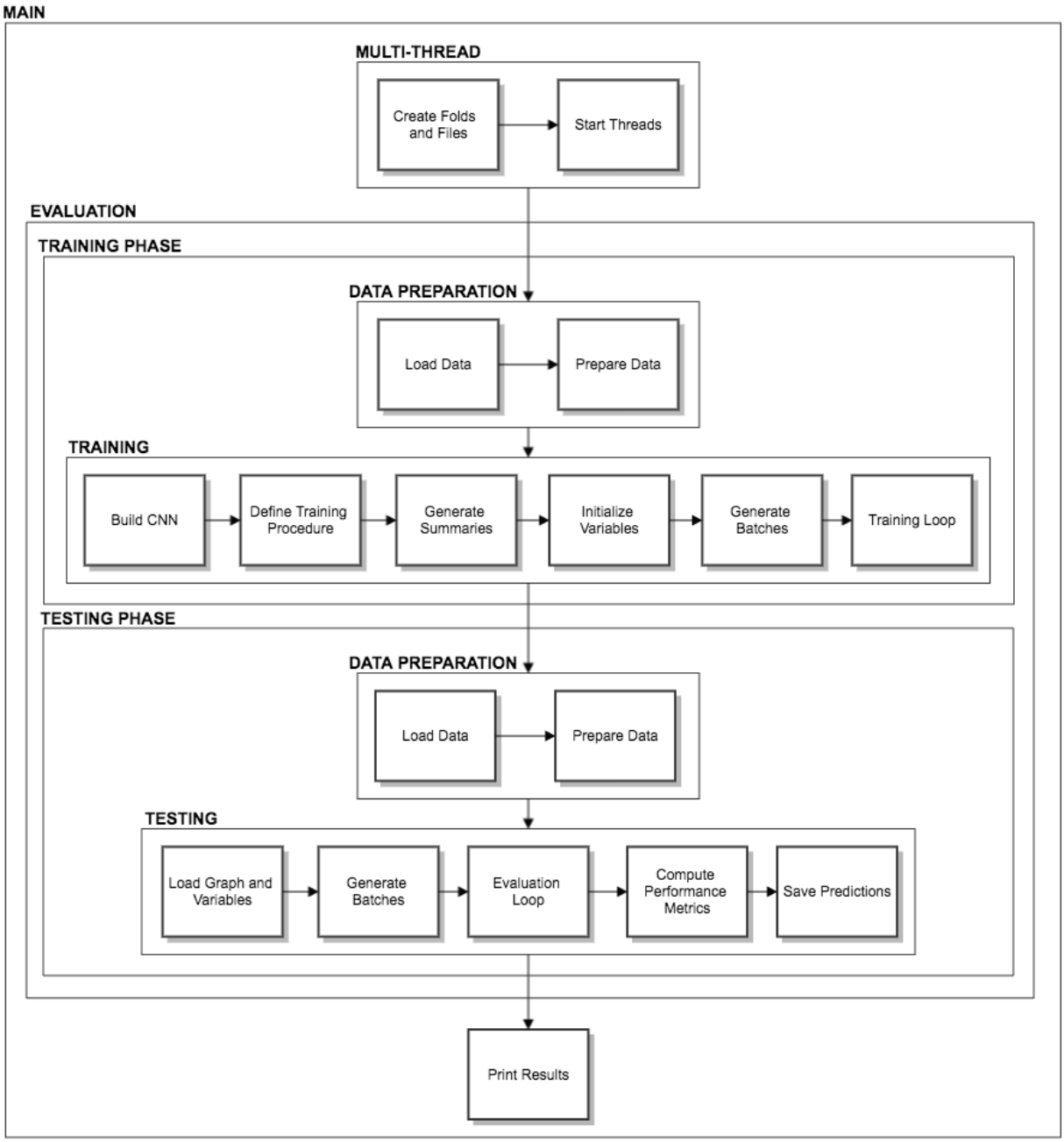

Figure 5.3 Process for building and applying a CNN.

