UNIVERSITY OF MACEDONIA
SCHOOL OF INFORMATION SCIENCES
DEPARTMENT OF APPLIED INFORMATICS

A Study on Machine Learning Techniques for Technical Debt Estimation and Forecasting

Dimitrios Tsoukalas
PhD Dissertation

Supervisor: Prof. Alexander Chatzigeorgiou
Advisors: Dr. Dionysios Kehagias, Dr. Apostolos Ampatzoglou

Thessaloniki, 12/2021

“A Study on Machine Learning Techniques for Technical Debt Estimation and Forecasting”

Dimitrios Tsoukalas

B.Sc. in Applied Informatics 2012
M.Sc. in Applied Informatics 2014
M.Sc. in Advanced Computer and Communication Systems 2016

PhD Dissertation

A dissertation submitted in fulfillment
of the requirements for the degree of
Doctor of Philosophy

University of Macedonia
Thessaloniki, Greece
2021

Summary

Technical Debt (TD) is a successful metaphor in conveying the consequences of software inefficiencies and their elimination to both technical and non-technical stakeholders, primarily due to its monetary nature. In recent years, TD has attracted the attention of both academia and industry. As a result, there has been a considerable increase in the number and provided functionality of methods and tools that support TD management activities.

However, due to the lack of a commonly accepted standard for identifying and quantifying TD, it is not clear how the existing TD tools map to these activities, as each adopts its own ruleset. As a result, the divergent TD estimates produced by these tools call into question the reliability of their findings. This highlights the need for novel methods that would incorporate the collective knowledge extracted by multiple TD tools, exploiting in that way a commonly agreed "TD knowledge base" to improve TD identification and estimation activities.

On the other hand, existing methods and tools focus mostly on assessing the TD of a software system at its current state. However, a decision on whether or not to repay a TD item has different consequences depending on when it is made. This stresses the need for methods and accompanying tools that would enable long-term effective software maintenance by forecasting the TD evolution of a software system, and providing insights regarding where and when to apply refactoring.

To this end, the purpose of the present dissertation is to advance the state of the art in the fields of TD estimation and TD forecasting, exploring new approaches, based mainly on ML techniques, that facilitate TD management activities and enable decision-making under uncertainty.

More specifically, given the lack of concrete approaches regarding the TD Forecasting concept, the present dissertation examined a multitude of different forecasting methods, ranging from simple time series to more sophisticated ML models, for their ability to accurately predict the future TD evolution of long-lived, open-source software systems. The results showed that statistical time series models are able provide satisfactory TD predictions for short-term forecasting horizons. However, it was observed that their predictive power dropped significantly when trying to forecast longer into the future. On the other hand, ML techniques proved more effective in modelling TD patterns when considering both short- and long-term forecasting horizons. More specifically, it was shown that ML models constitute a suitable and effective approach for TD forecasting, as they are able to fit and provide meaningful estimates of TD evolution over a relatively long horizon, while in most of the cases, the future TD value is captured with a sufficient level of accuracy.

Besides predicting the TD evolution of long-lived software systems, the present thesis also investigated the ability of data clustering techniques to improve the accuracy of cross-project TD forecasting, allowing in that way the provision of reliable forecasts for projects with insufficient historical data, such as new projects or projects whose past evolution cannot be retrieved. By grouping "similar" software projects into clusters based on code metrics, we observed that the cross-project prediction accuracy of cluster-specific ML models was higher when forecasting for projects assigned to the same cluster, compared to forecasting for projects assigned to a different cluster. This suggests that similarity between software projects may be a key factor for enhancing cross-project TD forecasting, and, in turn, that clustering may be a viable solution for achieving better results in cross-project forecasting.

Finally, by acknowledging the lack of a commonly accepted basis regarding the identification and estimation of TD, this dissertation investigated the ability of ML models to leverage the collective knowledge extracted by combining the results of different leading TD tools. Using various metrics pertaining to source code and repository activity as features and an empirical benchmark of classes sharing similar levels of TD as ground truth, the proposed classification framework was proven able to accurately identify candidate high-TD software classes in software systems, eliminating in that way the need to resort to a multitude of tools for establishing the ground truth. Based on the results, a tool prototype for automatically assessing the TD software projects has been implemented.

Keywords: Technical debt, Machine learning, Technical debt forecasting, Technical debt identification

Περίληψη

Ο όρος «τεχνικό χρέος» είναι μια επιτυχημένη μεταφορά η οποία εισήχθη το 1992 ώστε να περιγράψει το φαινόμενο κατά το οποίο τεχνικοί συμβιβασμοί που αφορούν την ποιότητα σε ένα σύστημα λογισμικού (π.χ. σχεδίαση ή κώδικας) και πραγματοποιούνται ώστε να αποφέρουν βραχυπρόθεσμα οφέλη, ενδέχεται να οδηγήσουν σε μακροπρόθεσμα προβλήματα εάν δεν επιλυθούν άμεσα. Λόγω της οικονομικής φύσης του, χρησιμοποιήθηκε ευρέως για να περιγράψει τις συνέπειες που φέρουν τα προβλήματα στην ποιότητα του λογισμικού, αλλά και να τονίσει τις ανάγκες εξάλειψής τους, τόσο σε τεχνικά- όσο και σε μη τεχνικά-καταρτισμένους ενδιαφερόμενους. Τα τελευταία χρόνια, το τεχνικό χρέος έχει προσελκύσει την προσοχή τόσο του ακαδημαϊκού χώρου όσο και της βιομηχανίας. Ως αποτέλεσμα, έχει υπάρξει μια σημαντική αύξηση στον αριθμό και την παρεχόμενη λειτουργικότητα των μεθόδων και των εργαλείων που υποστηρίζουν δραστηριότητες διαχείρισης τεχνικού χρέους.

Ωστόσο, λόγω της έλλειψης ενός κοινά αποδεκτού προτύπου για την αναγνώριση και την ποσοτικοποίηση του τεχνικού χρέους, δεν είναι σαφές πώς τα υπάρχοντα εργαλεία διαχειρίζονται τις παραπάνω δραστηριότητες, καθώς το καθένα υιοθετεί το δικό του σύνολο κανόνων. Ως αποτέλεσμα, οι αποκλίνουσες εκτιμήσεις τεχνικού χρέους που παράγονται από αυτά τα εργαλεία θέτουν υπό αμφισβήτηση την αξιοπιστία των ευρημάτων τους. Το παραπάνω πρόβλημα τονίζει την ανάγκη για νέες μεθόδους που θα ενσωματώνουν τη συλλογική γνώση που εξάγεται από πολλαπλά εργαλεία τεχνικού χρέους, αξιοποιώντας με αυτόν τον τρόπο μια κοινά αποδεκτή «βάση γνώσης τεχνικού χρέους» για τη βελτίωση των δραστηριοτήτων αναγνώρισης και ποσοτικοποίησης του τεχνικού χρέους.

Από την άλλη πλευρά, οι υπάρχουσες μέθοδοι και εργαλεία επικεντρώνονται κυρίως στην αξιολόγηση του τεχνικού χρέους ενός συστήματος λογισμικού στην τρέχουσα κατάστασή του. Ωστόσο, μια απόφαση σχετικά με το εάν θα αποπληρωθεί ή όχι ένα αντικείμενο τεχνικού χρέους έχει διαφορετικές συνέπειες ανάλογα με το πότε λαμβάνεται. Το παραπάνω τονίζει την ανάγκη δημιουργίας μεθόδων και συνοδευτικών εργαλείων τα οποία θα επιτρέπουν την μακροπρόθεσμη αποτελεσματική συντήρηση λογισμικού, προβλέποντας την εξέλιξη του τεχνικού χρέους ενός συστήματος λογισμικού και παρέχοντας πληροφορίες σχετικά με το πώς και πότε πρέπει να αποπληρωθεί.

Με βάση τα παραπάνω ζητήματα, σκοπός της παρούσας διατριβής είναι να συνδράμει στους τομείς της εκτίμησης και της πρόβλεψης τεχνικού χρέους, προτείνοντας νέες προσεγγίσεις οι οποίες βασίζονται κυρίως σε τεχνικές μηχανικής μάθησης. Οι προσεγγίσεις αυτές στοχεύουν στην διευκόλυνση των δραστηριοτήτων διαχείρισης τεχνικού χρέους και επιτρέπουν τη λήψη αποφάσεων υπό συνθήκες αβεβαιότητας.

Πιο συγκεκριμένα, δεδομένης της έλλειψης συγκεκριμένων προσεγγίσεων σχετικά με την έννοια της πρόβλεψης τεχνικού χρέους, η παρούσα διατριβή εξέτασε ένα πλήθος διαφορετικών μεθόδων πρόβλεψης, οι οποίες κυμαίνονται από απλά μοντέλα χρονοσειρών έως πιο εξελιγμένα μοντέλα μηχανικής μάθησης, για την ικανότητά τους να προβλέπουν με ακρίβεια τη μελλοντική εξέλιξη του τεχνικού χρέους σε συστήματα λογισμικού με μακροχρόνιο ιστορικό ανάπτυξης. Τα αποτελέσματα έδειξαν ότι τα μοντέλα χρονοσειρών

είναι σε θέση να παρέχουν ικανοποιητικές προβλέψεις τεχνικού χρέους για βραχυπρόθεσμους ορίζοντες πρόβλεψης. Ωστόσο, παρατηρήθηκε ότι η προγνωστική τους ισχύς μειώνεται σημαντικά όταν προσπαθούν να προβλέψουν στο πιο μακρινό μέλλον. Από την άλλη πλευρά, οι τεχνικές μηχανικής μάθησης αποδείχθηκαν πιο αποτελεσματικές στη μοντελοποίηση της μελλοντικής εξέλιξης του τεχνικού χρέους κατά την εξέταση τόσο βραχυπρόθεσμων όσο και μακροπρόθεσμων οριζόντων πρόβλεψης. Πιο συγκεκριμένα, αποδείχθηκε ότι τα μοντέλα μηχανικής μάθησης αποτελούν μια κατάλληλη και αποτελεσματική προσέγγιση για την πρόβλεψη τεχνικού χρέους, καθώς είναι σε θέση να παρέχουν ουσιαστικές εκτιμήσεις της εξέλιξης του τεχνικού χρέους για μια σχετικά μεγάλη περίοδο, πετυχαίνοντας στις περισσότερες περιπτώσεις επαρκή επίπεδα ακρίβειας.

Εκτός από την πρόβλεψη της εξέλιξης του τεχνικού χρέους σε συστήματα λογισμικού με μακροχρόνιο ιστορικό ανάπτυξης, η παρούσα διατριβή διερεύνησε επίσης την ικανότητα των τεχνικών ομαδοποίησης δεδομένων να βελτιώσουν την ακρίβεια της πρόβλεψης τεχνικού χρέους μεταξύ έργων, επιτρέποντας με αυτόν τον τρόπο την παροχή αξιόπιστων προβλέψεων για έργα με ανεπαρκή ιστορικά δεδομένα, όπως νέα έργα ή έργα από τα οποία δεν είναι δυνατό να ανακτηθούν παρελθοντικά δεδομένα. Χρησιμοποιώντας μετρικές κώδικα για την ομαδοποίηση «παρόμοιων» έργων λογισμικού, παρατηρήσαμε ότι η ακρίβεια πρόβλεψης μοντέλων μηχανικής μάθησης τα οποία είχαν εκπαιδευτεί σε κάποιο έργο λογισμικού μιας συγκεκριμένης ομάδας ήταν υψηλότερη για τα έργα λογισμικού που ανήκαν στην ίδια ομάδα, σε σύγκριση με την ακρίβεια για έργα λογισμικού που εκχωρήθηκαν σε διαφορετικές ομάδες. Αυτό υποδηλώνει ότι η ομοιότητα μεταξύ έργων λογισμικού μπορεί να αποτελέσει ένα βασικό παράγοντα για τη βελτίωση της πρόβλεψης τεχνικού χρέους μεταξύ έργων λογισμικού και, κατά συνέπεια, ότι η ομαδοποίηση μπορεί να είναι μια βιώσιμη λύση για την ανάπτυξη μοντέλων πρόβλεψης για αυτό το σκοπό.

Τέλος, αναγνωρίζοντας την έλλειψη μιας κοινά αποδεκτής βάσης σχετικά με την αναγνώριση και την ποσοτικοποίηση του τεχνικού χρέους, η παρούσα διατριβή διερεύνησε την ικανότητα των μοντέλων μηχανικής μάθησης να αξιοποιούν τη συλλογική γνώση που εξάγεται συνδυάζοντας αποτελέσματα διαφορετικών εργαλείων τεχνικού χρέους. Χρησιμοποιώντας διάφορες μετρήσεις που σχετίζονται με τον πηγαίο κώδικα και τη δραστηριότητα αποθετηρίου ως είσοδο, αλλά και ένα σύνολο κλάσεων που μοιράζονται παρόμοια επίπεδα τεχνικού χρέους ως σημείο αναφοράς, η μέθοδος ταξινόμησης που προτείνουμε αποδείχθηκε ικανή να προσδιορίσει με ακρίβεια υποψήφιες κλάσεις υψηλού τεχνικού χρέους, εξαλείφοντας αυτό τρόπο την ανάγκη καταφυγής σε ένα πλήθος εμπορικών εργαλείων για την παραπάνω διαπίστωση. Επιπλέον, με βάση τα εξαγόμενα αποτελέσματα, υλοποιήσαμε ένα πρωτότυπο εργαλείο το οποίο αυτοματοποιεί την παραπάνω διαδικασία.

Dedicated to my beloved parents, Konstantinos Tsoukalas and Kalliopi Mpouti,

my brother, Prodromos Tsoukalas, and my sister, Emilianna Tsoukala

... for their endless love, support, and encouragement.

Acknowledgments

The present dissertation is the result of a long, yet fruitful journey that could not have been completed had I not been surrounded and supported by many special people, whom I would like to thank.

First and foremost, I would like to express my deep gratitude and sincere appreciation to my principal supervisor, Prof. Alexander Chatzigeorgiou, for his priceless support and care throughout not only my PhD, but also my bachelor and master studies at the University of Macedonia (UoM). It was an honor and a real privilege for me to work under the supervision of a top scientist with renowned contributions in the broader scientific community. Under his supervision, I gained valuable knowledge and developed important skills that significantly helped me in becoming not only a better researcher, but also a better person.

I would also like to thank and express my sincere gratitude to my advisor at the Information Technologies Institute (ITI), Dr. Dionysios Kehagias, for believing in me and giving me the chance to start my PhD journey. His support and assistance at every stage of the research project were valuable for the present dissertation. Of course, I would also like to thank my advisor at the University of Macedonia (UoM), Dr. Apostolos Ampatzoglou, for his insightful comments and assistance.

Special thanks go to my dear friend and colleague, Miltiadis Siavvas, for his valuable constructive feedback and constant support, as well as my friends, Mary and Costas, for their encouragement and happy distractions to rest my mind outside of my research.

Last but not least, I would like to thank my family for their continuous and unparalleled love, help, and support throughout all the years of my studies. This journey would not have been possible without them, and therefore I dedicate this success to them.

Contents

List of Figures

List of Tables

Chapter I Introduction

"Technical debt has a direct correlation to how many expletives you yell when making any changes to an existing codebase."

Evan Fribourg - American software engineer

1 Motivation

The term "Technical Debt" (TD) was first introduced in 1992 by Ward Cunningham [1] as a metaphor intended to describe the problem of introducing long-term problems to software products, by not resolving existing quality issues early enough in the overall software development lifecycle (SDLC). In other words, TD represents the consequences of shortcuts (i.e., quality compromises) taken in a software product, usually to meet business goals such as limited time or budget. While initially related to software implementation (i.e. at the code level), the TD metaphor was gradually extended to all phases of the SDLC, i.e., software architecture, design, documentation, requirements, and testing [2]. Its monetary nature has established it as a means of communication between technical and management stakeholders. The fact that TD was inspired by the concept of the financial debt of economic theory has also led to the adoption of a multitude of financial theories for its identification, repayment, quantification, etc. [3]. However, managing TD is more complicated than managing financial debt because of the uncertainty involved [4].

The efficient management of TD requires a clear understanding of the state-of-research on Technical Debt Management (TDM) activities. Since TD combines elements from both software engineering and financial theory [3], the multitude of theories, methods, and tools that researchers and practitioners have developed and adopted for TDM follow two different paths and thus, can be classified into two broad schools of thought. The first one is the financial aspect of TD, which includes approaches such as Portfolio management, Real options and software economics [3]. The second one is the software engineering aspect of TD, which includes estimation methods such as calculation models, code metrics, operational metrics, etc. [5].

Although the number of various financial and software engineering approaches for managing TD continues to proliferate, there are still many open issues that require further investigation. First of all, none of the already proposed methods and tools have reached the desired level of maturity [5]. In addition, since no commonly accepted standard for managing TD exists [5], it is not clear how the associated tools map to TDM activities like identification, estimation, or repayment. In fact, in order to assess TD, most of these tools rely on predefined rules that can be asserted by static source code analysis. However, due to their different rulesets and adopted TD indexes [6], none of these tools can be considered as an ultimate oracle [7], while using multiple tools to aggregate their results is neither a practical nor a cost-effective solution. As a result, researchers, managers and developers perceive the concept of TD in different ways and are unable to distinguish between the software quality compromises that can be attributed as TD and those that cannot. In this respect, an interesting research topic would be to examine whether the collective knowledge extracted by combining the results of leading TD tools can be leveraged and incorporated into novel methods, exploiting in that way a commonly agreed “TD knowledge base” to improve the TD identification and estimation activities.

Aside from the aforementioned open issues, another important gap in the TDM field is the fact that existing methods and tools are mostly focusing on estimating the TD of a software system at its current state. Nevertheless, as a software system evolves, so does its TD [8]. In fact, in some cases it is the TD itself that drives the evolution of a software system, as it can highly affect future software maintenance activities if left unattended [9]. Under these circumstances, a method or tool that would predict the future TD evolution of a software system and thus, assist software stakeholders in taking proactive actions regarding TD repayment, is of paramount importance for effective long-term software maintenance. However, while researchers have extensively examined the topic of predicting the evolution of various aspects directly or indirectly related to the TD of a software system, such as code smells [10], fault-proneness [11] and evolution trends [12], limited contributions have been made so far regarding the forecasting of TD itself. This opens a new area of research on TDM, which can initially be approached by investigating whether forecasting approaches originally proposed for various other software quality aspects can be applied (and potentially extended) for predicting TD as well, and in what ways.

Taking under consideration the aforementioned open issues, the ultimate objective of this dissertation is to study two important TD-related aspects, namely TD estimation and TD forecasting, while also exploring the ability of various Machine Learning (ML) techniques as a means to estimate TD and predict its future evolution. Towards this effort, the objective of this dissertation can be decomposed into six more specific goals, which are described in the next section.

2 Dissertation Goals and Research Questions

This section presents the six goals of this dissertation, in summary. The studies that have been conducted to achieve each goal are developed in a dedicated chapter. Before reading the following six goals, the reader can find the timeline of the research conducted during this dissertation in Table 1. The timeline contains the chronicle of dissertation’s goals along with the resulting publication.

Timeline	**Goal**	**Publication**
July 2018	Explore the most significant contributions in the broader field of TD estimation and forecasting to date and identify the existing open issues that have not been adequately addressed yet. (Chapter III)	[13]
March 2019	Investigate whether the usage of time series models is a meaningful and accurate approach to forecasting TD in a long-lived, open-source software. (Chapter IV)	[14]

June 2019	Investigate whether the usage of machine learning models on a specific set of TD indicators is a meaningful and accurate approach to forecasting TD in a long-lived, open-source software. (Chapter V)	[15]
December 2019	Investigate whether the usage of machine learning models for TD forecasting is a meaningful practical approach for the prioritization of TD liabilities at class-level of granularity in a long-lived, open-source software. (Chapter VI)	To be submitted
May 2020	Investigate whether data clustering techniques can improve the accuracy of cross-project TD forecasting models. (Chapter VII)	[16]
January 2021	Investigate whether the usage of machine learning models on various code- and repository-related factors is a meaningful and accurate approach for classifying software classes based on their TD level (High/Not-High TD). (Chapter VIII)	[17]

2.1 *Goal 1: Related Work on Technical Debt Estimation and Forecasting*

TDM is one of the fastest-growing research areas of software technology (90% of the research was published after 2010 [18]). The efficient management of TD includes several different activities that assist managers and developers in making TD visible and controllable [5], such as TD identification, estimation, prioritization, prevention and repayment. However, although the number of various approaches for managing TD continues to proliferate, it is not clear how existing theories, methods and tools map to specific TDM activities. As a result, researchers, developers and managers are unable to distinguish between the software quality compromises that can be attributed as TD and those that cannot.

On the other hand, the evolution of a software system usually implies an analogous evolution of its TD as well [8]. The opportunity to predict the evolution of TD is of paramount importance to software maintainability, since it would allow system engineers and project managers to identify software artifacts that are prone to accumulate significant levels of TD and assess the point at which the software product could become unmaintainable. Despite the numerous endeavors towards forecasting the evolution of various aspects related to the TD of a software project, it is not clear if concrete approaches exist so far regarding the forecasting of TD itself.

To this end, to set the ground for this dissertation, two important TD-related aspects, namely TD estimation and TD forecasting, were theoretically examined. In particular, the first step of this dissertation aimed to review the most significant attempts in the broader field of TD estimation and forecasting, identify existing open issues of high interest, and pave the way for future research directions. Hence, this part of the dissertation acted as a reference for formulating the rest of the research goals by providing a solid understanding of existing solutions and identifying open issues that require further research. Based on the aforementioned first goal of the dissertation, the following research question has been formulated:

RQ: *What are the most significant contributions in the broader field of Technical Debt estimation and forecasting to date and what are the existing open issues that have not been adequately addressed yet?*

The detailed work on the state-of-the-art survey on methods and tools for TD estimation and forecasting is presented in Chapter III.

2.2 Goal 2: Time Series Techniques for Technical Debt Forecasting

After identifying the most significant attempts and existing open issues in the broader field of TD estimation and forecasting, a critical issue arose. More specifically, no concrete approaches had been proposed up to that point regarding the forecasting of TD, which is opposite to extensive research that has been conducted for predicting the evolution of individual software features or quality attributes that are directly or indirectly related to the TD of a software project. A possibility to predict the future TD evolution of a software project is primarily crucial for long-term effective software maintenance, which is recognized as one of the most-effort intense activities in the SDLC [3], In addition, forecasting the evolution of TD could be valuable for estimating the point in which the software product could become unmaintainable. Consequently, software architects and project managers would be able to gain a better understanding of future TD issues and plan appropriate refactoring activities.

Time series models have been widely used in previous studies for predicting software evolution trends, future change requests or software defects [12], [19]–[22]. To this end, this part of the dissertation attempted to empirically evaluate the ability of time series to adequately forecast future TD trends. Towards this goal, a code repository was initially constructed comprising five real-world open-source Java applications retrieved from the GitHub online repository. For each application, 150 snapshots (commits) were collected in weekly intervals, spanning up to 3 years of each system's evolution. This approach led to a dataset containing up to 750 snapshots in total. For each snapshot, the TD value was calculated using SonarQube[1], a popular static code analysis tool. Subsequently, for each application, the Box-Jenkins modelling method [23] was employed to construct and identify the parameters of various ARIMA models. Finally, the accuracy of these models was compared to a random walk model for various steps ahead into the future in order to reach safer conclusions regarding the significance of the observed results. To the best of our knowledge, this is the first study in the field of TD that examines the applicability of time series models for TD forecasting. The problem that this part of the dissertation attempts to solve can be summarized in the following research question:

***RQ**: Is the usage of time series models a valid and accurate approach to forecasting Technical Debt in a long-lived, open-source software?*

[1] https://www.sonarqube.org/

The detailed work on the applicability of time series models for TD forecasting is presented in Chapter IV.

2.3 *Goal 3: Machine Learning Techniques for Technical Debt Forecasting*

As a first step towards the TD forecasting concept, in the work introduced in Section 2.2, statistical time series ARIMA models were studied and applied [14]. Statistical time series models are mostly univariate, i.e., they require only the historical data of the variable of interest to forecast its future evolution behavior. A dataset of 5 real-world open-source Java applications was used and it was found that the ARIMA(0,1,1) can provide accurate TD Principal predictions over a sufficiently long time period for all sampled applications. However, even though the overall ARIMA model performance was satisfactory for short-term TD forecasting (up to 8 weeks ahead), it was observed that its predictive power dropped significantly for longer predictions. Moreover, the ARIMA models were proven difficult to tune, as one has to follow the entire Box-Jenkins methodology.

Taking into account the aforementioned challenges of ARIMA models, the next step of this dissertation constituted a logical continuation and extension of the previous efforts [14], in order to provide a more complete approach for TD forecasting. The motivation was to examine more advanced models, able to support feature engineering, i.e., take into account various TD-related features and their combinations to generate better TD predictions. Therefore, this part of the dissertation attempted to empirically evaluate the ability of more sophisticated multivariate ML methods to adequately forecast future TD trends of software applications and achieve better and more practical results in both short- and long-term predictions.

To shed light on this direction, an empirical study was conducted. Initially, the relevant literature and identified TD indicators that could act as predictors, such as TD-related features and various Object Oriented (OO) metrics, were studied. Afterwards, a relatively large code repository was constructed comprising 15 real-world open-source Java applications retrieved from the GitHub online repository. For each application, a subsequent number of snapshots (commits) ranging from 100 to 150 was collected in weekly intervals, spanning up to almost 3 years of each application's evolution. This approach led to a dataset containing 1,850 snapshots in total (171M lines of code). In order to extract the features that could act as predictors, two popular tools were used, namely SonarQube[2] and CKJM Extended[3] respectively. This process led to 15 independent application-specific datasets containing TD indicators and TD values for each snapshot. Subsequently, techniques like correlation analysis, univariate and multivariate analysis were employed, which allowed to select the most statistically significant TD predictors and thus retain as much discriminatory information as possible. Finally, various ML forecasting models were examined and their accuracy was compared for various forecasting horizons in order to reach safer conclusions regarding the significance of the observed results. To the best of our knowledge, this is the first study in the field of TD that

[2] https://www.sonarqube.org/

[3] http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

examines the applicability of ML models for TD forecasting. The problem that this part of the dissertation attempts to solve can be summarized in the following research question:

RQ: *Is the usage of machine learning models on a specific set of Technical Debt indicators a meaningful and accurate approach to forecasting Technical Debt Principal in a long-lived, open-source software?*

The detailed work on the applicability of ML models for TD forecasting is presented in Chapter V.

2.4 Goal 4: Technical Debt Prioritization and Repayment based on Forecasting Techniques

In a software affected by TD, refactoring is the only effective way to reduce it on existing source code. However, companies usually cannot afford to repay all the TD that is generated continuously [24], since strict production deadlines often force them to focus on the delivery of new functionality, leading to the accumulation of TD. Therefore, before applying refactoring activities, it is necessary to identify which TD items should be repaid first and which items can be tolerated until later releases, by prioritizing them based on their TD values, but also based on business’s objectives and preferences [25]. Several different TD prioritization approaches have been proposed by researchers, based mostly on code smells [26], [27], time [28], cost to fix a violation [29], and quality rules [30]. However, while all previous research works investigate TD prioritization and repayment based on historical TD data, there are no research endeavors considering the future evolution of TD to address this issue.

Taking into account the aforementioned gap in the field, the next step of the dissertation built on top of the previous work introduced in Section 2.3 [15]. More specifically, we leveraged the ability of ML models to accurately forecast the future TD evolution in order to propose a practical approach for effective prioritization of TD items not only based on their current TD, but also based on their potential future TD accumulation. The proposed approach considered both the frequency of changes and the future TD evolution to enable the early identification of classes that are more likely to become unmaintainable in the future, and therefore to allow prompt refactoring of their liabilities. It also emphasized on the class-level of granularity, since it provides highly actionable results to the developers and project managers, compared to system-level of granularity.

To demonstrate this approach, the artifacts (i.e., classes) of a software application with their history (past versions) were initially retrieved. Next, these artifacts were analyzed using static code analysis in order to calculate several TD measures for the construction of the initial class-level dataset. Afterwards, change proneness analysis was applied to detect the most change-prone classes in terms of size and TD. Then, class-level TD forecasting models were built to assess the TD evolution of the selected classes. Finally, visualization techniques were used to facilitate the understandability of the results and, in turn, the decision-making tasks of the developers and project managers regarding the repayment of the identified TD liabilities. The problem that this part of the dissertation attempts to solve can be summarized in the following research question:

***RQ**: Is Technical Debt forecasting using machine learning models a meaningful practical approach for the prioritization of Technical Debt liabilities at class-level of granularity in a long-lived, open-source software?*

The detailed work on the applicability of TD forecasting for the prioritization of TD liabilities is presented in Chapter VI.

2.5 Goal 5: Data Clustering for Cross-project Technical Debt Forecasting

Previous work within the context of the dissertation approached the TD forecasting challenge by examining a multitude of different forecasting methods, ranging from simple statistical time series models to more sophisticated ML models, revealing that the proposed approaches are able to provide meaningful results [14], [15]. However, these approaches presuppose that reliable and sufficient past data (in the form of past commits) are available for a particular software project, in order to initiate a proper tailored model-training process and produce accurate forecasts. This obviously affects the practicality of the produced models, as they cannot be easily applied to software projects with insufficient historical data, such as new projects or projects whose past evolution cannot be retrieved.

Taking into account the aforementioned limitations, the next step of this dissertation focused on cross-project TD forecasting, that is, building a forecasting model based on data retrieved from one project and using it to get reliable forecasts for a new, previously unknown software project. Such a method would allow project managers and developers leverage the benefits of TD forecasting in cases where a long history of commits is not available, e.g., from the very early stages of the development.

To this end, this part of the dissertation introduced a clustering-based approach that aims to improve the performance of TD forecasting models in cross-project prediction. Initially, a relatively large repository of software projects was utilized and its projects were grouped into clusters based on their similarity with respect to important TD aspects (e.g., size, complexity, Object-oriented metrics, etc.). Consequently, TD forecasting models were built using regression algorithms for each one of the produced clusters. The idea is that when a new project becomes available, it is assigned to one of the defined clusters and the dedicated pre-trained forecasting model that is associated to this cluster is used to predict its future TD. Hence, this approach enables the provision of TD forecasts, for software projects that do not exhibit a sufficiently long history of commits. In other words, it enables accurate TD forecasting from the early stages of software development. In brief, the overall problem that this part of the dissertation attempts to address can be summarized in the following research question:

RQ: *Is the usage of data clustering a meaningful approach towards cross-project Technical Debt forecasting?*

The detailed work on the clustering-based approach that aims to improve the performance of models in cross-project TD forecasting is presented in Chapter VII.

2.6 Goal 6: Machine Learning Techniques for Technical Debt Identification

TD that is accumulated due to problematic design and implementation choices needs to be repaid early enough in the software development life cycle. If not done so, then the software development process can be significantly impeded by its presence [31] and in extreme situations may even lead to technical bankruptcy [9]. To address this challenge, a major part of the work conducted within the context of this dissertation dealt with the TD forecasting concept, aiming to assist project managers and developers in planning appropriate refactoring activities and taking proactive actions regarding TD repayment.

Besides forecasting the future evolution of TD, however, another challenging open issue is the lack of a commonly accepted standard for estimating and managing TD across the entire software development lifecycle [5]. While there exists a plethora of tools that claim to help towards TD Management [32], it is not clear how these tools map to activities like identification, measurement, or repayment, since most of them rely on different TD indexes [6]. As a result, none of them can be considered as an ultimate oracle [7], while using multiple tools to aggregate their results is not a feasible solution. Ultimately, the aforementioned shortcoming leads to important problems affecting both the academia (e.g., construct validity issues) and the industry (e.g. which tool is to be trusted).

By acknowledging the lack of a commonly accepted TDM basis among the existing TD tools, the next (and last) step of the present dissertation focused on examining whether the collective knowledge extracted by combining the results of three leading TD tools can be leveraged and incorporated into novel models for TD identification/estimation. In particular, to shed light on this direction, the ability of various ML classification models to detect high-TD software classes was investigated, considering as input a wide set of factors spanning from code metrics to repository activity, and as ground truth a commonly agreed "TD knowledge base". Such models would enable the accurate identification of candidate high-TD classes in software systems, while providing the opportunity for further experimentation and analysis without having to resort to a multitude of commercial and open-source tools for establishing the ground truth.

Initially, an empirical benchmark [7] of classes sharing similar levels of TD was used on 25 OSS Java projects to obtain the ground truth for the proposed classification framework. Specifically, the classes that all three leading TD tools identified as TDIs were classified as high-TD. Then, supposing that the use of multiple sources of information will result in more accurate models, a set of independent variables was built based on a wide spectrum of software characteristics spanning from code-related metrics to metrics that capture aspects of the development process, retrieved by open-source tools. Finally, various statistical and ML algorithms were applied to select the most fitting one for the given problem. Ultimately, the derived models subsume the collective knowledge that would be extracted by combining the results of three TD tools, exploiting a "commonly agreed TD knowledge base" [7]. To facilitate their adoption in practice, the derived models were also implemented in a form of a tool prototype that relies on other open-source tools to automatically retrieve all independent variables and identify high-TD classes for any software project. The problem that this part of the dissertation attempts to address can be summarized in the following research question:

RQ: *By leveraging the collective knowledge extracted by combining the results of three leading TD tools, is the usage of machine learning models on various code-and repository-related factors a meaningful and accurate approach for classifying software classes based on their Technical Debt level (High/Not-High TD)?*

The detailed work on the evaluation of ML algorithms on their ability to detect high-TD software classes is presented in Chapter VIII.

3 Dissertation Outline

The rest of the dissertation is organized as follows:

Chapter II provides an overview of relevant background in order to introduce unfamiliar readers with the main concepts of this dissertation. Particularly, it introduces the Forecasting concept and its most popular techniques, as well as the TD concept and its main components, types, indicators and management strategies.

In Chapter III through Chapter VIII, the work to achieve the six goals of this dissertation is developed. Particularly:

- Chapter III explores the most significant contributions in the broader field of TD estimation and forecasting to date and identifies the existing open issues that have not been adequately addressed yet.
- Chapter IV investigates whether the usage of time series models is a meaningful and accurate approach to forecasting TD in a long-lived, open-source software.
- Chapter V examines whether the usage of ML models on a specific set of TD indicators is a meaningful and accurate approach to forecasting TD in a long-lived, open-source software.
- Chapter VI investigates whether the usage of ML models for TD forecasting is a meaningful and practical approach for the prioritization of TD liabilities at class-level of granularity in a long-lived, open-source software.
- Chapter VII examines whether data clustering techniques can improve the accuracy of cross-project TD forecasting.
- Chapter VIII investigates whether the usage of ML models on various code-and repository-related factors is a meaningful and accurate approach for classifying software classes based on their TD level (High/Not-High TD).

Finally, Chapter IX concludes the dissertation by highlighting its main contributions and by presenting suggestions for future work.

Chapter II Background

“*Any fool can know. The point is to understand.*”

Albert Einstein - German theoretical physicist

In this chapter, basic concepts of Technical Debt (TD) and Forecasting are introduced to familiarize the reader with the analysis that follows in the next chapters. Particularly, this chapter briefly discusses key components, types and indicators of TD, as well as popular ML and time series models that are widely used in various prediction and forecasting tasks.

1 Technical Debt Concepts

1.1 The Technical Debt Metaphor

Inspired by the financial debt of economic theory, Ward Cunningham introduced the metaphor of Technical Debt [1] in 1992 as follows:

"Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise".

In other words, TD indicates quality compromises that can yield short-term benefits in the software development process, but may negatively affect the long-term quality of software products.

There are several causes for creating TD. Fowler [33], [34] states that software development debt is usually a consequence of time pressure. Software engineers and developers often make non-optimal design decisions to quickly address fast-changing requirements, which is leading to a poorly designed system and increased TD over time. Kruchten et al. [35] assign TD to YAGNI decisions (You Ain't Gonna Need It) that often result in unjustified and unnecessary investments in new features, architecture, over engineering, etc. Martin Fowler [34] proposes TD quadrant, a 2×2 matrix (Intentionality x Wisdom), to visualize four different pathways that lead to TD. According to his study, it is not enough to discuss if something is a TD or not, but it is crucial to analyze intention (deliberate or inadvertent) and awareness (reckless or prudent). McConnell [36] suggests a similar categorization, arguing that TD may be unintentional and intentional. Unintentional debt is often a consequence of poor coding practices, while intentional debt is a result of non-optimal decisions that are committed on purpose. He proposes a further classification into short-term and long-term debt. Short-term debt is taken tactically to cover smaller gaps with the goal to speed up software release, and it is expected to be paid off quickly. On the other hand, long-term debt is taken strategically having in mind significant software improvements and can be carried out for years. Suryanarayana et al.

[9] point out that extreme situation when accumulated TD is enormous and cannot be paid off could lead to technical bankruptcy.

1.2 Technical Debt Main Components

The main component of TD is the *Principal*, which refers to the cost that has to be paid in order to eliminate the debt, i.e., the effort required to address the difference between the current and the optimal level of design-time quality. Depending on the type of TD, this can be translated into different kinds of activities, such as code refactoring, documentation updates or improving test coverage [5].

The second main component of TD is *Interest*, which is composed of two parts: i) the *interest amount*, i.e., the potential penalty in terms of increased effort and decreased productivity that will have to be paid in the future as a result of not completing these tasks in the present [4], and ii) the *interest probability*, i.e., the probability that the artefact that contains the debt will undergo maintenance. When this additional effort (interest) reaches a level that makes maintenance so difficult and expensive that the system is no longer financially viable, the project is declared bankrupted [37].

1.3 Technical Debt Types

Several recent studies have highlighted the need to analyze TD from SDLC point of view. Li et al. [5] classify different TD types into ten levels based on the occurrence during the main phases of a software development process (i.e., requirements, design and architecture, implementation, testing, building, documentation, infrastructure, versioning, and defects). According to the authors, *requirements TD* refers to compromises made between the optimal requirements specification and the actual system implementation, while *architectural TD* is occurred by not-optimal architecture decisions that affect some horizontal quality aspects, such as maintainability. *Design TD* refers to code smells such as intensive coupling, God classes, high-complexity, etc. [1]. *Code TD* is the poorly written code that violates best coding practices or coding rules, such as duplicated code. *Test TD* refers to incomplete testing coverage while *build TD* refers to flaws or complexity in the build process of a software system. *Documentation TD* refers to insufficient or incomplete documentation. Finally, *versioning TD* refers to the problems in source code versioning, while defect TD refers to defects, bugs, or failures found in software systems.

However, several researchers and practitioners are sharing the opinion that visible symptoms of low software quality, such as defects or bugs, should not be considered as TD liabilities [38]. Instead, they suggest that TD should be limited to internal system qualities, primarily maintainability and evolvability. Similarly, Kruchten et al. [35] outline the TD that occurs in various phases of the development process (i.e., code, tests, documentation). They attribute the TD of architecture to bad structural or architectural choices or technological gaps. Finally, Sterling [39] observes that the SDLC process may affect the size of the TD. For example, an agile software development process would create less TD than a waterfall model due to a more flexible response to change.

1.4 Technical Debt Indicators and Indexes

TD indicators allow to discover TD items by analyzing different artefacts created during the SDLC. Most TD indicators proposed in the literature are related to *software metrics* [5], [40] that allow the assessment of attributes, features, or characteristics of software artefacts. In the context of object-oriented (OO) programming, various sets of metrics, such as the metric suit proposed by Chidamber and Kemerer (C&K) [41] or the Quality Model for Object Oriented Design (QMOOD) [42], make it possible to characterize the size, complexity, coupling and cohesion of the code among others. These metrics have been widely used in the literature to predict maintenance effort and maintainability [43], [44], which is the quality attribute that is most closely related to TD. Besides OO metrics, *code smells* are also a well-known indicator of the presence of code TD [40], [45]. Code smells are warning signs indicating possible deeper problems in the design or code of software, often resulting from the violation of at least one programming principle [46]. These problems may impede the software maintenance process and impose the need for code refactoring [26]. In addition, *Automatic Static Analysis (ASA) tools*, such as FindBugs[4] or Checkstyle[5], are also widely used to indicate TD [47]–[49]. ASA tools allow the analysis of source code in search for bugs or violations of good programming practices that can cause failures or quality decay of the software. Most of these violations can be removed through refactoring to avoid unforeseen problematic situations [50].

In an attempt to provide an empirical TD estimation and assessment, various TD indexes, that is, indexes that offer an evaluation of the overall quality (in terms of TD) of a software application, have been proposed by researchers and subsequently implemented as industrial tools [6]. To quantify their TD indexes, these tools, initially gather their atomic data by calculating several TD indicators, such as OO metrics, software quality metrics, violations or code and architectural smells. Subsequently, to assess the quality of both the architecture and the code of an application they employ well-known models for modelling TD, such as the ISO/IEC 25010 standard [51], and the Software Quality Assessment based on Lifecycle Expectations (SQALE) [52] methodology among others [53]–[55].

1.5 Technical Debt Management Activities

Technical Debt Management (TDM) is one of the fastest-growing research areas of software technology and even has a dedicated conference, namely International Conference on Technical Debt (TechDebt). TDM includes several different activities that assist managers and developers in making TD visible and controllable [5], such as TD *identification*, *estimation*, *prioritization*, *prevention* and *repayment*. There are many methods and tools proposed in the literature for supporting TDM activities [56] and, as in the case of financial debt, the management of TD must be programmed founded on the amount of interest and the possibility of repayment over time. Concerning specific approaches to TDM, Brown et al. [2] stress the need to develop new models and techniques for assessing, managing, identifying causes, and repaying TD based on its economic

[4] http://findbugs.sourceforge.net/

[5] https://checkstyle.sourceforge.io/

impact. They argue that compensation must be made in such a way that there is a balance between short-term deadlines and long-term viability. Additionally, Seaman et al. [57] identify four approaches to TDM, including Cost-Benefit Analysis, Analytic Hierarchical Process (AHP), Portfolio Management Model and Real Options.

2 Forecasting Concepts

Forecasting is the process of making predictions of the future based on past and present data, usually by analysis of trends. Being able to predict future values of an observed attribute plays an important role in nearly all fields of science and engineering [58]. Due to the increasing variety and complexity of forecasting problems over the years, many forecasting techniques have been developed, and continue to be developed until today, each for a special use. The forecasting domain has been influenced, for a long time, by statistical methods that can be classified under two broad categories: *causal* (or associative) and *time series* models. Causal models (including the widely used regression analysis) assume that there is a cause-and-effect relationship between the variable of interest and other variables, and therefore try to discover that relationship to forecast future values. Time series models (including the widely used ARIMA model) assume that information needed to produce forecasts is contained in a set of time-dependent data that will continue to follow same patterns as in the past [59]. During the last decades however, *Machine Learning (ML)* models have drawn attention and have established themselves as serious contenders to classical statistical models in the forecasting community [60]. These models, also called black box or data-driven models, are self-correcting learning algorithms that utilize supervised, unsupervised or reinforcement learning to acquire knowledge of the stochastic dependency between the past and the future, based only on historical data.

The experts' opinions regarding which of the two approaches (i.e., time series and ML) yields more accurate predictions vary. In a recent study by Makridakis [61], the authors claim that ML methods need to become more accurate, requiring less computer time, and be less of a black box. A major contribution of their work is in showing that traditional statistical methods are more accurate than ML ones and pointing out the need to discover the reasons involved, as well as devising ways to reverse the situation. However, in their comparisons they made clear that the results might be related to the specific data set being used. They believe that if the series are much longer in length, ML methods can train their weights more optimally. On the other hand, in related studies Werbos showed that Artificial Neural Networks (ANNs) can achieve better results compared to traditional statistical methods such as linear regression and Box-Jenkins (ARMA, ARIMA) approaches [62], [63]. A similar study by Lapedes and Farber [64] concludes that ANNs can be successfully used for modelling and forecasting nonlinear time series. Recently, other models appeared such as regression trees, support vector regression and nearest neighbor regression [65], [66].