Table 5.1 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%

5.3 Study Definition

The goal of this new approach is to assess the prediction performance of a convolutional neural network in recommending technical debts to self-admit. The focus is the same as for TEDIOUS, enhancing the source code quality by keeping track of TDs. The perspective is to be able to suggest to developers, more accurately than with TEDIOUS, when to admit technical debts. We aim to address four research questions:

- **RQ1**: How does a CNN work for recommending SATD with source code comments only?
- **RQ2**: How does a CNN work for recommending SATD with source code with comments?
- **RQ3**: How does a CNN work for recommending SATD with source code without comments?
- **RQ4**: How does a CNN work for recommending SATD with source code partially with comments?

5.3.1 Dataset

To evaluate this new approach, the same dataset was used as in TEDIOUS (Maldonado et al., 2017b). The methods are already classified as SATD or not, and Table 5.1 summarizes the characteristics of all studied projects. Various information describe the content and nature of each project. This table was already presented in Section 3.2.1. However, a brief overview is still necessary to reiterate important facts.

There are some disparities between the results we obtained when analyzing the studies and what Maldonado et al. (2017b) obtained. However, this does not really represent an issue since many of these differences concern classes while our CNN approach is method-level based, like TEDIOUS. This aspect is also important since we clearly see a prevalence of method-related rather than class-related SATD. Out of all methods in a project, only a very small amount contains a technical debt, making the dataset highly unbalanced. As we will discuss in the analysis of results, the lower the amount of technical debts in a system, the lower the prediction performance. Since the dataset from Maldonado et al. (2017b) classified classes instead of methods, we performed pattern matching between known SATD comments and comments attached to methods in the dataset.

5.3.2 Analysis Method

For **RQ1**, we want to know how a convolutional neural network with source code comments only work for recommending SATD within-project. We also want to compare the results with the within-project predictions of TEDIOUS. A 10-fold cross validation was performed on each project, like for TEDIOUS, and the performance values are averaged over the 10 iterations. The same process is followed for **RQ2**, **RQ3** and **RQ4**.

Standard performance metrics on the SATD category were computed to evaluate our automated classification approach: precision, recall, F_1 score and accuracy. Precision is the percentage of relevant instances of methods predicted as SATD among all retrieved instances. Recall is the percentage of relevant instances of SATD methods that have been retrieved over all relevant instances. F_1 score is the harmonic mean between precision and recall. Accuracy is the total number of methods correctly predicted, whether it is SATD-related or not, among all analyzed methods.

Unfortunately, metrics such as MCC, ROC and importance of features were not computed for this approach. However, we still have enough information to evaluate and compare each approach. The downside is that it will be more difficult to take into account the effect of chance on predictions. Overall, what we look for in a good classifier is a balance between precision and recall while aiming for the highest F_1 score. We want to detect as many technical debts as possible while being correct in our predictions.

5.4 Study Results

This section reports the results obtained using various combinations of source code. Performance metrics are presented in tables and further analysis is provided textually, discussing the metrics and comparing them with TEDIOUS.

5.4.1 Source Code Comments Only

Table 5.2 reports the within-project performance results of a 10-fold cross validation on each system using our convolutional neural network and source code comments only. We computed the average for the 10 folds of each system and for the complete dataset. Since we already had the comments preprocessed for TEDIOUS, this dataset was a good start to test our CNN approach.

Two systems perform significantly worse than the others, namely Ant and jEdit. We face the same problem as in TEDIOUS, where the very low percentage of SATD methods

Table 5.2 Within-project prediction: results of CNN for each system using source code comments only

Source Code Comments Only				
System	**Pr**	**Rc**	**F_1**	**Acc**
Ant	93.33	10.77	19.31	97.02
ArgoUML	91.21	90.94	91.07	97.24
Columba	97.12	71.05	82.07	99.08
Hibernate	95.29	73.19	82.79	95.12
jEdit	82.95	29.32	43.32	98.09
jFreeChart	100.00	69.38	81.92	98.50
jMeter	95.25	75.95	84.51	98.71
jRuby	96.14	87.17	91.43	97.89
Squirrel	95.81	65.59	77.87	98.51
Total	**93.63**	**76.17**	**84.00**	**97.99**

(Ant has 0.5% and jEdit has 2%) in some systems directly affects the performance of the CNN. Ant has a precision of 93.33%, recall of 10.77% and F_1 score of 19.31%. jEdit is a little better, it has a precision of 82.95%, recall of 29.32% and F_1 score of 43.32%. However, if we compare with results from TEDIOUS, we notice a significant improvement in precision while maintaining a similar recall for these two systems.

As for the other systems, the precision is $>$ 91%, the recall is $>$ 65% and the F_1 $>$ 77%. Differently from TEDIOUS where the best results were obtained with jFreeChart, ArgoUML is the system where the CNN performed the best (precision 91.21%, recall 90.94% and F_1 91.07%). jFreeChart still obtained the best precision (100%) but other projects such as jRuby or jMeter provided more balanced performance values. We have to be careful when comparing results in this scenario because all types of technical debts were considered whereas TEDIOUS only analyzed design debts.

However, on first look, it seems that the CNN approach using source code is an improvement of TEDIOUS using source code features. The total average precision, recall and F_1 score is even higher than the best system using TEDIOUS. Further testing is needed to provide a better understanding of the performance of the CNN, which will be accomplished in the next sections.

5.4.2 Source Code With Comments

Using source code comments only is an interesting first step to train a convolutional neural network to predict technical debts, especially if we aim at addressing the issue of self-admitted technical debts. However, it would also be interesting to see how well a CNN can perform using the entirety of a project's source code, code and comments included. We experimented with such a dataset and obtained the results reported in Table 5.3 for a within project 10-fold cross validation.

Again, the worst results were obtained for ANT and JEDIT. However, compared to source code comments only, ANT improved its precision by +1.91%, recall by +55.90% and F_1 by +59.12, and JEDIT improved its precision to 100% but suffered some losses in its recall and F_1 score. We notice that the unbalance of the dataset is still an issue for the CNN but these results are still better than what we obtained with TEDIOUS.

As for the other systems, the precision varies between $[93.62\% - 98.57\%]$, the recall between $[76.80\% - 90.72\%]$ and the F_1 score between $[84.67\% - 94.09\%]$. Precision is generally slightly better and the recall gains a decent improvement compared to the source code comments only dataset. The best results were obtained with JFREECHART (precision 97.73%, recall 90.72% and F_1 score 94.09%), like in TEDIOUS but not the previous dataset. This can be explained by the fact that we only used design and implementation debts, compared to all types of debt for source code comments only. Consequently, results with this dataset should normally be more similar to the results of TEDIOUS.

If we look at the average total performance values, there is an improvement compared to the source code comments only dataset: precision +2.75%, recall +6.23% and F_1 +4.84%. Consequently, using the whole source code is also an improvement compared to TEDIOUS, in a pretty significant manner. We just tested the best case scenario where we have as much information as possible to train on, however, we also want to know how well a CNN can perform on a less than ideal software project.

5.4.3 Source Code Without Comments

We trained our convolutional neural network on source code comments only and on source code in addition to comments. We obtained promising results, but we want know how well our machine learner can work with less training information. We generated another dataset, this time only containing the source code of the studied projects, eliminating the comments. We built such a dataset to replicate the worst case where a software project is lacking comments and to remove SATD comments related to their respective methods. Table 5.4 reports the

Table 5.3 Within-project prediction: results of CNN for each system using source code with comments

Source Code With Comments				
System	**Pr**	**Rc**	**F_1**	**Acc**
Ant	95.24	66.67	78.43	99.81
ArgoUML	97.32	87.88	92.36	98.84
Columba	93.62	80.00	86.27	99.66
Hibernate	95.61	89.71	92.56	99.72
jEdit	100.00	21.00	34.71	98.25
jFreeChart	97.73	90.72	94.09	99.72
jMeter	96.45	82.61	88.99	99.45
jRuby	94.35	76.80	84.67	98.70
Squirrel	98.57	85.19	91.39	99.92
Total	**96.38**	**82.40**	**88.84**	**99.44**

performance results for each system using a 10-fold cross validation and source code without comments as the training set.