2.1 Causal or Associative Models

Causal, or associative, models assume that the variable that needs to be forecasted is somehow related to other variables in the environment through a cause-and-effect relationship. In this case, the forecasting challenge is to discover the relationships between the variable of interest and these other variables. These relationships, which can be very complex, take the form of a mathematical model, which is used to forecast future values of the variable of interest. Some of the best-known causal models are regression models, such as Linear and Multivariate regression, or regularization models, such as Ridge and Lasso regression.

Linear and Multivariate regression are the most commonly used techniques for modelling the relationship between two or more independent variables and a dependent variable by fitting a linear equation to observed data. In the simplest case, the Linear regression model allows for a linear relationship between the forecast variable and a single predictor variable. When there are two or more predictor variables, Multivariate regression is used. The main advantages of these techniques are their simplicity and that they are supported by many popular statistical packages. During Linear and Multivariate regression, the coefficients of these variables are estimated using the least squares method. However, quite often simple linear regression models are suffering from over-fitting or under-fitting. Ridge [67] and Lasso [68] regression are some of the simple techniques to reduce model complexity, reduce multi-collinearity and prevent over-fitting by applying regularization, i.e., add some more constrains to the loss function. In the case of Ridge regression, those constraints are the sum of squares of the coefficients multiplied by the regularization coefficient (lambda). This regularization type is called L2. Lasso regression, works in a similar way but instead of adding squares to the loss function it adds absolute values of the coefficients. As a result, during the optimization process, coefficients of unimportant features may become zero, which allows for automated feature selection. This regularization type is called L1.

2.2 Time series models

A time series is a collection of consecutive observations made at equally spaced time intervals. A fundamental assumption in time series analysis is stationarity, as statistical properties such as mean, variance, and autocorrelation are constant over time. Box and Jenkins introduced the Autoregressive Integrated Moving Average (ARIMA) technique to deal with the modeling of non-stationary time series [23]. ARIMA models have been successfully used in software evolution modelling [19]–[22] for forecasting based on historical data. These models are suited for stationary and non-stationary series and can be adjusted easily if significant changes take place in the trends.

The ARIMA model is parameterized by adjusting three distinct integers: *p*, *d* and *q*. Parameter *p* represents the auto-regressive (AR) part of the model, i.e., regression of the time series onto itself. Basic assumption is that current series values depend on previous values with some lags, where *p* is the maximum lag in the model. Parameter *d* stands for the integrated (I) part of the model and incorporates the amount of differencing (i.e. the number of past time points to subtract from the current value) to apply to the time series. Parameter *q* is the moving average (MA) part of the model. Basic assumption is that current error depends on the previous with some lag, which is referred to as *q*. This allows to set the error of the model as a linear combination of the error values observed at previous time points in the past.

The famous Box and Jenkins [23] strategy used to tune ARIMA models involves four steps: i) *Identification*: In order to successfully apply an ARIMA model for prediction, the observed time series has to be analyzed for stationarity. If non-stationarity is identified during this process, the effect can be removed by differencing or seasonal differencing the series. The number of differences performed before a time series becomes stationary corresponds to the *d* parameter of the model; ii) *Estimation*: Once stationarity is ensured, the next step is to plot the Auto-Correlation Function (ACF) and Partial Auto-Correlation Function (PACF) of the time series to determine the appropriate values of *p* and *q* parameters. The ACF correlogram reveals the correlation between the residuals of the data at specific lags. Then, the actual time series has to be modelled using the ARIMA(*p*,*d*,*q*) parameters previously defined; iii) *Diagnostic testing*: Once a series has been estimated, the next step is to test the model against competing models. Appropriate tests, such as fit statistics and goodness of fit are used to select the best model, by analyzing the residuals of the series for any possible correlations; iv) *Application*: During a training phase, the model is optimized for the data it is built from. As a final step, it is critical to test the model on observations that have not been used to train it. Hence, to ensure the ability of the model to generalize well, it is important to hold out some observations that can be used to evaluate the predictive power of the model on unseen data.

2.3 Machine Learning Methods

ML models are self-correcting learning algorithms that utilize various forms of learning, such as supervised, unsupervised or reinforcement, to predict new outcomes based on previously known results. Although most of these methods have existed for a long time, it is only during the last decades that they have drawn attention due to the constantly improving models, data and processing capacities. While traditional statistical forecasting techniques use only strictly formatted historical data, ML forecasting can take advantage of several data sources, since data can be of different source, format, dimensionality, etc. However, if not handled correctly, these methods can suffer from serious drawbacks such as the lack of interpretability (black box), expensive computational requirements or overfitting. Some of the most widely used ML forecasting models are Support Vector Machine, K-Nearest Neighbor, Decision Trees, Random Forest and various ANN variants, such as Multi-Layer Perceptron, Bayesian and Generalized Regression Neural Networks [61].

Support Vector Machines (SVM) were originally developed for solving classification problems but have been later extended to the domain of regression problems. The goal of Support Vector regression (SVR) [69] is to find a function that approximates the actually obtained target values for all the training data, and has a minimum generalization error. To achieve this, it tries to learn a non-linear function by linearly mapping features into high-dimensional, kernel-induced feature space. K-Nearest Neighbor regression [70] is a nonparametric regression method basing its forecasts on a similarity measure, i.e., the Euclidean distance between the points used for training and testing the method. Thus, given a number of inputs, the method picks the closest training data points and sets the prediction as the average of the target output values for these points.

As in the case of SVM, Decision Trees were originally developed for solving classification problems but were later extended to the domain of regression problems. A Regression Tree (RT) [71] is a variant of decision trees that is built through an iterative process of splitting the data into partitions, and then splitting it up further on each of the branches. Initially all of the samples in the training set are put together in one node. The algorithm chooses an independent variable with values that minimize the sum of the squared deviations from the mean in the separate parts. An enhanced version of the RT algorithm is Random Forest (RF) method [72]. RF is an ensemble of Decision Trees trained with the "bagging" method [73]. Bagging repeatedly selects a random sample with replacement of the training set and fits trees to these samples. After training, predictions for unseen samples can be made by averaging the predictions from all the individual regression trees or by taking the majority vote.

Chapter III Related Work on Technical Debt Estimation and Forecasting

"*Concentrate all your thoughts upon the work in hand. The sun's rays do not burn until brought to a focus.*"

Alexander Graham Bell - Scottish inventor, scientist, and engineer

Published as:

D. Tsoukalas, M. Siavvas, M. Jankovic, D. Kehagias, A. Chatzigeorgiou, and D. Tzovaras, "Methods and Tools for TD Estimation and Forecasting: A State-of-the-art Survey," in *IEEE International Conference on Intelligent Systems (IS 2018)*, 2018, pp. 698–705. doi: http://dx.doi.org/10.1109/IS.2018.8710521.

Chapter Summary

Technical debt (TD), a metaphor inspired by the financial debt of economic theory, indicates quality compromises that can yield short-term benefits in the software development process, but may negatively affect the long-term quality of software products. Numerous techniques, methods, and tools have been proposed over the years for estimating and managing TD, providing a variety of options to the developers and project managers of software applications. However, apart from managing TD, predicting its future value is equally important since this knowledge is expected to facilitate decision-making tasks regarding software implementation and maintenance, such as incurring or paying off TD instances. To this end, the purpose of the present chapter is to (i) summarize the work that has been conducted until today in the field of TD estimation and forecasting, and (ii) to identify existing open issues that have not been adequately addressed yet and require further research. The present survey led to two interesting observations. Firstly, none of the existing TD estimation methods and tools has reached a desired level of maturity, while a large volume of previously uninvestigated metrics and techniques exist that could potentially increase the completeness of TD estimation. Secondly, no notable contributions exist in the field of TD forecasting, indicating that it is a scarcely investigated field. The latter constitutes the main finding of the present literature review, since TD forecasting could lead to the development of practical decision-making mechanisms, which could assist developers and project managers in taking proactive actions regarding TD repayment.

1 Technical Debt Estimation Methods and Tools

TD estimation (or quantification) is a specific activity of TDM that quantifies the benefit and cost of known TD in a software system through estimation techniques. As it is the case for all TDM activities, TD estimation methods and tools can be also separated into two broad categories, the financial approaches and the software engineering approaches.

1.1 Financial Approaches

In recent years, various approaches based on economic theory have been applied to quantify TD principal and interest. One of the first TD modeling and visualization efforts include the Highsmith curve [74], which quantifies TD as the difference between actual and optimal Cost of Change (COC) over time.

Following the financial aspects of TD estimation, Guo and Seaman [75] leverage the Portfolio Management theory in the finance domain to determine the optimal collection of TD items that should be incurred or held. Through their approach, the researchers try to quantify TD principal as the effort required to resolve TD items and TD interest as the probability of interest to occur. Based on the Portfolio Management approach, Holvitie and Leppanen introduce DebtFlag [76], a tool designed to support TDM by capturing, tracking and resolving TD of software projects at the implementation level. This tool provides developers with lightweight documentation functionalities to capture TD and link them to corresponding parts in the implementation phase.

In another approach towards TD estimation, Alzaghoul and Bahsoon [77] state that the web service selection decision might incur a TD that is essential to be quantified and managed. Towards this aim, they exploit the Real Options theory by introducing a new method that aims to quantify TD in service level for cloud-based system architectures. In their approach, they construct a two-step binomial tree to quantify and predict the period during which TD is reduced to zero, taking into consideration several dimensions including Service Level Agreement (SLA), none-compliance, quick selection decisions and underutilization of the web service capacity.

Finally, in their study, Curtis et al. [53] are based on Software Economics theories and quantify TD as the cost of violating architectural rules, code rules, and best practices, giving three levels of severity to violations: high, medium and low. In order to achieve that, they introduce a function that quantifies principal and interest taking as input software artefacts, metrics, historical effort, or personnel activity. To further support their findings, they integrate their formula into CAST[6], a tool that quantifies TD by identifying violations in source code and categorizing them by quality attributes. This tool is designed to manage TD by analyzing multi-tiered, multi-technology applications for technical vulnerabilities and adherence to architectural and coding standards.

1.2 Software Engineering Approaches

The software engineering aspect of TD estimation lays its foundations on the notion that software quality metrics and the time and effort required for a software change can be used to quantify its impact. Therefore, TD Principal is quantified, in most of the approaches that exist in the literature, by summing up the estimated effort to fix each individual inefficiency that is identified through automated analysis tools [78]. For instance, if a software is vulnerable or does not satisfy all system requirements, vulnerabilities must be fixed and the requirements met. Therefore, the number of vulnerabilities or unsatisfied

[6] https://www.castsoftware.com/

requirements is an indicator of TD. In addition, if a software has been produced with excessively complex code, then its future changes are more expensive. In this case, metrics like coupling, cohesion, complexity, etc. can also be applied to assess TD [79].

Over the last years, several software engineering methods have been proposed to quantify a software system's level of TD. In a related study, Gaudin [80] introduces a new TD estimation formula that takes as input custom source code metrics to calculate a global indicator of TD. This indicator reflects how much effort is required to get a flawless score on the Seven Axes of Quality analysis, namely the bad distribution of the complexity, duplications, lack of comments, coding rules violations, potential bugs, no unit tests and bad design. One of the most representative TD assessment tools that used to make use of the Seven Axes of Quality formula was the Technical Debt Evaluation plugin for SonarQube[7], an open source platform for continuous inspection of code quality. SonarQube provides a dashboard for visualizing quality attributes of code, tests, design, and architecture. Under the hood, it performs static analysis of the source code to detect bugs, code smells and security vulnerabilities, among others. In the latest versions of the tool however, TD assessment is performed by using a new in-house formula. More specifically, SonarQube now checks code compliance against a set of classified coding rules and if the code violates any of these rules, it considers it as a violation or a TD item.

In another study, Bohnet and Döllner [81] calculate TD by using Software Maps to monitor code quality and development activity. In their approach, they argue that software maps enable managers to express and combine information about software development, software quality, and system dynamics. They also claim that software maps can support decision-making processes by investing the scarce developers' time to improve code quality and facilitate the future maintenance of the system.

Similarly, Nugroho et al. [29] propose an approach for quantifying TD principal and interest based on an empirical assessment method of software quality developed by the Software Improvement Group (SIG). Their method comprises of two parts, the estimation of repair effort and the estimation of maintenance effort. Following a different approach, the work of De Groot et al. [82] introduces three models to determine software value based on the notions of TD by using the Rebuild Value, i.e. the cost to rebuild a system from scratch using similar technology. In one of their models, they calculate the TD as the amount of work (in person-months) that is required to improve the level of software quality. However, to the best of knowledge, no tools exist to implement the methods mentioned above.

In addition, Ernst [83] proposes an approach for TD estimation on the requirements level by introducing Solution comparison, a method that calculates the distance between the optimal specification and the actual implementation of the system. To validate his method, in the same study he also introduces RE-KOMBINE, a requirements modeling tool that enables useful measures and models the TD present in requirements tradeoffs. Another tool based on the same notion is proposed by Strasser et al. [84]. The Automated Software

[7] https://www.sonarqube.org/

Tool for Validating Design Patterns based on the Role Based Metamodeling Language (RBML) is a compliance checker that quantifies TD on the design level by calculating the distance between a realization of a design pattern and the intended design. For that purpose, the tool compares UML class diagrams of instances of design patterns with their RBML representations and reports back if the given UML diagram is compliant or not.

Moreover, Curtis et al. [85] present a formula for TD calculation with adjustable parameters for estimating the principal of TD from structural quality data. On the other hand, Letouzey [55] presents the widely used SQALE method for monitoring and assessing the quality and TD of the source code of a software application. One of the most representative tools for assessing the TD of a software product using the SQALE method is SQuORE[8], a commercial quality management tool that uses four indicators namely: efficiency, portability, maintainability, and reliability to calculate code TD. For each of these indicators, a set of quality rules is assigned. One of the advantages of this tool is that it takes into account source code, unit tests, documentation quality, available functional requirements, etc. resulting in a more accurate and complete calculation of TD. In previous years, SonarQube also used the SQALE method (besides the Seven Axes of Quality formula mentioned above) to assess the TD of a software product before switching to its new in-house method.

In the same way, Nord et al. [86] follow an architecture-focused and measurement-based approach to introduce a metric for the rapid management of TD associated with architecture level in order to optimize development costs. To support their work, they argue that making the architectural debt visible provides all necessary information for making informed decisions for managing the potential impact of rework over time. In another work, Marinescu [54] introduces a novel framework for assessing TD using a technique for identifying architectural smells (called design disharmonies), detected by evaluating different metric-based rules that cover the majority of the aspects of design, such as complexity, coupling, and encapsulation. The impact of disharmonies is formulated as an index that uses three factors for its calculation, namely influence, granularity, and severity. This framework was integrated into inFusion[9], a tool that evaluates software quality by providing a global score known as the Quality Deficit Index (QDI).

Finally, in a recent study, Sanchez et al. [87] introduce TEDMA, an open tool that quantifies TD by computing TD metrics and integrating techniques implemented by third party tools. The novelty of this tool is that it supports analysis of the evolution of the metrics over the software evolution of the project. This kind of approach has not been introduced in any of the previous methods and tools presented in the study.

[8] https://www.vector.com/int/en/products/products-a-z/software/squore/

[9] inFusion tool is no longer supported and has been evolved into http://www.aireviewer.com/

Other popular quality assessment tools that worth mentioning are Sigrid[10], Structure101[11], NDepend[12], and Teamscale[13].

2 Software Evolution and Technical Debt Forecasting

Numerous techniques, methods, and tools have been proposed over the years for estimating and managing TD, providing a variety of options to the developers and project managers of software applications. However, apart from managing TD, predicting its future value is deemed equally important, since this knowledge is expected to facilitate decision-making tasks regarding software implementation and maintenance, such as incurring or paying off TD instances. In this section, we investigate the state-of-the-art and examine studies that can contribute towards the TD forecasting concept, though the lens of Software Evolution and Software Quality forecasting.

Software evolution is a term used in software engineering to refer to the process that starts with the development and then provides incremental updates of the software. According to Lehman's laws of software evolution, software systems must evolve over time or they will become irrelevant [88]. Gaining a higher level of information about the evolution of large software systems is a key challenge in dealing with increasing complexity and decreasing software quality [89]. For this reason, the attempts to analyze, understand and predict the evolution of a software system have increased considerably in the last years [90], and nowadays, the terms software evolution and software maintenance are often used as synonyms [91]. In his work, Mens [91] stresses the need to develop better predictive models for measuring and estimating the cost and effort of software maintenance and evolution activities with higher accuracy. Therefore, the improvement of these models can be proven of great value in software development, since being able to estimate the future evolution of a software product, could provide valuable insight for its quality as well.

According to ISO/IEC 25010 [51], which is a well-accepted international standard, the notion of software quality is hierarchically decomposed into a set of quality attributes, like maintainability, reliability, and security. A multitude of quality models have been proposed over the years allowing the assessment and/or prediction of these quality attributes individually [92]–[94]. For instance, Wagner [92] implements a model based on Bayesian Belief Networks for assessing and predicting the maintainability of a software application based on a set of software metrics. Similarly, Van Koten et al. [93] try to predict object-oriented software maintainability by applying a Bayesian network, while Zhou et al. [94] approach the same problem by using multivariate adaptive regression splines.

Since quality attributes are relatively abstract and difficult to be measured directly from the artifacts of software products (e.g., source code), ISO/IEC 25010 [51] further

[10] https://www.softwareimprovementgroup.com/solutions/sigrid-software-assurance-platform/

[11] http://structure101.com/products/workspace/

[12] https://www.ndepend.com/

[13] https://www.cqse.eu/en/products/teamscale/landing/

decomposes them into a set of more concrete quality properties (e.g., complexity), which can be directly quantified through common metrics (e.g., McCabe's Cyclomatic Complexity). Similarly, to the high-level quality attributes, a large number of methods have been proposed to estimate the future evolution of software quality properties and metrics used to calculate them, such as future number of changes [19], [22], [95], [96], software defects [20], [97], fault-proneness [11], [21], [98], [99], code smells [10], and vulnerabilities [100]. The majority of these methods try to approach the subject by applying time series or ML models on individual software properties based on the analysis of available information (historical data, trends, source code metrics, etc.).

2.1 Time Series Approaches for Software Quality Forecasting

A commonly used technique to analyze the evolution of software systems is time series analysis. In their study, Yazdi et al. [19] model the evolution of the design of software systems by applying ARMA time series to several typical projects successfully. Based on the empirical results the authors point out that time series models can predict the future changes of the next revisions of the systems with sufficient accuracies. In another study, Kenmei et al. [22] use time series models to forecast future change requests evolution and to identify trends based on data collected from three large open source applications. They highlight that time series are capable to model change requests and act as a support tool for project staffing and planning. Likewise, Raja et al. [20] use the time series approach to predict defects in software evolution. They use defect reports for eight open source projects and build time series models to predict software defects which lead to the conclusion that the model may be used to facilitate planning for software evolution budget and time allocation. Similarly, Goulão et al. [21] build a time series model to forecast the change requests evolution based on data collected from Eclipse's change request tracking system. Additionally, they include the identification of seasonal patterns and tendencies, which is important to validate that usage of seasonal information significantly improves the estimation ability of this model, when compared to other ARIMA models. Finally, in a recent study by Antoniol et al. [101], the authors apply time series to model the evolution of software metrics, such as size and complexity, based on the analysis of historical data (500 releases) of the Linux Kernel. They claim that forecasting the evolution of these software metrics can be used to improve the management of software artifacts or processes. By tuning and building an ARIMA model, their results show average errors below 15% for predictions up to three steps ahead.

2.2 Machine Learning Approaches for Software Quality Forecasting

In addition to time series analysis, multiple studies address the problem of predicting the future evolution of various TD-related software quality aspects by employing ML techniques. First and foremost, TD is an indicator of software quality with an emphasis on maintainability. During the past years, a plethora of studies have dealt with the aspect of software maintainability prediction, usually by employing ML regression techniques [95], [96], [102]. In their study, Chug and Malhotra [95] introduce a benchmarking framework for predicting the number of changes, and therefore the maintainability of a software application, using OO metrics as predictors. Through their framework, they compare the effectiveness of 17 ML techniques (including linear regression, decision trees, SVM and genetic algorithms) over seven open source systems. They conclude that although good predictive performance is achieved by almost all ML techniques, the genetically adaptive learning models perform better than the others do. In a similar study, Elish and Elish [96] compare various ML techniques, such as multivariate linear regression, SVM, ANN, TreeNet, and regression trees, also for predicting maintainability through the number of line changes. Their results indicate that competitive prediction accuracy is achieved when applying the TreeNet model. Finally, in a similar study, Malhotra and Lata [102] investigate the generalizability of 15 ML models in predicting cross-project software maintainability using OO metrics as predictors. More specifically, three individual models are trained on three Java open-source projects and subsequently each model is validated against the other projects. The results show that cross-project prediction can be successfully applied to predict software maintenance effort of open-source software.

Code smells are considered one of the key indicators of TD [40]. Recently, a plethora of work related to code smell detection has been proposed in the literature [10], [103]–[105]. Most of these studies evaluate different ML classifiers on the task of detecting various types of code smells and conclude that with the right optimization, ML algorithms can achieve good performance. In a systematic literature review by Azeem et al. [106], the authors summarize the research on ML algorithms for code smell prediction. From the selected 15 studies, they conclude that Decision Trees and SVM are the most widely used ML algorithms for code smell detection, while Random Forest seems to be one of the most effective algorithms in terms of performance. In another study by Fontana et al. [10], 16 different supervised ML techniques for code smell detection are compared on 74 software projects. They report that code smells can be detected with very high accuracy, while the highest performance is obtained by using J48 and Random Forest algorithms. A similar empirical study by Cruz et al. [103] evaluates seven different ML classifiers (including Logistic Regression, Naive Bayes, Decision Tree, K-Nearest Neighbors, Random Forest and XGBoost) on the task of detecting four types of code smells. The authors conclude that with the right optimization, ML algorithms can achieve good performance (F1 score up to 0.86) for two smells: God Class and Refused Bequest and report that Random Forest and XGBoost outperform the other classifiers. In an empirical study by Pecorelli et al. [104], a ML-based approach is proposed (including Random Forest, Logistic Regression, and Naive-Bayes) to classify code smells according to the perceived developers' criticality. They evaluate their approach on nine open-source projects and report that Random Forest is the best-performing algorithm with an F1 score up to 85%. On the other hand, a study by Lujan et al. [105] investigates the role of static analysis warnings as features of ML models for the detection of three code smell types. The authors use five open-source projects to build classifiers (including Random Forest, Naive Bayes, and SVM) exploiting the most relevant features and conclude that using Random Forest can predict code smells fairly accurately. Finally, in a recent study by Sharma et al. [107], the authors explore the feasibility of applying Deep Learning models not only to detect specific code smells (direct-learning), but also to perform transfer-learning in the context of smell detection. They conclude that CNN, RNN, and autoencoders models can be effectively used for code smell detection (though with varying performance), and that transfer-learning is feasible for implementation smells with performance comparable to that of direct-learning.

Besides code smells, software change proneness and defect proneness are also popular TD indicators; a change-prone module tends to produce more defects and accumulate more TD [108]. To date, various studies have investigated the applicability of ML models for the identification of change- and defect-prone modules [11], [97]–[99], [108]–[112]. As regards the defect-proneness prediction using ML techniques, in a study conducted by Arisholm et al. [11], the authors propose a multivariate regression model for predicting defect-prone components of object-oriented legacy systems by using history change and fault data from previous releases. Moreover, Gondra et al. [98] employ sensitivity analysis to select software metrics that are more likely to indicate the existence of errors, and afterward, train an ANN to predict future fault-proneness. In another study by Nagappan et al. [97], principal component analysis is applied on code metrics to build regression models that accurately predict the likelihood of post-release defects. In a study by Khoshgoftaar et al. [99], the authors use regression and classification trees to identify fault and non-fault prone modules on multiple releases of a large-scale legacy telecommunications system, concluding that these algorithms result in predictions with satisfactory accuracy and robustness. Finally, a study by Challagulla et al. [112] evaluates different ML models (including Decision Trees, Naïve-Bayes, Logistic Regression, Nearest Neighbor) for identifying faulty real-time software modules on four different software defect data sets. The results show that there is no particular model that performs consistently better for all the data sets.

Similarly to defect-proneness prediction, various researchers have investigated ML techniques for the prediction of software change proneness. A study by Pritam et al. [109] investigates the effectiveness of code smells in predicting the change proneness of software modules. For this purpose, the authors use different ML algorithms (including Naive Bayes, Random Forest, and Decision Tree) on a dataset comprising 8200 Java classes from 14 software systems. The results suggest that code smells are indeed a good predictor of class change proneness, while sensitivity and specificity values for all models are well over 70%. In another study by Abbas et al. [108], the authors apply 10 single and ensemble ML classifiers (including Naive Bayes, Random Forest, and Decision Trees) on a dataset from a commercial software to investigate the effectiveness of OO metrics in predicting change-prone software modules. The results indicate a high prediction performance for many of examined classifiers, while Random Forest is the best-performing model with an F1 score slightly above 90%.

Finally, TD is closely related to software refactoring, since the later constitutes the only effective way to reduce it on existing source code. However, researchers have only recently begun to explore how ML can be used to help in identifying refactoring opportunities [113]. In a study by Aniche et al. [113], six different ML classifiers (including Logistic Regression, Naive Bayes, SVM, Decision Trees, Random Forest) are applied on a dataset comprising over two million refactorings from 11,149 open-source projects. The results show that the models are able to predict 20 different refactoring types at class, method, and variable-levels and report that Random Forest is the best-performing algorithm with accuracy higher than 90%.

2.3 The Unexplored Importance of TD Forecasting

The multitude of models that are available in the literature for predicting the current value or the future evolution of specific quality attributes and quality properties reveal the importance of quality prediction and forecasting in the software engineering community. However, with the evolution of a software system, accumulated TD is evolving as well. Since TD is an indicator of software quality (with an emphasis on maintainability), predicting its future value is considered equally important.

Various studies have focused on analyzing the evolution of TD and its impact on software development, from different perspectives [8], [18], [37], [114]–[116]. In their study, Ampatzoglou et al. [18] highlight the need for knowing TD evolution, while stressing the need for project managers to be able to preserve a software product maintainable for as long as possible. For that purpose, Chatzigeorgiou et al. [116] introduce the term “breaking point”, which refers to the point in time when the accumulated interest will be equal to the TD principal, i.e., the cost becomes higher than the benefit. Trying to expand this work, Ampatzoglou et al. [37] instantiate and validate FITTED, a framework that assesses the breaking point of source code modules to support decision making with respect to investments on improving quality of a software, thus providing managers with an insightful decision-making tool. Hence, forecasting the evolution of TD principal and interest could be valuable for estimating the point in which the software product could become unmaintainable.

To effectively predict how the TD of a software system will progress in the future in order to improve the TD repayment strategy, it is necessary to constantly monitor and analyze its evolution. While the previously mentioned studies indicate that there has been extensive research with respect to forecasting the evolution of quality attributes and properties, directly or indirectly related to TD, no concrete contributions exist so far regarding TD forecasting [117], indicating that it is a scarcely investigated field. In a first attempt towards this issue, Skourletopoulos et al. [117] introduce the concept of predicting TD for Software as a Service (SaaS) systems, by exploiting COCOMO, a software cost model proposed by Boehm [118]. However, their study is limited only to cloud computing systems.

Under those circumstances, being able to forecast not only the evolution of software quality but also the evolution of TD principal and interest of a software system in the future is of great significance and value. Through our study in this chapter, we identified some interesting open issues that should be addressed through further research. In particular, no concrete contributions exist in the related literature regarding TD forecasting, while there is still a large volume of potential metrics and techniques that have not been used and that could potentially enhance the completeness of the software quality forecasting concept. Such a work would enable project managers and developers to support decision-making in uncertainty and plan precise payback strategies, in order to manage TD promptly and avoid unforeseen situations long-term.

3 Open Issues and Contributions

Despite the multitude of methods proposed in the bibliography for the estimation of TD, there are still **many** open issues that require further investigation. First of all, none of the already proposed methods and tools have reached a desired level of maturity, and according to recent studies [5], there is no commonly accepted standard for estimating and managing TD. As a result, developers and managers perceive the concept of TD in different ways, while current methods and tools are not able to map software quality attributes to TDM activities. Moreover, the majority of well-established TD estimation methods, such as the widely used SQALE method [52], mainly analyze the source code of the software. There is a large volume of potential metrics and techniques that have not been used yet for estimating TD, and which could potentially increase the completeness of the TD estimation concept. In addition, most of the already existing tools provide different TD indexes [6], creating confusion in the community about which of the current metrics should be selected, or how they should be combined [48].

Therefore, an interesting topic would be to investigate whether the combination of software-related metrics extracted from repositories and already existing TD estimation techniques and tools may lead to better and more accurate TD estimation methods. In addition, having in mind that new metrics and techniques for TD are emerging rapidly [119], there is a need for a single tool that combines software metrics and TD estimation techniques implemented by different approaches.

Another critical issue is that no particular approaches have been proposed for the forecasting of TD, which is opposite to extensive research that has been performed for predicting the evolution of individual software features or quality attributes that are directly or indirectly related to the TD of a software project, such as code smells [10], fault-proneness [11] and evolution trends [12]. A contribution to this challenge has high value since TD forecasting could lead to the development of practical decision-making mechanisms aiming to improve the TD repayment strategy. Furthermore, the decision making mechanisms should be integrated into an application or a tool to facilitate efficient identification of the aspects that might cause potential TD accumulation.

Hence, another interesting topic is whether the combination of software-related metrics and already existing software evolution approaches, along with existing forecasting methods could lead to the development of novel models that provide predictions about the evolution of a software’s future TD. Towards this goal, statistical methods such as causal models (including the widely used regression analysis) or time series models (including the widely used ARIMA model) [59] could be investigated. In addition, ML models like Artificial Neural Networks (ANNs) [62], [63], regression trees, support vector regression and nearest neighbor regression [65], [66] could also be examined.

Last but not least, software repositories such as versioning, project management and issue-tracking systems, as well as archived communication between project personnel could be a potential source of TD related data. We believe that there is great potential in mining this information to extract software related metrics and thus, unveil ways that can help to support the development of better TD estimation and prediction methods. In fact, we believe that by analyzing multiple sources of information and predicting the evolution of TD on specific software artifacts, triangulation can be achieved and yield more accurate estimates. To further ease and automate this process, a tool that would utilize multiple sources of information accompanying a software project by pairing existing TD estimation methods with specialized techniques for forecasting, code analysis, software evolution analysis and natural language processing could pave the way for the advance in the state of the art in this domain. By doing so, another important contribution to the research community would be to provide highly balanced, publicly available datasets of TD related metrics that could be reused by future researchers for relevant studies towards comparing or validating TDM methods and tools.

4 Conclusions and Future Work

In the present chapter, we investigated the state-of-the-art and examined the major contributions that have been made until today in the field of TD estimation and forecasting. Through our study, we identified some interesting open issues that should be addressed through further research. In particular, already existing methods and tools for TD estimation have not reached a satisfactory level of maturity yet, while there is still a large volume of potential metrics and techniques that have not been used and that could potentially increase the completeness of the TD estimation concept. In addition, although there has been extensive research with respect to predicting the evolution of individual software features, quality attributes, and quality properties that are directly or indirectly related to the TD of a software project, no concrete contributions exist in the related literature regarding TD forecasting.

Therefore, the improvement of already existing TD estimation methods, by incorporating previously uninvestigated software-related factors with potential relevance to TD is an interesting direction for future research. Another interesting topic would be to investigate different efficient ways to produce TD forecasting models for accurate prediction of TD principal and interest evolution. In addition, it would be useful to examine if TD forecasting could foster the development of high-quality software products. To the best of our knowledge, this is the first study that raises the awareness of this gap in the field of TD.

Chapter IV Time Series Techniques for Technical Debt Forecasting

“*Prediction is very difficult, especially if it's about the future.*”

Nils Bohr - Danish physicist

Published as:

D. Tsoukalas, M. Jankovic, M. Siavvas, D. Kehagias, A. Chatzigeorgiou, and D. Tzovaras, “On the Applicability of Time Series Models for Technical Debt Forecasting,” in *15th China-Europe International Symposium on Software Engineering Education (CEISEE 2019)*, IEEE TEMS, 2019. doi: 10.13140/RG.2.2.33152.79367.

Chapter Summary

Technical debt (TD) is commonly used to indicate additional costs caused by quality compromises that can yield short-term benefits in the software development process, but may negatively affect the long-term quality of software products. Predicting the future value of TD could facilitate decision-making tasks regarding software maintenance and assist developers and project managers in taking proactive actions regarding TD repayment. However, no notable contributions exist in the field of TD forecasting, indicating that it is a scarcely investigated field. This chapter constitutes an initial attempt towards this direction. To this end, in the present study, we empirically evaluate the ability of time series analysis to model and predict TD evolution. To create our dataset, we obtain weekly snapshots of five open source software projects over three years and compute their TD values. We find that the autoregressive integrated moving average model ARIMA(0,1,1) can provide accurate predictions over a fairly long time period for all sampled projects. The model can be used to facilitate planning for software evolution budget and time allocation. The approach presented in this chapter provides a basis for predictive TD analysis, suitable for projects with a relatively long history.

1 Introduction

The Technical Debt (TD) metaphor, a term inspired by the financial debt of economic theory, was introduced in 1992 by Ward Cunningham [1] to describe the problem of making quality compromises that can yield short-term benefits in the software development process, but may negatively affect the long-term quality of software products. The TD metaphor was initially related to software implementation (i.e. at the code level) but was gradually extended to other phases of the software development lifecycle (SDLC), i.e. software architecture, design, documentation, requirements, and testing [2]. In the same manner like financial debt, TD incurs interest payments in the form of increased future software costs usually caused by poor design and code quality. To effectively manage the identification, quantification, and repayment of TD during the SDLC, researchers and practitioners have developed and adopted a multitude of theories, methods and tools [5].

However, prediction of accumulated TD during the software evolution is an open and challenging research issue, as both software system and its TD emerge in parallel [8]. A possibility to predict TD is primarily crucial for software maintainability, which is recognized as one of the most-effort intense activities in the SDLC [3]. System engineers and project managers need the right tools and appropriate training support to be able to perform long-term effective software maintenance [3]. Hence, forecasting the evolution of TD could be valuable for estimating the point in which the software product could become unmaintainable.

Although various aspects, which are relevant for the TD concept, such as code smells [10], fault-proneness [11] and evolution trends [12], have attracted the attention of both academia and industry, to the best of our knowledge no studies are focusing on TD itself [13]. Hence, a method or tool that would provide practical decision-making support by predicting future TD of a software system in uncertainty is of eminent importance. Consequently, software architects and project managers would be able to gain a better understanding of future TD issues and plan appropriate refactoring activities.

As a first step towards TD Forecasting realization, we have developed and applied specific time series models for TD forecasting. Time series models have been widely used in previous studies for predicting software evolution trends, future change requests or software defects [12], [19]–[22]. To this end, in the present chapter we attempt to empirically evaluate the ability of time series to adequately forecast future TD trends. The problem that the present work attempts to solve can be summarized in the following research question:

> ***RQ****: Is the usage of time series models a valid and accurate approach to forecasting Technical Debt in a long-lived, open-source software?*

A positive answer to this question will suggest that time series models can potentially be used as the basis for the construction of the TD forecasting toolbox. We will also investigate the extent to which these models can properly capture the evolution of TD values.

In order to provide answers to the research question mentioned above, we conducted an empirical study. In particular, we initially constructed a relatively large code repository comprising five real-world open-source Java applications retrieved from the GitHub[14] online repository. For each application, we collected 150 snapshots (commits) in weekly intervals, spanning up to 3 years of each system's evolution. This approach led to a dataset containing up to 750 snapshots in total. For each snapshot, we calculated the TD value using SonarQube[15], a popular static code analysis tool. Subsequently, for each application, we employed the Box-Jenkins modelling method to construct and identify the parameters of our models. Finally, we compared the accuracy of these models with a random walk model for various steps ahead into the future in order to reach safer conclusions regarding

[14] https://github.com

[15] https://www.sonarqube.org/

the significance of the observed results. To the best of our knowledge, this is the first study in the field of TD that examines the applicability of time series models for TD forecasting.

The rest of the chapter is structured as follows: Section 2 describes the experiment setup, while Section 3 presents the analysis and a discussion on the results of the experiment. Finally, Section 4 concludes the chapter and discusses ideas for future work.

2 Experimental Setup and Methodology

An overview of the methodology followed within this chapter is presented in Figure 1.

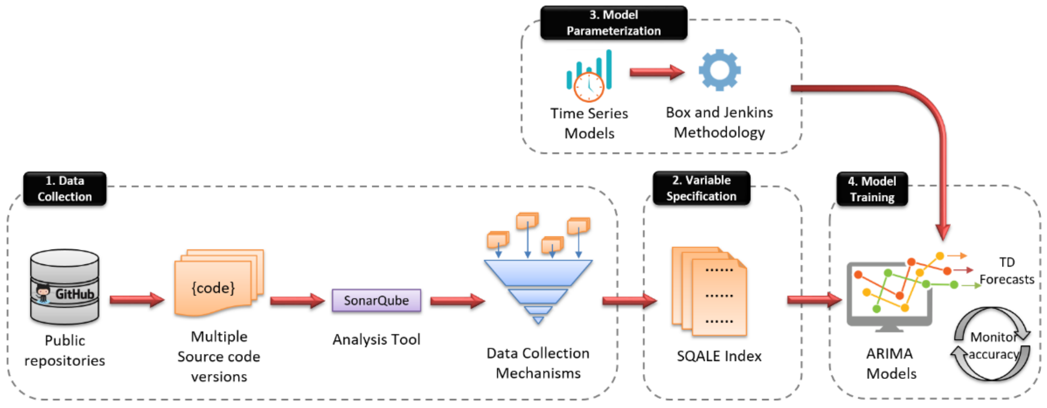


Figure 1: Chapter IV roadmap

2.1 Data Collection

The data used in this study were obtained from five popular open source projects from GitHub repository, namely Apache Kafka, Apache SystemML, Square OkHttp, Square Retrofit and Jenkins. The selection criteria were based on the popularity, activity level, data availability, and the Java programming language. For each system, we collected 150 snapshots (commits) in weekly intervals, spanning up to 3 years of each system's evolution.

2.2 Variable Specification

In the present work, we decided to use the Software Quality Assessment based on Lifecycle Expectations (SQALE) method [52] for quantifying the TD of the selected software products. Specifically, we use the SQALE Index plugin that is provided by SonarQube, a popular open source platform for continuous inspection of code quality. The SQALE Index quantifies the effort that is required to fix all the code violations (i.e. code smells, bugs and vulnerabilities) that reside in the analyzed software product. SonarQube expresses the effort in minutes. Finally, in order to avoid being biased by the size of the software products, similarly to [8], we decided to express the SQALE Index as a ratio i.e., SQALE Index divided by size (lines of code) of the software products. We term the normalized SQALE Index as TD Density and we model it via time series.

2.3 ARIMA Technique

Although there exist several forecasting methods that have proven to yield high predictive power, like Artificial Neural Networks (ANNs), support vector regression (SVR), or Regression Trees (RT), the application of these methods usually requires extensive parameter tuning and computational power. Moreover, these methods depend on the availability and reliability of data on independent variables over the forecasting period, which requires further efforts in data collection and estimation. Thus, as an initial approach towards TD forecasting we decided to employ time series models. The reason is that time series models, compared to machine learning models, are more straightforward, less computationally expensive and less data hungry, in a sense that they provide another modelling approach, which only requires the historical data of the variable of interest to forecast its future evolution behavior.

3 Analysis, Results and Discussion

This section reports the empirical study results for the five analyzed projects. Due to space limitation, we describe the ARIMA approach in detail only for the Apache Kafka project, while for the rest of the projects we present only the results of the selected models.

3.1 Apache Kafka

3.1.1 Identification

As described in Section 2, the first step is the time series plot, which provides an understanding of the nature of the series. When visualizing the weekly Apache Kafka TD evolution over the last three years, we can observe an increasing variation of TD density values. To eliminate this variation and linearize the series, we performed a natural *log(ln)* transformation. As a result, the fluctuations in the transformed series were far less than the original series.

After the logarithmic transformation, we removed the mean and analyzed the transformed series for stationarity, i.e. seasonality and trends. The decomposition of time series is a statistical task that deconstructs a time series into several components, each representing one of the underlying categories of patterns, i.e. trend, seasonality, and residual components of the data. By applying seasonal decomposition to the series, the existence of trend and seasonality becomes even more clear. This can be depicted in Figure 2.

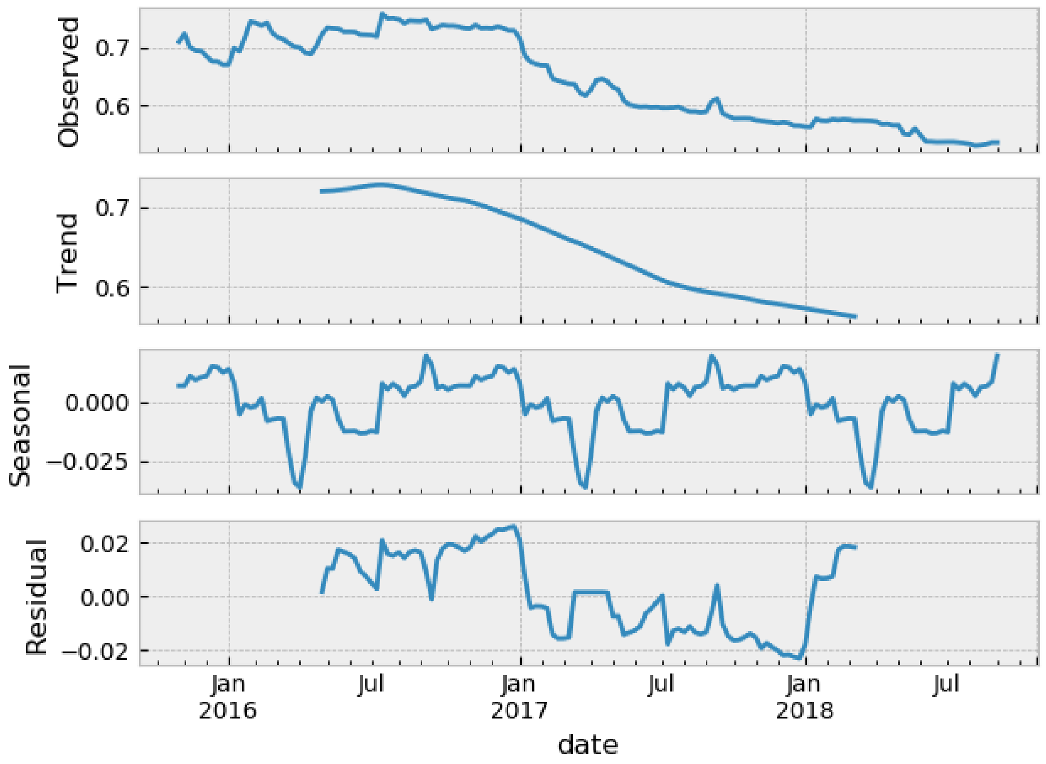


Figure 2: Seasonal decomposition of the TD evolution

By looking at Figure 2, we observe that the seasonal component and the decreasing trend of the data are nicely separated, leading to the conclusion that the series is not stationary in nature and needs to be adjusted in order to successfully apply an ARIMA model for prediction. The most common practice for making a series stationary is to transform the series through differencing. The process below describes all the required steps to make the series stationary and involves ACF and PACF correlograms analysis (see Chapter II2.2), as well as Dickey–Fuller tests [120], which test the null hypothesis that a unit root is present in an autoregressive model. Detailed results of the Dickey-Fuller test on original data are presented in Table 1.