Ant is still the system where the CNN performs the worst (precision 50.00%, recall 1.67% and F_1 3.23%), in line with every other experiment we performed so far. However, jEdit is performing decently compared to previously and, surprisingly, jFreeChart is average even though it contains the most design debts. It seems the unbalance of the dataset does not correlate as much with the results since jRuby obtains the best performance (precision 95.27%, recall 56.40% and F_1 70.85%) only with 2% of methods with design SATD.

As for the other systems, performance metrics all decreased compared to the last two datasets, especially the recall. If we compare the total average, at the most, precision decreases of -8.21%, recall of -48.72 and F_1 of -40.09%. Compared to TEDIOUS, the precision is generally better but the recall rate worst. It would be difficult, with the available information, to determine which machine learner performs better between TEDIOUS and CNN with source code without comments. However, it is safe to say that they are at least similar performance wise, with TEDIOUS providing better recall and the CNN better precision.

Overall, it is obvious that removing comments from the dataset impacts the performance of the CNN at all levels. It is clear that they represent an important piece of information for the machine learner to train on and should be kept in the dataset. We also notice that

Table 5.4 Within-project prediction: results of CNN for each system using source code without comments

Source Code Without Comments				
System	**Pr**	**Rc**	**F_1**	**Acc**
Ant	50.00	1.67	3.23	99.48
ArgoUML	89.78	37.27	52.68	94.67
Columba	84.21	14.55	24.81	98.81
Hibernate	79.66	27.65	41.05	98.43
jEdit	72.73	24.00	36.09	98.11
jFreeChart	73.68	17.72	28.57	97.85
jMeter	83.61	22.17	35.05	97.77
jRuby	95.27	56.40	70.85	97.84
Squirrel	88.89	17.78	29.63	99.56
Total	**88.18**	**33.68**	**48.75**	**98.12**

not only the amount of technical debts in a system is important to obtain good performance, but the nature of technical debts and the content of the source code as well. When removing comments, we observed that different systems would perform better than previously and other worst, despite having the same amount of SATD-methods. Knowing the importance of comments, one more experiment would be interesting to complete to better understand why.

5.4.4 Source Code Partially With Comments

So far, the best results were obtained with a dataset consisting of source code with comments. When removing all comments, we observed a significant loss in recall, and a modest one in precision. We now want to test an intermediate dataset where we keep all comments except the SATD comments manually analyzed by Maldonado et al. (2017b). The main reason behind the generation of this new dataset is to quantify the impact of these specific comments on the prediction performance and to know if they act as a self-prophecy. In other terms, we want to know how well our convolutional neural network can predict technical debts in methods if we remove comments self-admitting them. Table 5.5 reports the performance results for each system using a 10-fold cross validation and source code partially with comments.

AJOUTER LA TABLE CROSS PROJECT DE GIULIO. LES COMMENTS LIES AUX METHODS SATD SONT ENLEVES POUR EVITER SELF PROPHECY

All systems perform decently well and previously problematic ones, such as Ant and

Table 5.5 Within-project prediction: results of CNN for each system using source code partially with comments

Source Code Partially With Comments				
System	**Pr**	**Rc**	**F_1**	**Acc**
Ant	96.88	51.67	67.39	99.74
ArgoUML	95.40	60.81	74.28	96.65
Columba	100.00	39.09	56.21	99.18
Hibernate	87.68	54.41	67.15	98.95
jEdit	100.00	40.00	57.14	98.67
jFreeChart	93.68	68.78	79.32	99.13
jMeter	97.46	50.00	66.09	98.61
jRuby	97.44	68.60	80.52	98.45
Squirrel	87.23	45.56	59.85	99.68
Total	**94.84**	**58.83**	**72.61**	**98.82**

jEdit, are not outsiders anymore. The precision varies between [87.23% − 100.00%], the recall between [39.09% − 68.78%] and the F_1 score between [56.21% − 80.52%]. In other terms, the CNN is able to predict at least half of the technical debts in a system with a very high precision.

Compared to the dataset with all comments where results seemed somewhat dependent on the system, this one provides more balanced but weaker overall results. However, there is a net improvement compared to the dataset without comments. In fact, source code partially with comments positions itself right between the previous two datasets with a precision of 94.84%, recall of 58.83% and F_1 score of 72.61%. Compared to TEDIOUS, our CNN with source code partially with comments is also an improvement.

We observe that SATD comments have an important impact on the prediction performance. The amount of comments removed is very small compared to the total, however, the impact on performance is not negligible, mostly on recall. It seems like comments, especially SATD-related, are important features for our machine learner to train on. We observe this fact in the way each system position themselves compared to others, some will benefit from keeping SATD-related comments and others will not. In addition, performance metrics globally decrease when removing comments.

CHAPTER 6 CONCLUSION

6.1 Summary of Work

This thesis describes two approaches to recommend to developers when they should self-admit design technical debts. The main approach is TEDIOUS, a machine learner that uses features extracted from source code as independent variables and knowledge of SATD as dependent variables to recommend TD to be self-admitted. It is based on size, complexity, readability and checks from static analysis tools. The second approach, less developed than TEDIOUS, is a convolutional neural network using source code directly to recommend TD to be self-admitted. It is based on source code and comments.

Both approaches were evaluated on 9 projects manually analyzed and made publicly available by Maldonado et al. (2017b). For TEDIOUS, the within-project prediction gave on average a precision of 50%, recall of 52% and accuracy of 93% when recommending TDs. Oversampling was performed to compensate the highly unbalanced dataset (very few positive compared to negative instances) but without much benefits; it increased recall at the expense of precision. Cross-project prediction was also performed, with improved results compared to within-project. It gave on average a precision of 67%, recall of 55% and accuracy of 92% when recommending TDs. The best systems achieved a precision and recall rate $> 88\%$, namely jFreeChart and ArgoUML. Code readability, size and complexity played a major role in recommending design SATD, with static analysis tools being less important.

For the CNN approach, four experiments were conducted with no balancing performed. For *source code comments only* within-project prediction, an average precision of 93.63%, recall of 76.17% and accuracy 97.99% were obtained. For *source code with comments* within-project prediction, an average precision of 96.38%, recall of 82.40% and accuracy of 99.44% were obtained. Adding the implemented code to the dataset improved the results. For *source code without comments* within-project prediction, an average precision of 88.18%, recall of 33.68% and accuracy of 98.12% were obtained. Performance metrics were negatively affected by the removal of all comments. For *source code partially with comments* within-project prediction, an average precision of 94.84%, recall of 58.83% and accuracy of 98.82% were obtained, which is a better than no comments but worst than all comments included. **ADD CROSS PROJECT RESULTS.** Overall, each CNN experiment performed better than TEDIOUS.

6.2 Limitations of the Proposed Solution

TEDIOUS and our CNN approach have their share of limitations and constraints. The two approaches can't act as perfect classifiers because of the intrinsic nature of technical debts. You can try to describe methods with metrics and static analysis warnings, or use them as a whole, but there will always be a deeper level to their existence and a larger context that can be difficult for a machine learner to understand. We observed this aspect when qualitatively analyzing false negative examples of TEDIOUS. In other terms, the prediction performance of our proposed solution have a ceiling which is difficult to surpass.

For TEDIOUS, we are limited with the number of metrics and warnings we can use to train the machine learners. For the CNN, the number of LOC from the source code used for the dataset is also subject to a limit. The main reason behind this limitation is the processing time and computation power required (i) to extract these metrics and source code, and (ii) to train TEDIOUS and the CNN. We can't use as many training features as we want and have models trained in a decent amount of time with reasonable processing power.

In addition, there may be differences in feature values depending on the method to obtain them; the heavy preprocessing required to build datasets of TEDIOUS and the CNN approach can induce errors, and the manual classification of SATD comments is subject to human error. Also, default configurations only were used to collect warnings and train machine learners. Performance may be limited and could be increased by applying optimization on configuration parameters. Consequently, the results we obtained could be completely different, even though the same features are extracted, only because the process to extract them is different. In other terms, the replicability of our research is limited.