Table 1: Dickey-Fuller Test on Original Data

Test Statistic	-0.572788
p-value	0.877013
Critical Value (1%)	-3.475325
Critical Value (5%)	-2.881275
Critical Value (10%)	-2.577293

For a time series to pass the stationarity test, the “Test Statistic” value should be lower than the “Critical Value”. For the original time series, clearly this is not the case. This can be further supported by the fact that the ACF chart is characterized by the slow linear decay in the spikes. As a next step, we performed the test on the ln-transformed time series rather than the original. Detailed results of the Dickey-Fuller test on ln-transformed data are presented in Table 2.

Table 2: Dickey-Fuller Test on LN-Transformed Data

Test Statistic	-0.453466
p-value	0.900782
Critical Value (1%)	-3.475325
Critical Value (5%)	-2.881275
Critical Value (10%)	-2.577293

While the ln-transformation helped to improve the stationarity of the data, it did not completely eliminate the fluctuations and alternating factors. Again, the ACF chart is characterized by a slow linear decay in the spikes. The next step is to take a first-order difference of the data to eliminate the overall trend from the series. If the original series is Y_t, then the differenced series is indicated by $Y_t = Y_t - Y_{t-1}$. The ACF and PACF plots of first-order differenced time series are presented in Figure 3, while Dickey-Fuller test results are presented in Table 3.

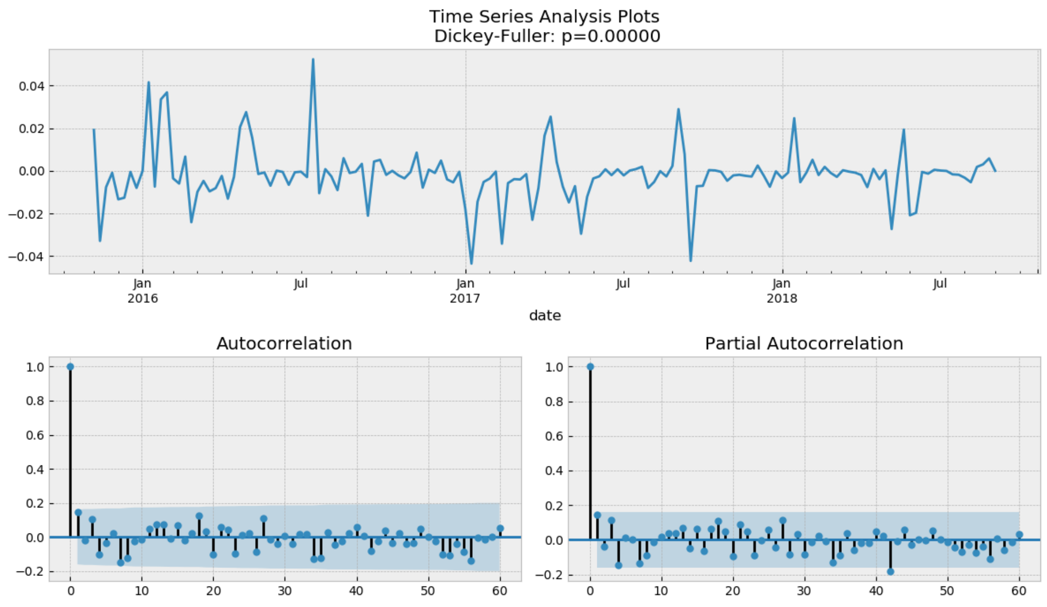


Figure 3: Dickey-Fuller, ACF and PACF of the first-order differenced data

Table 3: Dickey-Fuller Test on First-order Differenced Data

Test Statistic	-10.53053
p-value	9.177006e-19
Critical Value (1%)	-3.475325
Critical Value (5%)	-2.881275
Critical Value (10%)	-2.577293

Table 3 indicates that the time series is now stationary, as the Test Statistic value is lower that the Critical Value. The significance of *p-value* ($p \leq 0.05$) also confirms that taking the first-order difference has made the data stationary. This is also illustrated on the plot itself, as there is no visible trend. The number of required transformations until the series becomes stationary corresponds to the *d* parameter of the ARIMA(*p*,*d*,*q*) model, thus setting the value of $d = 1$ can be safely supported by the analysis performed in this step.

3.1.2 Estimation

During the previous analysis, we applied first-order differencing on the original data to make it stationary and then identified the *d* parameter of the ARIMA model. During this phase of the analysis, we have to identify the auto-regressive (AR) *p* and moving average (MA) *q* parameters. The selection process of these parameters is not trivial task, but we followed the practical recommendations for selection of parameters *p* and *q* through visual inspection of the ACF and PACF correlograms [121]. In short, if the ACF of the series disappear gradually, and the PACF of the series disappear abruptly, it indicates an AR component. An opposite behavior, i.e., ACF disappear abruptly and PACF disappear gradually, indicates an MA component. If both ACF and PACF disappear gradually or disappear abruptly, various models can be tested to identify the *p* and *q* parameters accurately.

We analyzed the ACF and PACF plots of the first-order differenced values illustrated in Figure 3 in accordance with the ARIMA guidelines. In general, the *x*-axis of the ACF plot indicates the lag at which the autocorrelation is computed. The *y*-axis indicates the value of the correlation (between -1 and 1). For the dataset, both ACF and PACF plots indicate a cut off at lag 1. In this case, three competing models, ARIMA(1,1,0), ARIMA(0,1,1) and ARIMA(1,1,1), need to be compared using goodness of fit tests and residual analysis to accurately identify the *p* and *q* parameters. We excluded ARIMA(0,1,0) from further analysis as we later used it as a “random walk” for the purpose of comparing it with the selected model during the application step. The next step is to compare the three models for goodness of fit by applying fit statistics and select the most suitable one based on the optimal results.

3.1.3 Diagnostic testing

During this phase of the ARIMA analysis, we analyzed goodness of fit of the selected models as well as the residuals (i.e., the difference between the predicted and the actual values) for any possible correlations to verify adequacy of these models. A summary of the fitted models ARIMA(1,1,0), ARIMA(0,1,1) and ARIMA(1,1,1) is presented in Table 4.

Table 4: ARIMA Candidate Model Results

Model	**Model Results**				
	AIC	***Ljung-Box (Q)***	***Prob(Q)***	***coef P* >|*z*|**	
ARIMA(0,1,1)	-987.52	39.62	0.49	***ma***	0.001
ARIMA(1,1,0)	-985.69	40.66	0.44	***ar***	0.016
ARIMA(1,1,1)	-985.366	41.19	0.42	***ma***	0.727
				ar	0.880

The *coef P*>|*z*| column indicates the significance of each feature weight. MA parameter of the first model has a *p-value* below 0.05, so it is reasonable to retain it in our model. The same applies to AR parameter of the second model. However, the AR and MA parameters of the third model have *p-values* above 0.05, which indicates that there is room for adjustments, and that retaining both AR and MA parameters may decrease the predictive performance of the model.

One of the most common goodness of fit tests is the Ljung–Box [122] statistics for residual analysis. The Ljung–Box Q test indicates, through its high significance value (>0.05), that the residuals are independent and all three models are suitable and well-adjusted to the time series. Another indicator that can facilitate our selection process is the Akaike information criterion (AIC) [123], which measures the goodness of fit of statistical models for a given set of data. Given a collection of models for the data, AIC estimates the quality of each model, relative to each of the other models. Hence, models that have a better fit will receive a better (lower) AIC score than similar models that fit worse. Both ARIMA(1,1,0) and ARIMA(1,1,1) models have similar AIC scores, with values -985.687 and -985.366 respectively. For ARIMA(0,1,1), the value is slightly lower (better), i.e. -987.523. In general, the three models seem identical and their scores are close. However, the fact that the first model has a slightly better AIC and Ljung–Box Q test score indicates that it is a more appropriate candidate. Therefore, we chose ARIMA(0,1,1) as the most suitable model.

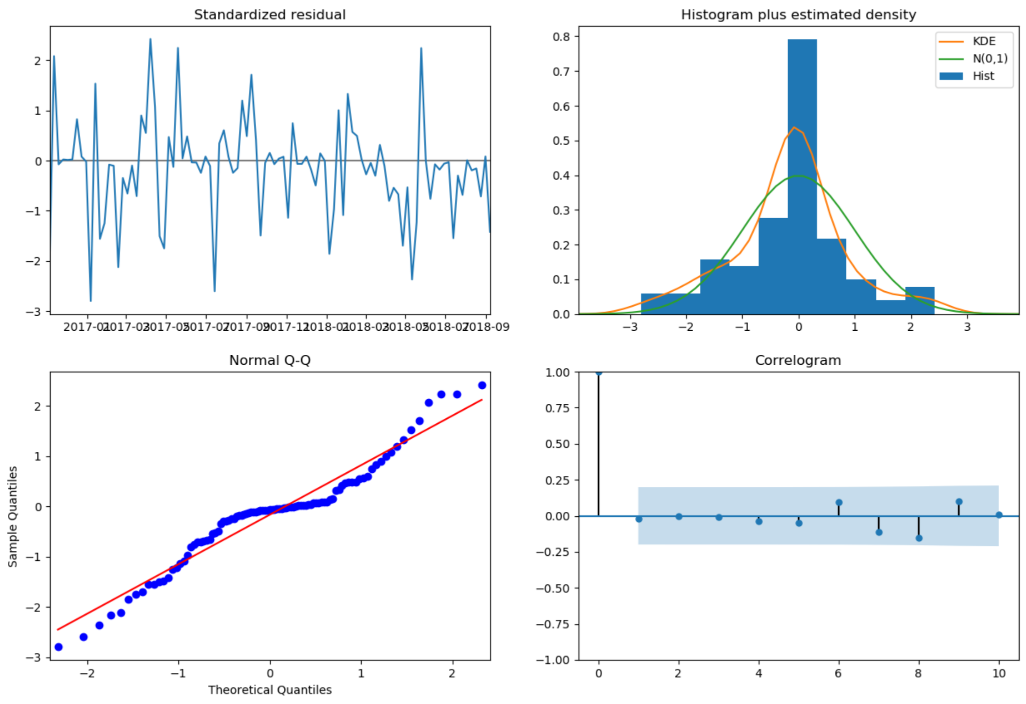


Figure 4: ARIMA(0,1,1) residual analysis

A residual analysis also evaluates the model’s goodness of fit and helps to investigate for any unusual behavior. Figure 4 presents the residual analysis and diagnostics of the ARIMA(0,1,1) model. The primary concern is to ensure that the residuals of the model are uncorrelated and normally distributed with zero-mean. If the ARIMA model does not satisfy these properties, it is a good indication that it can be further improved. In our case, the model diagnostics suggests that the model residuals are normally distributed based on the following:

- In the top right plot, the red kernel density estimation (KDE) line follows the *N(0,1)* line. This is a good indication that the residuals are normally distributed.
- The QQ-plot on the bottom left shows that the ordered distribution of residuals (blue dots) follows the linear trend of the samples taken from a standard normal distribution with *N(0, 1)*. This is also an indication that the residuals are normally distributed.
- The residuals over time (top left plot) do not display any obvious seasonality. This can be further confirmed by the ACF on the bottom right, which shows that the time series residuals have low correlation with lagged versions of itself.

Those observations led us to conclude that the model produces a satisfactory fit that could help forecast future values. The next step is to evaluate the selected model's performance on the dataset.

3.1.4 Application

The fourth and final ARIMA modelling step is Application. During this step, the selected model is optimized for the data it is built from and then tested on observations that have not been used during training to ensure the ability of the model to generalize well. For this purpose, it is important to hold out some observations that can be used to evaluate the predictive power of the model on unseen data. Validation methods extensively used in machine learning, such as k-fold cross-validation, cannot be directly used with time series data due to the temporal order in which values were observed. Hence, observations cannot be randomly split into groups without respecting the temporal order.

To assess prediction accuracy and compare different models we adopted *walk-forward validation* [124], a strategy inspired by k-fold cross-validation. Walk-forward validation is a commonly used way to evaluate time series models' performance, based on the notion that models are updated when new observations are made available. In brief, during walk-forward validation a subset of *n* consecutive points extracted from the original time series is used to train an initial model. Then, accuracy of the model is tested against future time steps and prediction is evaluated against the known value to compute prediction errors. Finally, the time window is moved one-step forward to include the known value into the training set and the process is repeated.

We choose the *n* = 52 (one year) as our sliding training window. Then, we applied three independent walk-forward validation processes, where predictions were made for the next *n*+4 (1 month), *n*+8 (2 months), and *n*+12 (3 months) future steps respectively. In time series analysis, it is common to include a random walk model for the purpose of comparing it with the selected model. The random walk model excludes the auto-regressive (AR) and moving average (MA) parameters. Since the proposed model is ARIMA(0,1,1), the random walk model to be used is ARIMA(0,1,0).

We evaluated forecasts for both, the proposed model as well as the random walk model using *Root Mean Squared Error (RMSE).* The benefit of RMSE is that it penalizes large errors and the scores are in the same units as the forecast values (TD density per week). We also computed *Mean Absolute Percentage Error (MAPE)* as well as *Mean Absolute*

Error (MAE). In Table 5, we report a comparison of prediction errors of both, our selected model as well as the random walk model for multiple (4, 8 and 12) time steps into the future.

Table 5: ARIMA(0,1,1) and Random Walk Model Comparison

Model	**Steps ahead**	**Model Fit statistics**		
		RMSE	***MAPE***	***MAE***
ARIMA(0,1,1)	4	0.010	1.349	0.008
	8	0.017	2.430	0.014
	12	0.025	3.623	0.021
ARIMA(0,1,0)	4	0.011	1.372	0.010
	8	0.018	2.455	0.015
	12	0.025	3.612	0.021

Results indicate that the model is stable over the holdout sample for all steps ahead and that the proposed model outperforms the random walk model for 4 and 8 steps ahead. Of course, the predictive power of this modelling approach decreases as we forecast longer into the future. For 12 steps ahead, RMSE and MAE errors are equal for both models, while the MAPE error of our model is bigger that the MAPE of the random walk.

3.2 Apache SystemML, Square OkHttp, Square Retrofit and Jenkins

3.2.1 Identification

For the rest of the applications, similar to the Apache Kafka case, we performed a *log(ln)* and first-order differencing to eliminate the non-stationarity of the series. The results of applying a Dickey-Fuller test on the initial and first-order differenced time series are presented in Table 6.

Table 6: Dickey-Fuller Tests on First-order Differenced Data

Series	**Test Statistic (Critical Value)**
Apache SystemML	
Initial	-1.043719 (3.475018)
1st difference	-11.31556 (3.475018)
Square OkHttp	
Initial	-1.800630 (-3.456355)
1st difference	-10.03932 (-3.456355)
Square Retrofit	
Initial	-1.426334 (-3.498198)
1st difference	-10.24958 (-3.498198)
Jenkins	
Initial	-1.796879 (-3.475637)
1st difference	-9.991324 (-3.475637)

Table 6 indicates that for every project, taking the first-order difference of the values has made the time series stationary. The Dickey-Fuller test statistic is lower than the critical value so we reject the null hypothesis of unit root. As stated above, the number of required transformations until the series becomes stationary corresponds to the *d* parameter of the ARIMA(*p*,*d*,*q*) model, thus setting the value of *d* = 1 can be safely supported by the above analysis.

3.2.2 Estimation

During this phase of the analysis, the auto-regressive (AR) *p* and moving average (MA) *q* parameters have to be defined. We analyzed the ACF and PACF plots of the four series in accordance with the ARIMA guidelines in Section 2 to accurately identify the *p* and *q* parameters. Based on this analysis, competitive models for each project were identified. These models are presented below:

- *Apache SystemML*: ARIMA(1,1,0), ARIMA(0,1,1) and ARIMA(1,1,1).
- *Square OkHttp*: ARIMA(0,1,2), ARIMA(1,1,2), ARIMA(0,1,1), ARIMA(1,1,0) and ARIMA(1,1,1).
- *Square Retrofit*: ARIMA(1,1,0), ARIMA(0,1,2), ARIMA(0,1,1) and ARIMA(1,1,1).
- *Jenkins*: ARIMA(1,1,0), ARIMA(0,1,2), ARIMA(0,1,1) and ARIMA(1,1,1).

3.2.3 Diagnostic testing

During this phase of the ARIMA analysis, goodness of fit of the selected models as well as the residuals (i.e., the difference between the predicted and the actual values) have to be analyzed for any possible correlations to verify adequacy of these models. A summary of the fitted models for each project is presented in Table 7.

Table 7: ARIMA Candidate Model Results

Model	**Model Results**				
	AIC	***Ljung-Box (Q)***	***Prob(Q)***	***coef P >|z|***	
Apache SystemML					
ARIMA(0,1,1)	**-1265.54**	**25.99**	**0.96**	***ma***	**0.562**
ARIMA(1,1,0)	-1265.49	25.99	0.96	***ar***	0.581
ARIMA(1,1,1)	-1264.89	24.90	0.97	***ma***	0.124
				ar	0.056
Square OkHttp					
ARIMA(0,1,1)	**-1069.41**	**6.89**	**1.00**	***ma***	**0.040**
ARIMA(1,1,0)	-1068.50	9.14	1.00	***ar***	0.626
ARIMA(0,1,2)	-1052.98	8.06	1.00	***ma***	0.504
ARIMA(1,1,2)	-1067.47	6.96	1.00	***ar***	0.899
				ma	0.342
ARIMA(1,1,1)	1067.81	7.96	1.00	***ar***	0.752
				ma	0.709
Square Retrofit					
ARIMA(0,1,1)	**-632.89**	**27.70**	**0.93**	***ma***	**0.035**
ARIMA(1,1,0)	-632.87	27.90	0.93	***ar***	0.926
ARIMA(0,1,2)	-631.93	26.92	0.94	***ma***	0.494
ARIMA(1,1,1)	-631.32	27.10	0.94	***ar***	0.156
				ma	0.099

Jenkins					
ARIMA(0,1,1)	**-1047.50**	**34.84**	**0.70**	***ma***	**0.042**
ARIMA(1,1,0)	-1047.22	35.57	0.67	***ar***	0.103
ARIMA(0,1,2)	-1046.86	29.55	0.89	***ma***	0.116
ARIMA(1,1,1)	-1045.99	33.71	0.75	***ar***	0.854
				ma	0.793

Models with the lowest AIC value are presented in bold. Based on the AIC test score we identified ARIMA(0,1,1) as the best candidate for all projects under analysis. A residual analysis also evaluated that the residuals of the selected model are uncorrelated and normally distributed with zero-mean. Those observations lead us to conclude that ARIMA(0,1,1) produced a satisfactory fit that could help forecast future values for the four projects under analysis.

3.2.4 Application

The fourth and final ARIMA modelling step is Application. Similarly, to the Kafka project, we compared our models to the random walk. We choose the n = 52 (one year) as our sliding training window. Then, we applied three independent walk-forward validation processes, where predictions were made for the next n+4 (1 month), n+8 (2 months), and n+12 (3 months) future steps respectively. In Table 8, a comparison of prediction errors is reported for multiple (4, 8 and 12) time steps into the future for each project.

Table 8: ARIMA(0,1,1) and Random Walk Model Comparison

Model	**Steps ahead**	**Model Fit statistics**		
		RMSE	***MAPE***	***MAE***
Apache SystemML				
ARIMA(0,1,1)	4	0.005	0.476	0.003
	8	0.006	0.740	0.005
	12	0.008	1.024	0.006
ARIMA(0,1,0)	4	0.005	0.482	0.004
	8	0.006	0.741	0.006
	12	0.008	1.016	0.006
Square OkHttp				
ARIMA(0,1,1)	4	0.006	0.998	0.004
	8	0.010	1.502	0.006
	12	0.012	2.074	0.009
ARIMA(0,1,0)	4	0.008	1.348	0.006
	8	0.010	1.784	0.007
	12	0.011	1.921	0.008
Square Retrofit				
ARIMA(0,1,1)	4	0.011	1.215	0.007
	8	0.015	2.082	0.013
	12	0.020	3.000	0.018
ARIMA(0,1,0)	4	0.011	1.223	0.008
	8	0.016	2.096	0.014
	12	0.020	2.948	0.018
Jenkins				

ARIMA(0,1,1)	4	0.009	0.962	0.005
	8	0.012	1.602	0.009
	12	0.016	2.271	0.013
ARIMA(0,1,0)	4	0.011	0.987	0.007
	8	0.012	1.599	0.009
	12	0.016	2.283	0.013

As in the Apache Kafka case, results indicate that the ARIMA(0,1,1) model is stable over the holdout sample for all steps ahead and outperforms the random walk model for 4 and 8 steps ahead. Of course, also in this case it is reasonable to expect that the predictive power decreases as we forecast longer into the future. In the following section, we discuss our results in more details.

3.3 Discussion

Across the five independently developed, maintained, and managed open source projects, a single first-order moving average model with one order of non-seasonal differencing and constant term ARIMA(0,1,1) was shown to fit and forecast the pattern of weekly TD density evolution. The comparison of ARIMA(0,1,1) with the random walk indicates that a more complex model performs better when the forecast horizon is from 4 to 8 weeks. ARIMA model has shown better fitness statistics and higher predictive power compared to the random walk, which leads to the conclusion that the TD patterns can be modeled adequately by time series techniques.

However, when trying to forecast for 12 weeks ahead, the random walk model performs equally or even better in almost all of the cases. This is a reasonable finding as trying to forecast longer into the future, increases the uncertainty of the predictions. The only exception is the Jenkins case, where our model performs better. This finding deserves more investigation and will be the subject of future work.

Furthermore, the fact that the same ARIMA(0,1,1) model fits across the five time series could possibly point out that the pattern of TD is persistent across the five studied projects. This is a very interesting finding indicating that we can sacrifice the benefits of fine-tuning the prediction models to each of the products in favor of the simplicity of always applying the same model. However, the validity of this model assumes that no dramatic changes are made into the evolution process of the projects. In any case, the applicability of this model needs to be investigated on more projects in order to further support this conclusion.

More generally, we conclude that time series analysis is a useful technique for analyzing the evolution of the TD density over a relatively long period. However, these models build on the assumption that reliable historic data are available and that the data is collected in constant time intervals (e.g. daily, weekly, or monthly frequency). Generally, this is hardly the case as sufficient historic data are usually hard to acquire. Moreover, although time series models can yield satisfactorily accuracy, they demand a relatively long history of past data, require frequent re-training on new data, and are difficult to tune properly.

4 Conclusions and Future Work

The research work presented in this chapter examines the usage of time series models as a valid and accurate approach to forecasting TD in long-lived, open-source, software projects. To the best of our knowledge, this is the first time that time series forecasting, and more specifically ARIMA methodology has been employed for this purpose.

For the purpose of our study we obtained 3 years of TD evolution history, computed from the static code analysis of five independently developed, maintained, and managed open-source software systems. Based on this data, we observed that a single time series model, ARIMA(0,1,1), can accurately predict the pattern of software evolution TD for each of five projects. The results shown indicate that ARIMA time series models outperform purely random processes and random walks up to 8 weeks into the future.

However, we have observed that predictive power decreases considerably for longer forecasting horizons. In addition, the fact that sufficient historic data are rarely available and hard to acquire, leads us to the conclusion that we should also consider other forecasting techniques. Machine learning algorithms, compared to classical methods for time series forecasting, include the ability to handle irrelevant features, as well as to support complex relationships and tolerance to noise between variables.

Chapter V Machine Learning Techniques for Technical Debt Forecasting

“The goal of forecasting is not to predict the future but to tell you what you need to know to take meaningful action in the present.”

Paul Saffo - American technology forecaster

Published as:

D. Tsoukalas, D. Kehagias, M. Siavvas, and A. Chatzigeorgiou, “Technical Debt Forecasting: An empirical study on open-source repositories,” in *Journal of Systems and Software,* 2020, vol. 170, p. 110777. doi: 10.1016/j.jss.2020.110777.

Chapter Summary

Technical debt (TD) is commonly used to indicate additional costs caused by quality compromises that can yield short-term benefits in the software development process, but may negatively affect the long-term quality of software products. Predicting the future value of TD could facilitate decision-making tasks regarding software maintenance and assist developers and project managers in taking proactive actions regarding TD repayment. However, no notable contributions exist in the field of TD forecasting, indicating that it is a scarcely investigated field. To this end, in the present chapter, we empirically evaluate the ability of machine learning (ML) methods to model and predict TD evolution. More specifically, an extensive study is conducted, based on a dataset that we constructed by obtaining weekly snapshots of fifteen open source software projects over three years and using two popular static analysis tools to extract software-related metrics that can act as TD predictors. Subsequently, based on the identified TD predictors, a set of TD forecasting models are produced using popular ML algorithms and validated for various forecasting horizons. The results of our analysis indicate that linear Regularization models are able to fit and provide meaningful forecasts of TD evolution for shorter forecasting horizons, while the non-linear Random Forest regression performs better than the linear models for longer forecasting horizons. In most of the cases, the future TD value is captured with a sufficient level of accuracy. These models can be used to facilitate planning for software evolution budget and time allocation. The approach presented in this chapter provides a basis for predictive TD analysis, suitable for projects with a relatively long history. To the best of our knowledge, this is the first study that investigates the feasibility of using ML models for forecasting TD.

1 Introduction

The Technical Debt (TD) notion, a term inspired by the financial debt of economic theory, was introduced in 1992 by Ward Cunningham [1] as a metaphor intended to describe the problem of introducing long-term problems to software products, by not resolving existing quality issues early enough in the overall software development lifecycle (SDLC). The TD metaphor was initially related to software implementation (i.e., at the code level) but was gradually extended to other phases of the SDLC, i.e., software architecture, design, documentation, requirements, and testing [2]. In the same manner like financial debt, TD incurs interest payments in the form of increased future software costs, usually caused by poor design and code quality. To effectively manage the identification, quantification, and repayment of TD during the software development lifecycle, researchers and practitioners have developed and adopted a multitude of theories, methods and tools [5].

However, predicting the accumulated TD during the evolution of a software application is an open and challenging research issue, as both the software system and its TD emerge in parallel [8]. The opportunity to predict TD is of paramount importance to software maintainability, which is recognized as one of the most effort-intense activities in the SDLC [3]. System engineers and project managers need the right tools and appropriate training support to be able to perform long-term effective software maintenance [3]. Therefore, forecasting the evolution of TD could be valuable in assessing the point at which the software product could become unmaintainable and to identify software artifacts, which are prone to accumulate significant levels of TD.

Although the topic of predicting the evolution of various aspects directly or indirectly related to the TD concept, such as code smells [10], fault-proneness [11] and general software evolution trends [12], has attracted the attention of both academia and industry, to the best of our knowledge no studies are focusing on the forecasting of TD itself [13]. Hence, a method or tool that would provide practical decision-making support by predicting future TD of a software system that is expected to evolve over time can be valuable to software development teams. Consequently, software architects and project managers would be able to gain a better understanding of future TD issues and plan well in advance appropriate refactoring activities for saving maintenance costs.

As a first step towards TD forecasting, in our previous work, we have studied and applied statistical time series models for TD Principal forecasting [14]. Statistical time series models are mostly univariate, i.e., they require only the historical data of the variable of interest to forecast its future evolution behavior and have thus been widely used in the literature for predicting software evolution trends, future change requests or software defects [12], [19]–[22]. We used a dataset of 5 real-world open-source Java applications and found that the Autoregressive Integrated Moving Average model ARIMA(0,1,1) can provide accurate TD Principal predictions over a sufficiently long time period for all sampled applications. However, even though the overall ARIMA model performance was satisfactory for short-term TD Principal forecasting (up to 8 weeks ahead), we observed that its predictive performance dropped significantly for long-term predictions. Moreover, we concluded that ARIMA models might prove difficult to tune, as one has to follow the entire Box-Jenkins methodology [23].

The work presented in this chapter is a logical continuation and extension of our previous efforts [14], in order to provide a more complete approach for TD forecasting. More specifically, we believe that we can achieve better scores by trying out more advanced multivariate models able to support feature engineering, i.e., take into account various TD-related features and their combinations to generate better TD predictions. Therefore, while in our previous study [14] the main focus lied on univariate time series forecasting methods, in the present chapter we attempt to empirically evaluate the ability of multivariate Machine Learning (ML) methods to adequately forecast future TD trends of software applications and achieve better and more practical results in both short-and long-term predictions. Building multivariate models that, alongside the evolution of the target variable, learn also from the evolution of additional features related to the target variable is a widely-used strategy [125], [126], since the covariation of time series that follow similar time-based patterns can model interesting interdependencies and therefore improve forecast accuracy. To this end, in this chapter we extend our initial dataset by adding 10 more software applications (15 in total) and investigate whether the combination of software-related metrics acting as TD indicators and already existing ML forecasting methods could lead to the development of novel models that provide predictions about the evolution of TD in a software project. Towards this direction, we have studied and applied various popular ML methods, such as Regression, Regularization, Support Vector Regression, and Regression Trees to forecast the evolution of TD Principal.

The problem that our work attempts to solve can be summarized in the following research question:

> ***RQ:*** *Is the usage of machine learning models on a specific set of Technical Debt indicators a meaningful and accurate approach to forecasting Technical Debt Principal in a long-lived, open-source software?*

The objective of this study is to evaluate the ability of ML methods to model and predict the TD evolution of a software application, based on a set of TD indicators selected as TD predictors. The viewpoint is that of researchers who intend to investigate how different ML approaches can be effectively adopted by project managers and developers to accurately forecast the evolution of TD Principal and thus, support planning and decision-making. The context is an empirical study on TD Principal values of 15 real-world open-source Java applications publicly available in the GitHub repository. The included TD Principal values cover almost 3 years of each application’s evolution, which corresponds to nearly 150 snapshots in weekly intervals. As such, the resulting models are expected to be meaningful in the context of the dataset constructed for this study. A positive answer to the formulated research question will suggest that ML models trained on a selected set of Technical Debt indicators can potentially be used as the basis for the construction of a TD Forecasting tool. We will also investigate the extent to which these models can properly capture the evolution of TD Principal values in terms of accuracy and forecasting length.

To shed light on this question, we conducted an empirical study following the roadmap illustrated in Figure 5. Initially, we studied the relevant literature and identified TD indicators that could act as predictors, such as TD-related features and various Object Oriented (OO) metrics. Afterwards, we constructed a relatively large code repository comprising 15 real-world open-source Java applications retrieved from the GitHub[16] online repository. For each application, we collected a subsequent number of snapshots

[16] https://github.com/

(commits) ranging from 100 to 150 in weekly intervals, spanning up to almost 3 years of each application's evolution. This approach led to a dataset containing 1,850 snapshots in total (171M lines of code). In order to extract the identified TD-related features and various Object Oriented (OO) metrics that could act as predictors, we used two popular tools, namely SonarQube[17] and CKJM Extended[18] respectively. This process led to 15 independent application-specific datasets containing TD indicators and TD values for each snapshot. Subsequently, we employed techniques like correlation analysis, univariate and multivariate analysis, which allowed us to select the most statistically significant TD predictors and thus retain as much discriminatory information as possible. Finally, we examined potential ML forecasting models and algorithms that could be applied for TD prediction and compared their accuracy for various forecasting horizons in order to reach safer conclusions regarding the significance of the observed results. To the best of our knowledge, this is the first study in the field of TD that examines the applicability of ML models for TD forecasting.

An overview of the methodology described above is presented in Figure 5.

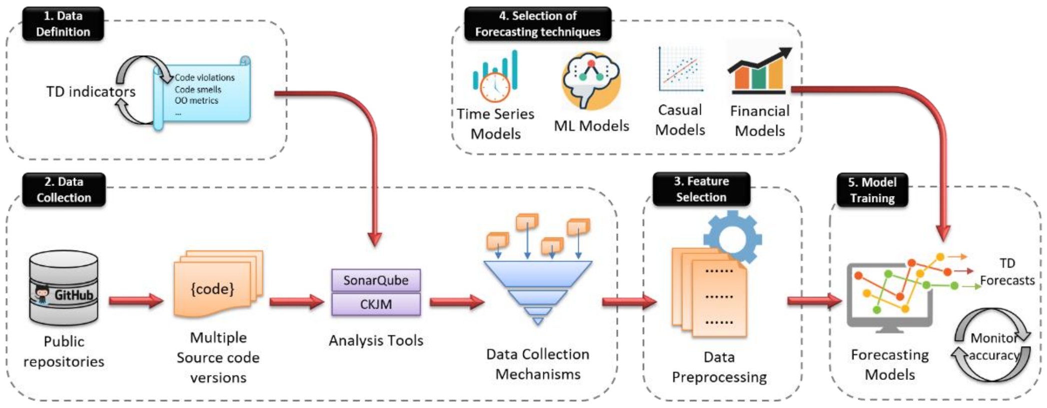


Figure 5: Chapter V roadmap

The meaningfulness of any forecasting model is also related to its ability of reflecting the developers' perspective on whether the modeled phenomena and predicted evolution are useful. To this end, we have performed a survey to empirically evaluate the meaningfulness of the TD forecasting approach introduced in this study and to investigate the usefulness of the TD Forecasting concept in general, via a questionnaire distributed to representatives of a software company.

The rest of the chapter is structured as follows: Section 2 thoroughly describes data definition, collection and pre-processing steps. Section 3 describes forecasting model training, testing and benchmarking, as well as the current state of technical implementation of the proposed approach. Section 4 presents the results of a survey that has been conducted to empirically evaluate the meaningfulness of the TD forecasting approach introduced in this study. Section 5 reports the limitations and validity threats of this empirical study,

[17] https://www.sonarqube.org/

[18] http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

while section 6 discusses significant implications for both research and practice. Finally, Section 7 concludes the chapter and discusses ideas for future work.

2 Data Definition, Collection and Preparation

For the execution of this study, we aimed at combining different TD-related features and metrics into a common dataset (source triangulation) with the purpose of investigating if and to what extent multivariate ML models can be used in order to accurately predict the TD evolution of software applications. This section describes in detail the definition, collection and pre-processing of the dataset that was used later as input by the produced TD forecasting models. As a first step towards creating the TD-related dataset, we studied the literature and selected an initial set of TD indicators. As soon as the appropriate TD indicators were selected, we downloaded multiple consecutive snapshots (commits) of 15 open-source projects and then used the SonarQube[19] and CKJM Extended[20] tools to extract these indicators, along with the TD Principal value of each snapshot. Once the data collection step had finished, we performed data pre-processing on the collected data. Techniques such as descriptive statistics, correlation analysis, and feature selection were applied on the dataset to prepare it as an input for forecasting models. Finally, we restructured each application specific dataset to a format that can be used as input to the forecasting models. In what follows, the above procedure is presented in detail.

2.1 TD Indicator Definition

As discussed in Section 1.4 of Chapter II, OO metrics, code smells, issues extracted from ASA tools, and software quality metrics extracted from quality assessment tools have been widely used in the literature as indicators able to monitor and quantify TD and the quality of software maintainability in general. In the approach presented in this chapter, we treat these indicators coming from different sources as potential TD predictors (source triangulation) and combine them with already existing forecasting methods to develop novel models that provide predictions about the future evolution of TD in a software application. Two of the most popular and widely used tools for calculating such TD indicators are SonarQube and CKJM Extended. SonarQube is an open source platform for continuous inspection of code quality that provides analysis functionalities and a wide range of metrics for measuring quality attributes of code, tests, and design. As of today, it has been adopted by more than 120K organizations[21] including nearly more than 100K public open-source projects[22]. In this study, SonarQube has been used as proof of concept for research purposes, since according to two recent studies on Technical Debt Management [3], [5], it is the most frequently used tool for estimating TD principal. In addition, another reason for selecting this tool is the fact that it is highly customizable,

[19] https://www.sonarqube.org/

[20] http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

[21] https://www.sonarqube.org/

[22] https://sonarcloud.io/explore/projects

allowing the users to adjust the standard out-of-the-box set of rules (named “sonar way”) that it provides in order to better meet their needs. In a relevant study [127], the authors suggest that companies should continuously re-consider the adopted SonarQube rules based on business’s objectives and preferences. In a similar way, developers and users of the TD Forecasting tool described in this study could fine-tune the rule-set of SonarQube prior to obtaining TD-related measurements, so that predictions can be tailored based on the company's critical needs.

Therefore, in the present work, we opted for the TD-related metrics[23] that are provided by SonarQube, as our primary TD Principal predictors. The version of SonarQube used within the context of this work is 6.7.4. Furthermore, SonarQube was also used to compute the target variable, i.e., to quantify the TD Principal of the selected software applications. To do so, SonarQube checks code compliance against a set of classified coding rules and if the code violates any of these rules, it considers it as a violation or a TD item. For each of the identified TD items, SonarQube computes the remediation time (i.e., estimated effort) needed to refactor it and considers it as TD.

To complement the TD predictor set we decided to account also for the popular Chidamber and Kemerer (C&K) metrics and Quality Model for Object Oriented Design (QMOOD) metrics [41], [42]. The reason behind this choice is that C&K metrics, such as *DIT*, *NOC*, *RFC*, *LCOM* and *WMC*, and QMOOD metrics, such as *DAM*, *MOA*, and *CAM* have been intensively studied for their ability to predict maintainability and maintenance effort [43], [44] (see “Studies” column in Table 9). One of the main limitations of SonarQube tool is the lack of OO detection mechanisms. Therefore, to collect OO metrics for our applications, we chose the popular CKJM Extended [128], an extended version of the CKJM open-source tool able to calculate a wide range of metrics[24] (including those defined in C&K and QMOOD suites), by processing the bytecode of Java files. Indicators extracted by CKJM Extended can be calculated at the source-code level, and can be used to assess well-known quality properties associated with the architecture of a software application, such as complexity, coupling, cohesion, and inheritance among others. In addition, CKJM Extended calculates C&K metrics strictly according to the original (1994) definition by Chidamber and Kemerer.

In Table 9, the metrics that were selected as TD indicators and therefore used as independent variables for the creation of our dataset are presented along with a short description. Moreover, to strengthen the TD indicators selection, we provide references to studies that relate each metric with TD and the quality of software maintainability in general. The first half of the table describes metrics computed by SonarQube, while the last half describes metrics extracted by CKJM Extended. The target variable, i.e., the variable that we want to forecast, is denoted here as *total_principal.* We define *total_principal* as the effort (in minutes) to fix all issues and we compute it as the sum of code smell, bug, and vulnerability remediation effort.

[23] https://docs.sonarqube.org/latest/user-guide/metric-definitions/

[24] http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/metric.html

Table 9: TD indicators

Metric	***Description***	***Studies***
Project-level metrics (computed by SonarQube)		
Technical Debt Metrics		
sqale_index	Effort to fix all code smell issues. The measure is stored in minutes.	
reliability_remediation_effort	Effort to fix all bug issues. The measure is stored in minutes.	
security_remediation_effort	Effort to fix all vulnerability issues. The measure is stored in minutes.	
total_principal	Effort to fix all issues. The sum of the three metrics mentioned above, i.e., code smell, bug and vulnerability remediation effort. The measure is stored in minutes.	
Reliability Metrics		
bugs	Total number of bug issues of a project.	[129] [130] [8]
Security Metrics		
vulnerabilities	Total number of vulnerability issues of a project.	[130] [8] [131]
Maintainability Metrics		
code_smells	Total number of code smell issues of a project.	[8] [26] [132] [133] [134] [135] [136]
Size metrics		
comment_lines	Number of lines containing either comment or commented-out code of a project.	[130] [137]
ncloc	Number of physical lines of a project that contain at least one character, which is neither a whitespace nor a tabulation nor part of a comment.	[8] [136] [138] [139] [96]
Coverage Metrics		
uncovered_lines	Number of lines of code of a project, which are not covered by unit tests.	[130]
Duplication Metrics		
duplicated_blocks	Number of duplicated blocks of lines of a project.	[130] [29] [54]
Complexity Metrics		
complexity	The Cyclomatic Complexity of a project calculated based on the number of paths through the code.	[140] [141] [142]
Class-level metrics aggregated at project-level (computed by CKJM Extended)		
Complexity Metrics		
AMC	Average Method Complexity: The average method size for each class (number of java binary codes in the method), averaged for all project classes.	[137] [143]
WMC	Weighted Methods per Class: The total number of methods that a class contains weighted by their complexity values, averaged for all project classes.	[139] [140] [142] [93] [144] [94] [96]
DIT	Depth of Inheritance Tree: The depth of inheritance tree for each class from the object hierarchy top, averaged for all project classes.	[140] [142] [93] [144] [94] [145] [96]

NOC	Number of Children: The number of immediate descendants (i.e., children) of a class, averaged for all project classes.	[140] [141] [144] [94] [145] [96]
RFC	Response for a Class: The number of local methods plus the number of methods called by class methods, averaged for all project classes.	[139] [140] [141] [142] [93] [144] [94] [145]
Coupling Metrics		
CBO	Coupling Between Objects: The total number of classes coupled to a given class, averaged for all project classes.	[139] [140] [142] [144] [145]
Ca	Afferent Coupling: The total number of other classes that call methods of the given class, averaged for all project classes.	[137] [145] [146]
Ce	Efferent Coupling: The total number of other classes that their methods are called by the given class, averaged for all project classes.	[137] [145] [146]
CBM	Coupling Between Methods: The total number of parent classes to which a given class is coupled, averaged for all project classes.	[139] [140] [142]
IC	Inheritance Coupling: The total number of new or redefined methods of a class to which all its inherited methods are coupled, averaged for all project classes.	[139] [140]
Cohesion Metrics		
LCOM	Lack of Cohesion in Methods: The number of methods pairs in a class that are not interrelated through the sharing of some of the class fields, averaged for all project classes.	[139] [140] [93] [144] [94] [145] [96]
LCOM3	Lack of Cohesion in Methods: Similar to LCOM but ranging from 0 to 2.	[142] [145]
CAM	Cohesion Among Methods: This metric computes the relatedness among methods of a class based on their parameter lists, averaged for all project classes (Range 0 to 1).	[139]
Other Metrics		
NPM	Number of Public Methods: The total number of methods in a class that are declared as public, averaged for all project classes.	[141] [142] [93] [94] [96]
DAM	Data Access Metric: The ratio of the number of private or protected fields to the total number of fields declared in the class, averaged for all project classes (Range 0 to 1).	[147] [148]
MOA	Measure of Aggregation: The total number of data declarations (class fields) whose types are user defined classes, averaged for all project classes.	[147] [148]

At this point, it should be noted that an obvious choice of related metrics to be included as independent variables in the multivariate models that we investigate in this work are the constituent components of TD Principal (i.e., total_principal) as computed by SonarQube, i.e., the code smells, bugs, and vulnerabilities. However, apart from the TD Principal constituent components that have an evident effect on the target variable itself, we decided to investigate also other, not directly related metrics (presented in Table 9), which however are known to act as TD indicators, in order to examine whether they have an equally (or more) significant impact on TD Principal. To this end, the extensive feature selection analysis reported in Section 2.3 will allow us to come up with the best predictors set, tailored to our dataset and experimental setup.

2.2 Collection of Data

To start the dataset construction process, we initially selected 15 popular open-source applications from the GitHub[25] repository. The selected 15 applications have different sizes and belong to different application domains, which range from Networking Software (e.g., Kafka, Dubbo, OKHttp, Retrofit, Openfire, WebSocket) to Business Software (e.g., OFBiz), and from Scientific Software (e.g., SystemML) to Utilities Software (e.g., Commons IO, Guava, Jenkins, ZXing). The selection criteria were based on the software popularity, activity level, data availability, and the Java programming language. More specifically, we exploited the advanced search mechanism provided by GitHub by selecting only Java projects in the “Languages” filter, and then sorting the results based on “Most Stars” filter. This resulted to an initial set of Java applications ranked by their popularity. Next, to assess the activity level of each application, we used the “Insights” GitHub functionality to examine the total number of commits and compare it to the lifespan of the application. We selected only applications whose commit activity was frequent (at least once per week) and long-lived (at least 3 years). Moreover, since SonarQube and CKJM Extended both work on compiled classes to compute metrics values, selected projects needed to be compilable, i.e., not producing any errors during compiling.

For each application that met the above criteria, approximately 150 snapshots (commits) in weekly intervals were fetched, spanning up to 3 years of each project’s evolution. More precisely, we opted for the last commit in every analyzed week as the time point of analysis. The rationale behind this option, i.e., to ensure fixed and weekly time intervals between the commits is twofold. First, ensuring fixed time distance between the retrieved samples (i.e., commits) is critical for the reliability of the produced forecasting models. Secondly, collecting snapshots at weekly rather than daily intervals is a more viable solution as rarely do projects keep daily commits. In addition to this, another reason that led us to the decision to take snapshots at weekly rather than daily intervals or even to analyze all consecutive commits, is to avoid as many periods of inactivity as possible, but not eliminate them. Periods of inactivity are consecutive snapshots in which no significant new changes were committed to a codebase. While including many such snapshots could potentially introduce a high level of noise in the dataset, including a reasonable number of ‘low or no activity’ periods is important for an accurate model: Generating forecasts that (potentially) indicate future periods of inactivity could prove useful in practice for project managers in decision-making activities. Choosing longer intervals (e.g., monthly) would probably reduce the periods of inactivity even further, but it would result in significantly fewer data and thus significantly lower forecasting performance, whereas the produced models would be able to provide forecasts only at a monthly basis and not for shorter

[25] https://github.com

periods (e.g., some weeks ahead), which would restrict their practicality in decision-making.

The approach described above led to a codebase containing up to 1850 snapshots in total (171M lines of code). A sufficiently high number of applications is fundamental to reach a conclusion that does not depend on a specific dataset, allowing to generalize the obtained results. For the purpose of constructing the dataset, a dedicated crawler was created. In order to facilitate the reproducibility and the extensibility of the present study, as well as the construction of similar datasets, this crawler has been made available online[26]. In Table 10, the applications that were selected for constructing the codebase are presented in detail.

[26] https://sites.google.com/view/technical-debt-forecasting/main

Table 10: Applications of the TD dataset

Application name	*Analyzed weekly snapshots*		*Last snapshot LOC*	*Total commits*	*GitHub contributors*	*GitHub stars*	*Description*
	#	*Timeframe*					
apache/kafka	150	30/10/2015 - 07/09/2018	116.000	7.055	621	14.8k	Kafka is a platform used for building real-time data pipelines and streaming apps.
apache/commons-io	150	11/09/2015 - 27/07/2018	30.000	2.303	53	657	Commons IO library contains utility classes, stream implementations, file filters, file comparators, and much more.
apache/ofbiz	100	04/11/2016 - 28/09/2018	243.000	24.427	20	678	OFBiz is an ERP system that houses a large set of libraries, entities, services and features to run all aspects of your business.
apache/systemml	150	02/10/2015 - 10/08/2018	200.000	6.039	59	798	SystemML provides an optimal workplace for machine learning using big data.
apache/groovy	150	25/12/2015 - 02/11/2018	210.000	16.806	290	3.6k	Groovy is a powerful, dynamic language, with static-typing and static compilation capabilities for the Java platform.
apache/nifi	100	04/11/2016 - 28/09/2018	289.000	5.599	280	1.9k	NiFi supports powerful and scalable directed graphs of data routing, transformation, and system mediation logic.
apache/incubator-dubbo	100	28/07/2017 - 01/02/2019	67.000	4.139	279	20.3k	Dubbo (incubating) is a high-performance, Java based open source RPC framework.
google/guava	150	25/12/2015 - 02/11/2018	114.000	5.190	219	35.8k	Guava is a set of core libraries that includes graphs, APIs/utilities for concurrency, I/O, hashing, string processing, and much more!
square/okhttp	150	18/12/2015 - 26/10/2018	24.000	4.438	198	35.8k	OKHttp is an HTTP & HTTP/2 client for Android and Java applications.
square/retrofit	100	27/01/2017 - 21/12/2018	7.900	1.782	131	34.8k	Retrofit is a type-safe HTTP client for Android and Java by Square, Inc.

jenkinsci/jenkins	150	25/03/2016 - 01/02/2019	147.000	29.265	599	14.8k	Jenkins is the leading open-source development workflow automation server.
spring-projects/spring-boot	100	04/11/2016 - 28/09/2018	15.000	25.004	646	28.6k	Spring Boot makes it easy to create Spring-powered, production-grade applications and services with absolute minimum fuss.
TooTallNate/Java-WebSocket	100	17/03/2017 - 25/01/2019	5.200	954	62	1.9k	WebSocket server and client implementation written in Java.
zxing/zxing	100	10/03/2017 - 01/02/2019	29.000	3.524	96	24.6k	ZXing is an open-source, multi-format 1D/2D barcode image-processing library implemented in Java.
igniterealtime/Openfire	100	18/11/2016 - 12/10/2018	100.000	9.214	114	2.1k	Openfire is a real time collaboration (RTC) server that uses the only widely adopted open protocol for instant messaging, XMPP.

After fetching the source code of each snapshot for the 15 selected applications, we proceeded to the next step, i.e., using SonarQube and CKJM Extended (as described in Section 2.1) in order to analyze each snapshot and build 15 application-specific datasets consisting of the TD indicators described in Table 9. We chose the format of each application-specific dataset to be the following: each row contains a specific snapshot of the application in chronological order (time series), whereas the columns contain the values of the TD indicators, plus one column containing the value of total TD principal for that particular snapshot. This format helped us also during the forecasting model construction phase described later.

Since the work presented in this chapter aims at modelling TD evolution of the entire software project (system), rather than predicting the TD of individual software artefacts (e.g., classes), we performed data collection for each application at the system level. In other words, each application snapshot (commit) provided a single observation in the application-specific dataset. TD-related metrics extracted by SonarQube analysis are computed at system-level by default, so no further modifications were needed. However, most of the metrics extracted by CKJM Extended, such as *DIT*, *RFC*, and *NOC*, are originally defined at the class level. Therefore, those metrics could not be directly used as independent variables. For this purpose, we aggregated CKJM metrics at system-level, i.e., we used their weighted mean among classes. More specifically, in our approach the system-level value of each metric is the aggregation of its class-level values weighted by the lines of code of each class, divided by the total lines of code of the system under analysis. This aggregation approach has been used in relevant studies by Wagner et al. [149] and Bagen et al. [150], but also in some of our previous studies as well [151], [152].

2.3 Data Preparation

Selection of independent (input) variables is a critical part in the design of a ML algorithm. Each additional input unit adds another dimension and contributes to the "curse of dimensionality" [153], a phenomenon in which performance degrades as the number of inputs increases. Furthermore, irrelevant or partially relevant features can negatively impact model performance. Thus, after constructing our dataset, the next step is to provide a clear understanding of the statistical attributes of our variables, and then to reduce the number of input variables described in Table 9 by keeping only the most important ones, i.e., the ones that are highly significant for TD Principal forecasting. Techniques like correlation analysis, univariate and multivariate analysis will allow us to retain as much discriminatory information as possible.

In order to study the statistical significance of each indicator over the TD quality and be able to safely perform dimensionality reduction of our dataset, also known as feature selection, we need to maximize diversity and representativeness by considering a comparable number of different heterogeneous applications. The dataset we constructed and described in Section 2.2 for forecasting purposes consists of 15 applications, a number that may not be suitable for generalizing our findings and reach to a generic conclusion regarding feature selection. Therefore, to increase the size of our dataset for feature selection purposes, in addition to those applications, we have also exploited a benchmark repository that consists of the 100 most popular Java libraries (e.g., Junit, Xerces, HyperSQL, etc.) retrieved from the Maven Repository[27]. The same dataset was used in the study by Siavvas et al. [151] for calibrating a Quality Assessment Model, as well as in a similar study [152] for investigating the interrelationship of software metrics and specific vulnerability types. For the purpose of this work, we further extended the repository by adding 110 more Maven applications, based on their popularity. As a result, the final benchmark repository contains 210 open-source software Java applications, comprising approximately 30 million lines of code, which is considered an adequate number for the purpose of identifying most significant TD indicators. To extract all required indicators from the extended dataset, as in the case of the initial 15 applications, we analyzed the source code of each application using SonarQube and CKJM Extended, as described in Section 2.2. Finally, we merged these metrics with the metrics obtained from the last snapshot of each of our analyzed applications presented in Table 10, leading to an extended dataset containing metrics from a total of 225 applications. We chose to add only the last snapshot of the analyzed applications presented in Table 10 in order to ensure equal representativeness, since each software application in the benchmark repository is represented by only one commit. Furthermore, we assumed that the last snapshot of each project would be more mature and bigger in size compared to its previous snapshots. This extended dataset can be found online[28].

At this point, it should be noted that the extended dataset was used only for correlation analysis and feature selection purposes, as the additional 210 applications obtained from the benchmark repository do not contain project history (past commits), and therefore, are not suitable for forecasting model experiments. After feature analysis and selection, we switched back to our original dataset, containing 15 applications with their commit history (1850 commits in total) for forecasting model training.

2.3.1 Descriptive Statistics

Descriptive statistics is the term given to the analysis of data that helps describe data in a meaningful way, allowing for simpler interpretation. It provides simple summaries about the sample and about the observations that have been made. Descriptive statistics include measures of central tendency, such as the mean, median, and mode, and measures of variability, such as standard deviation, variance, the minimum and maximum variables, and the kurtosis and skewness. After extracting the metrics of each application (using SonarQube and CKJM Extended) and merging them into a common dataset as described in the previous section, the descriptive statistics of TD indicators calculated based on the extended dataset are presented in Table 11. As a reminder, the extended dataset comprises a superset of the dataset that was constructed for forecasting purposes (i.e., the 15 applications presented in Table 10) and the additional 210 applications that were added at a later stage for feature selection purposes. Therefore, for the computation of the statistical metrics presented in Table 11 we have included all 225 applications. For the conduction

[27] https://mvnrepository.com/

[28] https://sites.google.com/view/technical-debt-forecasting/main

of our experiments, we used the Python programming language and more specifically the Pandas[29] data analysis library.

Table 11: Descriptive statistics of TD indicators (extended dataset)

Metric	*Mean value*	*Standard deviation*	*Min value*	*Lower quartile*	*Median value*	*Upper quartile*	*Max value*	*Skewness*	*Kurtosis*
bugs	29.67	61.15	0	3	9	30	585	5.29	37.60
vulnerabilities	38.82	125.94	0	1	7	32	1598	9.68	114.51
code_smells	858.66	2095.09	2	94	235	732	16442	5.55	35.26
comment_lines	5113.78	15587.26	1	387	1381	3889	170698	8.20	77.66
ncloc	19524.28	37253.22	175	2655	7068	20438	357664	5.26	38.09
uncovered_lines	9815.51	17856.10	32	1302	3297	10288	137326	3.91	19.21
duplicated_blocks	110.18	429.33	0	0	12	47	4219	7.77	67.18
complexity	3914.02	7420.96	11	421	1352	4284	61862	4.28	23.54
AMC	53.37	74.38	9.407	25.374	36.565	52.876	819.191	6.70	58.29
WMC	36.40	82.09	2.085	12.638	18.921	31.524	814.007	7.14	57.32
DIT	0.76	0.31	0	0.655	0.813	0.964	1.523	-0.94	0.98
NOC	0.53	1.90	0	0.080	0.261	0.481	26.770	12.83	176.92
RFC	71.45	90.00	11.618	39.508	54.295	78.112	993.299	7.63	69.34
CBO	17.18	21.07	0	9.056	13.554	18.983	245.469	7.14	68.78
Ca	6.24	8.15	0	2.463	3.850	7.399	73.761	4.98	31.90
Ce	12.43	18.10	0	5.922	9.664	13.024	210.225	7.71	73.96
CBM	0.14	0.23	0	0.026	0.085	0.164	2.340	5.37	41.71
IC	0.10	0.11	0	0.025	0.071	0.137	0.595	2.05	5.45
LCOM	442.53	1552.82	-12160	61.584	153.151	503.916	13693.182	0.89	46.39
LCOM3	0.86	0.15	0.435	0.776	0.840	0.916	1.648	1.38	5.06
CAM	0.34	0.08	0.050	0.278	0.333	0.377	0.683	0.86	3.01
NPM	23.25	51.91	0.250	8.009	12.287	21.058	655.642	9.55	109.34
DAM	0.66	0.18	0.024	0.574	0.682	0.779	1.000	-0.95	1.53
MOA	3.58	7.85	0	0.761	1.591	3.263	81.424	6.70	56.12

Metrics that vary little are not likely to be useful predictors. In our case, from Table 11 we observed that for all metrics there are significant differences between the lower 25th (lower) percentile, the median, and the 75th (upper) percentile, thus showing strong variations. Therefore, all metrics were selected to be used for subsequent analysis. We also observed that, as it is the case with software engineering data [41], most of our metrics are highly skewed, which means that few outlier observations may substantially affect the results, if not treated carefully. To mitigate this risk, in the rest of the analysis that follows we opted for techniques that perform well when the distribution of values in the feature space cannot be assumed.

In Figure 6, histograms of each metric are presented to further complement our initial analysis. We used histograms to further examine the normality and skewness of each

[29] https://pandas.pydata.org/

metric. A normal distribution is symmetric and bell-shaped. We observed that most of the metrics are not normally distributed.

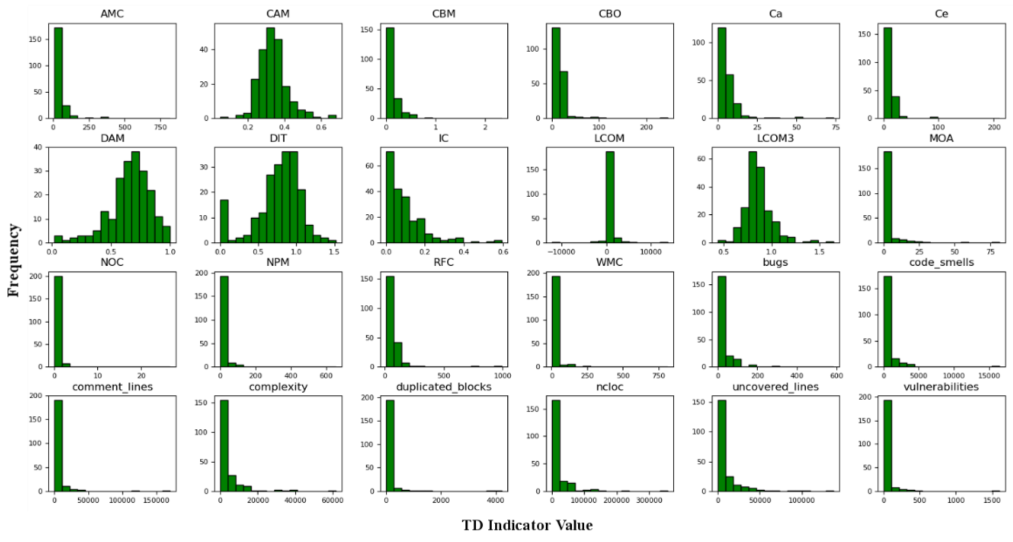


Figure 6: Histograms of TD indicators (extended dataset)

To further validate this finding we used boxplots, which can be found online[30] as supporting material. Boxplots provide a standardized way of displaying the distribution of data. These graphs divide the dataset into a five-number summary: the minimum, first quartile (Q1), median, third quartile (Q3), and maximum. Outlier observations are often easy to identify by inspecting a boxplot, since they are plotted as individual points lying outside the boundaries set by the minimum and maximum values. We observed that some of the metrics seem to have outlier observations. We further investigated these findings during the univariate (one variable outlier) and multivariate (two or more variable outlier) analysis described below.

2.3.2 Correlation analysis

As previously stated, performance of a ML algorithm degrades as the number of inputs increases. Irrelevant or partially relevant features can negatively affect model performance. In order to successfully reduce the number of input variables by keeping only the most important ones, we first applied Spearman's rank correlation coefficient (ρ) [154] analysis between TD Principal as computed by SonarQube and each TD indicator described in Table 11 for the 225 applications. Spearman's rank correlation was selected, as it is a nonparametric test that is not sensitive to outliers. Additionally, it does not assume any distribution for the studied data, which is important in our case, as our data did not seem to follow any known distribution. To interpret the strength of the correlations, Coehen et al. [155] suggestion was used. According to Coehen et al., a correlation less than 0.3 is considered weak, between 0.3 and 0.5 is considered moderate, and above 0.5 is considered strong. Finally, to ensure that the observed associations did not occur by chance, the correlations were tested for statistical significance. For this purpose, the *p-value* of each

[30] https://sites.google.com/view/technical-debt-forecasting/main

correlation was examined. A *p-value* of 0.05 means that we are 95% confident that the observed association has not occurred by chance. Hence, we examined the statistical significance of the observed correlations at 95% level of confidence.

In Figure 7, the correlation between each metric is illustrated based on color warmness, i.e., the more red a box, the higher the correlation between the corresponding metrics. We focused on the last row, which represents the correlation between our dependent variable, i.e., TD Principal and each independent variable, i.e., TD indicators.

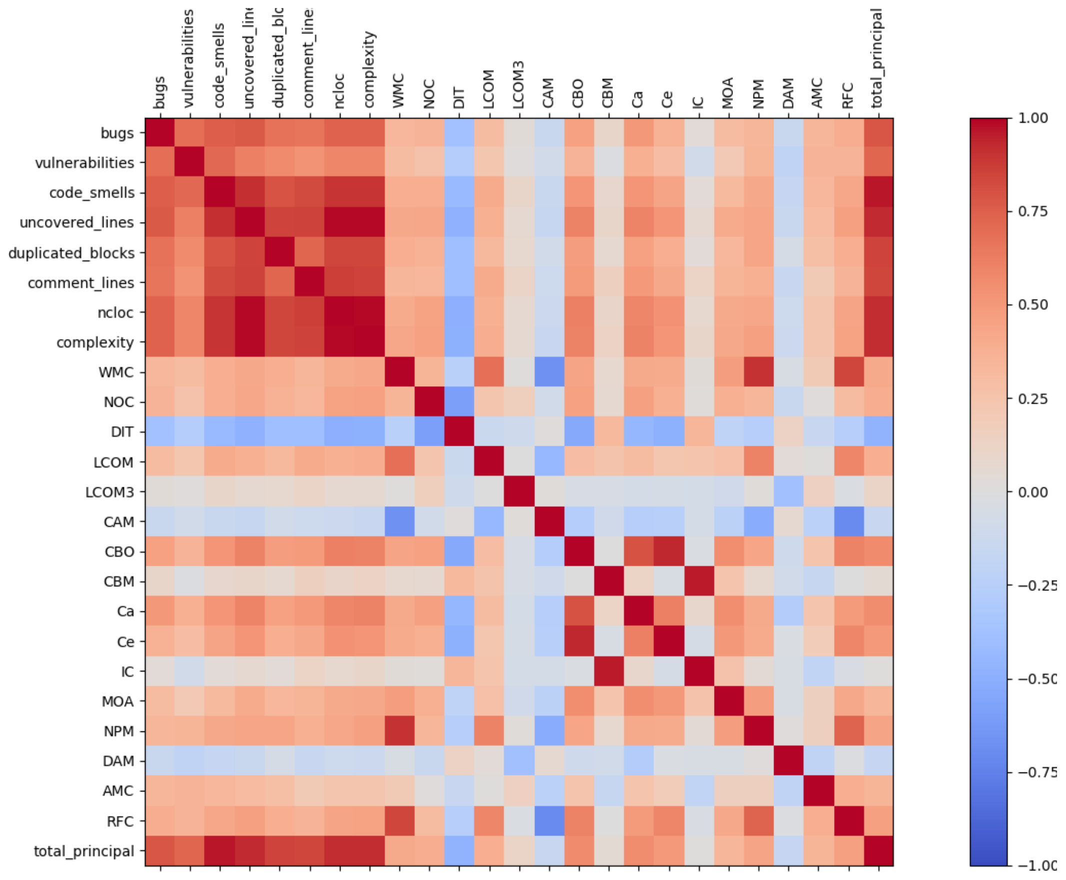


Figure 7: Spearman's rank correlation of TD indicators (extended dataset)

To further complement Spearman's correlation analysis, the correlations between TD and each TD indicator, as well as the significance (*p-value)* of each correlation are presented in numbers in Table 12. To facilitate the readability of the correlation table, asterisk (*) symbols are used to denote the strength of each correlation based on the Coehen's et al. suggestion described above. In particular, the values marked with one (*), two (**), and three (***) asterisks correspond to weak, medium and strong correlations respectively. In addition, statistically significant *p-values* (*p* =< 0.05) are marked in bold, while not statistically significant *p-values* (*p* > 0.05) are in regular font.

Table 12: Spearman's rank correlation of TD indicators (extended dataset)

Metric	*Correlation with Total Principal*	*p-value*
Project-level metrics (computed by SonarQube)		
Reliability Metrics		
bugs	0.784***	**8.10023e-45**
Security Metrics		
vulnerabilities	0.722***	**5.25492e-35**
Maintainability Metrics		
code_smells	0.962***	**1.0044e-118**
Size metrics		
comment_lines	0.838***	**3.77265e-91**
ncloc	0.917***	**2.20628e-58**
Coverage Metrics		
uncovered_lines	0.929***	**2.45155e-56**
Duplication Metrics		
duplicated_blocks	0.846***	**1.23839e-84**
Complexity Metrics		
complexity	0.915***	**2.35973e-83**
Class-level metrics aggregated at project-level (computed by CKJM)		
Complexity Metrics		
AMC	0.349**	**8.79842e-10**
WMC	0.408**	**2.83379e-09**
DIT	-0.471**	**6.05013e-13**
NOC	0.396**	**8.26175e-09**
RFC	0.463**	0.106134
Coupling Metrics		
CBO	0.568***	**0.0376558**
Ca	0.562***	**2.88947e-19**
Ce	0.492**	0.442171
CBM	0.053*	**7.94597e-19**
IC	0.003*	**3.53632e-14**
Cohesion Metrics		
LCOM	0.385**	0.958871
LCOM3	0.112*	**6.03566e-07**
CAM	-0.144*	**2.4988e-11**
Other Metrics		
NPM	0.440**	**0.015035**
DAM	-0.168*	**2.09667e-07**
MOA	0.337**	**1.66596e-12**

where (*): weak correlation, (**): medium correlation, (***): strong correlation

As a first step towards feature selection, metrics that have either a low correlation score (i.e., correlation < 0.3), or a non-significant statistical correlation (i.e., *p-value > 0.05)* with respect to TD were marked as candidates for removal. In particular, five metrics were identified as non-correlated (i.e., *CBM*, *IC*, *LCOM3*, *CAM*, *DAM*), while 3 metrics had statistically insignificant correlations (i.e., *RFC*, *Ce*, *LCOM*) with respect to TD and consequently were filtered out, leaving 16 out of 24 TD indicators for further analysis.

2.3.3 Univariate Analysis

After applying the correlation analysis as the first filter towards feature selection, we considered to apply a univariate regression analysis between each remaining metric (TD indicator) and the TD for the extended dataset (225 applications). The importance of controlling for potential confounders in empirical studies of object-oriented products has been emphasized in the study by el Emam et al. [156]. Univariate regression focuses on determining the relationship between one independent variable (i.e., each metric) and the dependent variable (i.e., TD Principal) and has been widely used in software engineering studies to examine the effect of each metric separately [11], [143], [156]. Thus, we used this method as a second filter, to help us with the process of removing metrics whose underlying relationship is not statistically significant to TD. During descriptive statistics however, we observed that most of the metrics were highly skewed. In order to render the data suitable for univariate regression analysis we applied the natural logarithm *log(ln)* transformation to the values of the remaining metrics [61]. Using the natural logarithm reduces the skew of the response and predictors (linear regression assumptions include normal distribution of the residuals).

Table 13 summarizes the results of the univariate linear regression analysis for each metric, applied on the extended dataset (225 applications). Column “R2” gives the coefficient of determination, i.e., the proportion of the total variation in the dependent variable that is explained by the model. Columns “p-value”, “Standard error”, and “Relationship” show the statistical significance, the standard error, and the sign of the regression coefficient for the independent variable, respectively.

Table 13: Univariate analysis results of TD indicators (extended dataset)

Metric	*R2*	*p-value*	*Standard error*	*Relationship*
bugs	0.625	**0.000**	0.048	+
vulnerabilities	0.528	**0.000**	0.045	+
code_smells	0.931	**0.000**	0.019	+
comment_lines	0.685	**0.000**	0.036	+
ncloc	0.823	**0.000**	0.033	+
uncovered_lines	0.847	**0.000**	0.029	+
duplicated_blocks	0.692	**0.000**	0.031	+
complexity	0.812	**0.000**	0.032	+
AMC	0.107	**0.000**	0.164	+
WMC	0.131	**0.000**	0.131	+
DIT	0.143	**0.000**	0.518	-
NOC	0.037	**0.005**	0.311	+
CBO	0.298	**0.000**	0.136	+
Ca	0.317	**0.000**	0.132	+
NPM	0.160	**0.000**	0.127	+
MOA	0.098	**0.000**	0.141	+

We set the significance level at $\alpha = 0.05$. Metrics with *p-values* lower than 0.05 (*p-value* $\leq$ 0.05) are considered statistically significant to TD, and therefore can be selected as TD indicators for further analysis. On the contrary, metrics with *p-values* greater than 0.05 can be removed from further analysis, since they are not considered statistically significant. In our case, all 16 remaining metrics had *p-values* lower than 0.05. Therefore, no further metrics were dropped during this step.

2.3.4 Multivariate Analysis

Since univariate analysis did not filter out any metrics, we proceeded with applying multivariate regression analysis [157] as a final filtering step towards feature selection. While univariate analysis is used to examine the effect of each independent variable on the target variable separately, multivariate analysis examines the common effectiveness of a set of independent variables at predicting the dependent variable. Multivariate analysis is usually combined with Stepwise regression [157], a feature selection method in which the choice of predictive variables is carried out by an automatic procedure, thus allowing for removing independent variables based on their significance (*p-values*). Backward Elimination, a special type of Stepwise regression, involves starting with all candidate variables, testing the deletion of each variable using a multiple linear regression, deleting the variable whose loss gives the most statistically insignificant deterioration of the model fit (i.e., highest *p-value*), and repeating this process until no further variables can be deleted without a statistically significant loss of fit (i.e., until all remaining variables have *p-values* less than the user-defined significance level). This technique has been widely used in empirical software engineering studies to examine the effects of combined metrics on the software quality, defect or code smells prediction [11], [112], [158]–[160]. During this step, all metrics reported in Table 13 were examined since all of them were found to be statistically significant (i.e., *p-value* < 0.05) during the univariate regression analysis.

While one can argue that another round of feature selection might be unnecessary, we decided to perform it mainly for two reasons. First, this additional filtering layer will capture instances where an independent variable that was found to be significant with respect to the dependent variable while being independently examined (univariate analysis), may not have significant predictive power when combined with other variables. Therefore, including it to the final set may lead to redundant information and increase model complexity. Second, the "sliding window" method described in Section 2.4 will extend each initial sample of the dataset by including past information and future information simultaneously into a single row. If for example, we decide to leave the size of independent variable set as is, i.e., 16 features, and we want to include information up to 2 lags in the past (+1 for the current lag), the final set will comprise 3 x 16 = 48 independent variables. While ML models generally support complex relationships between variables, this data reframing approach may result in a dramatic increase in the number of features, and therefore increase the complexity of the prediction algorithms to a point where the performance drops significantly. That being said, we considered to keep our independent variables set as small as possible, but without losing much of its explanatory power.

Before starting the Backward Elimination process, a significance level has to be set. This value acts as a significance threshold that determines the stopping point, i.e., the point at which we no longer need to drop any independent variables (i.e., predictors). In our case, we set the significance level value at 0.05 to examine the statistical significance of each TD indicator to act as a predictor at a 95% level of confidence. Subsequently, the Backward

Elimination process involved performing multiple iterations of fitting a multiple linear regression model with all possible predictors, inspecting the *p-values* of each predictor, and then finding and removing the most insignificant predictor, i.e., the predictor with the highest *p-value*. As long as there was a predictor that could be removed (i.e., its *p-value* is greater than 0.05), the process was repeated by fitting a new model excluding the previously removed predictors. The process stopped when all remaining predictors had *p-values* less than the significant level of 0.05.

After multiple iterations of applying Backward Elimination, details of the final multivariate linear regression model are shown in Table 14. As can be seen, the final model has four covariates, meaning that four metrics, namely *bugs*, *code smells*, *duplicated blocks*, and *afferent coupling (Ca)*, seem to have the most significant impact on TD and act as good TD predictors, at least for the dataset under investigation (225 applications). For each covariate, we provide its coefficient, the standard error, the *t-ratio* and the statistical significance (*p-value*) of the coefficient. The *t-ratio* is the ratio of the coefficient estimate to its standard error. Since our sample is relative large, a *t-ratio* greater than 1.96 (in absolute value) suggests that our coefficients are statistically significantly different from zero at the 95% confidence level. We observe that all remaining predictors have high *t-ratio* values (>1.96). In addition, the *p-value* for each covariate tests the null hypothesis that the coefficient is equal to zero (no effect). We observe that after performing Backward Elimination, all remaining predictors have low *p-values* (< 0.05), thus they are likely to be meaningful since the null hypothesis is rejected. The intermediate results that we obtained throughout the various iterations of the Backward Elimination process can be found online[31] and provide information regarding the metric that was eliminated at each iteration until reaching the final set of TD indicators shown in Table 14. We also provide details regarding the coefficient, the standard error, the t-ratio, and the statistical significance (*p-value*) of each metric during every iteration of the process.

Finally, to strengthen the feature selection process followed, we tested the optimal TD predictors selected above for multicollinearity. Multicollinearity is a phenomenon where two or more predictors show high intercorrelations, i.e., they are highly linearly related. While correlation between a predictor and the target variable is an indication of good model performance, correlation among the predictors is usually an issue. If this issue is not taken care of during the feature selection analysis, it can later cause unpredictable variance and lead to overfitting, as the model cannot ascertain how important a feature is to the target variable. One of the most common ways to identify and quantify the severity of multicollinearity in a linear regression analysis is the Variance Inflation Factor (*VIF*) [161]. The *VIF* is calculated by taking each predictor, regressing it against every other predictor in the model and then using the produced coefficient of determination (R^2) into the following formula:

$$VIF = \frac{1}{1-R^2}$$

[31] https://sites.google.com/view/technical-debt-forecasting/main

VIF values range from 1 upwards. As a rule of thumb, a *VIF* value between 1-5 indicates that a predictor is moderately correlated with the other predictors, while a value between 5-10 indicates that multicollinearity is likely present and thus, the predictor should be removed. We computed *VIF* factors for each of the predictors presented in Table 14. As can be seen, all *VIF* values are considerably less than 5, indicating that our final TD predictor set does not suffer from multicollinearity.

Table 14: Multivariate analysis model of TD indicators (extended dataset)

Metric	*Coefficient*	*Standard error*	*t-ratio*	*p-value*	*VIF*
bugs	0.1075	0.027	4.056	0.000	2.503
code_smells	0.7489	0.029	25.899	0.000	3.574
duplicated_blocks	0.1276	0.020	6.494	0.000	2.752
Ca	0.1816	0.040	4.567	0.000	1.247

All the analysis described above was performed by using the Python programming language and more specifically the scikit-learn[32] ML library. To conclude, among the initial 24 metrics (TD indicators) under investigation, four of them were found to have statistically significant effects on TD. Therefore, the optimal TD predictors extracted through this process were *bugs*, *code smells*, *duplicated blocks*, and *afferent coupling (Ca)*. These metrics will be considered as input to the forecasting models during the model training phase described in Section 3.

2.4 Sliding window method

In general, ML models do not directly support the notion of observations over time. As a result, time series data usually need to be re-framed in a form suitable for supervised learning problems before used for forecasting tasks. To understand this notion, an example of dummy data collected in temporal order is presented in Table 15. Each row represents a sample of data collected at a specific lag (timestamp). Columns 2 to 4 hold the values of independent variables *X1* to *X3* respectively, while column *Y1* holds the value of the target variable. One thing that is apparent in this table is that the structure of the data does not quite fit the supervised learning framework. Two problems arising from this particular data format are the following: First, if the dataset is used in this format during model training, no past information will be included in the samples, due to the fact that each row only includes information about one specific lag. Second, since the target variable of each row points to a current lag value, the model will learn to make estimations only for the current lag (rather than forecasts).

A benefit of using ML models over traditional statistical approaches (e.g., ARIMA) is their ability to support more than one input features. Trying to take advantage of this, we used a method called "sliding window" [162] to transform the dataset in a format that integrates into a single sample multiple prior time steps as inputs (*X*) to predict future time steps as output (*Y*). In short, this method extends each initial sample of the dataset by including

[32] https://scikit-learn.org/stable/

past information and future information simultaneously into a single row. This approach is described in more detail below.

Table 15: Dataset collected in temporal order

X				*Y*
timestamp	***X1***	***X2***	***X3***	***Y1***
0	10	100	1000	10000
➡ **1**	20	200	2000	20000
2	30	300	3000	30000
3	40	400	4000	40000
4	50	500	5000	50000
…	…	…	…	…

The number of past time steps that we want to include as input into each sample is called the "window width" or size of the lag. As a first step, the width of the sliding window needs to be chosen. Window width, illustrated as a red box in Table 15, corresponds to the number of rows, i.e., the current lag (indicated with a red arrow) plus a number of past lags that will be merged into a new single row. In this example, supposing that *t* is the current lag, the red box in Table 15 indicates that independent variables of the samples at lags *t* and *t*-1 (one step in the past) will be merged into one new row that incorporates not only current but also past information. Additionally, the desired forecasting horizon, illustrated as a blue box in Table 15, needs to be chosen. More specifically, the blue box in this example indicates that we want forecasts for 1 step-ahead, thus the *Y* value of *t*+1 sample will be selected as the target variable. In case we wanted to prepare the dataset for 2 steps-ahead forecasts, *t*+2 value would be selected as the target variable, and so on. The above process will result in a new row, as depicted in Table 16. The process is repeated by shifting the two boxes simultaneously over the samples, one step at a time, creating new rows until the window reaches the end of the table. Applying the above transformation will result in a reframed dataset that uses one past lag plus the current lag of independent variables to forecast 1 step-ahead. The reframed dataset is presented in Table 16.

Table 16: The reframed dataset after applying the sliding window approach

X				*Y*			
index	***X1(t-1)***	***X2(t-1)***	***X3(t-1)***	***X1(t)***	***X2(t)***	***X3(t)***	***Y1(t+1)***
0	10	100	1000	20	200	2000	30000
1	20	200	2000	30	300	3000	40000
2	30	300	3000	40	400	4000	50000
3	40	400	4000	…	…	..	…

There is no standard answer regarding the choice of the window width, i.e., the number of past lags that will be merged per row. This choice usually depends on the number of independent variables, the length of the forecasting horizon and the forecasting model itself. Therefore, during the initial window width selection, a balance needs to be found between the model complexity and the optimal prediction quality. It is often a good idea to test different numbers by training an algorithm and see what values work better for different forecasting horizons, based on an error minimization criterion. For instance, we found out that choosing a window width of 2 lag observations resulted in the minimum Mean Absolute Error (MAE) when trying to forecast for 5 steps ahead, for most of the application-specific datasets across different models. Adding more than 2 lags simply increased models complexity, without profound impact on the model accuracy. Respectively, for longer forecasting horizons, a larger window appeared to be more suitable and resulted in better model performance.

We restructured each application-specific dataset using this method, depending on the forecasting length we wanted to test our models, to make it suitable for supervised ML. Once a time series dataset is prepared this way, any of the standard linear and non-linear ML algorithms can be applied, as long as the order of the rows is preserved. A fragment of the Apache Kafka reframed dataset, after applying the sliding window approach is presented in Table 17.

Table 17: The Apache Kafka dataset reframed for 1 step-ahead forecasts using a sliding window with a width of 2

X													***Y***
index	***cs(t-2)***	***bu(t-2)***	***db(t-2)***	***ca(t-2)***	***cs(t-1)***	***bu(t-1)***	***db(t-1)***	***ca(t-1)***	***cs(t)***	***bu(t)***	***db(t)***	***ca(t)***	***TD(t+1)***
0	1277.0	89.0	29.0	7.985	1267.0	91.0	31.0	7.961	1102.0	88.0	33.0	7.986	23782.0
1	1267.0	91.0	31.0	7.961	1102.0	88.0	33.0	7.986	1104.0	89.0	37.0	8.064	23791.0
2	1102.0	88.0	33.0	7.986	1104.0	89.0	37.0	8.064	1107.0	89.0	37.0	7.995	23852.0
3	…	…	…	…	…	…	…	…	…	…	…	…	…

where cs = code smells, bu = bugs, db = duplicated blocks, ca = afferent coupling, t = time step

Another particular challenge that emerges from the concept of TD forecasting is the need to make multi-step forecasts, that is, forecasts for more than one time-step into the future. This need is driven by the fact that we are trying to capture the entire future TD evolution of a software application, rather than the TD value at a particular time step in the future. There are three main approaches that ML methods can use to make multi-step forecasts: i) the *Direct* approach, where a separate model is developed to forecast each forecast lead time, ii) the *Recursive* approach, where a single model is developed to make one-step forecasts, and the model is used recursively where prior forecasts are used as input to forecast the subsequent lead time, and iii) the *Multiple output* approach, where a single model with multiple outputs is developed, capable of predicting the entire forecast sequence in a one-shot manner.

Most ML-based regression models, with the exception of ANNs, do not directly support more than one outputs. Hence, we excluded *Multiple output* approach. Moreover, the sliding window method described above assumes that the dataset is reformed in a multivariate way, i.e., it includes lag observations from independent variables. As a result, *Recursive* approach is also excluded because it would require also forecasted values of the independent variables to predict further than one step ahead. Therefore, we adopted the *Direct* approach, which means that separate models will be developed to forecast each forecasting horizon. In practice, this means that when trying to forecast for *N* steps-ahead, the dataset will be reframed *N* times by following the sliding window approach described above, where each time the dependent variable *Y* will point to a value from *t*+1 to *t*+*N* steps ahead. Subsequently, *N* separate models will be created, each dedicated to forecast one future point starting from *t*+1 up to t+ *N*. Finally, the outputs of the models will be merged into a common vector that depicts the entire forecasted TD evolution up to *N* steps ahead.

3 Machine Learning Approach for TD Forecasting

In the previous section, we first introduced various software-related metrics that have been widely used in the literature as TD indicators and, then described the data collection process we followed in order to prepare our initial application-specific datasets. Subsequently, during the feature selection process described in Section 2.3, we reduced the initial 24 features (TD indicators) to 4 in order to reduce model complexity. The optimal TD predictors selected were *Code Smells, Bugs, Duplicated Blocks* and *Afferent Coupling (Ca)*. Finally, we restructured each application-specific dataset using the sliding window method to make it suitable for supervised ML. In this section, we examine the ability of various ML models to forecast the evolution of TD Principal for each application-specific dataset based on the selected TD predictors. To do so, we train and test the selected models for various forecasting horizons ranging from 1 to 40 steps (weeks) ahead by means of time series validation. Obtained prediction errors of the investigated algorithms are compared among the various forecasting horizons and their benchmarking and evaluation results are documented thoroughly.

3.1 Model Training, Testing and Benchmarking

In this section, we investigate the ability of linear and non-linear ML models to forecast TD evolution of 15 software applications. To do so, we applied a collection of ML models such as Multivariate Linear Regression (MLR), Ridge and Lasso regression, Stochastic Gradient Descent (SGD), Support Vector Regression (SVR) with both linear and Gaussian kernel, and Random Forest regression and compared their results for each application-specific dataset. Most of these models have been extensively compared and evaluated in the literature for their ability to predict software quality attributes, such as Maintainability [43], [95], [163], [102], [96] and Security [164]–[166], or lower-level software properties, such as code smells [10] and defects [112]. However, choosing the most appropriate ML model is often the result of trial and error, as the predictive performance of these algorithms strongly depends on the size and structure of the data. Therefore, within the context of this chapter, we considered investigating a broad spectrum of ML models in order to account for highly diverging data relationships that may govern the different application-specific datasets and overcome the limitations of different techniques. The selected models are briefly described below:

- *Multivariate linear regression (MLR)* is the most commonly used technique for modelling the relationship between two or more independent variables and a dependent variable by fitting a linear equation to observed data. During MLR, the coefficients of the variables are estimated using the least squares method. The main advantages of this technique are its simplicity, interpretability and the fact that it performs well when the relationship to be modelled is not extremely complex. In addition, it is supported by many popular statistical packages. However, quite often simple MLR models are suffering from overfitting.

- *Ridge and Lasso regression* are simple techniques that aim to reduce model complexity and thus, prevent overfitting by applying regularization, i.e., add some constrains to the loss function. In the case of Ridge regression, those constraints are the sum of squares of the coefficients multiplied by the regularization coefficient (lambda). This regularization type is known as L2. Lasso regression works similarly but instead of adding the squares of the coefficients to the loss function, it adds absolute values. As a result, during the optimization process, coefficients of unimportant features may become zero, which acts as an automated feature selection. This regularization type is known as L1. The main advantages of these regularization techniques, apart from the fact that they prevent overfitting, are the simplicity and computational efficiency of the produced model. However, regularization models are often suffering from high bias error.

- Gradient descent is the process of minimizing a function by following the gradients of the loss function. This involves knowing the form of the loss as well as the derivative so that the function can move towards the minimum value. *Stochastic Gradient Descend (SGD)* regression is based on gradient descent, but instead of updating coefficients based on the derivative of the data, the algorithm updates the coefficients based on the derivative of a randomly chosen sample. In that way, SGD allows the function to converge and overcome local minima faster. Because of the randomness involved, the main advantage of SGD is its ability to perform well with noisy data.

- The goal of *Support Vector regression (SVR)* is to find a function that approximates the target values for all the training data with the minimum generalization error. To achieve this, it tries to learn a non-linear function by linearly mapping features into high-dimensional, kernel-induced feature space. The main advantage of SVR is the efficient non-linear data handling by using the kernel trick. In addition, SVR supports regularization capabilities (L2 Regularization) that prevent overfitting. However, hyper-parameter tuning and choosing an appropriate kernel function can be proven a difficult task.

- Starting with the base case, a *Regression Tree (RT)* is a variant of decision trees that is built through an iterative process of splitting the data into partitions on each of the decision nodes, known as binary recursive partitioning. An enhanced version of the RT is *Random Forest (RF)* method. RF is an ensemble of RTs trained with the “bagging” method. Bagging repeatedly selects random samples by replacing the training set and fits trees to these samples. After training, predictions for unseen samples are made by averaging the predictions, or by taking the majority vote of all the individual RTs. The main advantages of RF are its interpretability and the fact that it is great at learning complex, highly non-linear relationships. However, RF models are slower and require more memory compared to the other models presented. In addition, RF models are prone to major overfitting due to the training nature of decision trees.

For the conduction of our experiments, we used the Python programming language and more specifically the *scikit-learn*[33] ML library. For reasons of brevity, the ML approaches presented below will focus mainly on the Apache Kafka software system. However, results and model comparisons will include also the rest of the applications.

Once our dataset is ready for supervised learning, the next step is to train and validate the performance of the selected algorithms. Validation methods extensively used in ML, such as k-fold cross-validation, cannot be directly used with time series data due to the temporal order in which values were observed. Hence, observations cannot be randomly split into groups without respecting the temporal order. To better assess prediction accuracy and compare different models we adopted the Walk-forward Train-Test validation method [124], a strategy inspired by k-fold cross-validation. Walk-forward Train-Test validation is a commonly used way to evaluate time series models performance, based on the notion that models are updated when new observations are made available. In brief, during Walk-forward Train-Test validation a subset of n consecutive points extracted from the original time series is used to train an initial model. Then, accuracy of the model is tested against future time steps and prediction is evaluated against the known value to compute prediction errors. Finally, the time window is expanded to include the known values into the training set and the process is repeated. Validation results are combined (e.g., averaged) over the rounds to give an estimate of the model's predictive performance. Using Walk-forward Train-Test validation will result in more models being trained, and in turn, a more accurate estimate of the performance of the models on unseen data. Figure 8 below provides a visualization of the Walk-forward Train-Test validation behavior.