We can't assure that our results can be generalized to all Java projects even though we used systems previously studied (Maldonado et al., 2017b) which cover a wide variety of domains. Our dataset is limited, it only consists of 9 systems, and we cannot claim that TEDIOUS and the CNN can be accurate on every possible Java system. We already observe this fact in the various experiments we pursued. In addition, the applicability of our approaches is limited to Java programs only. Finally, the conclusion deduced from the analysis of the CNN results has to be taken with skepticism since it is still in its preliminary steps and the impact of randomness was not taken into account in its evaluation.

6.3 Future Work

Future work can be divided in two themes: applicability scenarios and improvement paths. Applicability scenarios are ideas on how TEDIOUS and the CNN approach could be used

concretely by developers in the industry. Improvement paths are tasks that have to be accomplished to improve and extend our approach.

For future applications, our approaches could act as recommendation systems for developers, suggesting when to self-admit technical debts with comments, which was the main goal of our research. Also, they could complement static analysis tools by helping to customize warning checks raised by them. Our machine learners could learn rules from previously detected SATDs. Additionally, TEDIOUS and our CNN could complement existing smell and anti pattern detectors, such as DECOR (Moha et al., 2010), to enhance their performance. Indeed, our research proved that TEDIOUS could outperform a smell detector of Long Methods and Long Parameter List.

Many improvement and work paths are already planned for the future. Both TEDIOUS and the CNN could benefit from configuration optimization. Only default parameters were used by the machine learners and extraction tools. More specifically, CNN results are still preliminary and performance metrics are still lacking. Further testing is required to better understand the real value of this approach. SATD comments were matched to methods with pattern matching, however, we retained only perfect matches, some comments did not match and others were very close to matching but were ultimately ignored. A revision of this process would be necessary to gather a more accurate picture of technical debts in systems.

As mentioned previously, our dataset is small and our approaches would benefit from a larger pool of examples to train on. More information could also be provided by adding other metrics or warnings to define methods. The same idea can be translated to the CNN by adding more source code. TEDIOUS only detects design debts and it would be interesting to see how well it performs for the prediction of all types of technical debts. The CNN already predicts all types of debts and performs better than TEDIOUS. We could also extend the application of our concept to other programing languages and domains.

Finally, TEDIOUS is based on a Random Forests algorithm and we have a convolutional neural network which gives good prediction results. It would be interesting to test another popular machine learner used with Natural Language Processing, which is a Recurrent Neural Network (RNN).

BIBLIOGRAPHY

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

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

"word2vec. `https://code.google.com/archive/p/word2vec/` (last access: 06/21/2017)".

E. Allman, "Managing technical debt", *Communications of the ACM*, vol. 55, no. 5, pp. 50–55, 2012.

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

S. Ambler. (2013) 11 strategies for dealing with technical debt. En ligne: `http://www.disciplinedagiledelivery.com/technical-debt/`

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

J. Bansiya et C. G. Davis, "A hierarchical model for object-oriented design quality assessment", *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.

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

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

L. Breiman, "Bagging predictors", *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

——, "Random forests", *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya *et al.*, "Managing technical debt in software-reliant systems", dans *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 47–52.

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

——, "A metric for software readability", dans *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 121–130.

N. V. Chawla, K. W. Bowyer, L. O. Hall, et W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique", *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

J. Cohen, *Statistical power analysis for the behavioral sciences*, 2e éd. Lawrence Earlbaum Associates, 1988.

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

M. L. Collard, M. J. Decker, et J. I. Maletic, "srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration", dans *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, série ICSM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 516–519. DOI: 10.1109/ICSM.2013.85. En ligne: http://dx.doi.org/10.1109/ICSM.2013.85

J. Cong et B. Xiao, "Minimizing computation in convolutional neural networks", dans *ICANN*, 2014.

C. Couto, J. E. Montandon, C. Silva, et M. T. Valente, "Static correspondence and correlation between field defects and warnings reported by a bug finding tool", *Software Quality Journal*, vol. 21, no. 2, pp. 241–257, 2013.