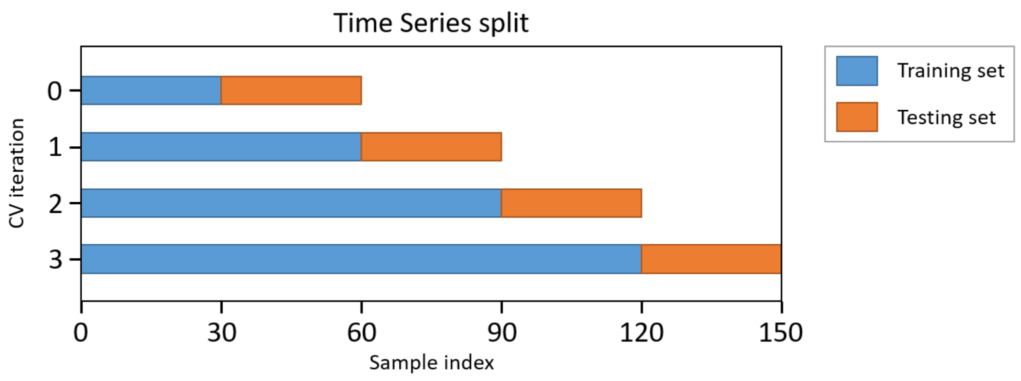


Figure 8: Walk-forward Train-Test validation

The Apache Kafka dataset consists of 150 observations (snapshots). For Walk-forward Train-Test validation we chose the number of splits = 5, meaning that training set will start from 25 samples and will expand up to 125 samples during the last iteration. The test set will constantly contain 25 observations. The number of splits = 5 was chosen such that each train/test group of data samples is large enough to be statistically representative of the broader dataset. A larger number of splits would result in overly small train/test groups, which in turn would suffer from large variability [167]. It is worth mentioning here that number of splits = 5 was chosen also for the other application datasets that contain 150

[33] https://scikit-learn.org/stable/

observations. For those that contain 100 observations, we chose the number of splits = 4 to maintain the train/test group size analogy. To test predictive performance of our models for different future horizons, we repeated the whole validation process five times, where predictions were made for the next n+1 (1 week), n+5 (5 weeks), n+10 (10 weeks), n+20 (20 weeks), and n+40 (40 weeks) future steps respectively.

Before the learning process begins, a hyper-parameter tuning process must take place in order to increase models' predictive performance. A model hyper-parameter is an external attribute of the model. In contrast to typical model parameters, e.g., the coefficients of a Linear Regression model, the value of a hyper-parameter cannot be estimated from data during the training process. Hyper-parameter examples may include the penalty parameter C of the error term in SVM, the number of trees in the Random Forest, etc. In order to tune our models in the best possible way, we used the GridSearchCV[34], a python implementation of the Grid-search method [168]. Grid-search is commonly used to find the optimal hyper-parameters of a model that result in the most accurate predictions, by performing an exhaustive search over specified parameter values for an estimator. We chose R^2 (coefficient of determination) as the objective function of the estimator to evaluate a parameter setting. R^2 is the proportion of the variance in the dependent variable that is predictable from the independent variable(s). We performed hyper-parameter selection on every application-specific dataset during the 5-fold Walk-forward Train-Test validation described above to avoid overfitting and ensure that the selected models have a good degree of generalization.

We evaluated and compared the forecasting performance of the investigated models using the Mean Absolute Percentage Error (MAPE). The MAPE is a popular measure for forecast accuracy that uses absolute values to measure the size of the error in percentage terms. MAPE has two advantages. First, the absolute values keep the positive and negative errors from cancelling out each other. Second, because relative errors do not depend on the scale of the dependent variable, this measure allows for comparing forecast accuracy between differently scaled time-series data (e.g., different software applications). The equation of MAPE is given below:

$$MAPE = \frac{100}{n}\sum_{i=1}^{n}\frac{|y_i - \hat{y}_i|}{y_i}$$

where n is the number of observations, y_i is the actual value and $\hat{y}_i$ is the forecast value.

To further complement model evaluation, we also computed the Mean Absolute Error (MAE) as well as the Root Mean Squared Error (RMSE). Both of these errors are widely used in forecasting tasks. MAE measures the average magnitude of the errors in a set of predictions, without considering their direction. RMSE is a quadratic scoring rule that also measures the average magnitude of the error. Both MAE and RMSE express average model prediction error in units of the variable of interest. The equations of MAE and RMSE are given below:

[34] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

$$MAE = \frac{\sum_{i=1}^{n}|y_i - \hat{y}_i|}{\text{n}}$$

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Again, n is the number of observations, y_i is the actual value and $\hat{y}_i$ is the forecast value.

In Table 18, we report a comparison of prediction errors of the regression models trained on the Apache Kafka dataset for multiple (1, 5, 10, 20 and 40) time steps (weeks) into the future. Prediction errors in each cell of the table are averaged values of the testing errors for all train-test splits that were performed during Walk-forward Train-Test validation. Prediction errors indicated in bold are averaged values of the specific models that were created for each week-ahead prediction category (i.e., 1-week, 5-weeks, 10-weeks ahead models, etc.).

As a reminder, since we adopted the *Direct* approach described in Section 2.4, each examined model provides a single output, that is, the predicted value of the horizon that it was trained to provide forecasts for. In practice, this means that the values of the errors presented below refer to a forecast for a specific individual point in the future, not the entire forecasted evolution of up to that point. We present aggregated forecasts that illustrate the entire evolution of the examined applications in the Model Execution phase described in Section3.2.

Table 18: Apache Kafka TD predictions using Walk-forward Train-Test validation

Model	*Weeks ahead*	*MAE (minutes)*	*RMSE (minutes)*	*MAPE (%)*
MLR	1	594.799	819.085	1.267
	5	2655.770	3094.180	5.906
	10	3105.604	3613.806	6.595
	20	4163.185	4948.930	8.146
	40	6214.496	6790.829	11.072
	Average	**3346.771**	**3853.366**	**6.597**
Lasso regressor	1	430.106	676.957	0.881
	5	1474.881	1768.435	3.055
	10	2240.932	2615.345	4.455
	20	2894.261	3239.653	5.444
	40	4700.503	5004.199	8.655
	Average	**2348.137**	**2660.918**	**4.498**
Ridge regressor	1	438.024	682.877	0.898
	5	1579.645	1869.698	3.260
	10	2579.991	2922.568	4.979
	20	3239.642	3627.422	5.977
	40	5061.284	5428.619	9.177
	Average	**2579.717**	**2906.237**	**4.858**
SGD regressor	1	789.139	1049.557	1.690
	5	2812.558	3253.977	6.225
	10	3069.528	3606.347	6.419
	20	3940.336	4718.533	7.617
	40	6383.504	7063.643	11.311
	Average	**3428.834**	**3970.349**	**6.706**

SVR regressor (linear)	1	571.416	830.770	1.214
	5	3457.010	4004.375	7.567
	10	2811.954	3344.931	6.153
	20	3760.544	4551.495	7.564
	40	6569.625	7304.984	11.89
	Average	**3428.834**	**3970.349**	**6.706**
SVR regressor (rbf)	1	4758.278	5254.790	9.561
	5	5753.172	6257.845	11.819
	10	4657.056	5348.217	9.798
	20	3687.333	4485.870	7.591
	40	5240.294	5794.354	9.835
	Average	**4819.227**	**5428.215**	**9.721**
Random Forest Regressor	1	4273.391	4731.381	8.824
	5	5454.023	5936.521	11.259
	10	4430.855	4994.763	9.252
	20	3425.365	4173.618	6.798
	40	3664.173	3947.600	7.007
	Average	**4262.788**	**4767.657**	**8.660**

Figure 9 illustrates the MAPE of the forecasting models, averaging the five forecasting horizon cases (i.e., 1, 5, 10, 20 and 40 weeks ahead) under investigation. Figure 10, Figure 11, Figure 12, Figure 13 and Figure 14 illustrate the MAPE of the forecasting models for 1, 5, 10, 20 and 40 steps (weeks) ahead respectively.

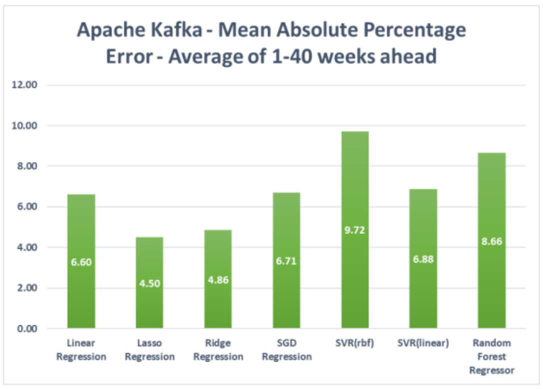


Figure 9: Apache Kafka TD predictions – MAPE averaged for all steps-ahead using Walk-forward Train-Test validation

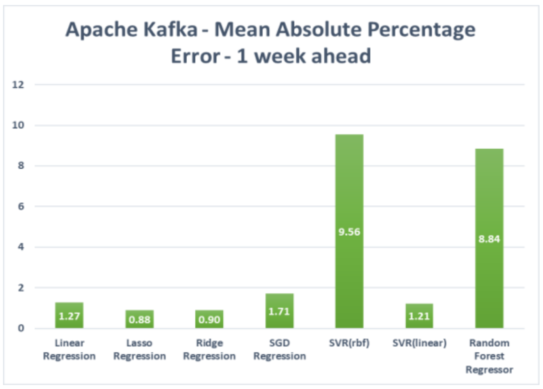


Figure 10: Apache Kafka TD predictions – MAPE for 1 step-ahead using Walk-forward Train-Test validation

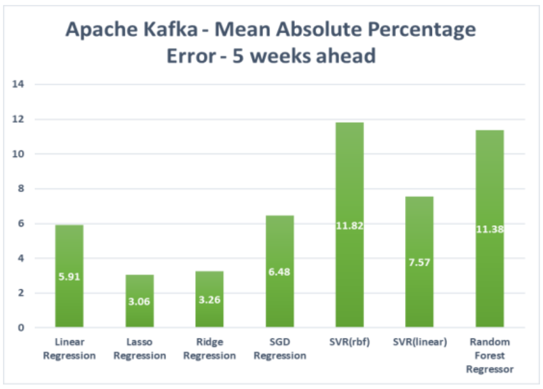


Figure 11: Apache Kafka TD predictions – MAPE for 5 steps-ahead using Walk-forward Train-Test validation

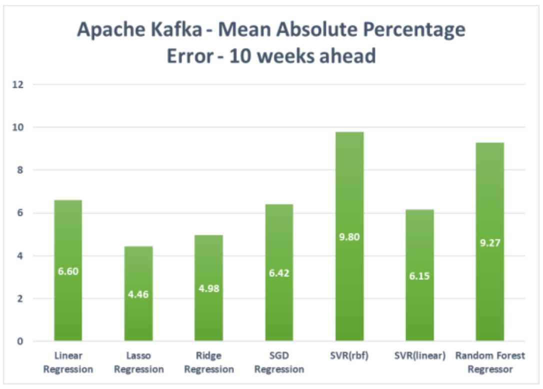


Figure 12: Apache Kafka TD predictions – MAPE for 10 steps-ahead using Walk-forward Train-Test validation

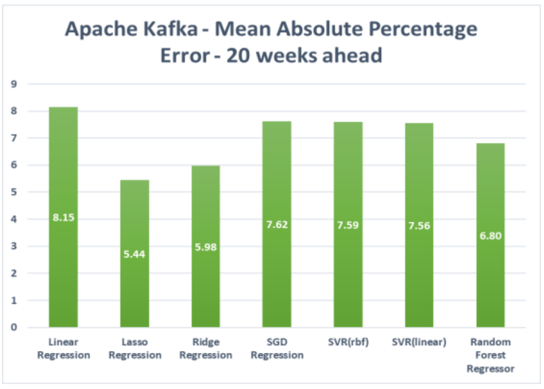


Figure 13: Apache Kafka TD predictions – MAPE for 20 steps-ahead using Walk-forward Train-Test validation

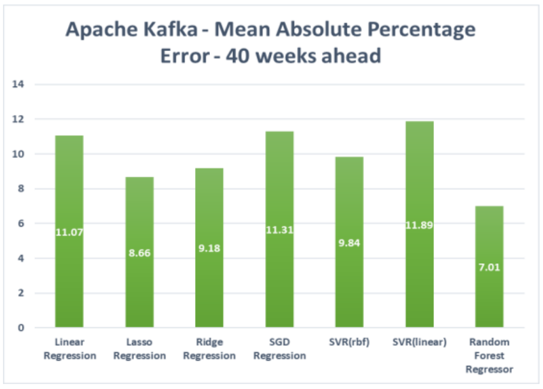


Figure 14: Apache Kafka TD predictions – MAPE for 40 steps-ahead using Walk-forward Train-Test validation

By observing Figure 9, it is clearly depicted that linear models, such as MLR, Lasso, Ridge and SVR(linear) Regression, have generally lower MAPE values and perform better that non-linear models, such as SVR(rbf) and Random Forest Regression. Moreover, we observe that among linear models, the best accuracy is demonstrated by models that apply Regularization in order to prevent overfitting, i.e., Lasso and Ridge Regression. More specifically, Lasso Regression is the best candidate with an average MAPE value of 4.5%, followed by Ridge Regression with an average MAPE value of 4.86%.

When it comes to shorter forecasting length (i.e., 1-10 weeks ahead), the difference between linear and non-linear model performance becomes even clearer. By having a look at Figure 10, we notice that forecasting the TD of the Apache Kafka project for 1 step ahead (1 week) using Lasso Regression gives a MAPE of 0.88%, while for the same horizon SVR with a Gaussian kernel gives 9.56% and Random Forest Regression gives 8.84%. Correspondingly, by having a look at Figure 11 and Figure 12 we observe that Lasso Regression for 5 steps (5 weeks) and 10 steps (10 weeks) ahead gives a MAPE of 3.06% and 4.46%, while for the same horizons SVR with a Gaussian kernel gives 11.82% and 9.80% respectively.

Linear models are the best candidates even for a forecasting length of 20 steps (20 weeks) ahead, as can be seen by Figure 13. However, an interesting observation is that while their predictive power drops significantly as we try to forecast longer into the future, non-linear models seem to have an almost stable performance over the holdout sample for all steps ahead. This could be an indicator that for even longer lengths, non-linear models could perform better than the linear ones. Indeed, by having a look in Figure 14, we observe that for a forecasting horizon of 40 steps (40 weeks) ahead, Random Forest Regression is the best candidate and gives the lowest MAPE (7.01%).

To further examine the ability of the investigated algorithms to forecast TD Principal and get an understanding of how the models perform, we repeated the same experiments for each of the 15 applications in our dataset. We will not go through each project one by one, but instead we will provide averaged scores. Detailed results of applying Walk-forward Train-Test validation for the rest of the applications can be found at the online Appendix [169] (Table 1 to Table 14). Figure 15 below illustrates the MAPE of the forecasting models, averaging the five forecasting horizon cases (1, 5, 10, 20 and 40 weeks ahead) and the 15 software applications under investigation. Figure 16, Figure 17, Figure 18, Figure 19 and Figure 20 illustrate the MAPE of the forecasting models for 1, 5, 10, 20 and 40 steps (weeks) ahead respectively, averaging the 15 software applications under investigation.

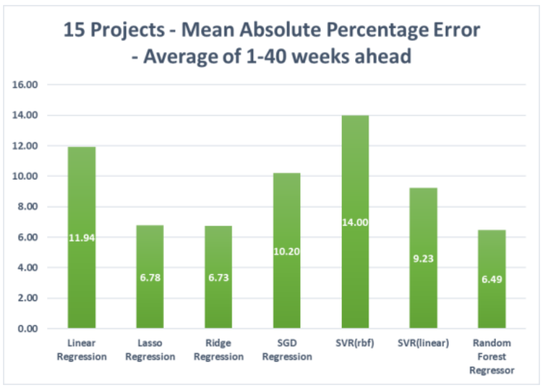


Figure 15: 15 projects TD predictions – MAPE averaged for all steps-ahead using Walk-forward Train-Test validation

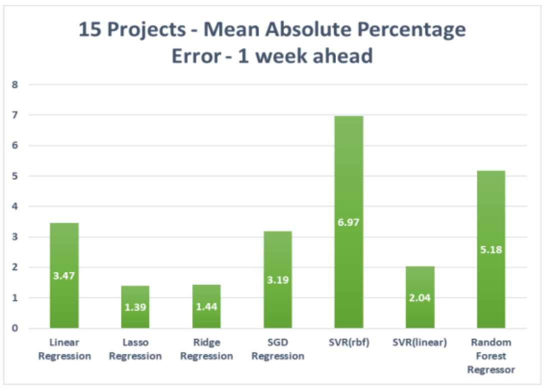


Figure 16: 15 projects TD predictions – MAPE for 1 step-ahead using Walk-forward Train-Test validation

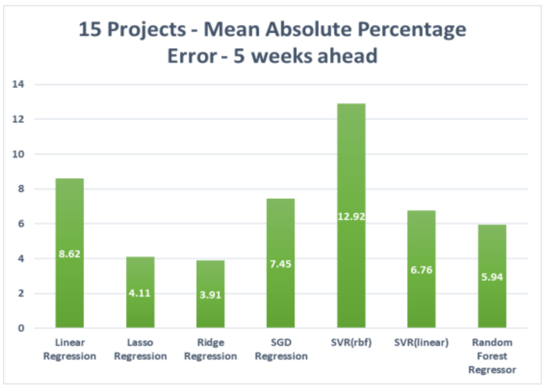


Figure 17: 15 projects TD predictions – MAPE for 5 steps-ahead using Walk-forward Train-Test validation

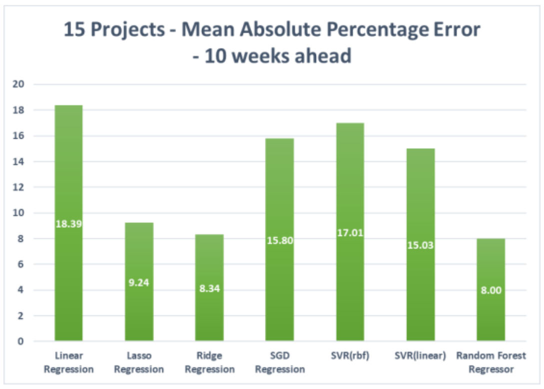


Figure 18: 15 projects TD predictions – MAPE for 10 steps-ahead using Walk-forward Train-Test validation

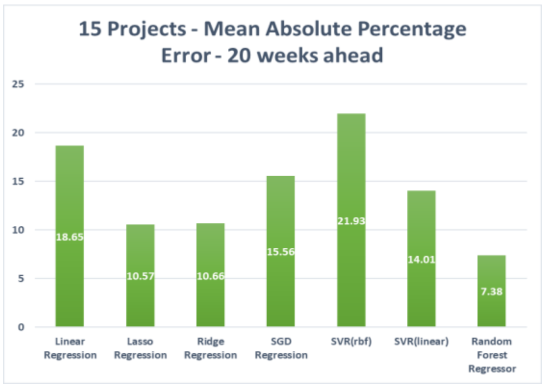


Figure 19: 15 projects TD predictions – MAPE for 20 steps-ahead using Walk-forward Train-Test validation

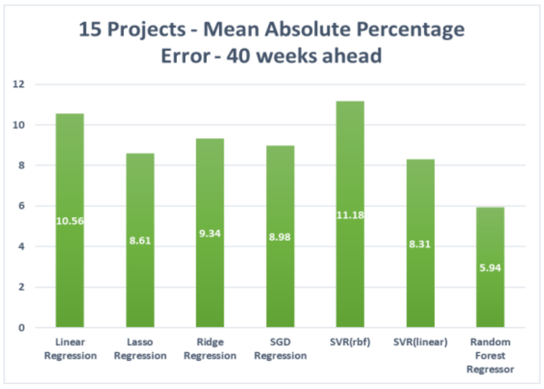


Figure 20: 15 projects TD predictions – MAPE for 40 steps-ahead using Walk-forward Train-Test validation

Similarly to the Apache Kafka case, we observe that for shorter forecasting lengths, linear models that apply Regularization, such as Lasso and Ridge regression, have generally lower MAPE values and higher performance compared to the non-linear models. As depicted in Figure 16 and Figure 17, Ridge and Lasso Regression models are the best candidates with MAPE values of 1.44% - 3.91% and 1.39% - 4.11% respectively. We also observe that again, the predictive power of linear models drops significantly as we forecast longer into the future. However, the non-linear Random Forest Regression algorithm seems to have an almost stable performance over the holdout sample for all steps (weeks) ahead. In fact, starting from 20 (Figure 19) up to 40 steps ahead (Figure 20), we observe that Random Forest Regression is the best candidate and performs better than the other models giving the lowest MAPE value of 7.38% and 5.94% respectively.

To sum up, an interesting finding that we can extract from the analysis of the experiments is that linear models that apply Regularization, i.e., Lasso and Ridge Regression, are capable of achieving high forecasting performance for shorter forecasting lengths (<10 weeks ahead), while the non-linear Random Forest Regression is performing better than the rest of the investigated models for longer forecasting lengths (>10 weeks ahead). The fact that the above results are observed in all of the 15 application-specific datasets is also interesting and of high significance, whereas it increases our confidence regarding the generalizability of the aforementioned findings. Although we cannot be sure that these results may apply to similar applications, they do make a valuable contribution to the beginning of the TD forecasting landscape composition. Therefore, a TD forecasting tool could leverage the predictive power from both of these algorithms combined to deliver good predictions and adequately forecast future TD Principal trends of software applications.

3.2 Model Execution

Following the construction and benchmarking of our models described in Section 3.1, this section presents indicative examples of model execution as well as indicative visualizations of the forecasting results. As reported in Section 2.4, we decided to adopt the *Direct* approach, meaning that separate models were developed to forecast each forecast lead time. In Figure 21 below, we provide an example of forecasting the TD Principal evolution of Apache Kafka application for 20 steps (weeks) ahead using Random Forest regression, which during the model validation phase was reported to have a stable performance and perform better than the other examine models for longer forecasting lengths. The red line denotes the forecast, while the blue line denotes the ground truth. It should be noted that the samples covered by the red line (i.e., test set) were excluded during the model-training phase. Behind the scenes, according to the adopted *Direct* approach, 20 models were executed, one for each specific length of interest (starting from 1 step to 20 steps), while their forecasted TD values where aggregated into a common vector, and then plotted as the projected TD evolution.

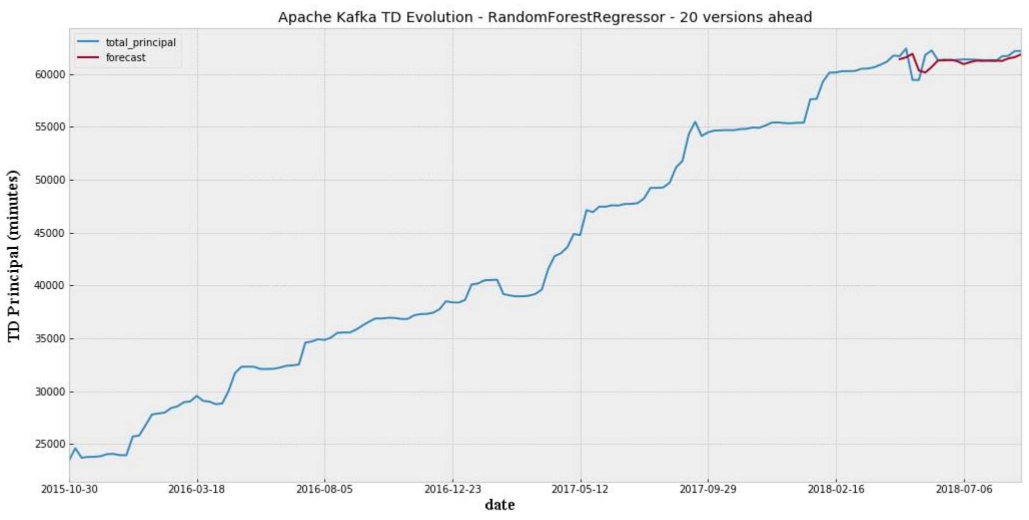


Figure 21: Apache Kafka TD Principal forecasting for 20 steps ahead using Random Forest and the Direct approach

Indicative visualizations illustrating the forecasting results of Random Forest regression for 20 steps ahead for the rest of the 14 applications are provided in the online[35] supportive material. As can be seen in both Figure 21 and the online material, similar observations can be made for all 15 applications under investigation. In particular, the Random Forest regression seems to provide meaningful long-term forecasts for each one of the studied cases (i.e., software applications). In fact, the selected algorithm is able to capture the trend of the future evolution of the TD Principal, whereas in most of the cases the future value of the TD Principal is also captured with a sufficient level of accuracy. For reasons of brevity, we do not provide illustrations of TD evolution forecasts for the rest of the algorithms. However, similarly to the case of long-term TD Principal forecasting using Random Forest regression, satisfying forecasts were also obtained by using Regularization models (i.e., Lasso and Ridge regression) for shorter forecasting lengths, as expected by the results that were reported during model benchmarking in Section 3.1.

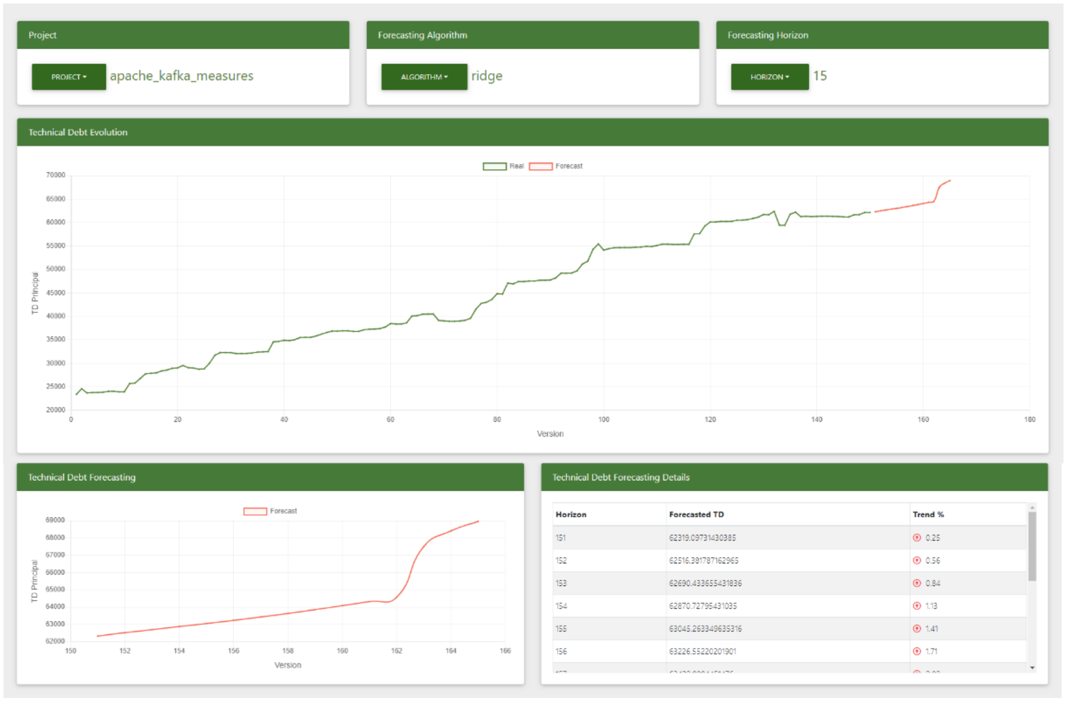
3.3 Technical Implementation

The work presented in this chapter introduces an approach aiming to cover the existing gap in the field and set the foundations towards methods and accompanying tools able to deliver TD forecasts and therefore assist developers and project managers in taking proactive actions regarding TD management activities. The SDK4ED[36] European project aims to address this challenging issue by implementing the proposed approach in the form of a tool, i.e., the TD Forecasting tool, as a part of the integrated TD Management (TDM) framework. To this end, an envisaged TD Forecasting tool has been implemented as individual standalone tool in order to facilitate its applicability in practice. This tool consists of a backend server dedicated to the deployment of a set of forecasting models, a web service that exposes the server, and an interactive Graphic User Interface (GUI) that

[35] https://sites.google.com/view/technical-debt-forecasting/main

[36] https://sdk4ed.eu/

allows the invocation of forecasting models and displays the results, providing users with insightful information for the future evolution of TD. Both the backend and frontend of the TD Forecasting tool are components of the overall SDK4ED Dashboard, which forms the final outcome of the SDK4ED project. The TD Forecasting tool, integrated into a preliminary version of the SDK4ED Dashboard, can be found online[37] (currently being used for development purposes). The main screen of the tool is provided in Figure 22.



Project: apache_kafka_measures

Forecasting Algorithm: ridge

Forecasting Horizon: 15

Horizon	Forecasted TD	Trend %
151	62319.09731430385	0.25
152	62516.381787162965	0.56
153	62690.433655431836	0.84
154	62870.72795431035	1.13
155	63045.263349635316	1.41
156	63226.55220201901	1.71

Figure 22: Main screen of the TD Forecasting tool

The main screen of the TD Forecasting tool comprises a dropdown button, two interactive plots and one table. The dropdown button allows the user to select the forecasting horizon for which they would like to see predictions for. Once the forecasting horizon is selected, the backend server invokes the proper forecasting algorithm (depending on the selected horizon) and returns the predictions back to the GUI, which in turn parses the result. Then, the interactive plots showing the ground truth (green) and the predicted (red) TD Principal evolution appear on the screen. The first plot shows the entire evolution followed by the forecasted evolution of the application, whereas the second plot focuses solely on the forecast, giving a more fine-grained view. In addition to the plots, a complementary table comprising the detailed results of the forecasts is presented at the bottom-right part of the screen. This table presents the forecasted TD values for the upcoming weeks, as well as the difference between the current TD value and the forecasted TD values per week, which may serve as an indicator of whether the TD Principal will increase or decrease, and to what extent. This additional information is expected to help the developers take even more informed decisions regarding the prioritization of their TD repayment activities.

[37] http://160.40.52.130:3000/tdforecast

4 Case Study

In this section, we present the results of an industrial study conducted to empirically evaluate the meaningfulness of the TD forecasting approach introduced in this work and to investigate the extent to which this approach can provide valuable insights and affect developers' decisions regarding the evolution of software, via a questionnaire distributed to representatives of a software company.

4.1 Survey Design

To minimize the possibility that current work will remain a statistical exercise detached from real software development practices, we have designed a survey through which we seek feedback from practitioners. The goal of this survey is twofold: a) to empirically evaluate the meaningfulness and accuracy of the TD forecasting methodology proposed in this study, and b) to empirically assess the usefulness and acceptance of the TD Forecasting concept in general, especially for software companies that deal with daily TD management activities.

For the purposes of this survey, we have involved a Greek department (located in Thessaloniki, Greece) of a large European software company (hereafter referred to as *Company*) that provides IT development services in multiple technologies to private and public organizations. The *Company* employs more than 2.200 highly-skilled professionals worldwide. However, the *Company* wants to keep its anonymity, thus all records in the dataset have been anonymized, and no personalized information can be provided, either about the company and its projects, or the case study participants. In the case of this survey, although participants might not be extremely familiar with the TD concepts and terminology, they are all experienced in issues related to quality assessment, since the *Company* uses SonarQube for continuous inspection of code quality during its software development process. A possible lack of experience in TD terminology has been considered during the design of the data collection instrument. Furthermore, the fact that the *Company* uses SonarQube as a quality inspection tool, allows us to partially validate also the SonarQube TD measurement mechanism, in a sense that we can determine, through communication with the developers, if the TD measurements are in line with real events occurring during the software development process.

As a survey instrument, we opted for a questionnaire, which is described in detail below and can be found also online[38]. The most important part of developing a questionnaire is the selection of questions. In our survey, this process was governed by the guidelines provided by Kitchenham and Pfleeger [170]: (a) keep the amount of questions low, (b) questions should be purposeful and concrete, (c) answer categories should be mutually exclusive, and (d) the number, the order and the wording of questions should avoid biasing the respondent. To this end, we constructed a questionnaire with 13 main questions (4 multiple-choice and 9 short-answer), organized into three main parts (see Table 19), and an introductory part (2 questions). The questionnaire begins with the introductory part (i.e.,

[38] https://forms.gle/Jjg8RoA55m1EwMJ77

Part-1) where participants are asked to provide some demographic information, such as their role and years of experience in the *Company*. Subsequently, in Part-2, participants are first introduced to some background information on the concept of TD and its main components (such as TD Principal, inefficiency types, etc.) and then asked to rate, on a Likert scale, a group of questions that aim to evaluate the usefulness of TD Forecasting. The last two parts of the questionnaire, i.e., Part-3 and Part-4, refer to some project-specific questions (to be able to provide valid answers in Part-3 and Part-4, the participants should have been actively involved in the development of these projects). The process of designing and formulating the questions of Part-3 and Part-4 are thoroughly described below.

During our visit to the premises of the *Company*, we were given access to a dedicated SonarQube instance hosting the analysis results of a large set of software applications, in order to find the most suitable candidates for the design of our survey. The criteria we relied on to select an application are as follows. First, the application needs to be developed in Java programing language. Second, it needs to be constantly maintained and thus, to provide a relative long history of commits, as well as the associated SonarQube analysis measurements available for these commits. Finally, its SonarQube analysis measurements need to contain (at the minimum) *code smells*, *bugs*, *duplicated blocks*, *lines of code* and the TD Principal itself. The first three metrics are required for the construction of the TD forecasting models, since they were selected as the most statistically significant TD predictors based on the analysis conducted within Section 2. Unfortunately, the fourth TD predictor, i.e., *afferent coupling (Ca)* was not available for this analysis, since the *Company* does not use the CKJM Extended tool. Furthermore, we were not given access to the source code of the applications, in order to execute the CKJM Extended tool ourselves. However, we believe that the unavailability of one independent variable will not significantly affect the forecasting performance of our models, especially since *afferent coupling (Ca)* was found to have the highest standard error among the final four variables, as presented in Table 14.

Based on the above criteria, we ended up with two software applications belonging to the Business Software domain, namely *Project A* and *Project B*, with a size of 58K LoC and 384K LoC respectively. Specifically, analysis data from the SonarQube database of the *Company* included 62 commits for *Project A* and 51 commits for *Project B*. At this point, it should be noted that although committing code updates at a weekly basis (i.e., at the end of each week) is considered an integral part of the *Company's* routine, we were informed that there existed a few cases where SonarQube did not run for a particular week, mainly due to technical issues. As a result, collecting SonarQube analysis data at fixed weekly intervals (as introduced in Section 2.2 to establish the dataset) was no longer a viable approach. To overcome this issue, we decided to replace the concept of weekly snapshots with that of consecutive commits. Therefore, within the context of this survey and more specifically in Part-4 of the questionnaire where we present TD forecasts for each application, forecasting for 10 steps ahead is referring to 10 commits rather than 10 weeks ahead. The SonarQube measurements of the two anonymized software applications selected during this step of the process can be found online[39].

As a first step, we parsed the collected SonarQube analysis data, performed some required pre-processing steps and extracted two plots (one for each application) illustrating the entire TD Principal evolution of the two applications under investigation. Subsequently, we manually inspected the plots and extracted selected periods where we identified abrupt (but interesting) TD Principal trends, i.e., a trend showing a gradual increase|decrease, sharp increase|decrease, temporary increase|decrease, etc. Therefore, in Part-3 of the questionnaire, for each of the identified cases, participants are asked what the root cause of these abrupt trends was. For instance, if a trend shows a sharp TD Principal decrease during some period, maybe that is due to a code deletion or code refactoring that removed TD-ill code. If the participants are aware of specific actions they performed that justify these abrupt trends (e.g., refactoring, code additions or deletions, deadlines, etc.), then it means that SonarQube TD measurement mechanism can capture these changes and is in line with real events occurring during the software development process.

As a second step, we applied forecasting models to predict the future TD Principal evolution of both *Project A* and *Project B* for 10 steps ahead. To do so, we exploited a prototype of the TD Forecasting tool described in Section 3.3 in order to extract forecasting plots in an automated way, that is, without repeating the tedious process of model training, testing and benchmarking through the usage of Python scripts. Therefore, in Part-4 of the questionnaire, a TD forecast for each application is presented to the participants and they are asked if they would be willing to change anything in the planned development process based only on the projected TD Principal evolution. For instance, if the TD evolution of a specific application has been constant up to a point but forecasts show a sharp TD increase in the future, we ask them if they would consider performing refactoring to prevent that increase. Respectively, if the TD evolution has been gradually increasing up to a point but forecasts show a sharp TD decrease, we ask them if they would consider investing in enhancing new functionalities instead of performing refactoring. In that way, the practical usefulness and meaningfulness of TD forecasts is, at least partially, evaluated using qualitative feedback from the participants.

Table 19: Survey Instrument

ID	***Question***
	Part 1 - Demographics
Q1.1	What is your role in the company?
Q1.2	How many years of experience do you have in this position?
	Part 2 - The usefulness of TD Forecasting
Q2.1	How useful is it to have an estimation of the current TD Principal of a software project?
Q2.2	How useful is it to have a forecast of the future TD Principal of a software project?
Q2.3	To what extent would a forecast of the TD Principal make you consider changing the planned future development of a project?
Q2.4	Supposed that a forecast shows an increasing trend of the TD Principal, what actions would you take to repay TD?

[39] https://sites.google.com/view/technical-debt-forecasting/main

Part 3 - TD Principal Evolution	
Project A	
Q3.1	Case 1: A temporal TD increase (6/1/19 – 9/1/19). What is the cause of this change?
Q3.2	Case 2: A sharp TD increase (18/1/19 – 27/2/19). What is the cause of this change?
Q3.3	Case 3: A sharp TD increase (4/4/19 – 12/5/19). What is the cause of this change?
Q3.4	Case 4: A gradual TD increase (27/10/19 – 14/2/20). What is the cause of this change?
Project B	
Q3.5	Case 1: A gradual TD increase (23/2/19 – 28/7/19). What is the cause of this change?
Q3.6	Case 2: A sharp TD decrease (28/7/19 – 18/8/19). What is the cause of this change?
Q3.7	Case 3: A sharp TD increase (21/11/19 – 19/2/20). What is the cause of this change?
Part 4 - TD Principal Forecasting	
Project A	
Q4.1	The latest commits of Project A show relatively stable TD evolution. However, the forecast for 10 commits ahead shows a gradual increase in the TD principal. By having a look at this forecast, would you change anything in the planned development process? Would you consider performing code refactoring in order to prevent this increase?
Project B	
Q4.2	The latest commits of Project B show a gradual increase in the TD evolution. However, the forecast for 10 commits ahead shows a slight decrease in TD principal increasing rate. By having a look at this forecast, would you change anything in the planned development process? Would you consider investing in enhancing already existing, or adding new functionalities?

The majority of the questions in Part 2 of the questionnaire have been answered on a Likert Scale ranging from 1 to 5, with the exception of the last question (Q2.4) which gives the respondents the following options: a) “Refactoring”, b) “Writing new code that is TD-free”, c) “No actions”, and d) “Other: ___”. However, the last two parts of the questionnaire, i.e., Part-3 and Part-4, refer to some project-specific questions and require a short description, so they have been answered by providing a “short answer” text box.

4.2 Survey Analysis and Results

In this section, we present the results of the survey study, through presenting some demographics and subsequently analyzing and discussing the answers of the participants. In total, we obtained four (4) complete answers. The reason that we received only four complete answers is due to the fact that Part-3 and Part-4 of the questionnaire require deep knowledge of the applications under analysis. In fact, during our communication with the company, we specifically requested that participants should be actively involved in the development of these projects so that they can provide valid answers. To facilitate the process of distinguishing between the responses of the four participants but at the same time maintain their anonymity, the participants of this survey are hereafter referred to as P1 to P4. Regarding the demographics of the participants extracted from Part-1 of the questionnaire, three out of four participants (P2, P3, and P4) are working in the *Company* as Software Developers, while one participant (P1) is working as a Software Architect. Moreover, participants’ years of experience in this position range from 2 to 12 years, with a mean value of 6.25 years.

First, we analyzed the answers from Part-2 to understand the usefulness of TD forecasting in an industrial context. More specifically, regarding Q2.1 “How useful is it to have an estimation of the current TD Principal of a software project?”, on a Likert scale ranging from: 1 - “Not Useful” to 5 -“Very Useful”, two out of four of the participants (P1 and P4) chose “Useful” (option 4), while the remaining two (P2 and P3) chose “Very Useful” (option 5). These responses suggest that TD Principal monitoring is perceived as highly important for software development companies, as it can provide valuable information regarding the effort and, in turn, the cost that is required for maintaining and extending a software application.

Regarding Q2.2 “How useful is it to have a forecast of the future TD Principal of a software project?”, on a Likert scale ranging again from: 1 - “Not Useful” to 5 -“Very Useful”, two out of four participants (P1 and P2) chose “Useful” (option 4), while the remaining two (P3 and P4) chose “Very Useful” (option 5). These responses suggest that TD Principal forecasting is of great significance and value for software development companies, since they would be able to gain a better understanding of future TD issues and plan well in advance appropriate refactoring activities for saving maintenance costs.

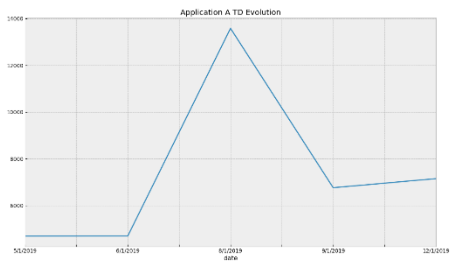
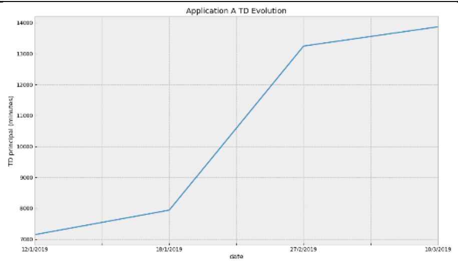
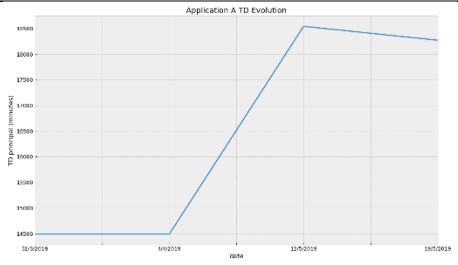
Regarding Q2.3 “To what extent would a forecast of the TD Principal make you consider changing the planned future development of a project?”, on a Likert scale ranging from: 1 - “Not at all” to 5 - “To a great extent”, three participants (P1, P2, and P4) chose “To a moderate extent” (option 4), while the remaining one (P3) chose “To a great extent” (option 5). These responses suggest that all participants would consider changing the planned future development of a project based on a forecast of the TD Principal. In fact, this statement is further evaluated through specific questions presented to the participants in Part-4 of this questionnaire.

Finally, regarding Q2.4 “Supposed that a forecast shows an increasing trend of the TD Principal, what actions would you take to repay TD?”, three out of four participants (P1, P2, and P4) responded with “Refactoring”, while the remaining one (P3) responded with “*First make sure that the new code will have less TD and then, when time plan allows it, refactor existing code*”. While code refactoring is a well-established approach for TD repayment [46], clean code has recently emerged as a promising TD prevention strategy. By inspecting respondents’ answers, we notice that while P1, P2, and P4 would opt for refactoring the already existing code to repay TD, the latter answer indicates that P3, possibly forced by strict deadlines that require the delivery of new functionalities, would prefer to increase the overall quality of the project by writing new TD-free code, i.e., clean code that contributes positively to the overall TD. This question is also further assessed through specific questions presented to the participants in Part-4 of this questionnaire.

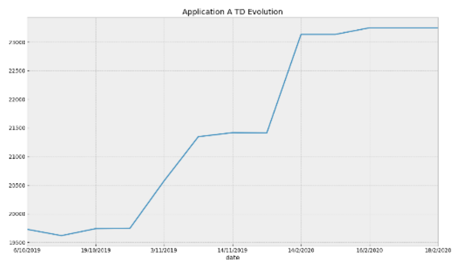
Subsequently, we analyzed the answers from Part-3 of the questionnaire. In Part-3, for a series of identified cases where the TD Principal of Project A and Project B showed abrupt trends, participants were asked what was the root cause of these changes. The main goal of this part is to assess whether participants are aware of specific actions the development team had performed that justify these abrupt trends (e.g., refactoring, code additions or deletions, deadlines, etc.), and therefore to validate that SonarQube TD measurement mechanism can capture these changes and is in line with real events occurring during the software development process. Participants’ answers are summarized in Table 20 and Table 21, which refer to comments regarding observed TD Principal trends of Project A and Project B respectively. By inspecting the tables, it can be seen that three participants

(P2, P3, and P4) are working on Project A, while two participants (P1 and P3) are working on Project B (with P3 working on both projects). The figures of the identified cases illustrated in the tables can be also found online[40].

Table 20: Participants comments on TD Principal trends of Project A

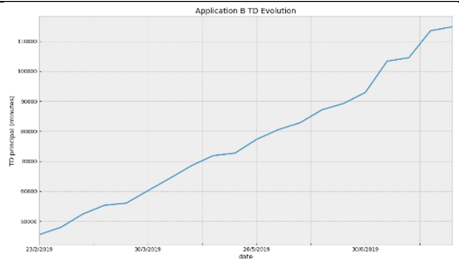
Cases	**TD Principal Trend**	**Participants Comments** **(what is the cause of these TD changes?)**
Q3.1	 <u>Temporal increase</u> (6/1/19 – 9/1/19)	**P3:** "*The project was new and a large amount of functionality was added for the first time from less experienced engineers. Then improvements were made to the initial code and this is why there is this decrease after 08/01.*" **P2:** "*Added 5 new forms to the project along with all the front end and back end code, entities, domains, repositories, services. (06/01-08/01). The decrease from (08/01-09/01) is due to work made on fixing sonar issues regarding TD.*" **P4:** "*Rapid code development (code additions) in order to meet deadlines without testing by junior engineers*"
Q3.2	 <u>Sharp increase</u> (18/1/19 – 27/2/19)	**P3:** "*1st iteration to the customer was coming up. Features had to be completed in limited time.*" **P2:** "*Added 2 new main forms to the project along with all the front end and back end code, entities, domains, repositories, services with code duplication*" **P4:** *Same as Q3.1*
Q3.3	 <u>Sharp increase</u> (4/4/19 – 12/5/19)	**P3:** "*New functionality added after the comments of iteration*" **P2:** "*New forms, new module, with code duplication*" **P4:** *Same as Q3.1*

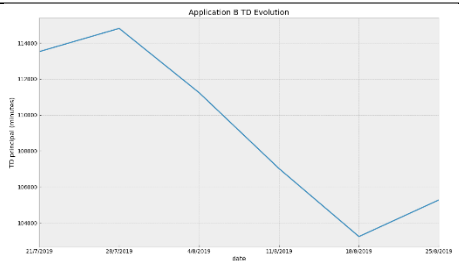
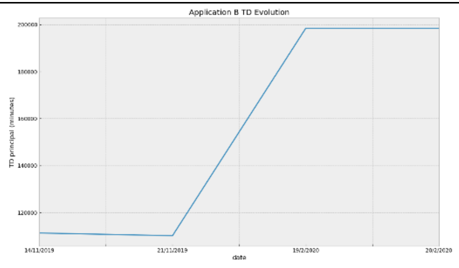
40 https://sites.google.com/view/technical-debt-forecasting/main

Q3.4	 Gradual increase (27/10/19 – 14/2/20)	**P3:** "*Beta testing of the module was creating requests for bug fixing and some new functionality.*" **P2:** "*Code expansion. New features with code duplication*" **P4:** "*Bug fixing, code additions and deletions and new features implementations all at the same time without testing if previous implementations are broken and without testing if the new ones are stable*"

By reading participants' comments regarding the identified TD Principal trends in Table 20, we can note that in almost all cases, developers involved in Project A are aware of specific actions they performed that justify these abrupt trends. More specifically, according to the participants, the temporal TD Principal increase of Project A depicted in Q3.1 can be attributed to a large amount of functionality that was added from less experienced engineers in order to meet deadlines, followed by improvements to the code that led TD Principal to decrease again. Similarly, the sharp TD Principal increase depicted in Q3.2 can be attributed to rapid code additions that had to be completed in limited time and without proper testing, in order to deliver the 1st version of the project to a customer. Subsequently, the sharp TD Principal increase depicted in Q3.3 can be attributed to some new additional functionalities that were requested by the customer and thus were quickly added after the 1st delivery of the project. Finally, the gradual TD Principal increase depicted in Q3.4 can be attributed to rapid bug fixing and code expansion, without properly testing previous and new implementations. Based on the above, we could state that the SonarQube TD measurement mechanism is indeed able to capture real events occurring during the software development process.

Table 21: Participants comments on TD Principal trends of Project B

Cases	TD Principal Trend	Participants Comments (what is the cause of these TD changes?)
Q3.5	 Gradual increase (23/2/19 – 28/7/19)	**P1:** "*Many developers writing code with strict deadlines.*" **P3:** "*1st and 2nd iteration to the customer plus a demo for a new contest in a new country at the same time.*"

Q3.6	 Sharp decrease (28/7/19 – 18/8/19)	**P1:** "*The cause is refactor of code written quickly.*" **P3:** "*Free time in summer to spend on refactoring and solving bugs vulnerabilities and code smells*"
Q3.7	 Sharp increase (21/11/19 – 19/2/20)	**P1:** "*Sonarqube did not run for a long time.*" **P3:** "*New customer came up asking new features that caused refactor to specific parts of code, plus simultaneously iterations to both customers in limited time.*"

Similarly to the case of Project A, by reading participants' comments regarding the identified TD Principal trends in Table 21, we can observe that developers involved in Project B are also aware of specific actions they performed that justify these abrupt trends. More specifically, according to the participants, the gradual TD Principal increase of Project A depicted in Q3.5 can be attributed to specific strict deadlines, such as the delivery of the 1st and 2nd version of the product to the customer and a demo for a new contest, that forced the *Company* to involve more developers into this project. Similarly, the sharp TD Principal decrease depicted in Q3.6 can be attributed to code refactoring and defects fixing that the developers performed during the summer period. Finally, the sharp TD Principal increase depicted in Q3.7 can be attributed to new features that were added for a new customer and affected specific parts of code, thus making developers constantly having to switch between two customers in a limited time. The comment from the developer stating that SonarQube did not run for a long time does not affect the magnitude of the TD increase but implies that in that case, the increase could be gradual instead of sharp. Similarly to the observations made regarding project A, in the case of Project B we could also state that SonarQube TD measurement mechanism can capture real events occurring during the software development process.

Summarizing the answers of the respondents regarding Q3.1 to Q3.7, that is, Part-3 of the questionnaire, we observe that most of the TD pattern types observed during the TD evolution of Projects A and B can be attributed to similar events that occurred during the software development cycle. More specifically, TD growth (either sharp or gradual) is mainly attributed to rapid code additions, usually without proper testing, in order to implement new features and functionalities that were requested by clients of the Company under strict time constraints. Similarly, temporal TD growth is related to quick and "dirty" code expansions in order to meet deadlines, which however were followed by prompt refactoring actions that improved the TD quality of the recently-added code. On the other hand, TD drop is attributed to heavy refactoring cycles that were performed during relatively relaxed periods, to repay the large amount of accumulated TD and thus, improve the quality and maintainability of the suboptimal code introduced in the two projects during a long period of rapid code expansions mentioned above. The above events are in line with the definition of the TD metaphor and verify the necessity for which it was inspired in the first place. The quality compromises made by the Company during the studied period may have yielded the desired short-term benefits, such as the quick delivery of code to the clients, but have resulted in quality decay of the Company's software products, which made developers aware of the need to spend additional time on refactoring actions in order to bring the software back to a maintainable state.

Finally, we analyzed the answers from Part-4 of the questionnaire. In Part-4, TD forecasts for Project A and Project B were presented to the participants and they were asked if they would be willing to change anything in the planned development process based only on the projected TD Principal. In that way, the main goal of this part is to evaluate the practical usefulness and meaningfulness of TD forecasts using qualitative feedback from the developers. Figure 23 and Figure 24 illustrate 10 steps-ahead forecasts (red line) for Project A and Project B respectively, using Ridge Regression that performed better for short-term predictions. In this point, it should be noted that in the figures included in the questionnaire, the ground truth (blue line) from the starting point of the forecasts and onwards was hidden from the participants. However, we include this information here for reasons of completeness and validation of the forecasting approach presented in this work. In addition, the time point from which we decided to start our forecasts was carefully selected to signal a significant change in the current trend up to that point. The reason behind this choice is that we want to assess the willingness of developers to change their planned development processes, based on a change they cannot foresee just by looking at the past trend. Participants' answers to Q4.1 and Q4.2 are summarized below.

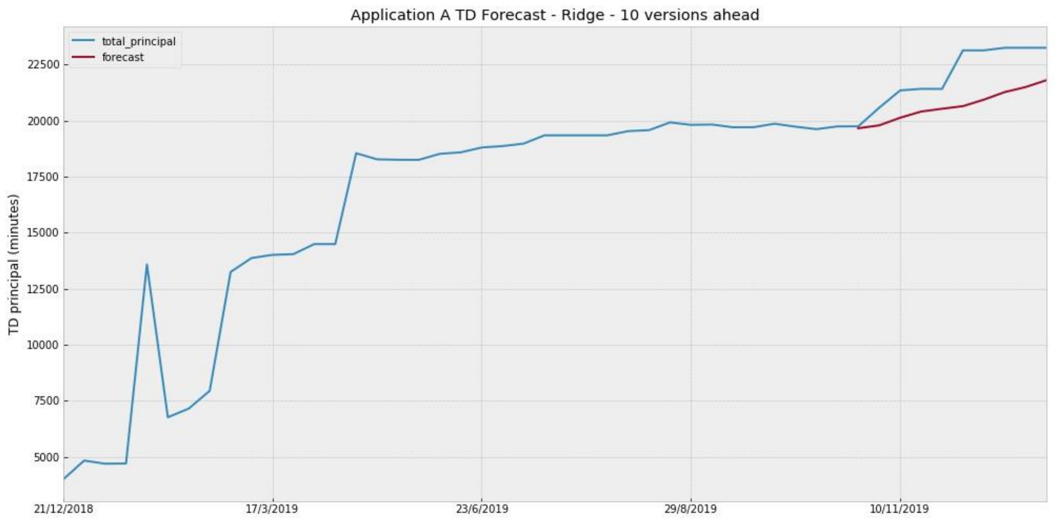


Figure 23: Project A TD Principal forecasting for 10 steps ahead using Ridge (the ground truth from the starting point of the forecasts and onwards was hidden from the participants)

Q4.1 "The latest commits of Application A show relatively stable TD evolution. However, the forecast for 10 commits ahead shows a gradual increase in the TD principal. By having a look at this forecast, would you change anything in the planned development process? Would you consider performing code refactoring in order to prevent this increase?"

P3: "*This project is almost completed and minor changes and additions to functionality are expected. I would focus to solve any blocking, critical and major bugs to prevent software misbehavior during the use from the end user.*"

P2: "*Yes, code refactoring is of utmost importance for not only preventing TD increase but also to decrease TD principal future rate.*"

P4: "*Code refactoring is mandatory at this point but the manager does not approve that. That means that even if the development team wants to implement better architectures and refactor the code in order to be maintainable and re-usable this must be approved by the management. If the management has low priority on producing quality software the TD increase will be continuous. Also, if the code is mainly developed by junior engineers without guidance by senior engineers the TD will be incremental.*"

By revisiting Q3.4 in Part-3 of the questionnaire, we observe that this gradual TD Principal increase was attributed by the participants to rapid code expansion without properly testing previous and new implementations. This means that the developers invested more in new functionalities rather than refactoring the already existing code. However, in this question (Q4.1), the developers are presented with a forecast that predicts a gradual increase in the TD principal of the application during that period. Based on their answers, we notice that by having a look at this forecast they would reconsider this decision and proceed with refactoring instead of adding new code. In fact, by reading participants' answers to Q4.1, we observe that all respondents are actually willing to take action in order to reduce the TD Principal that is expected to increase based on the forecast shown in Figure 23. More specifically, the first two respondents state that they would perform refactoring to prevent future TD principal increasing rate and solve any defects that might arise to hinder the expected software behavior. The third respondent also agrees that code refactoring is mandatory at that point in order to prevent this increase. However, he/she states that any deviation from the planned development process must first be approved by the project manager. As a matter of fact, the perfect balance between repaying TD (and therefore increasing software quality) and reducing time to market of a software project is usually hard to achieve and lies at the decision-making abilities of the manager. This confirms the fact that future evolution of software quality depends heavily on business-related parameters such as planned features, release deadlines etc.

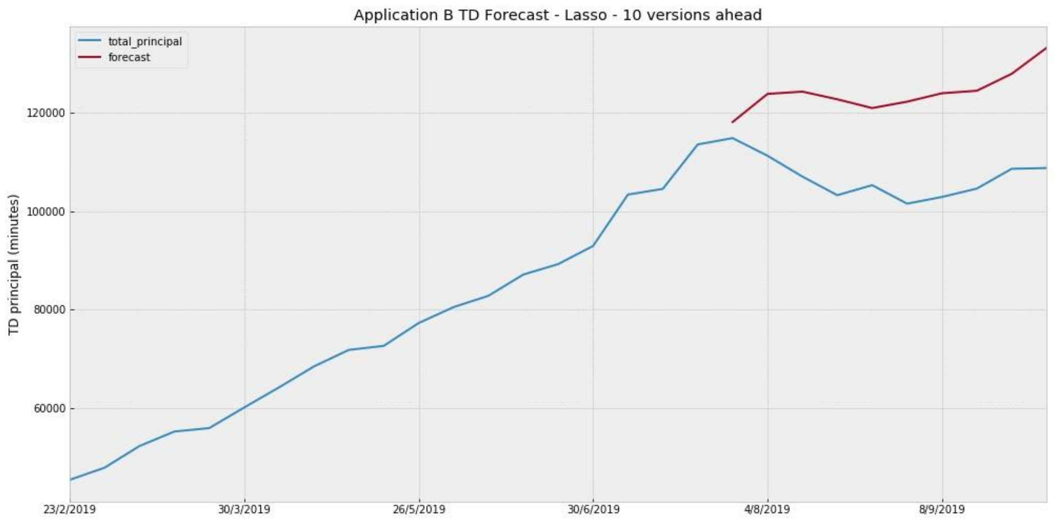


Figure 24: Project B TD Principal forecasting for 10 steps ahead using Ridge (the ground truth from the starting point of the forecasts and onwards was hidden from the participants)

Q4.2 "The latest commits of Application B show a gradual increase in the TD evolution. However, the forecast for 10 commits ahead shows a slight decrease in TD principal increasing rate. By having a look at this forecast, would you change anything in the planned development process? Would you consider investing in enhancing already existing, or adding new functionalities?"

P1: "*For sure, we already have done some critical refactorings of code to decrease TD and from now on, that we have better handling of the project, we reassure that each line of code written, will not increase TD, but if it does, we apply refactors at the end of a sprint.*"

P3: "*This is a still developing project with lots of functionality to be added. Ideally, I would pause the developing process and refactor the existing code. However since future TD seems to decrease I want to try to decrease the TD in the new code, then when time plan allows it go back and refactor problematic areas.*"

By revisiting Q3.6 in Part-3 of the questionnaire, we observe that this slight decrease in TD principal was attributed by the participants to code refactoring and defects fixing that the developers performed during the summer period. This means that during a relatively relaxed period, they invested time in cleaning up the already existing code and therefore decreasing the continuously growing TD Principal rate. While there is nothing wrong with this strategy, an alternative solution could be to invest in writing new but TD-free code instead of refactoring the existing one. In fact, writing new 'clean' code could be (in the long term) as efficient as code refactoring, especially when considering the difficulty of introducing heavy refactoring cycles in the industry, due to time limitations. In this question (Q4.1), the developers are presented with a forecast that predicts a slight decrease in TD principal increasing rate of the application during that period. Based on their answers, we notice that by having a look at this forecast they would reconsider this decision and proceed with adding new TD-free code instead of refactoring the existing one. In fact, by reading participants' answers to Q4.2, we observe that both respondents state that they are willing to take advantage of the forecasted slight decrease in TD principal in order to pause refactoring activities and start adding new functionalities to the software application, by focusing more on adding new TD-free code. This would result in both preventing TD from increasing and at the same time delivering new functionalities. More specifically, the first respondent states that she/he would reassure that each new line of code written will not increase TD, but in case it does, they will apply refactoring at the end of the sprint. Similarly, the second respondent states that ideally she/he would pause the developing process and apply refactoring. However, since there is a lot of pending functionality to be added and the forecast shows a slight TD decrease, she/he would focus more on writing new TD-free code.

Through the industrial study reported in this section, we have a first level of validation that a) actual TD trends reflect the circumstances and/or decisions taken during past development, and b) forecasts derived through well-studied ML-models can provide valuable insights and affect developers' decisions regarding the evolution of software. Of course, further research would be needed to solidify any claims on the usefulness of TD forecasting approaches, taking into account the numerous human- and business-related factors that drive the evolution of any software project.

5 Limitations and Threats to Validity

In this section, we discuss the limitations and validity threats of this empirical study. The accuracy of any forecasting model is by definition constrained, especially in the software domain, where future evolution of software quality depends heavily on numerous business-related factors such as planned features, release deadlines and fluctuations in the size of the development team. Therefore, anticipating such scheduled or unanticipated events would be a challenging endeavor beyond the scope of our study. We believe that over longer time horizons, repeating phenomena are captured by a project's history and building a prediction model based on historical data can provide some knowledge on future evolution. As described in Section 4, we have performed a study in an industrial setting to investigate the value of predictions regarding TD evolution. Nevertheless, we acknowledge the inability of the proposed approach to take into account planned or unforeseen business-related events. Apart from the aforementioned limitations, the methodology proposed in this chapter suffers from the usual threats to external and internal validity.

External validity refers to the ability to generalize results. The results of the study are unavoidably subject to external validity threats, since the applicability of ML models to forecast TD is examined on a sample set of 15 applications. It is always possible that another set of applications might exhibit different phenomena. Nevertheless, the fact that the selected applications are quite diverse with respect to application domains, size, etc. partially mitigates threats to generalization. In addition, a large part of the proposed methodology consists of constructing forecasting models that learn from past versions and therefore can be easily adapted to any software application, as long as sufficient and reliable historic data are available. A similar threat stems from the fact that our dataset consists of open source Java applications, thus limiting the ability to generalize the conclusions to applications of a different domain or programming language. However, the process of building TD forecasting models described in this chapter primarily builds upon the output of the tools used to compute software-related metrics that can act as indicators of the quality attribute of TD. This means that the proposed models can be easily adapted to forecast the TD of applications that are coded in a different programming language, as long as there are tools that support the extraction of software-related metrics that can act as TD indicators for the respective language. This also contributes to mitigating threats to generalization. However, since the dataset does not include industry applications, we cannot make any speculation on closed-source applications. Commercial systems as well as other object-oriented programming languages can be the subjects of further research. Finally, another possible threat to external validity is the small sample size of the survey performed within the context of the case study of this work, as reported in Section 4. In particular, the low number of participants and the small number of investigated software applications used for validation may have insufficient power to provide valuable insights regarding the meaningfulness of the proposed TD forecasting methodology in practice.

Concerning the *internal validity*, i.e., the possibility of having unwanted or unanticipated relationships between the parameters that might affect the variable that we are trying to predict, it is reasonable to assume that numerous other metrics that affect TD might have not been taken into consideration. However, the fact that we constructed our initial set of TD predictors based on software-related metrics that have been widely used in the literature as indicators of the presence of TD, such as OO software metrics, code smells and code issues extracted from ASA tools, limits this threat. Regarding the final selection of TD predictors, if we had limited our feature selection analysis to only correlations between the TD estimates and software-related metrics acting as TD predictors, then there would have been a threat to internal validity. However, we attempted to mitigate this threat through the use of univariate and multivariate regression analysis to further explore the relationships between the dependent and independent variables. Furthermore, in order to study the statistical significance of each indicator over the TD quality and be able to safely perform feature selection, we maximized diversity and representativeness by extending our dataset with additional 210 different heterogeneous applications.

Construct validity refers to the meaningfulness of measurements and that the independent and dependent variables are represented correctly. In this study, the main threats related to construct validity are due to possible inaccuracies in the identification of software-related metrics acting as TD indicators, as well as the identification and quantification of TD itself. In order to mitigate this risk, we decided to use two well-known and widely used tools, namely SonarQube and CKJM Extended. It should be noted that both of these tools were used as a proof of concept of the proposed forecasting methodology. The forecasting approach described in the present study is not dependent on the selected tools, as it could be applied to the measurements produced by other tools, based on user preference. However, the results presented in this study depend on the measurements obtained by these tools and, consequently, on the tools themselves. Therefore, more experimentation is required to assess the correctness of results obtained via other tools. As for the experimented prediction models, we exploited the ML algorithms implementation provided by the scikit-learn library, which is widely considered as a reliable tool. System-level forecasting also poses a threat to the validity of the findings as a tool for guiding TD repayment. In order for the refactoring activities to be more effective, recommendations for TD repayment need to be made at lower levels of granularity (e.g., class-level). However, the main goal of the proposed forecasting approach is to help developers and project managers make high-level decisions on whether there is a need to perform TD repayment in the next period of the project in general, and not to provide fine-grained recommendations on which software components the repayment activities should be focused.

Finally, *reliability* threats concern the possibility of replicating this study. To facilitate such replication studies, we provide an experimental package containing both the dataset and the scripts that were used for our analysis and forecasting model construction. This material can be found online[41]. Moreover, the source code repositories of the 15 selected projects are available on GitHub to obtain the same data.

6 Implications to Researchers and Practitioners

To the best of our knowledge, this is the first study in the field of TD that examines the applicability of ML models for TD forecasting. Across the 15 independently developed open source Java projects, our analysis indicates that linear Regularization models and the non-linear Random Forest regression are able to provide meaningful forecasts of TD evolution, and in most of the cases, with a sufficient level of accuracy. This work has significant implications for both research and practice, despite the limitations noted in the previous section.

6.1 Implications for Research

Through our study, we identified some interesting open issues that should be addressed through further research. In particular, although there has been extensive research with respect to predicting the evolution of individual software features, quality attributes, and quality properties that are directly or indirectly related to the TD of a software project, no concrete contributions exist in the related literature regarding TD forecasting. Therefore, we believe that this study has a high impact on the scientific community and therefore, we suggest and encourage researchers to further explore this direction. An interesting topic of future work would be to extensively evaluate TD forecasting techniques on a broader spectrum of real-world software applications covering different domains or programming languages. In addition, it would be useful to investigate different efficient ways to produce forecasting models for accurate prediction of TD principal and interest evolution, by bringing into the equation other types of software repositories that could be a potential source of TD related data, such as project management and issue-tracking systems. More specifically, mining TD related data from project-and issue-tracking systems, such as the reported effort of fixing bugs on Bugzilla or closing issues on Jira, could provide valuable information towards enhancing the TD forecasting approach. Ultimately, an approach that would pair all the above information with specialized techniques for forecasting, code

[41] https://sites.google.com/view/technical-debt-forecasting/main

analysis, software evolution analysis, and natural language processing could pave the way for the advance in the state of the art in this domain.

Predicting the future value of TD interest would be also critical for decision making, as it can be used to timely determine the point at which a software product would become unmaintainable, and therefore to respond promptly through appropriate refactoring activities in order to prevent this situation. More specifically, an interesting way of approaching the problem of TD interest forecasting would be to examine whether forecasting techniques could contribute towards enhancing the process of identifying the "breaking point" of an application, a term introduced by Chatzigeorgiou et al. [116] that refers to the point in time where the accumulated interest is equal to the TD principal and, thus, the cost becomes higher than the benefit.

6.2 Implications for Practice

Monitoring and forecasting the evolution of TD is highly important for software development companies, as it can provide valuable information regarding the effort and, in turn, the cost that is required for maintaining and extending a software application. Therefore, a TD forecasting methodology integrated into a relative tool, such as the outcome of the present research work, could be crucial for companies that want to remain competitive, while taking planned decisions regarding their TD management activities. In a hypothetical scenario where a software company has to make an investment to a specific application, our TD forecasting tool could provide an effective method to facilitate planning for budget and time allocation. More specifically, when the model predicts a declining number of TD, a project manager can then proactively allocate resources to software enhancements, or to other projects in more need. When the model predicts an increase in TD, an organization a priori can allocate the resources needed to quickly repay it by taking actions such as post development refactoring activities. This research has therefore the potential to make a great economic impact by helping software companies save budget by foreseeing TD accumulation and therefore avoid a potential bankruptcy in the future.

This empirical study has focused exclusively on modelling software evolution at the system level, thus allowing project managers to efficiently prioritize TD activities when dealing with different software applications. When it comes to a specific application however, the developers are often overwhelmed with a large volume of TD liabilities (e.g., code smells, bugs, vulnerabilities, etc.) that they need to fix. This renders the TD repayment procedure tedious, time consuming and effort demanding. In such cases, the significance of prioritizing which software components to refactor is highlighted even further, since fixing TD items in dormant parts of the code does not effectively affect maintenance costs [171]. As a future work, we will investigate the possibility of extending TD forecasting techniques to lower levels of granularity of a software project (e.g., package, class or function level). This would enable for a more granular prioritization of TD liabilities by incorporating information retrieved from TD forecasting techniques, allowing for a ranking of a software project's artifacts based on predictions of their long-term accumulated TD values.

7 Conclusions and Future Work

Technical Debt (TD) refers to inefficiencies during all phases of software development lifecycle that lead to extra maintenance effort. In recent years, TD has attracted the attention of both academia and industry. As a result, there has been a considerable increase in the number and provided functionality of methods and tools that support TD management. TD repayment, a high-level activity of TD management aims to resolve or mitigate TD in a software system by techniques such as reengineering and refactoring. However, a decision on whether or not to repay a TD item has different consequences depending on when it is made. This stresses the need for methods and accompanying tools that would enable system engineers and project managers to perform long-term effective software maintenance, by providing insights regarding where and when to apply refactoring. Therefore, what the stakeholders require is a decision-support system to help them make such choices and support decision-making under uncertainty. Under those circumstances, a method or tool able to track and forecast the evolution of TD of a software system could lead to the development of practical decision-making mechanisms aiming to improve the TD repayment strategy and estimate the point in which a software product could become unmaintainable.

The purpose of this chapter is to examine whether and to what extent is the usage of ML models a meaningful and accurate approach to forecasting TD Principal in long-lived, open-source software applications (RQ1). Across the 15 independently developed, maintained, and managed open source projects, we have shown that TD Principal patterns can be modeled adequately by ML techniques. More specifically, for forecasting horizons between 1 and 20 weeks ahead, Regularization models (i.e., Lasso and Ridge regression) are able to fit and provide meaningful forecasts of TD Principal evolution. Trying to forecast longer into the future however, we noticed that their predictive power drops significantly. On the contrary, the non-linear Random Forest regression seems to have an almost stable performance over the holdout sample for all examined steps ahead. In fact, for forecasting horizons longer than 20 weeks ahead, Random Forest regression was able to capture the trend of the future evolution of the TD Principal with higher predictive power compared to the linear models. This indicates that a more complex model performs better when the forecast horizon is longer. From the above analysis, we can conclude that ML models constitute a suitable and effective approach for TD Principal forecasting, as they are able to fit and provide meaningful estimates of TD evolution over a relatively long period, while in most of the cases, the future TD value is captured with a sufficient level of accuracy.

To the best of our knowledge, this is the first study in the field of TD that examines the applicability of ML models for TD forecasting and therefore, it constitutes a good basis for future research and experimentation. Future work includes the extensive evaluation of TD forecasting techniques on a broader spectrum of real-world software applications, as well as to lower levels of granularity of a software project (e.g., package, class or function level). To solidify any claims on the usefulness of TD forecasting approaches, we plan to conduct an extended case study where we will provide practitioners with future predictions, track the actual development, and observe the impact of the forecasting results adoption. We also plan to investigate the ability of already examined or new forecasting models to provide more accurate predictions for even longer forecasting horizons. Last but not least, we plan to investigate other types of software repositories that could be a potential source of TD related data, such as project management and issue-tracking systems, as well as archived communication between project personnel. More specifically, the analysis of the communication between project personnel could reveal indications of high-TD artifacts that concentrate a large part of maintenance effort. These indications could then be factored in TD forecasting techniques to target these critical - from a maintenance point of view - artifacts. In fact, we believe that there is great potential in mining this information to achieve source triangulation and thus, yield more accurate TD forecasting estimates.

Chapter VI Technical Debt Prioritization and Repayment based on Forecasting Techniques

“*Some problems with code are like financial debt. It’s OK to borrow against the future, as long as you pay it off.*”

Ward Cunningham - American computer programmer

Chapter Summary

Monitoring Technical Debt (TD) is considered highly important for software development companies, as it provides valuable information on the effort required to repay TD and in turn maintain the system. When it comes to TD repayment however, developers are often overwhelmed with a large volume of TD liabilities that they need to fix, which renders the procedure tedious, time consuming, and effort demanding. Hence, prioritizing TD liabilities is of utmost importance for effective TD repayment. To this end, in the present chapter, we propose a practical approach for granular prioritization of TD liabilities by incorporating information retrieved from TD forecasting techniques, emphasizing on the class-level of granularity to provide highly actionable results. More specifically, the proposed approach takes into account the change proneness and forecasted TD evolution of software artefacts and combines it with proper visualization techniques, to enable the early identification of classes that are more likely to become unmaintainable. To demonstrate the proposed approach and evaluate its usefulness, a case study is conducted based on a real-world open-source software application. This work is expected to facilitate project managers and developers better plan their refactoring activities, in order to manage TD promptly and avoid unforeseen situations long-term.

1 Introduction

Technical Debt (TD) [1] is commonly used to indicate quality compromises that can yield short-term benefits in the software development process, but may negatively affect the long-term quality of software. In a software affected by TD, refactoring is the only effective way to reduce it on existing source code. However, companies usually cannot afford to repay all the TD that is generated continuously [24]. In fact, TD issues need to be translated into economic consequences before choosing the TD items that should be removed. In addition to this, strict production deadlines often force companies to focus on the delivery of new functionality, reducing the time that they can invest on TD repayment activities, leading to the accumulation of TD. Therefore, before applying refactoring activities, it is necessary to identify which items should be resolved first, by prioritizing them based on their TD values, but also based on business’s objectives and preferences [25].

Among the various TD Management activities, TD Prioritization is considered one of the most important. The TD prioritization process is used for scheduling of planned refactoring initiatives, by ranking identified TD items according to certain predefined rules to support decisions regarding which TD items should be repaid first and which TD items can be tolerated until later releases. Several different prioritization approaches and methods have been proposed by researchers on how to prioritize TD. Regarding code TD, prioritization is mostly based on code smells [26], [27]. In addition, other metrics such as time [28], cost to fix a violation [29], and quality rules [30] have also been considered. However, while all previous research works investigate the feasibility of prioritization and repayment of TD liabilities based on historical TD data, to the best of our knowledge, there are no research endeavors considering the future evolution of TD to address this issue.

These facts stress the need for a method or tool that would provide companies with insights regarding where and when to apply refactoring activities. Therefore, what the stakeholders require is a decision-support system (DSS) to help them make such choices and support long-term effective TD repayment. However, a decision on whether or not to repay a TD item has different consequences depending on when it is made. For instance, the cost of refactoring a component in the current release is different than the cost of refactoring the same component in a future release [172]. A DSS that would also take into account the future TD evolution by incorporating TD forecasting techniques could prioritize TD items not only based on their current TD, but also based on their potential future TD accumulation.

To this end, in the present chapter, we propose a practical approach for a more granular prioritization of TD liabilities by incorporating information retrieved from change-proneness analysis and TD forecasting techniques. The proposed approach considers both the frequency of changes and the future TD evolution to enable the early identification of classes that are more likely to become unmaintainable in the future, and therefore to allow prompt refactoring of their liabilities. It also emphasizes on the class-level of granularity, since it provides highly actionable results to the developers and project managers, compared to system-level of granularity.

In order to demonstrate our approach, we conducted a case study on an open-source software application, namely Apache Kafka. During the first step, we retrieved the artifacts (i.e., classes) of a software application with their history (past commits). Next, we analyzed these artifacts using a static analysis platform in order to calculate several TD measures for the construction of the initial class-level dataset. Afterwards, we applied a change proneness analysis to detect the most change-prone classes of the given software application in terms of size and TD. We filtered out the classes whose change proneness was below a predefined threshold or were not suitable for the construction of forecasting models. Then, we used the remaining data to build class-level TD forecasting models and assess the TD evolution of the selected classes. Finally, we used visualization techniques to facilitate the understandability of the results and, in turn, the decision-making tasks of the developers and project managers regarding the repayment of the identified TD liabilities. To provide confidence that proposed prioritization approach captures the actual criticality of the project's classes from a TD viewpoint, we performed a qualitative evaluation using the project's Jira issue tracker.

The rest of the chapter is structured as follows: Section 2 thoroughly describes the proposed methodology, while Section 3 presents the empirical validation of the methodology, using a case study based on a real-world application. Finally, Section 4 concludes the chapter and discusses ideas for future work.

2 Approach and Methodology

The TD prioritization approach that we propose in the present chapter is inspired by the approach introduced by Guo et al. [173]. According to their methodology, TD artifacts are initially identified and their respective TD measures are computed. Subsequently, based on these measures, a decision on which TD items should be repaid or ignored is reached. In our approach, emphasis is given on class-level of granularity, whereas the future value of TD is also considered for the prioritization of the TD liabilities of the analyzed software. The proposed approach for TD liabilities prioritization is summarized in Figure 25.

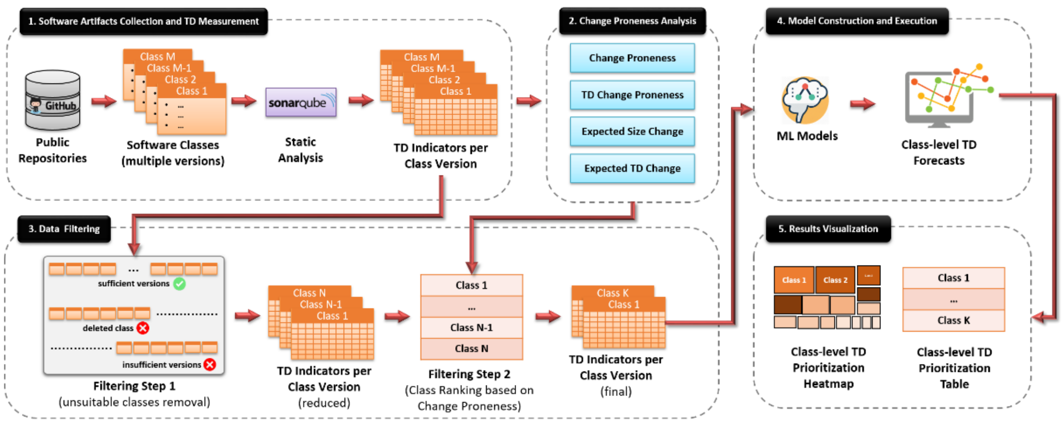


Figure 25: Chapter VI roadmap

As can be seen by Figure 25, our approach comprises five steps:

1. **Software Artifacts Collection and TD Measurement**. During the first step of our approach, the source code of a software application is retrieved from a code repository, along with its history (i.e., past commits). Next, the artifacts (i.e., classes) of the retrieved versions of the application are analyzed using a static analysis platform and several TD measures are calculated for the construction of the initial class-level dataset.

2. **Change Proneness Analysis**. During the second step of our approach, a change proneness analysis is applied on the initial class-level dataset in order to detect the most change-prone classes of the given software application in terms of both size and TD.

3. **Data Filtering**. During the third step of our approach, a two-step filtering process is applied on the dataset. First, it filters out the classes that are not interesting from a TD viewpoint (i.e., they are not modified frequently) or they are not suitable for the next steps of the analysis (i.e., due to insufficient version history for the construction of forecasting models). Second, it ranks the classes based on their

calculated change proneness and keeps only those that their change proneness is above a predefined threshold.

4. **Model Training and Execution**. During the fourth step of our approach, the data that pass the *Data Filtering* step are used to build class-level TD forecasting models by using Machine Learning (ML) methods. These models are then applied to assess the TD evolution of the selected classes.

5. **Results Visualization**. Finally, during the fifth step of our approach, the forecasting results are visualized through dedicated graphs to facilitate their understandability, and, in turn, the decision-making tasks regarding the prioritization of the identified TD liabilities.

More details about each step of the proposed approach are provided in the rest of this section.

2.1 Software Artifacts Collection and TD Measurement

As already mentioned, this step is responsible for the collection of all the artifacts of a selected software application along with their history, as well as for the calculation of important artifact-level TD indicators. Since the proposed approach operates at class-level of granularity, the type of artifacts that we emphasize on are software classes. The reasoning behind the selection of this type of artifacts is that class-level of granularity usually leads to predictors with more practical results [41]. In fact, the higher the level of granularity, the lower the practicality of the produced results, since the focus of the developers is not pointed to an actionable subset of TD liabilities. Software classes are the main constructs that developers work on, whereas the number of the liabilities that they contain is usually manageable. At this point, it should be also noted that the history (or at least an adequate number of past instances of each class) is also required for the construction of the class-level TD forecasting models that are produced in Step 4 of the approach (see Figure 25), as well as for the final prioritization of TD liabilities.

After retrieving all the classes, the *TD Principal* of each class of the selected software application is calculated. TD Principal is quantified, in most of the approaches that exist in the literature, by summing up the estimated effort to fix each individual inefficiency that is identified through automated analysis tools [78]. SonarQube[42], a popular open source platform for static code analysis and continuous inspection of code quality, is used for the calculation of the *TD Principal*, since according to recent studies [3], [5], it is the most frequently used tool for estimating and monitoring TD. Due to the adoption of SonarQube, the *TD Principal* of each class is actually the summation of the *Reliability Remediation Effort* (i.e., sum of remediation effort of bugs), the *Security Remediation Effort* (i.e., sum of remediation effort of vulnerabilities), and the *TD Remediation Effort* (i.e., sum of remediation effort of code smells). However, it should be noted that the proposed approach

[42] https://www.sonarqube.org/

is platform-agnostic, and therefore the *TD Principal* can be calculated based on any TD platform (or tool) of choice.

In the approach presented in this chapter, in addition to *TD Principal*, we also considered various TD indicators (capable of acting as TD predictors) that should be included as input to the TD forecasting models, in order to enhance their predictive performance. Since SonarQube calculates *TD Principal* based on different issue categories, similarly to a recent relevant study [15] on TD forecasting, we opted for the TD-related metrics that are provided by this tool as our primary *TD Principal* predictors. As a result, additional TD indicators are also computed for each class of the selected software application. More specifically, for each class we also compute the number of: (i) *bugs*, (ii) *code smells*, (iii) *vulnerabilities*, and (iv) *lines of duplicate code*, among others.

The same measurements are also applied for each commit of the classes of the selected software application. This step results in a long dataset of classes with TD-related measurements, which render valuable sources for the construction of class-level TD forecasting models.

2.2 Change Proneness Analysis

This step is responsible for assessing the change proneness of the classes of the selected software application. The change proneness of a given class is of high interest from a TD viewpoint since the frequent changes that are applied to a class increase the probability of TD accumulation, mainly due to the potential introduction of quick fixes, bugs, vulnerabilities, and code smells [174]. Several studies have shown that change-prone artifacts are more likely to contain important bugs and vulnerabilities [166], [175].

In addition to this, the change proneness is also valuable for selecting the classes that are more suitable for the construction of class-level forecasting models. Attempting to apply forecasting techniques on classes that have little to no change in lines of code as well as in TD across multiple versions would not be efficient because their projected evolution would usually be identical to previous versions. Furthermore, classes that have not been changed frequently (or at all) in the past are less probable to undergo maintenance in the future, and thus, fixing their violations is less urgent to a development team.

Hence, we decided to consider the change proneness of the classes in the filtering process of the proposed approach (described in Section 2), and, in turn, in the final prioritization of the TD liabilities. More specifically, as will be discussed in Section 2.3, we included the results of the change proneness analysis to apply a second filtering process with the purpose to remove classes that are not so change prone either in terms of Lines of Code (LoC), or in terms of their TD values.

The first step of the *Change Proneness Analysis* is to define the metrics on which the analysis will be based. Initially, we define the *Change Proneness* (*CP*) of a class as the metric which represents the probability of this class to change in the next version with respect to its Lines of Code (LoC). This is a statistical measure derived by dividing the number of examined commits where changes in the LoC were observed ($n_{altered}$), to the total number of examined commits (n_{total}) of the corresponding class:

$$\mathrm{CP} = \frac{n_{altered}}{n_{total}}$$

Based on this metric, we also define the *TD Change Proneness* (*CP_{TD}*) of a class, which corresponds to the probability of the TD of the class to change in the next commit. Similarly to the *CP* metric, *CP_{TD}* is derived by dividing the number of the examined commits in which an alteration in the TD of the selected class was observed *$n_{TDaltered}$*, to the total number of its examined commits (*n_{total}*):

$$\mathrm{CP}_{TD} = \frac{n_{TDaltered}}{n_{total}}$$

Apart from the probability of change, we are also interested in knowing how much the LoC and the TD of a given class change on average between examined commits. These values are useful for TD repayment planning since they provide an estimate of the magnitude of change that is expected to be observed in the next commit of a given class.

More specifically, we define the *Expected Size Change* (*$E[D_{LOC}]$*) of a class as the average change in its size (expressed in LoC) that is observed between two sequential examined commits. This metric is given by the following formula:

$$\mathrm{E}[\mathrm{D}_{LOC}] = \frac{\sum D_{LOCi}}{n_{total} - 1}$$

where:

- *D_{LOCi}*: The total LoC of the class that changed between commit *i* and commit *i-1*
- *n_{total}*: The total number of examined commits of the selected class

Similarly, we define the *Expected TD Change* (*$E[D_{TD}]$*) of a class, which corresponds to the average change in its TD that is observed between two sequential examined commits. This metric is given by the following formula:

$$\mathrm{E}[\mathrm{D}_{TD}] = \frac{\sum D_{TDi}}{n_{total} - 1}$$

where:

- *D_{TDi}*: The total TD of the class that changed between commit *i* and commit *i-1*
- *n_{total}*: The total number of examined commits of the selected class

At this point it should be noted that although the latter two metrics are not used directly by the proposed approach for the filtering of the selected classes (see Section 2.3), reporting these values can be proved very useful for facilitating decision making regarding the TD repayment planning. In fact, these values can actually supplement the results of the proposed approach, in order to help the developers and project managers make more informed decisions.

2.3 Data Filtering

This step is responsible for preparing the final dataset that will be used for the construction of the class-level TD forecasting models in the next step of the proposed approach (see Section 2.4). More specifically, the *Data Filtering* step is responsible for (i) removing classes that are not suitable for the construction of TD forecasting models, and (ii) keeping the classes that are more interesting from a TD viewpoint. Hence, a two-step approach is adopted for the filtering of the classes of the selected software application.