W. Cunningham, "The wycash portfolio management system", dans *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, série OOPSLA '92. New York, NY, USA: ACM, 1992, pp. 29–30. DOI: 10.1145/157709.157715. En ligne: http://doi.acm.org/10.1145/157709.157715

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

C. N. Dos Santos et M. Gatti, “Deep convolutional neural networks for sentiment analysis of short texts.” dans *COLING*, 2014, pp. 69–78.

N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, et I. Gorton, “Measure it? manage it? ignore it? software practitioners and technical debt”, dans *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, série ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 50–60. DOI: 10.1145/2786805.2786848. En ligne: http://doi.acm.org/10.1145/2786805.2786848

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

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

F. A. Fontana, R. Roveda, et M. Zanoni, “Technical debt indexes provided by tools: a preliminary discussion”, dans *Managing Technical Debt (MTD), 2016 IEEE 8th International Workshop on*. IEEE, 2016, pp. 28–31.

I. Griffith, D. Reimanis, C. Izurieta, Z. Codabux, A. Deo, et B. Williams, “The correspondence between software quality models and technical debt estimation approaches”, dans *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*. IEEE, 2014, pp. 19–26.

Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. Da Silva, A. L. Santos, et C. Siebra, “Tracking technical debt—an exploratory case study”, dans *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 528–531.

I. Guyon et A. Elisseeff, “An introduction to variable and feature selection”, *Journal of machine learning research*, vol. 3, no. Mar, pp. 1157–1182, 2003.

M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, et I. H. Witten, “The weka data mining software: an update”, *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman, et F. Shull, “Organizing the technical debt landscape”, dans *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 23–26.

F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, et H. Sahraoui, “A bayesian approach for the detection of code and design smells”, dans *Quality Software, 2009. QSIC’09. 9th International Conference on*. IEEE, 2009, pp. 305–314.

Y. Kim, “Convolutional neural networks for sentence classification”, *arXiv preprint arXiv:1408.5882*, 2014.

A. Krizhevsky, I. Sutskever, et G. E. Hinton, “Imagenet classification with deep convolutional neural networks”, dans *Advances in neural information processing systems*, 2012, pp. 1097–1105.

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

G. Louppe, L. Wehenkel, A. Sutera, et P. Geurts, “Understanding variable importances in forests of randomized trees”, dans *Advances in neural information processing systems*, 2013, pp. 431–439.

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

E. Maldonado, E. Shihab, et N. Tsantalis, “Using natural language processing to automatically detect self-admitted technical debt”, *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017. DOI: 10.1109/TSE.2017.2654244

——, “Using natural language processing to automatically detect self-admitted technical debt”, *IEEE Transactions on Software Engineering*, 2017.

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

——, “Assessing technical debt by identifying design flaws in software systems”, *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9–1, 2012.

B. W. Matthews, “Comparison of the predicted and observed secondary structure of t4 phage lysozyme”, *Biochimica et Biophysica Acta (BBA)-Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.

T. J. McCabe, “Reverse engineering reusability redundancy: the connection”, *American Programmer*, vol. 3, pp. 8–13, October 1990.

——, “Reverse engineering, reusability, redundancy: the connection”, *American Programmer*, vol. 3, no. 10, pp. 8–13, 1990.

N. Moha, Y.-G. Gueheneuc, L. Duchien, et A.-F. Le 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.

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

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

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

R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann Publishers, 1993.

G. Suryanarayana, G. Samarthyam, et T. Sharma, “Chapter 1 - technical debt”, dans *Refactoring for Software Design Smells*, G. Suryanarayana, , G. Samarthyam, et T. Sharma, éds. Boston: Morgan Kaufmann, 2015, pp. 1 – 7. DOI:

https://doi.org/10.1016/B978-0-12-801397-7.00001-1. En ligne: http://www.sciencedirect.com/science/article/pii/B9780128013977000011

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

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

F. Wedyan, D. Alrmuny, et J. M. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction", dans *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*. IEEE, 2009, pp. 141–150.

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

R. K. Yin, *Case study research: Design and methods*. Sage publications, 2013.

N. Zazworka, M. A. Shaw, F. Shull, et C. Seaman, "Investigating the impact of design debt on software quality", dans *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 17–23.