As already mentioned, the first step of the *Data Filtering* process is responsible for removing classes that are not suitable for the construction of TD forecasting models. First of all, classes that do not have sufficient version history (i.e., a sufficient number of past commits) are removed from the analysis, since training forecasting models for predicting the TD evolution of individual classes requires a substantial number of past instances. More specifically, in our approach, we exclude classes that the number of their past commits is below a specific threshold. This threshold is defined in a heuristic manner by taking into account the specific characteristics of the corresponding software application under analysis. For instance, in the case study described in Section 3, since 150 commits of the overall software application were examined, classes that had fewer than 100 past instances were excluded from the analysis. Apart from the version history, the proposed approach also eliminates classes that are not present in the latest commit of the software application. This is reasonable since these classes no longer exist in the code base of the application, and therefore they are not of interest for the developers of the application.

The second step of the *Data Filtering* process is responsible for identifying and keeping classes that are more interesting from a TD viewpoint. As already mentioned, classes that are modified frequently are more likely to affect the future value of the TD of the corresponding software application, as these source code modifications normally lead to an alteration (either positive or negative) of the class's TD. Hence, the results of the *Change Proneness Analysis* (described in Section 2.2) are exploited, for the final selection of the classes that will be used for the production of class-level TD forecasting models.

More specifically, the final classes that passed the first step of the *Data Filtering* process are ranked based on their *Change Proneness* (*CP*) metric in a descending order. Subsequently, the top *N* classes are selected to be part of the final dataset that will be used for the construction of the class-level TD forecasting models. The value of *N* is defined by the user (e.g., developer) based on the number of classes that he/she would like to have TD forecasts. It should be noted that instead of the *CP* metric, the *TD Change Proneness* (CP_{TD}) metric can be used as a measure of the class change proneness. Several empirical evaluations that we performed revealed that in the vast majority of the cases, a statistically significant strong correlation exists between the two metrics, and therefore they can be used interchangeably for measuring change proneness. An example of this empirical evaluation is provided in the use case that is described in Section 3.

2.4 Model Construction and Execution

The final dataset that is produced by the *Data Filtering* step is provided as input to the *Model Construction and Execution* step. This step is responsible (i) for the construction of a class-level TD forecasting model for each one of the classes of the received dataset, and (ii) for the execution of the produced forecasting models in order to retrieve class-level TD forecasts.

The procedure that is adopted for the construction of the class-level TD forecasting models is as follows. For each one of the selected classes, their commit history is retrieved along with their TD metrics that were computed in a previous step of the overall process. The resulting class-specific dataset is restructured based on the “sliding window” approach [162], in order to transform it in a format that is suitable for supervised ML tasks. In short, this method extends each initial sample of the dataset by including past information (i.e., multiple prior time steps as inputs (X)) and future information (i.e., future time steps as output (Y)) simultaneously into a single row.

Subsequently, several linear, non-linear, and ensemble ML models (e.g., Linear Regression, Support Vector Regression, Random Forest, etc.) are built and tested on the dataset for various forecasting steps ahead (since commits were collected in weekly intervals, steps actually refer to weeks). To better assess prediction accuracy of the produced models, the Walk-forward Train-Test validation [124] is adopted. The produced models are compared based on three different performance metrics, particularly the *Root Mean Square Error (RMSE)*, the *Mean Absolute Error (MAE)*, and the *Mean Absolute Percentage Error (MAPE)*. The model that demonstrates better values in these three metrics is selected as the TD forecasting model of the given class.

The aforementioned approach is repeated for each one of the *N* classes that were selected by the *Data Filtering* step. This process results in *N* class-level TD forecasting models. Finally, the produced models are applied to their corresponding classes, in order to calculate the future value of their *TD Principal*. This value is necessary for the final prioritization of the TD liabilities, which is achieved through appropriate visualization techniques.

2.5 Results Visualization

The purpose of the proposed approach is to help the developers and project managers of a software application better prioritize their TD repayment activities. To better prioritize the TD repayment activities, the results of the proposed approach need to be properly visualized, so that the underlying information is effectively conveyed to the developers and project managers of the software application, assisting them in making more informed decisions regarding TD repayment. Several approaches for visualizing the produced results can be adopted. In the proposed approach, emphasis is given on heat maps. An example of a heat map that can be produced by our approach is depicted in Figure 26.

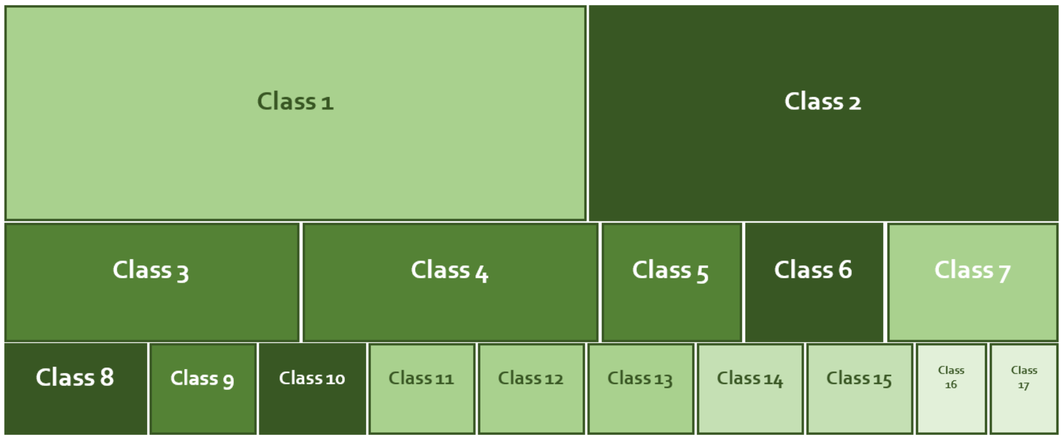


Figure 26: Heat map visualizing the future value of the TD Principal and the change proneness of the selected classes

As can be seen in Figure 26, the heat map consists of a number of rectangles. Each rectangle corresponds to a specific class of the software application. The size (i.e., area) of the rectangle is proportional to the future value of the *TD Principal* of the class as reported by the associated class-level TD forecasting model. The color of the rectangle denotes the change proneness of the corresponding class. The greener the rectangle, the higher the probability to change in the next versions. Hence, the heat map allows the developers to take into account two different criteria for prioritizing their TD repayment activities, namely the future value of the TD Principal and the change proneness of the selected classes. For example, a class that is expected to have relatively higher *TD Principal* in the upcoming versions and that it is highly likely to change (e.g., Class 2 in Figure 26), may probably require immediate remediation actions compared to a class that changes less frequently (e.g., Class 1 in Figure 26), in order to avoid further TD accumulation.

It should be also noted that, apart from the heat map, a table comprising the detailed results of the analysis is considered necessary. This table should contain supplementary information including the additional metrics of the *Change Proneness Analysis* that were defined in Section 2.2. This additional information is expected to help the developers take even more informed decisions regarding the prioritization of their TD repayment activities.

3 Empirical Validation (Case Study)

In this section, the proposed approach is demonstrated through a case study on a real-world open-source software application. This case study also acts as a test bed for evaluating the correctness of the proposed approach, and for assessing the feasibility of generating class-level TD forecasting models of sufficient predictive performance.

3.1 Data Collection, Analysis and Filtering

3.1.1 Software Artifact Collection and TD Measurement

The data used in this study were obtained from a popular open source Java project, namely Apache Kafka[43]. Apache Kafka is a platform for building real-time data pipelines and streaming apps, whose code is hosted on GitHub[44] with more than 8.000 commits. As a first step towards building our dataset, we collected 150 versions (commits) in weekly intervals, spanning up to almost 3 years of the system's evolution (i.e., from 30/10/2015 to 7/9/2018). Subsequently, we used SonarQube in order to extract the specific TD indicators that act as predictors, namely number of *bugs*, *code smells*, *vulnerabilities*, and *lines of duplicate code*, as well as the *TD Principal* for each version of the classes across the 150 different commits. We calculated the *TD Principal* of each class as the summation of the *Reliability Remediation Effort*, the *Security Remediation Effort*, and the *Maintainability Remediation Effort*. By statically analyzing each version, we derived from the process 150 CSV files containing 1718 unique classes. This process resulted in a long dataset of multiple versions of classes with TD-related measurements, which render valuable sources for the construction of class-level TD forecasting models.

3.1.2 Change Proneness Analysis

After acquiring a long dataset of multiple versions of classes with TD-related measurements, the next step is to apply a *Change Proneness Analysis* in order to detect which are the most change-prone classes from the given software application. This process is of high interest not only from a TD viewpoint, but also for selecting the classes that are more suitable for the construction of class-level forecasting models. By following the *Change Proneness Analysis* process described in Section 2.2, we computed the 4 metrics of interest, namely *Change Proneness* (*CP*), *TD Change Proneness* (CP_{TD}), *Expected Size Change* ($E[D_{LOC}]$), and *Expected TD Change* ($E[D_{TD}]$) for each of the 1718 classes of our dataset.

As already mentioned, these metrics are very important for the *Data Filtering* step of the overall approach. More specifically, the *CP* and CP_{TD} metrics are actually used as the main criteria for the selection of the main classes of interest, which are chosen for the construction of class-level TD forecasting models. Apart from the *Data Filtering* step, these four metrics provide additional useful information to the developers and project managers of the software application, allowing them to reach more informed decisions regarding the relevant TD repayment activities.

3.1.3 Data Filtering

The *Data Filtering* step is responsible for preparing the final dataset that will be used for the construction of the class-level TD forecasting models. As already mentioned in Section 2.3, the first part of this step involves the removal of classes that are not suitable for the construction of TD forecasting models, i.e., they do not have sufficient commit history. In our approach, since 150 versions of the Apache Kafka application were examined, we decided to set the threshold to 100 versions. This number was computed in a heuristic

[43] https://kafka.apache.org/

[44] https://github.com/apache/kafka

manner, after applying dedicated experiments in order to assess what would be the minimum number of samples that would result in an acceptable forecasting error, when given as input into various ML forecasting algorithms. As a result of applying this filtering process, 1099 out of 1718 classes were filtered out, leaving us with 619 classes. Apart from classes with insufficient number of commit history, we also eliminated classes that were not present in the latest commit of the software application (i.e., 7/9/2018), as they no longer exist in the code base. This extra filtering step removed another 26 classes, leaving us with 593 classes for further analysis.

The second part of the *Data Filtering* step involves identifying and keeping classes that are modified frequently and therefore are more interesting from a TD viewpoint. Hence, we exploited the results of *Change Proneness Analysis* for the remaining 593 classes that passed the first filtering step, by ranking them based on their *Change Proneness* (*CP*) metric in a descending order so as to focus on the classes that are more likely to affect the future *TD Principal* value the corresponding Apache Kafka application. An indicative number of 10 classes along with their computed *Change Proneness Analysis* metrics ranked by *Change Proneness* (*CP*) in a descending order are presented in Table 22. The complete ranked set of the 593 classes can be found at the online supporting material [176].

Table 22: Change Proneness Analysis metrics for first 10 classes of the Apache Kafka ranked by CP

Class name	CP	CP_{TD}	$E[D_{LOC}]$	$E[D_{TD}]$
StreamThread	0.533	0.393	3.141	-0.597
Fetcher	0.400	0.240	4.148	1.168
StreamTask	0.387	0.120	2.161	0.101
KafkaConsumer	0.353	0.087	1.799	0.698
StreamsConfig	0.341	0.101	3.453	0.701
ConsumerCoordinator	0.320	0.133	1.732	0.530
KafkaProducer	0.313	0.127	1.718	-0.839
KafkaStreams	0.312	0.159	3.606	1.255
RocksDBStore	0.312	0.152	1.730	0.219
KTableImpl	0.283	0.166	2.278	1.097

It should be noted that alternatively, the CP_{TD} could have been used as the measure of the class's change proneness, and, in turn, as the basis for the ranking. However, by inspecting Table 22, we can observe that a correlation may exists between *Change Proneness* (*CP*) and *TD Change Proneness* (CP_{TD}) metrics. In order to reach safer conclusions, formal statistical testing was applied. More specifically, we ranked the selected 593 classes of Apache Kafka based on the *CP* and CP_{TD} metrics, leading to the generation of two individual rankings. Subsequently we compared the two resulting rankings in order to determine whether a statistically significant and strong positive correlation exists. For this purpose, we defined the following Null Hypothesis (H_0), along with its corresponding alternative hypothesis (H_1), and tested it in the 95% confidence interval:

- H_0: No statistically significant correlation exists between the two rankings
- H_1: A statistically significant correlation exists between the two rankings

In order to test the Null Hypothesis the *Spearman rank correlation coefficient* (ρ) was used, which is a non-parametric test, not affected by outliers. The calculated ρ was found to be 0.85, which is a positive and strong (according to Cohen et al. [155]) correlation. The *p-value* was found to be 0.0048, which is lower than the threshold of 0.05, which led us to the rejection of the Null Hypothesis, and, thus to the acceptance of the alternative hypothesis. As a result, we can conclude that a statistically significant positive and strong correlation exists between the rankings of *CP* and CP_{TD}, at least for the selected dataset.

Hence, this observation suggests that instead of the *CP* metric, the CP_{TD} metric can also be used as a measure of the class change proneness. However, in the remaining parts of this empirical validation study we decided to use the *CP* metric for measuring change proneness, as it is a well-known metric in the literature.

Qualitative Evaluation of TD Prioritization Approach

This chapter essentially proposes a practical approach that aims to facilitate refactoring activities planning by providing a granular prioritization of TD liabilities. Therefore, a qualitative evaluation of its usefulness in practice (i.e., to investigate whether the selected classes are actually critical from a TD viewpoint) would normally require seeking feedback from practitioners, that is, developers of the investigated Kafka application, in order to ask their opinion on how relevant is the proposed ranking of critical classes to the actual effort and, in turn, the cost that is required for maintaining and extending these classes. However, developers of open-source projects are difficult to reach and are usually unresponsive.

To overcome this obstacle, we decided to evaluate our approach through a proxy that would give us an insight into the actual development workflow. More specifically, we exploited the Jira[45] issue tracking system of the Apache Kafka with the purpose of investigating whether the specific ranking of classes as suggested by our approach is in line with the actual ranking of classes that were indeed critical for the developers, based on the reported fault information. In brief, we measured the criticality of a given class based on how many times it has been reported in the project's Jira issue tracker. To do so, we performed queries to the Jira API and we fetched the total number of issues related to each of the 593 classes through the investigated 3 years of the system's evolution (i.e., from 30/10/2015 to 7/9/2018). This information can be found online [176]. Subsequently, we ranked the selected 593 Apache Kafka classes based on the total number of Jira issues and compared the resulting ranking with the ranking based on the *CP* metric (described above) in order to determine whether a statistically significant and strong positive correlation exists. We defined the Null (H_0) and alternative hypothesis (H_1), and tested it in the 95% confidence interval.

Again, in order to test the Null Hypothesis we used the *Spearman rank correlation coefficient* (ρ). The calculated ρ was found to be 0.741, which is a positive and strong (according to Cohen et al. [155]) correlation. The *p-value* was found to be 4.32e-104, a value significantly lower than the threshold of 0.05, which led us to the rejection of the null hypothesis, and thus, to assume that a statistically significant positive and strong

[45] https://bit.ly/36VLV0K

correlation exists between the rankings of Jira issues and *CP* between the 593 selected classes.

Hence, this observation suggests that the classes that are prioritized higher by our approach are really a problem for the developers of the analyzed software. More specifically, the classes that are presented to the user and used for the construction of TD forecasting models are highly likely to correspond to classes that have been frequently revisited by the developers in the past in order to fix TD-related issues. This provides confidence that the prioritization that is proposed by our approach captures the actual criticality of the project's classes from a TD viewpoint.

3.2 Model Construction and Execution

The *Model Construction and Execution* step is responsible for the construction and execution of a class-level TD forecasting model for each one of the classes of the received dataset in order to retrieve class-level *TD Principal* forecasts. For reasons of brevity, we decided to focus on the top 10 classes of the Apache Kafka software application in terms of *CP* metric (depicted in Table 22) as part of the dataset used for the construction of the class-level TD forecasting models. A snapshot of this dataset, including TD-relevant metrics for the selected 10 classes can be found online [176]. We believe that the selected number of classes is sufficient for evaluating the correctness of the proposed approach (i.e., the feasibility of constructing class-level TD forecasting models), as well as for demonstrating the overall usefulness of the proposed TD prioritization methodology. However, the reader can easily replicate the present analysis using a much larger number of software classes, depending on their preferences. In addition, in case that the proposed approach is incorporated by a dedicated tool, the user may be equipped with the option to manually define the number of classes for which they would like to have TD forecasts.

3.2.1 Model Construction

As an initial step towards constructing the class-level TD forecasting models, we retrieved the examined commit history for each of the selected classes, along with their *TD Principal* values and TD metrics that act as predictors, extracted during *Software Artifact Collection and TD Measurement* step of the overall process (see Section 3.1.1). Subsequently, due to ML models not directly supporting the notion of observations over time, we restructured each class-specific time series dataset in order to be suitable for supervised learning. To do so, we used the “sliding window” method that in short, extends each sample (class version instance) of the class-specific dataset by including past information and future information simultaneously into a single row. We found out that choosing a sliding window of size = 2 resulted in the minimum *MAPE* when trying to forecast for 5 steps ahead, for most of the class-specific datasets across different models. This means that two past lag values plus the current lag will be used to forecast future values. Respectively, for longer forecasting horizons (e.g., 10 steps ahead), a larger window appeared to be more suitable and resulted in better model performance.

After performing the above data restructuring process, we split each class-specific dataset into training and test sets. In particular, to better assess model prediction accuracy and at the same time respect the temporal order of our class-level time series data, we adopted the Walk-forward Train-Test validation, a commonly used way to evaluate time series models performance, based on the notion that models are updated when new observations are made available. The Apache Kafka class-level dataset is comprised of 150 observations per class. For Walk-forward Train-Test validation we chose the number of splits = 5, meaning that training set will start from 25 samples and will expand up to 125 samples during the last iteration. Test set will constantly contain 25 observations.

Subsequently, a set of Causal and ML models, namely Multiple Linear regression (MLR), Ridge and Lasso regression, Support Vector regression (SVR) and Random Forest regression were selected for a class-level evaluation. Most of these models have been extensively compared and evaluated in the literature for their ability to predict important software attributes [43], [95], [96], [163]. In order to tune selected models in the best possible way, we used the Grid-search method [168]. Grid-search is commonly used to find the optimal hyper-parameters of a model that result in the most accurate predictions, by performing an exhaustive search over specified parameter values. These models were built on each class-level training set, and then tested on the respective test set. To test their predictive performance for different future horizons, we repeated the Walk-forward Train-Test validation process three times, where predictions were made for the next n+1, n+5, and n+10 future steps (weeks) respectively.

As mentioned in Section 2.4, the produced models were compared based on three different performance metrics, particularly the *MAPE*, the *RMSE*, and the *MAE*. The *MAPE* is a popular measure for forecast accuracy that uses absolute values to measure the size of the error in percentage terms. *RMSE* and *MAE* are also widely used in forecasting to express average model prediction error in units of the variable of interest.

As an example, we will illustrate the results of training TD forecasting models on the StreamThread.java class, which was found to be the most change-prone class of the Apache Kafka project for the examined commits. The StreamThread.java class dataset is comprised of 150 versions.

In Table 23, we report a comparison of prediction errors of the regression models trained on the StreamThread.java dataset for multiple (1, 5 and 10) time steps (weeks) into the future. Prediction errors in each cell of the table are averaged values of the testing errors for all train-test splits that were performed during Walk-forward Train-Test validation. Prediction errors indicated in bold are averaged values of the specific models that were created for each week-ahead prediction category.

Table 23: StreamThread.java TD predictions using Walk-forward Train-Test validation

Model	Weeks ahead	MAE (mins)	RMSE (mins)	MAPE (%)
MLR	1	19.194	24.814	5.711
	5	41.923	48.236	12.548
	10	55.858	62.19	16.399
	Average	**38.992**	**45.080**	**11.553**
Lasso regressor	1	13.043	19.1	3.84
	5	37.803	44.077	11.249
	10	56.903	63.862	16.767
	Average	**35.916**	**42.346**	**10.619**
Ridge regressor	1	13.69	19.684	4.036

	5	35.827	42.102	10.653
	10	52.832	59.769	15.577
	Average	**34.116**	**40.518**	**10.089**
SVR regressor (linear)	1	9.723	16.925	2.866
	5	41.517	47.49	12.354
	10	58.377	65.681	17.152
	Average	**36.539**	**43.365**	**10.790**
SVR regressor (rbf)	1	61.27	73.061	18.433
	5	56.804	67.281	17.003
	10	89.915	103.4	26.348
	Average	**69.329**	**81.247**	**20.594**
Random Forest Regressor	1	29.434	35.912	8.679
	5	43.726	49.142	12.887
	10	52.388	59.118	15.524
	Average	**41.849**	**48.057**	**12.363**

As far as shorter forecasting horizons are concerned (i.e., 1 week ahead), it is clearly depicted in Table 23 that linear models, such as Multivariate, Lasso, Ridge and SVR (linear) Regression, have generally lower *MAPE* values and outperform non-linear models, such as SVR(rbf) and Random Forest Regression. In fact, we observe that forecasting the TD of the StreamThread.java class for 1 step (week) ahead using SVR with a linear kernel gives a *MAPE* of 2.866%, while for the same horizon SVR with a Gaussian kernel gives 18.433% and Random Forest Regression gives 8.679%.

By observing Table 23, we also notice that linear models that apply Regularization in order to prevent overfitting, i.e., Ridge and Lasso Regression, are the best candidates even for a mid-term forecasting horizon of 5 steps (5 weeks) ahead. However, while their predictive power drops significantly as we try to forecast longer into the future, non-linear Random Forest model seems to have an almost stable performance over the holdout sample for all steps ahead.

For longer horizons (i.e., 10 weeks ahead), linear models are in general performing equally. However, the non-linear Random Forest model seems to perform slightly better than the other models. In fact, forecasts for 10 steps ahead using Random Forest gives the lowest *MAPE* error (15.524%), as well as the lowest *MAE* and *RMSE* errors. These results were also verified during the model execution phase described in Section 3.2.2, where we observed that Random Forest regression is able to provide accurate forecasts that lie very close to ground truth, even for 10 weeks ahead.

To further examine the ability of the investigated algorithms to forecast *TD Principal* and get an understanding of how the models perform, we repeated the same experiments on each of the 10 classes under examination in our dataset. We will not go through each class one by one, but instead we will provide averaged scores. Detailed prediction scores for each class can be found online [176]. Figure 27 illustrates the *MAPE* of the forecasting models, averaging the three forecasting horizon cases (1, 5 and 10 weeks ahead) and the 10 Apache Kafka classes under investigation.

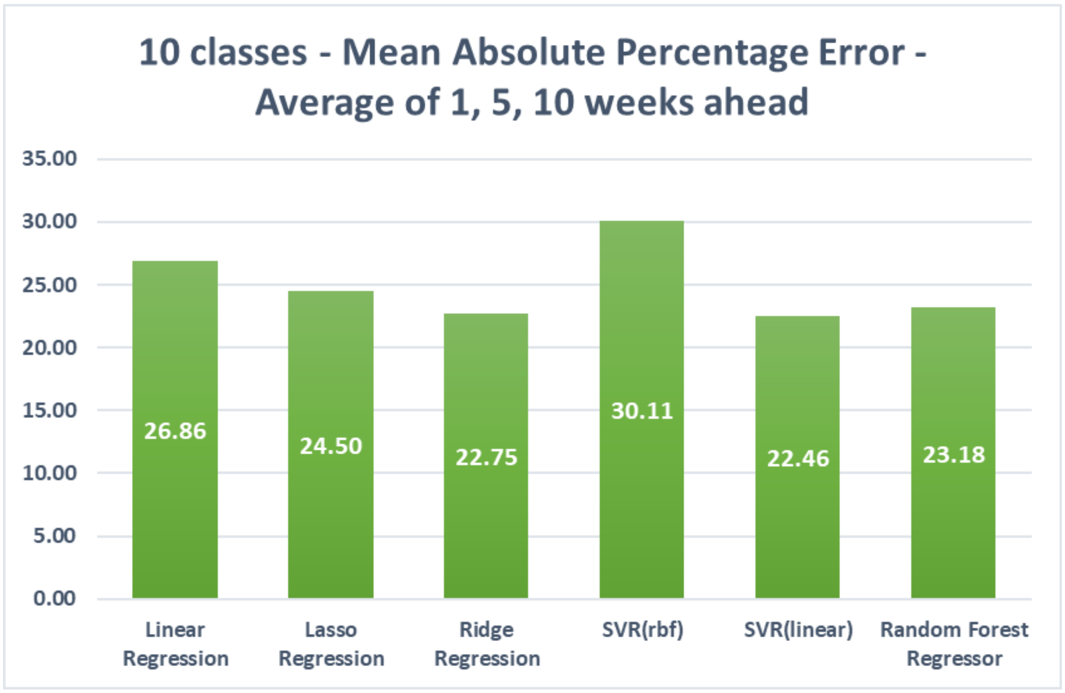


Figure 27: 10 classes TD predictions – MAPE averaged for all steps-ahead using Walk-forward Train-Test validation

The general outcome of the *Model Construction* phase is that selection of a forecasting model really depends on the horizon that we want to forecast. By inspecting the results presented both in the present chapter and online, we note that for shorter horizons the best accuracy is presented on models that apply regularization in order to prevent overfitting, i.e., Lasso and Ridge regression. However, when it comes to longer horizons the Random Forest regression model is generally performing better. These results will be visually presented in the rest of this section.

3.2.2 Model Execution

After constructing our models as described above, in this section we present the *Model Execution* phase. Towards executing our models for multi-step forecasts, we adopted the "Direct" approach, which means that a separate model is developed to forecast each forecast lead time. The main reason behind this decision is that since most of the ML models that we examined do not directly support more than one outputs, we excluded the Multi-step approach, i.e., single models with multiple outputs, where each output is used to forecast each forecast lead time simultaneously.

In Figure 28, we provide an example of forecasting the evolution of StreamThread.java class for 10 versions ahead using Random Forest regression, which during the model construction phase was reported to perform better than the other examined models for longer forecasting horizons. The red line denotes the forecast, while the blue line denotes the ground truth. Behind the scenes, 10 models were executed, one for each specific horizon of interest (starting from 1 step to 10 steps), while their forecasted TD values where aggregated into a common vector, and then plotted as the projected TD evolution.

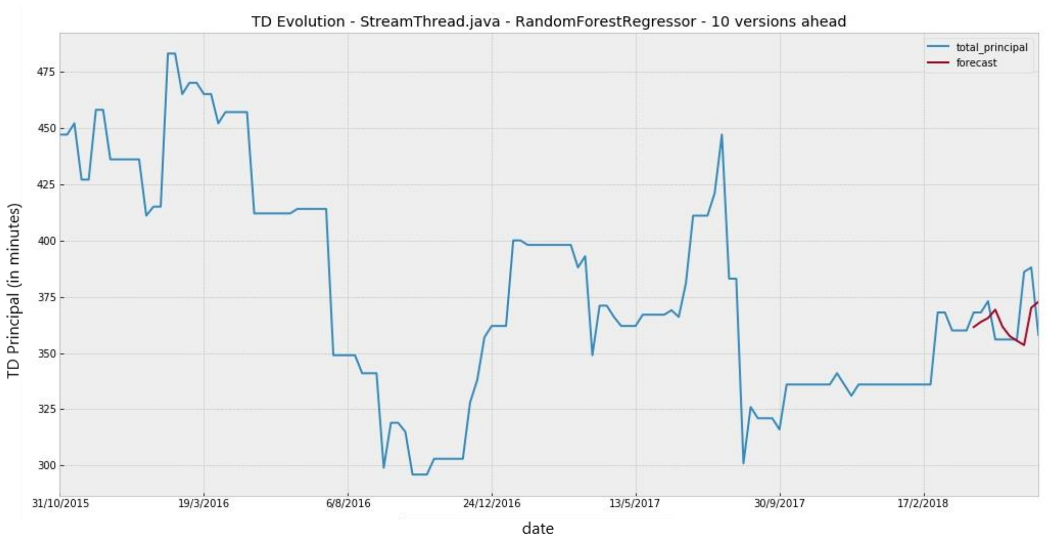


Figure 28: StreamThread.java TD forecasting for 10 versions ahead using the Direct approach

For reasons of brevity, forecasts for 10 versions ahead using Random Forest regression for the rest of the 9 classes are available online [176]. As can be seen, similar observations can be made for the other 9 classes of the selected subset of the Apache Kafka project. In particular, the Random Forest regression seems to provide meaningful long-term forecasts for each one of the studied cases (i.e., classes). In fact, the selected algorithm is able to capture the trend of the future evolution of the TD Principal, whereas in most of the cases the future value of the TD Principal is also captured with a sufficient level of accuracy.

3.3 Results Visualization

The approach proposed in this study introduces a more granular prioritization of TD liabilities by incorporating information retrieved from TD forecasting techniques. As described in Section 2, to properly combine the two proposed criteria for prioritizing TD repayment activities, namely the future forecasted value of the *TD Principal* and the change proneness of the selected classes, we decided to use a heat map as a means of visualization. Heat maps are easy to read and understand, since their underlying information is effectively conveyed to the developers and project managers of the software application, assisting them in making more informed decisions regarding TD repayment.

Figure 29 illustrates an indicative heat map that combines information retrieved from the analysis that we described above. In particular, the rectangles correspond to the specific 10 selected classes of the Apache Kafka application, as extracted and ranked (see Table 22) during the *Data Filtering* step of the methodology. The color of the rectangle denotes the change proneness of the corresponding class. The greener the rectangle, the higher the probability to change in the next versions. The size of each rectangle is proportional to the future value of the *TD Principal*, as reported by the class-level TD forecasting model. For the purpose of this indicative example, we have selected 4 steps-ahead (i.e., 1 month) as the horizon of the forecast. However, users could easily change the horizon of the forecasts depending on the TD repayment strategy that they are interested to, by simply using a drop-down menu, as shown at the bottom of the indicative screen.

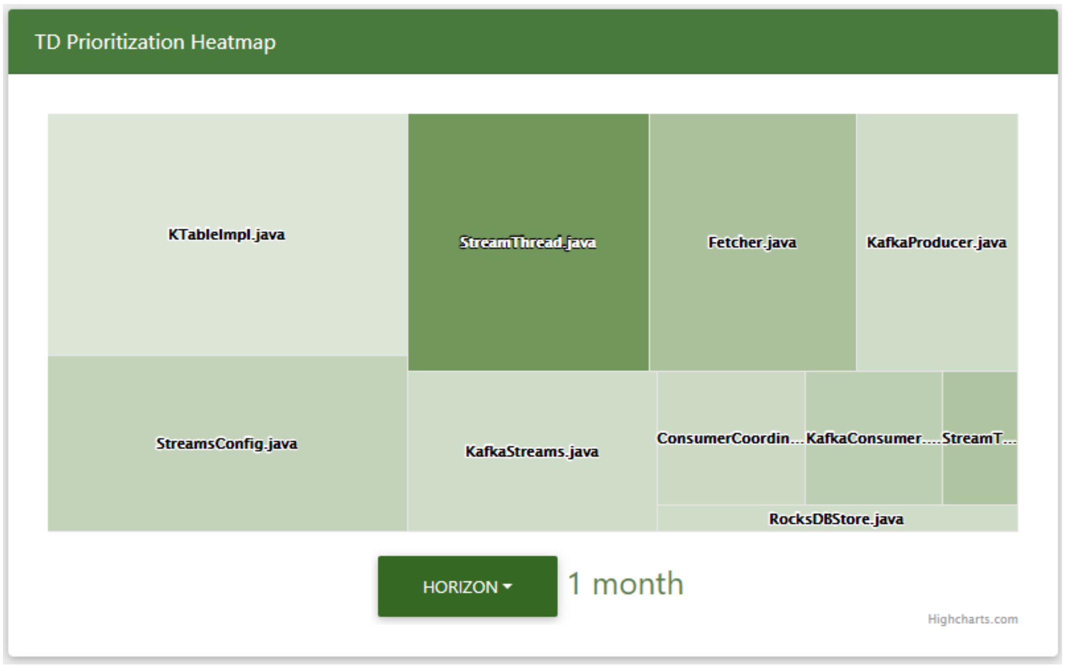
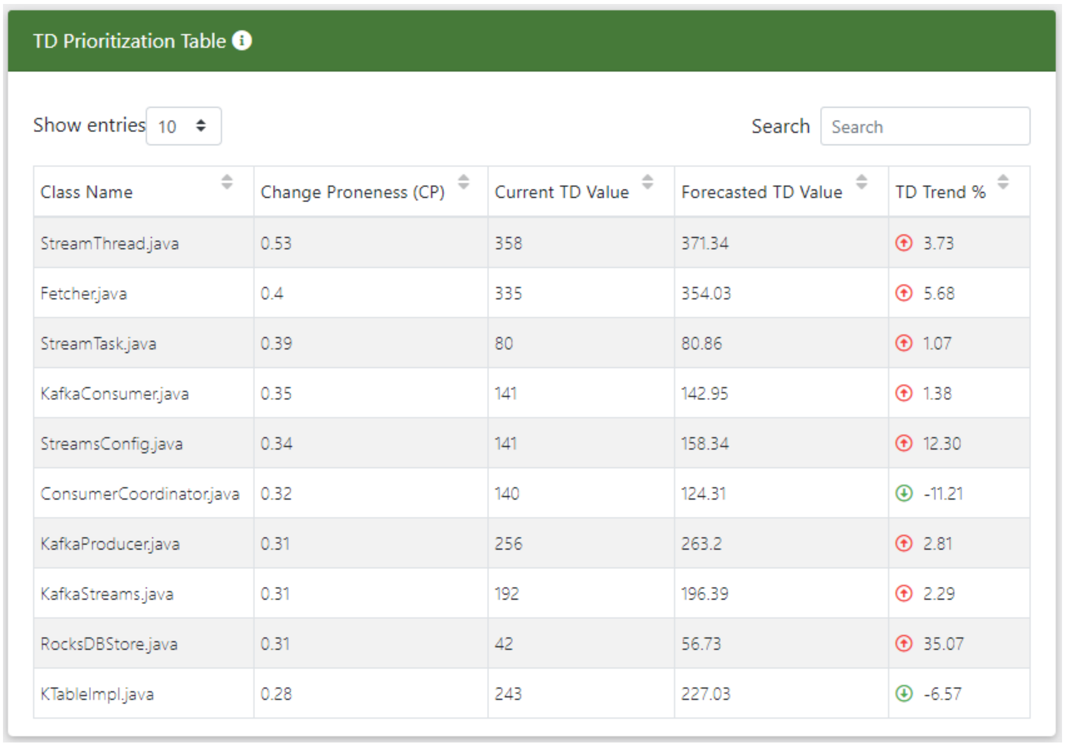


Figure 29: Heat map visualizing the future value of the TD Principal and the change proneness of the selected classes of Apache Kafka application

An example that illustrates the usefulness of the proposed method in enabling the early identification of classes that are more likely to become unmaintainable is the following: In the heat map depicted in Figure 29, class *StreamThread.java* is expected to have relatively high *TD Principal* (big rectangle size) in the upcoming versions, whereas it is also highly likely to change (deep green color). On the other hand, while class *KTableImpl.java* is expected to have relatively higher *TD Principal* compared to StreamThread.java (bigger rectangle size), it is less likely to change (light green color). As a result, *StreamThread.java* needs to be prioritized higher than *KTableImpl.java*, as it probably requires immediate remediation actions in order to reduce the risk of its TD accumulation.

Apart from the heat map, an indicative complementary table comprising the detailed results of the analysis is presented in Figure 30. This table contains supplementary information including the *Change Proneness (CP)* value, the forecasted class-level TD value, as well as the difference between the current TD value and the forecasted TD value, which can act as an indicator regarding whether TD of a specific class will increase or decrease. This additional information is expected to help the developers take even more informed decisions regarding the prioritization of their TD repayment activities.


TD Prioritization Table

Show entries 10

Search

Class Name	Change Proneness (CP)	Current TD Value	Forecasted TD Value	TD Trend %
StreamThread.java	0.53	358	371.34	↑ 3.73
Fetcher.java	0.4	335	354.03	↑ 5.68
StreamTask.java	0.39	80	80.86	↑ 1.07
KafkaConsumer.java	0.35	141	142.95	↑ 1.38
StreamsConfig.java	0.34	141	158.34	↑ 12.30
ConsumerCoordinator.java	0.32	140	124.31	↓ -11.21
KafkaProducer.java	0.31	256	263.2	↑ 2.81
KafkaStreams.java	0.31	192	196.39	↑ 2.29
RocksDBStore.java	0.31	42	56.73	↑ 35.07
KTableImpl.java	0.28	243	227.03	↓ -6.57

Figure 30: Table containing TD forecasting and CP supplementary metrics of the selected classes of Apache Kafka application

4 Conclusions and Future Work

The purpose of this chapter is to present a practical approach for a more granular prioritization of TD liabilities by incorporating information retrieved from static analysis, change-proneness analysis and forecasting techniques. Through our case study, across the 10 most change-prone investigated classes of the Apache Kafka application, we have shown that the proposed approach can effectively support TD prioritization during the TD management activities. More specifically, taking into account the future evolution of TD and combining it with proper change-proneness analysis and visualization techniques can enable the early identification of classes that are more likely to become unmaintainable in the future, and therefore allow project managers and developers plan effective TD repayment strategies.

Future work includes the extensive evaluation of class-level TD forecasting techniques on a broader spectrum of real-world software applications. We also plan to investigate the ability of already examined or new forecasting models to provide more accurate predictions for even longer forecasting horizons. Last but not least, we plan to investigate other types of software repositories that could be a potential source of TD related data, such as project management and issue-tracking systems, in order to achieve source triangulation and thus develop better and more informed TD prioritization approaches.

Chapter VII Data Clustering for Cross-project Technical Debt Forecasting

"*You cannot eat a cluster of grapes at once, but it is very easy if you eat them one by one.*"

Jacques Roumain - Haitian writer and politician

Published as:

D. Tsoukalas, M. Mathioudaki, M. Siavvas, D. Kehagias, and A. Chatzigeorgiou, "A Clustering Approach Towards Cross-Project Technical Debt Forecasting," in *SN Computer Science*, 2021, vol. 2, pp. 1–30. doi: 10.1007/s42979-020-00408-4.

Chapter Summary

Technical debt (TD) describes quality compromises that can yield short-term benefits but may negatively affect the quality of software products in the long run. A wide range of tools and techniques have been introduced over the years in order for the developers to be able to determine and manage TD. However, being able to also predict its future evolution is of equal importance in order to avoid its accumulation, and, in turn, the unlikely event of making the project unmaintainable. Although recent research endeavors have showcased the feasibility of building accurate project-specific TD forecasting models, there is a gap in the field regarding cross-project TD forecasting. Cross-project TD forecasting is of practical importance, since it would enable the application of pre-existing forecasting models on previously unknown software projects, especially new projects that do not exhibit sufficient commit history to enable the construction of project-specific models. To this end, in the present chapter we focus on cross-project TD forecasting, and we examine whether the consideration of similarities between software projects could be the key for more accurate forecasting. More specifically, we propose an approach based on data clustering. In fact, a relatively large repository of software projects is divided into clusters of similar projects with respect to their TD aspects, and specific TD forecasting models are built for each cluster, using regression algorithms. According to our approach, previously unknown software projects are assigned to one of the defined clusters and the cluster-specific TD forecasting model is applied to predict future TD values. The approach was evaluated through several experiments based on real-world applications. The results of the analysis suggest that the proposed approach comprises a promising solution for accurate cross-project TD forecasting.

1 Introduction

Technical Debt (TD), a term that has been inspired by the financial debt of economic theory, has been introduced by Ward Cunningham in 1922 [1] to describe in monetary terms the cost of additional software maintenance effort caused by technical shortcuts taken usually in favor of shorter time-to-market. Initially, TD was linked only to the software implementation process but soon enough it was extended to the whole software development cycle [2]. As in financial debt, TD incurs interest payments in the form of increased future costs owing to the earlier quick and dirty design and implementation choices. In that sense, TD can be considered as a precious tool that enables the understanding of when a software product becomes unmaintainable.

Numerous techniques, methods, and tools have been proposed over the last years for managing TD, providing a variety of options to the developers and project managers of software applications [5]. These approaches however, are mostly focusing on estimating the TD of a software project at its current state. On the other hand, predicting the future TD during the software evolution is a new emerging field of study that can be considered equally important, as the software system and its TD emerge in parallel [8]. TD forecasting could give project managers and developers the opportunity to timely react on the accumulation of TD by employing appropriate repayment activities (e.g., code refactoring) promptly [13] and thus, maintain a software application to a satisfactory quality level and avoid the unlikely event of reaching a point at which the software project becomes no longer maintainable. Whereas various researchers have addressed the topic of forecasting the evolution of various aspects directly or indirectly related to the TD of a software project, such as code smells [10], fault-proneness [11] and evolution trends [12], not too many concrete approaches have been proposed so far regarding the forecasting of TD itself. Hence, it is of paramount importance to produce and develop tools that will enable on-time decision making by predicting TD's future evolution.

In our previous work [14], [15], we have initially approached the TD forecasting challenge by employing statistical time series models, which have been widely used for predicting software evolution trends. In addition to statistical time series, we have also introduced and investigated other more sophisticated approaches that can be applied for TD forecasting, such as machine learning (ML) and causal models. The results of our studies revealed that the proposed approaches are promising and are able to provide accurate results. However, these approaches assume that reliable and sufficient historic data do exist in order for the models to be applied to a specific software project and provide accurate forecasts. More specifically, we expect that a rather long history of past commits of a given project is available for initiating a proper model-training process, as opposed to short series of historical data, which expose limited capacity for training forecasting models, due to insufficient information that they provide. This obviously affects the practicality of the produced models, as they cannot be applied to new software projects without a sufficient number of past commits. Cross-project TD forecasting, that is, building a forecasting model based on data retrieved from one project and using it to get reliable forecasts for a new, previously unknown software project, would allow project managers and developers leverage the benefits of TD forecasting in cases where a long history of commits is not available, e.g., from the very early stages of the development.

To this end, in the present chapter, in order to tackle the aforementioned issue, we introduce a novel clustering-based approach that aims to improve the performance of TD forecasting models in cross-project prediction. More specifically, according to our approach, a relatively large repository of software projects is utilized and its projects are grouped into clusters based on their similarity with respect to important TD aspects (e.g., size, complexity, Object-oriented metrics, etc.). Consequently, TD forecasting models are built using regression algorithms for each one of the produced clusters. When a new project becomes available, it is assigned to one of the defined clusters and the dedicated pre-trained forecasting model that is associated to this cluster is used to predict its future TD. Hence, this approach enables the provision of TD forecasts, for software projects that do not exhibit a sufficiently long history of commits. In other words, it enables accurate TD forecasting from the early stages of software development. In brief, the overall problem that the present work attempts to address can be summarized in the following research question:

RQ: *Can data clustering improve the accuracy of cross-project TD forecasting?*

For the purposes of the present study, a relatively large repository of real-world open-source software projects is utilized. This repository is used as the basis for the construction of the clusters, as well as of the TD forecasting models for each cluster. It is also utilized for the evaluation of the ability of the proposed approach to provide accurate cross-project TD forecasting.

The cross-project TD forecasting approach that is proposed in the present chapter is part of the SDK4ED[46] European project. The main goal of the SDK4ED project is to minimize cost, development time, and complexity of low-energy software development processes, by providing a set of innovative solutions (i.e., toolboxes) integrated into the form of an easy-to-use platform for automatic optimization and trade-off calculation among important design-time and run-time software quality attributes. The SDK4ED TD Forecasting tool, integrated into the TD Management (TDM) framework, aims to provide predictive forecasts regarding the evolution of the TD quality attribute. As mentioned previously, the existing work that forms the basis of this tool is built upon methods that assume the availability of a long history of past commits of a given project for initiating a proper model-training process. Therefore, the cross-project approach introduced in this chapter aims to set the foundations towards covering the existing gap in the field by proposing a method able to deliver cross-project TD forecasts and therefore assist practitioners (i.e., SDK4ED users) in performing TD management activities from the very early stages of the development.

The remainder of this chapter is structured as follows. Section 2 describes the proposed methodology, along with experimental setup, whereas in Section 3, a discussion of the experimental results is provided. Finally, Section 4 concludes the chapter and discusses directions for future work.

[46] https://sdk4ed.eu/

2 Methodology and Experimental Setup

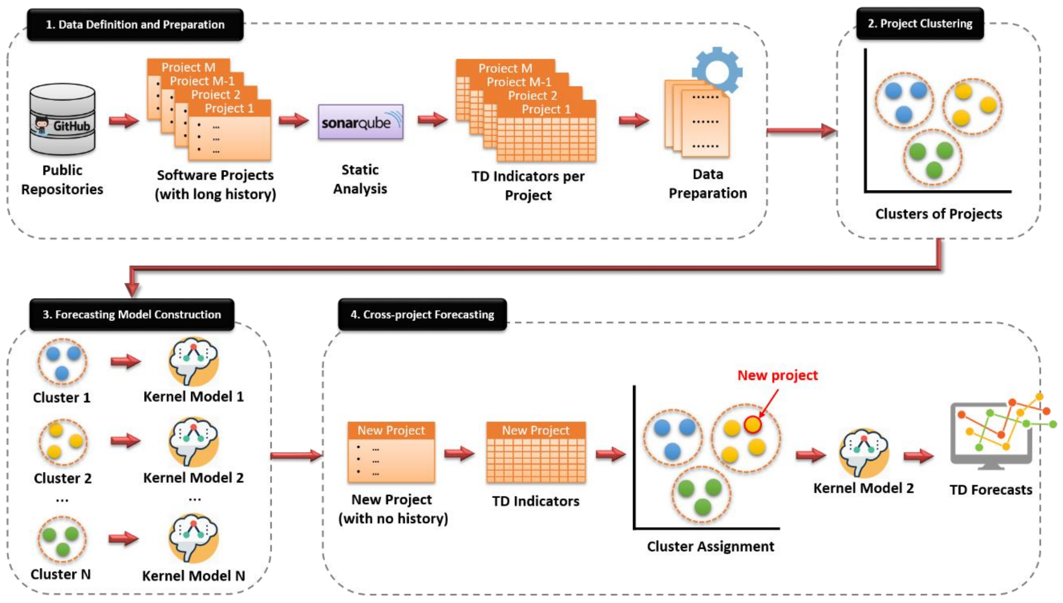


Figure 31: Chapter VII roadmap

In Figure 31, a high-level overview of the overall approach that is adopted in the present chapter for examining whether clustering of software projects could be the key for accurate cross-project TD forecasting is illustrated. As can be seen in Figure 31, the overall approach comprises four steps, which are briefly described below:

1. **Data Definition and Preparation**: During the first step of our approach, the source code of multiple software projects is retrieved from a code repository along with its history (i.e., past commits). Subsequently, the source code of the retrieved projects is analyzed using a static analysis platform and several TD measures are calculated for the construction of the initial dataset. Next, a data preparation process follows, which is responsible for applying feature selection techniques and bringing the dataset in a form ready to be used for model training purposes.

2. **Project Clustering**: During the second step of our approach, a selected clustering algorithm takes as input the dataset of analyzed projects, as produced during Step 1, and after finding the optimal number of clusters, it creates groups based on project “similarity” measures. These measures include various software metrics and TD indicators, such as size, complexity, code smells and bugs, which were computed during the static analysis performed in Step 1.

3. **Forecasting Model Construction**: During the third step of our approach, for each cluster generated during Step 2, the TD-related data of the projects assigned into the specific cluster are used to train a single cluster-representative TD forecasting model, which within the context of this work is referred to as “kernel” model. Therefore, the outcome of this step is the creation of *N* kernel models, where *N* is the total number of clusters produced by the clustering algorithm.

4. **Cross-project Forecasting**: During the fourth and final step of our approach, we assume that a new, previously unseen project becomes available. After performing

the required static analysis in order to extract its TD measures, the clustering algorithm assigns it to the appropriate similarity cluster and the dedicated pre-trained TD forecasting model for that cluster (produced during Step 3) is used to predict its future TD without requiring prior knowledge and historical data.

More details about each step of the proposed approach are provided in the rest of this section.

2.1 Dataset

As can be seen in Figure 31, the first step of the proposed approach starts with the construction of the dataset, that is, a relatively large code repository comprising the source code of multiple software projects along with their history (i.e., past commits). Subsequently, the source code of the retrieved projects needs to be analyzed using a static analysis platform in order to extract TD measures that will be used as input to the clustering algorithm.

For the purposes of the present study, we utilized a publicly available[47] TD dataset provided by Lenarduzzi et al. [177], which contains TD measurement data from 33 real-world open-source Java projects retrieved from the Apache Software Foundation. To collect TD measurements, the authors analyzed all available commits for each selected project using SonarQube[48], a popular open source platform that offers continuous code quality inspection and TD measurement mechanisms. Each project in the original dataset is provided as a csv file. The columns of the csv file correspond to 61 software-related metrics for a specific commit, as extracted from SonarQube across the whole commit history of each application. Respectively, each row of the csv file represents a specific commit of that project.

For the present work, 27 of these projects were used during the first three steps of the approach, while three of the remaining projects (randomly selected) were used during the last step of the approach as case studies in order to evaluate its usefulness in practice. The 27 projects that were utilized for clustering and forecasting model construction purposes (i.e., Step 1-3) are presented in Table 24, along with additional information regarding the total number of analyzed commits and the analysis timeframe.

Table 24: Projects of the TD Dataset

Project Name	**SonarQube**	
	Analyzed Commits	**Analysis Timeframe**
Atlas	2336	12/14 - 06/18
Aurora	4012	04/10 - 06/18
Batik	2097	10/00 - 06/06
Beam	2865	12/14 - 07/16
Commons-bcel	1324	10/01 - 04/18
Commons-beanutils	1192	03/01 - 06/18

[47] https://github.com/clowee/The-Technical-Debt-Dataset

[48] https://www.sonarqube.org/

Commons-cli	896	06/02 - 02/18
Commons-collections	2982	04/01 - 09/18
Commons-configuration	2895	12/03 - 05/18
Commons-daemon	980	09/03 - 08/11
Commons-dbcp	1861	04/01 - 06/18
Commons-digester	2143	05/01 - 08/17
Commons-exec	617	08/05 - 01/16
Commons-fileupload	922	03/02 - 10/17
Commons-io	2118	01/02 - 06/18
Commons-jexl	1551	04/02 - 05/18
Commons-jxpath	597	08/01 - 11/15
Commons-net	2088	04/02 - 08/17
Commons-ognl	608	05/11 - 09/13
Commons-validator	1339	01/02 - 04/18
Commons-vfs	2067	07/02 - 05/18
Felix	596	08/05 - 10/06
Httpcomponents-client	2867	12/05 - 06/18
Httpcomponents-core	1941	02/05 - 08/17
Mina-sshd	1370	12/08 - 06/18
Santuario	2697	10/01 - 06/18
Zookeeper	411	05/08 - 06/18

Within the context of this study, we did not exploit all 61 software-related metrics provided by SonarQube. Instead, we decided to narrow down the number of features by focusing only on those that act as the most prominent TD indicators. TD indicators are indicators usually related to software metrics [5], [40] that allow researchers and practitioners to point out TD-related quality issues residing in various attributes, features, or characteristics of a software artefact. Recent literature has identified a variety of software metrics that can act as potential TD indicators, while there is also a multitude of assessment tools [13] that support TD estimation based on these metrics. For instance, metrics such as cyclomatic complexity, code duplication or low test coverage (i.e. the absence of sufficient code tests) have been widely used to predict software maintainability decay and, in turn, the TD of a software application [54], [130], [140]. In addition, code issues that can negatively affect the quality of a software, such as code violations or bugs, have also been studied for their impact on TD [47]–[49]. Finally, code smells, i.e., sub-optimal design solutions that violate at least one programming principle [46] are considered in the literature as one of the first indications of the presence of TD [40], [45]. Code smells represent serious threats to the software maintenance process and impose the need for code refactoring [26].

In Table 25 we present the SonarQube metrics that were finally selected based on their ability to act as TD indicators. These metrics will be used as independent variables during the next steps of the methodology. In addition to the independent variables, the table also introduces the target variable, i.e., the variable we try to forecast, denoted as *total_principal* (in bold). In brief, *total_principal* is defined as the effort (in minutes) to fix all issues and is computed as the sum of code smell, bug, and vulnerability remediation effort.

Table 25: Brief description of the SonarQube metrics that were used as TD indicators

Metric	Description
Technical Debt Metrics	
sqale_index	Effort to fix all Code Smells. The measure is stored in minutes in the database.
reliability_remediation_effort	Effort to fix all bug issues. The measure is stored in minutes in the DB.
security_remediation_effort	Effort to fix all vulnerability issues. The measure is stored in minutes in the DB.
total_principal	**Effort to fix all issues. The sum of code smell, bug and vulnerability remediation effort. The measure is stored in minutes.**
Reliability Metrics	
bugs	Number of bug issues of a project.
Maintainability Metrics	
code_smells	Total count of Code Smell issues of a project.
Size Metrics	
comment_lines	Number of lines containing either comment or commented-out code of a project.
ncloc	Number of physical lines that contain at least one character which is neither a whitespace nor a tabulation nor part of a comment.
Security Metrics	
vulnerabilities	Number of vulnerability issues of a project.
Complexity Metrics	
complexity	It is the Cyclomatic Complexity calculated based on the number of paths through the code. Whenever the control flow of a function splits, the complexity counter gets incremented by one. Each function has a minimum complexity of 1. This calculation varies slightly by language because keywords and functionalities do.
Coverage Metrics	
uncovered_lines	Number of lines of code of a project, which are not covered by unit tests.
Duplication Metrics	
duplicated_blocks	Number of duplicated blocks of lines of a project.

2.2 Data Preparation

2.2.1 Feature Selection

After acquiring the data needed in order to proceed with further analysis, the next step is the preparation of the dataset in order to transform it in a form ready to be used for clustering and model training purposes. Therefore, the data preparation step first involves feature selection techniques and then the “sliding window” method, which is responsible for reframing the dataset in a form suitable for supervised ML problems.

The selection of independent (i.e. input) variables before designing or experimenting with an ML algorithm is a task that needs to be treated with special attention. A large number of input features, i.e. a high dimensional feature space, may lead to the “curse of dimensionality” [153]. According to this phenomenon, the increasing number of the model’s inputs leads to a degradation in its predictive performance. Features that are not associated (or they are partially associated) with the target variable may negatively affect the model accuracy. Hence, after building our dataset, a clear understanding of the statistical significance of the TD indicators needs to be gained, and lastly the feature selection algorithms need to be employed in order to reduce the number of the model's inputs by keeping only the TD indicators (described in Table 25) that are statistically important to act as TD predictors.

In order to examine which of the TD indicators are statistically significant for predicting the TD, four different feature selection algorithms were employed. More specifically, two filter-based methods were used, namely *Spearman Correlation* and *F-test*, one wrapper-based method named *Recursive Feature Elimination (RFE)*, and finally one embedded method named *Tree-based Elimination (TBE)*. Generally, filter-based methods filter the features based on a few metrics. On the other hand, wrapper-based methods consider the selection of a set of features as a search problem. Finally, embedded methods use algorithms that have built-in feature selection methods (such as Lasso and Random Forest).

- The **Spearman correlation** is a non-parametric approach used to quantify the monotonicity of the relationship between the values of two given datasets. As it is in other correlation techniques such as Pearson, Spearman correlation coefficients take values between -1 and +1. However, as a non-parametric test, Spearman correlation does not assume any distribution for the studied data. A coefficient value of -1 or +1 suggests monotony, whereas a coefficient value of 0 suggests no correlation at all. In this work, we explore the absolute values of the Spearman correlation coefficients between the independent (TD Indicators) and dependent (TD Principal) variable in our dataset, and we rank the former based on these values. Then, we select the top *N* features based on the results of this test.
- The **F-test** technique is a univariate linear regression test that employs a linear method to test the relationship of each of various independent features. This method consists of two main steps. First, the correlation between each indicator and the target variable is calculated. Then, the result is transformed into an F-score and then into a p-value. A relatively low p-value indicates the existence of a relationship between the examined variables, whereas a relatively high p-value indicates that there is no relationship. The F-test method calculates the correlation between an independent variable and the dependent variable and thus, it can be applied in order to determine and exclude the features that are likely to be insignificant for the evolution of the target variable and as a result, unsuitable to be used as TD indicators. In our occasion, the p-values between the independent variables (TD Indicators) and the dependent variable (TD Principal) are calculated and in this way we keep the top *N* features.
- The **Recursive Feature Elimination (RFE)** method, as its name implies, is a feature selection method that recursively eliminates features based on an estimator model trained on the initial set of features. The significance of each feature is implied either by a coefficient attribute (e.g. Logistic Regression) or by a feature-importance attribute (e.g. Random Forest) determined by the nature of the model. As this process evolves, the insignificant features are recursively excluded until we finally end up with only the required number of features. We select Logistic Regression as the estimator model. As a result, the RFE rates the significance of each of the features based on the coefficient given by the decision function of the

Logistic Regression estimator and crops the initial dataset until it contains only the top *N* independent variables in terms of significance.

- **Tree-based Elimination (TBE)** is an embedded method. As the name implies, this method uses models that have built-in feature selection methods to reduce the initial number of features. In this case, oppositely to the RFE method, we use the Random Forest model to determine feature importance based on node uncleanness in each decision tree. Then, the top *N* features are selected based on the average of all feature importance values calculated by each decision tree.

All of the four aforementioned feature selection algorithms were put in practice using Python in conjunction with the scikit-learn[49] ML library. Each of the algorithms described above was employed independently on the feature set and retained the top $N = 5$ features that were selected by each method. We set $N = 5$ mainly due to the fact that both RFE and TBE methods stopped the feature elimination process when the feature subset reached the number of five features. As a result, selecting the top 5 features from each independent method allowed us to directly compare the selected features' subsets among the four methods. Then, we aggregated the results by ranking each feature based on the number of times it was selected to be in the top 5 of a particular method. On Table 26 we observe the features ranked by the number of times they were selected by each of the algorithms. A value of True in a specific column indicates that the feature of that specific row has been selected to be in the top 5 features of the algorithm of this column.

Table 26: Results of the four feature selection methods

Feature	Spearman	F-test	RFE	TBE	Total
code_smells	**True**	**True**	**True**	**True**	**4**
bugs	**True**	**True**	**True**	**True**	**4**
complexity	**True**	**True**	**False**	**True**	**3**
ncloc	**False**	**True**	**True**	**True**	**3**
vulnerabilities	**True**	**True**	**True**	**False**	**3**
duplicated_blocks	False	False	False	True	1
uncovered_lines	False	False	True	False	1
comment_lines	False	True	False	False	1

By inspecting Table 26, it is obvious that *code_smells* and *bugs* are in the top 5 features of every feature selection algorithm. Furthermore, *complexity*, *ncloc*, and *vulnerabilities* are also high in the list since they were selected by three out of the four algorithms. We finally decided to keep the features that were indicated by at least three out of the four feature selection algorithms to be among the top five features. This basically means that features below *vulnerabilities* in the table will be excluded from the analysis that follows. To sum up, starting out with eight TD indicators that we examined, five of them turn out to be statistically significant for predicting TD, by three or four feature selection algorithms. At this point, it is worth mentioning that each of the aforementioned feature selection algorithms performs a statistical test in order to decide which of the features should be

[49] https://scikit-learn.org/stable/

excluded and which should be included during the selection process. Thus, the statistically significant impact of the final selection of features with respect to the target variable (i.e., *total_principal*) are the result of the independent statistical tests applied by each of the feature selection algorithms. Consequently, the TD indicators that were found to be the most promising TD predictors according to our feature selection approach are *code_smells*, *bugs*, *complexity*, *ncloc*, and *vulnerabilities*. These metrics will be considered as input during the clustering process described in Section 2.3 and during the creation of the TD forecasting models presented in Section 2.4.

2.2.2 Sliding Window Method

As we brought up earlier, the project-specific datasets that will be utilized through this work contain information provided by different indicators, throughout different points in time (i.e., timestamps) of each project. Our goal is to examine the evolution of the dependent variable, i.e., TD, through time. Hence, every sample of the dataset contains information for a specific point in time that is very probable to depend on the past (few) samples. However, the majority of ML models are not able to capture the temporal relationships that may govern the observations through time. Therefore, time series data need to be transformed into an appropriate form for supervised learning before put into practice for ML tasks.

In order to gain a better insight into the need for transforming time series data before providing it as input for ML models, a sample of data gathered in a temporal sequence is shown in Figure 32. The rows represent data samples gathered at specific points in time. Columns 2 to 4 represent features *X1* to *X3* accordingly, whereas column *Y1* represents the dependent variable, i.e., the variable we want to generate forecasts for. By inspecting the table, it becomes obvious that the current dataset structure is considered inappropriate for forecasting models training using supervised learning as there are two issues coming to light. First, if the dataset is utilized as it is for the training process, then the model will not be able to take under consideration any past information, as each row contains information only for a specific sample (i.e., point in time). Second, as the dependent variable of each sample holds only the value of that specific point in time, training a model using this format will result in a function that has learned how to make approximations only for that point in time (rather than generate future predictions).

One particular advantage of employing ML models instead of statistical models is their capability to support more than one input features. Taking advantage of this aspect of ML models, a method called "sliding window" was applied [162] on each project-specific dataset in order to bring it in a particular structure that integrates into one row information from multiple prior rows (steps) as inputs (*X*) in order to generate forecasts for future time steps as output (*Y*). Briefly, the sliding window expands each initial row of the dataset by including past and future information into one lag. More details on this approach are given below.

	X			***Y***
timestamp	***X1***	***X2***	***X3***	***Y1***
0	10	100	1000	10000
➡ **1**	20	200	2000	20000
2	30	300	3000	30000
3	40	400	4000	40000
4	50	500	5000	50000
…	…	…	…	…

Figure 32: Dataset collected in temporal order

The term that describes the number of past rows that we want to incorporate as input in the current lag is called the "window width". In the bibliography it may also be referred to as the "size of the lag". Firstly, we need to determine the sliding window width. On Figure 32, the red box represents the window width which corresponds to the present row (showed by the red arrow) plus a number of past rows. The current lag and the past lags containing past information will be merged into a new single row. For this specific case, the red box in Figure 32 contains the independent variables of the current lag (*t*) and also one step in the past lag (*t-1*) which will be merged into one new row. The forecasting horizon needs to also be determined. In our case, it is illustrated as the blue box in Figure 32. The blue box in our example indicates that forecasts will be generated for 1 step-ahead. This basically means that the *Y* value (i.e., the target variable) of *t+1* lag will be chosen as the target value for the sample that corresponds to lag *t*. If we wanted to generate forecasts for 2 steps-ahead, the *Y* value of *t+2* lag would be chosen as the target value, and so on. This process results in a new row, as shown in Figure 33. We repeat the process by "sliding" the two boxes at the same time over the rows, step by step, producing new lags until the window reaches the end of the dataset. When the whole process is over, a new re-framed dataset is produced, which now uses the current and one past step of each independent variable in order to generate forecasts for the target variable for one step ahead. The new dataset is shown in Figure 33.

	X						***Y***
index	***X1(t-1)***	***X2(t-1)***	***X3(t-1)***	***X1(t)***	***X2(t)***	***X3(t)***	***Y1(t+1)***
0	10	100	1000	20	200	2000	30000
1	20	200	2000	30	300	3000	40000
2	30	300	3000	40	400	4000	50000
3	40	400	4000	…	…	..	…

Figure 33: The reframed dataset after applying the sliding window approach

There are no specific guidelines that can be followed in order to determine the size of the window (i.e., the window width). Most of the times, selecting the number of past steps that will be merged per row is a process that strongly correlates to the number of the independent features, the size of the forecasting horizon and the model that is used itself. Consequently, as the selection of the window width takes place, we need to strike a balance between the model complexity and the forecasting accuracy. As a result, a good plan is to examine different window widths by training a model and check what values lead to better results for various forecasting horizons. The goal throughout this process is to minimize the errors. For example, we discovered that the optimal number to use as the size of the lag is 2. This provided us with the minimum Mean Absolute Percentage Error (MAPE) when forecasting takes place for 5 steps ahead, for most of the application-specific datasets when different models were tested. If more lags were added, the complexity of the models increased, with no remarkable effect on the model performance. Accordingly, for larger forecasting horizons, a larger window width was noticed to be more appropriate and as a result we observed improved model accuracy.

We re-framed each dataset using the sliding window method, taking into account the forecasting length we wanted to test our models for, in order to enable the usage of supervised ML. As long as the time series dataset is restructured following this approach and the sequence of the lines is conserved, any of the usual linear and non-linear ML algorithms can be employed.

Another problem that arises from the approach of TD forecasting is the necessity to generate multi-step predictions (i.e., predictions for more than one time-step), as we are interested not only in a single predicted TD value for a particular point in the future, but also in gaining an understanding of the overall evolution of the TD up to that point. There exist three main ML approaches that enable multi-step forecasts: i) the *Direct* approach, where different individual models are produced to generate forecasts for each forecast lead time, ii) the *Recursive* approach, where one model is produced to generate one-step ahead predictions, and the same model is employed repeatedly using as input the previous prediction to predict the following lead time, and iii) the *Multiple output* approach, where one model with multiple outputs is produced that predicts the whole forecast series simultaneously, i.e., in a one-shot way.

The ML models that we decided to consider in this work do not support more than one output at once thus, we reject the *Multiple* output approach. In addition, and as long as we are interested in multivariate forecasting, i.e., besides the dependent variable we consider also independent variables, we cannot put in practice the *Recursive* approach as this would demand predictions of the independent variables to forecast more than $t+1$ steps into the future. Thus, we decided to proceed with the *Direct* approach, which means that various independent models will be trained for each of the forecasting horizons. In practice, if we want to generate forecasts for N steps-ahead, the dataset needs to be transformed N times by applying the sliding window method exactly as we presented it before, where every time the target variable Y will lead to a point from $t+1$ to $t+N$ steps ahead respectively. Thus, N independent models will be developed, each one aiming to predict one future value starting from $t+1$ up to $t+N$. Lastly, the outputs of the models will be put together into a single array which eventually will hold the future evolution of the TD up to N steps ahead.

2.3 Project Clustering

In the previous section, we first gave a description on the data collection procedure we followed in order to prepare our initial application-specific datasets and then described the process under which the dataset is reframed in order to be ready to be used for clustering and model training purposes. This step of our approach involves the execution of a clustering algorithm that takes as input the refined dataset and creates groups based on project “similarity” measures.

As already mentioned, although TD forecasting models perform well when applied to software projects on which they have been trained (i.e., within-project forecasting), they fail to provide sufficient results in cross-project forecasting. This can probably be explained by the fact that each project exhibits intrinsic characteristics with respect to their type, size, complexity, etc., which may affect the produced models. However, it is reasonable to expect that the application of a TD forecasting model on a project that is highly similar (with respect to the aforementioned intrinsic characteristics) to the project that was used for building the forecasting model, may demonstrate sufficient predictive performance. Clustering is a viable solution for grouping software projects based on their similarity, and therefore for evaluating the correctness of the aforementioned hypothesis.

Clustering belongs to unsupervised machine learning. Hence, there are no labels we are trying to predict. To date, there exist numerous algorithms dedicated to perform clustering tasks, such as K-means, Agglomerative clustering, Mean Shift and Spectral Clustering, etc., each having its advantages and disadvantages. However, K-means is considered one of the simplest and most commonly-used clustering algorithms. Therefore, we decided to exploit its capabilities for the purposes of this work. K-Means is a centroid-based clustering algorithm. A centroid is basically a point in the center of a cluster. Each cluster is represented by a central vector or a centroid. The centroid point does not necessarily belong to the dataset that is used for clustering. The number of centroids is the number of clusters that we choose. Afterwards, all data points of the dataset we use for clustering are assigned to the closest centroid.

In our case, to proceed with applying K-means for the clustering task, we created a sub-set of our initial dataset in which each of the initial 27 projects is represented by one sample, so as to ensure equal representativeness for the clustering process. To do this, we selected the latest version of each one of the 27 projects in order for each software application in the new dataset to be represented by only one commit. Subsequently, to optimally group the software projects based on their similarity, we used the popular "Elbow" method, a heuristic commonly used in determining the number of clusters in a dataset. After applying the Elbow method, our 27 projects have been assigned to six different groups. We remind the reader at this point that the features we used for the clustering process are the following: *code_smells*, *bugs*, *complexity*, *ncloc*, *vulnerabilities*, and *total_principal*. In Figure 34, we offer a visualization of the six different clusters as they have been formed after applying K-means. For visualization purposes, we used only two features in order to draw the clusters into two dimensions. More specifically, in this particular illustration, the y-axis represents the TD (i.e., *total_principal*), while the x-axis represents a randomly-chosen feature among the ones selected as optimal TD predictors. In this case, this feature is *ncloc*.

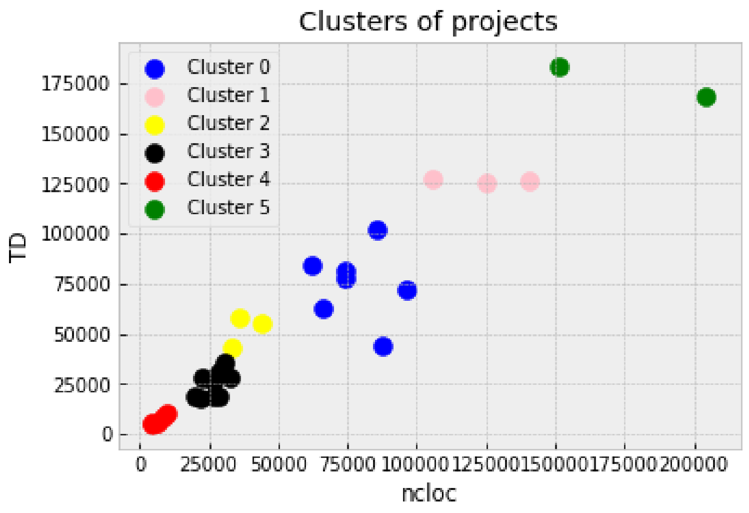


Figure 34: Clusters of the 27 datasets as they have been formed after K-means clustering with respect to TD and ncloc

In Table 27, the resulting clusters as produced by the K-means algorithm are presented. More specifically, the reader may observe the datasets (i.e., projects) that have been allocated to each cluster.

Table 27: Visualization of the 6 clusters and the datasets they consist of as generated by k-means clustering algorithm

Cluster 0	***Cluster 1***	***Cluster 2***	***Cluster 3***	***Cluster 4***	***Cluster 5***
Felix	*Santuario*	*Commons-beanutils*	*Commons-validator*	*Commons-cli*	*Batik*
Zookeeper	*Aurora*	*Commons-io*	*Commons-vfs*	*Commons-daemon*	*Atlas*
Httpcomponents-client	*Beam*	*Commons-bcel*	*Commons-dbcp*	*Commons-exec*	
Commons-configuration			*Commons-digester*	*Commons-fileupload*	
Mina-sshd			*Commons-jexl*		
Commons-collections			*Commons-jxpath*		
Httpcomponents-core			*Commons-net*		
			Commons-ognl		

2.4 Forecasting Model Construction

During the previous step of our approach, a carefully selected clustering algorithm receives as input the refined dataset of analyzed projects and after finding the optimal number of clusters, it creates groups based on TD-related measures. Therefore, as can be seen in Figure 31, the next step involves the construction of representative TD forecasting models (i.e., kernel models) for each cluster that will actually be used to provide TD forecasts on any new project that is assigned to a specific cluster in a later stage. Towards constructing kernel models for each cluster, we investigate the performance of a number of ML models trained on a specific project to forecast the TD Principal evolution of the other projects assigned within the same cluster and then, choose the best-performing model, based on its cross-project forecasting performance. The above process can be characterized as cross-project within-cluster forecasting, in a sense that a TD forecasting model trained on a specific project is tested on projects of the same cluster.

To further validate the idea that a kernel model of a specific cluster should provide better cross-project TD forecasting results when used on projects within the same cluster, we will also perform cross-project / cross-cluster predictions, in a sense that a TD forecasting model trained on a specific project belonging to a specific cluster is tested on projects of a different cluster. This will allow us to investigate whether the prediction accuracy of a kernel model is higher when forecasting for projects of the same cluster, compared to forecasting for projects of a different cluster. More details on this experiment, as well as the accompanying result are provided in Section 3.1.

As stated above, within this step of the methodology we investigate the performance of a number of ML models trained on a specific project to forecast the TD Principal evolution of another project. ML techniques are commonly used to make predictions for the future of a variable that interests the researchers. The goal is to maximize model accuracy, which means that the predictions need to be as close to the real values as possible, avoiding the risk of overfitting or underfitting the training data. In this work, we used the Mean Absolute Percentage Error (MAPE) as measure of prediction accuracy of a forecasting method. MAPE is a commonly used measure in order to validate the forecasting performance that takes into account absolute values to quantify the significance of the error expressed as a percentage. MAPE is a reliable measure and has two main advantages. The first one is that the absolute values lead to taking under consideration both the positive and negative errors preventing them from cancelling out each other which allows for no information to be missed. The second one is that due to the fact that respective errors do not correlate to the scale of the target variable, MAPE allows the comparison of forecast accuracy between data that are differently scaled (e.g. different software projects). The equation that describes the MAPE is presented below:

$$MAPE = \frac{1}{n}\sum_{t=1}^{n} \frac{|A_t - F_t|}{A_t}$$

where A_t is the actual value and F_t is the forecasted value. The difference between the actual and the forecasted value is divided by the actual value. The absolute value in this calculation is summed for every forecasted point in time and divided by the number of fitted points. Multiplying the equation by 100% makes it a percentage error.

Predicting the TD Principal is a regression problem, as this is a continuous variable. Thus, we need to use regression algorithms. In this chapter, five different algorithms have been examined for TD forecasting. Three of them are considered linear and the other two non-linear. More specifically, the linear algorithms that were investigated within the context of this work are Linear, Lasso, and Ridge Regression, whereas the nonlinear are Support Vector Regression (SVR) using the Gaussian (rbf) kernel, as well as Random Forest. The majority of these algorithms have been investigated in the literature for their capability to generate predictions for different software quality aspects, such as TD [15] and

Maintainability [95], [102], or lower-level software attributes, such as code smells [10] and defects [112]. Nevertheless, the task of determining the best-performing ML model per occasion is usually the outcome of a trial-and-error process, as the forecasting performance of any algorithm strongly relies on the distinctive characteristics of the data (such as size and structure). As a result, we decided to examine various ML algorithms to account for highly diverging data relationships that might rule the different software applications and thus, to overcome the restrictions of different techniques.

Again, for the utilization of the experiments, Python programming language was used and more specifically the scikit-learn ML library. For every algorithm under investigation, a grid search is initially performed in order to determine the optimal values of their parameters for the specific datasets that we have. The models that we chose to employ are described below.

2.4.1 Linear, Lasso and Ridge Regression

Linear regression is a method which explores the existence of linearity between a dependent variable and one or more independent variables. More specifically, given data with a number of variables and one single target variable, the purpose is to find a function that will return the best fit. The problem that occurs when a model learns exactly the details in the training data but fails to generate accurate predictions on new data is called "overfitting". In order to avoid this problem a regularization term can be introduced and this is exactly what Lasso and Ridge Regression do. Lasso Regression adds absolute values of the coefficients and thus, coefficients of insignificant variables will become zero. On the other hand, Ridge Regression, instead of adding absolute values, adds squares of the coefficients. Hence, the bigger the coefficient of a feature, the bigger the loss will be. However, the reader should keep in mind that Lasso and Ridge Regression do not necessarily provide us with better results than Linear Regression. This strongly depends on the characteristics of the different datasets.

2.4.2 Support Vector Regression – Radial Basis Function

Support Vector regression is a type of Support vector machine that supports linear and non-linear regression. In our case, we choose non-linear as we have another three linear algorithms. More specifically, we will use RBF (Radial Basis Function). RBF is a function with respect to the origin or a certain point c, i.e., $\varphi(x) = f(\|x - c\|)$, where the norm is usually the Euclidean norm but can be other type of measure.

The RBF learning model assumes that the dataset $D = (x_n, y_n), n = 1..N$ influences the hypothesis set $h(x)$, for a new observation x, which means that each x_i of the dataset influences the observation in a gaussian shape. Of course, if a data point is far away from the observation its influence is residual (the exponential decay of the tails of the gaussian makes it so).

2.4.3 Random Forest

A Regression Tree constitutes an alternative of decision trees that is constructed during a repeated procedure of separating the dataset into the decision branches. This procedure is called "binary recursive partitioning". An improved type of the Regression Tree is Random Forest (RF) paradigm. RF is a group of RTs which are trained by using the “bagging” approach. Bagging chooses random values over and over again by restoring the dataset and fits trees to these values. When the training process is completed, predictions for unknown data are produced by calculating the midpoint of the forecasts, or by taking the general contribution of all the independent RTs. Two are the main key benefits of RF. First, its accountability and second, its ability to learn complicated, highly non-linear relationships. Nonetheless, RF models require relatively long time to train and need more memory compared to the rest of the models that we consider. Last but not least, RF models are vulnerable to extensive overfitting because of the nature of decision trees.

2.5 Cross-project Forecasting

At this point, it is considered valuable to provide some clarifications with respect to main terms that are used in this chapter, in order to avoid confusion in the following stages of this study. With the term “within-project forecasting” we refer to the process of building and investigating the performance of models dedicated to predicting the evolution of the TD Principal in the software applications on which they have been trained. This means that the investigated models are trained on the data of a specific project, and evaluated based exclusively on data retrieved from the same project. On the other hand, in “cross-project forecasting”, whose applicability we aim to assess within the context of this work, emphasis is given on the ability of a given pre-trained TD forecasting model to accurately predict the future of the TD Principal in previously unknown software projects (i.e., software projects that were not used for the model training).

In the same manner, the term “within-cluster forecasting” refers to the procedure of constructing TD forecasting models on software projects assigned to a specific cluster to predict the future of the TD Principal in projects that are assigned to the same cluster. On the contrary, “cross-cluster forecasting” refers the process of using TD forecasting models trained on software projects of a specific cluster to forecast the TD evolution of projects assigned to different clusters. In order to ensure the ability of the proposed clustering approach to lead to satisfactory cross-project TD forecasts, the performance of the produced ML models is evaluated both in a within-cluster and in a cross-cluster manner in Section 3. In brief, in order for the proposed approach to be meaningful, the produced ML models should demonstrate lower errors in within-cluster evaluation. Hence, based on the above description, both the within-cluster and cross-cluster cases deal with cross-project TD forecasting, which is the main focus of the present work.

3 Results and Discussion

3.1 Within-cluster and Cross-cluster Evaluation

Having assigned all 27 projects to 6 clusters, in this section we aim to identify whether or not datasets that belong to the same cluster provide lower errors than datasets that belong to different clusters when used for cross-project TD forecasting. In order to investigate this assumption, experiments were employed for all combinations of the datasets (i.e., projects). This means that each dataset was used for training a forecasting model and then, this model was used to generate predictions for all the rest of the 26 datasets. The reader may assume that for these 27 projects, when a specific algorithm is used, 27x27 combinations (excluding the cases of the pairs referring to the same projects) are formed, thus 702 performance measurement values (MAPE values) are obtained. As stated in Section 2.4, five algorithms were employed for TD forecasting, namely Linear Regression, Lasso Regression, Ridge Regression, SVR and Random Forest. Hence, in total, we obtained 702×5=3510 MAPE values for all of the algorithms for five steps (commits) ahead. For reasons of brevity, we will not present the results for all possible combinations here. In Table 28, we present a fragment of the MAPE values for the case of Lasso Regression used to generate forecasts for five steps ahead so that the reader may have a quick look on a sample of the results. Detailed results for every algorithm can be found at the online Appendix [178]. More specifically, the MAPE values for SVR, Random Forest, Linear Regression, Lasso Regression and Ridge Regression can be found in Table 24, Table 25, Table 26, Table 27, and Table 28 (online Appendix) respectively.

To complement the forecasting performance evaluation, besides forecasting for 5 steps (commits) ahead, cross-project experiments were also extended for 10 steps ahead. However, since we believe that using the results of 5 steps-ahead forecasts is enough to answer the within-cluster and cross-cluster assumption that we aim to evaluate in this section, we decided to limit the 10 steps-ahead experiments only for those combinations of datasets that belong to the same cluster (within-cluster experiments). Thus, for these results we do not provide five tables as the case of five steps ahead results. Instead, we provide one table per each cluster, including within-cluster results for each algorithm. As we have six clusters and five possible algorithms there are 30 such tables. The reader may find these tables at the online Appendix [178] (Table 29 - Table 58).

Within the context of cross-project TD forecasting, training was performed on the entire dataset of the project that was chosen each time, while testing was performed also on the entire dataset of each project acting as testing dataset. This means that forecasts were generated at every sample (commit) of the testing dataset, throughout its whole evolution. In addition, since we aim at generating predictions for 5 and 10 steps (commits) ahead by following the *Direct* approach described in Section 2.2, predictions are generated first for 1 step ahead, then for 2 steps ahead, and so on. Subsequently, the predictions are aggregated into a common vector. This way, 5 or 10 different values (depending on the chosen horizon) are generated which represent the forecasts for 5 and 10 steps ahead respectively at a specific point of the testing dataset. These 5 or 10 values are then compared to the real values using the MAPE and a mean value of the MAPEs is calculated. This way, *N* such MAPEs are obtained for each testing dataset, where *N* is the number of points in the test dataset (i.e., number of project commits). The mean value of these *N* errors is calculated so that we have one single value representing the performance of the model when tested on each testing dataset.

To facilitate the readability of the tables, we define three categories of MAPE values to make it easier for the reader to examine and compare the results. MAPE values between 0-20% are marked with three asterisks (***), MAPE values between 21-50% are marked with two asterisks (**) and finally MAPE values between 51-80% are marked with one asterisk (*). MAPE values higher than 81% are marked with no asterisk at all. Finally, the parenthesis next to each project denotes the corresponding cluster number that this project belongs to for an even easier examination. The same notation holds for all tables in the rest of this chapter.

Table 28: Fragment of cross-project MAPE values obtained for all combinations of the 27 projects (within-cluster and cross-cluster) using Lasso Regression (the rest of the table can be found in Table 27 - online Appendix [178])

Train on:	*Zookeeper*	*Httpcompone*	*Commons-*	*Commons-*
Batik (5)	49%**	1%***	2%***	1%***
Felix (0)	42%**	3%***	5%***	3%***
Zookeeper (0)		0.6%***	13%***	1%***
Httpcomponents-client	28%**		12%***	2%***
Commons-configuration	57%*	3%***	5%***	1%***
Santuario (1)	70%*	6%***	6%***	3%***
Commons-beanutils (2)	33%**	1%***		1%***
Commons-validator (3)	84%	3%***	9%***	3%***
Commons-vfs(3)	39%**	2%***	6%***	
Commons-io (2)	75%*	4%***	7%***	2%***
Mina-sshd (0)	32%**	1%***	14%***	2%***
Aurora (1)	65%*	3%***	7%***	1%***
Beam (1)	8%***	4%***	16%***	5%***
Commons-collections (0)	16%***	2%***	2%***	1%***
Httpcomponents-core (0)	42%**	2%***	11%***	2%***
Commons-bcel (2)	37%**	1%***	4%***	1%***
Commons-cli (4)	157%	18%***	23%**	16%***
Commons-daemon (4)	336%	12%***	45%**	9%***
Commons-dbcp (3)	31%**	7%***	3%***	2%***
Commons-digester (3)	201%	48%**	34%**	8%***
Commons-exec (4)	309%	9%***	34%**	7%***
Commons-fileupload (4)	278%	11%***	39%**	10%***
Commons-jexl (3)	107%	10%***	12%***	7%***
Commons-jxpath (3)	9%***	3%***	5%***	1%***
Commons-net (3)	16%***	2%***	12%***	2%***
Commons-ognl (3)	13%***	2%***	7%***	1%***
Atlas (5)	128%	59%*	64%*	57%*

- where (*): MAPE between 51-80%, (**): MAPE between 21-50%, (***): MAPE between 0-20%

- the number in the parenthesis denotes the corresponding cluster

After carefully examining Table 28, one may easily notice that the MAPE values tend to be lower in within-cluster forecasting and higher in cross-cluster forecasting. In simple words, a project-specific TD forecasting model tends to provide better forecasts when it is applied to projects that belong to the same cluster, than when applied to projects that belong to different clusters. This supports our initial statement that the similarity of the different projects may play an important role on the predictive performance of the TD forecasting models in cross-project TD forecasting. The same observations can be made for the other regression algorithms, as can be seen in Table 24 - Table 28 (online Appendix [178]).

Hence, the similarity of the projects seems to affect the performance of TD forecasting models in cross-project forecasting, regardless of the selected regression algorithm.

As mentioned above, in all the studied cases the errors tend to be lower in within-cluster TD forecasting, than in cross-cluster TD forecasting. In order to reach safer conclusions, hypothesis testing is required. For this purpose, we defined the following null hypothesis (along with its alternative hypothesis) and tested it at the 95% confidence interval:

- **H_0**: *No statistical significant difference is observed between the errors of within-cluster and cross-cluster TD forecasting experiments*
- **H_1**: *A statistical significant difference is observed between the errors of within-cluster and cross-cluster TD forecasting experiments*

The hypothesis was tested for each one of the five regression algorithms that we examined. In order to conduct the statistical tests, the errors reported in Table 24 - Table 28 (online Appendix [178]) were properly grouped into two groups based on whether they correspond to within- or cross-cluster TD forecasting experiment. A fragment of this grouping is presented in Table 29.

Table 29: Fragment of the within-cluster and cross-cluster MAPE values, for each one of the five machine learning algorithms that were tested

SVR RBF		Random Forest		Linear Regression		Lasso Regression		Ridge Regression	
Within	**Cross**	**Within**	**Cross**	**Within**	**Cross**	**Within**	**Cross**	**Within**	**Cross**
24%	100%	29%	124%	8%	81%	13%	81%	32%	101%
29%	127%	28%	126%	25%	125%	26%	125%	28%	134%
32%	151%	40%	162%	16%	110%	19%	117%	28%	155%
72%	242%	83%	263%	75%	217%	73%	216%	81%	250%
40%	59%	57%	76%	40%	46%	40%	45%	39%	77%
40%	308%	42%	363%	40%	270%	20%	346%	38%	317%
37%	220%	72%	220%	15%	215%	18%	215%	49%	216%
67%	870%	68%	983%	19%	60%	6%	805%	67%	889%
68%	809%	70%	821%	6%	883%	41%	899%	74%	741%
145%	569%	160%	580%	40%	552%	13%	558%	162%	566%
120%	478%	129%	602%	12%	305%	12%	308%	129%	497%
59%	193%	65%	222%	12%	153%	7%	152%	70%	177%
156%	46%	164%	69%	8%	12%	13%	12%	162%	31%
14%	13%	17%	34%	13%	26%	7%	21%	18%	14%
24%	100%	29%	124%	8%	81%	13%	81%	32%	101%

In order to test the null hypothesis for each one of the five cases, the Wilcoxon's Rank Sum test was employed, which is a non-parametric test that does not assume any distribution for the analyzed data. More specifically, the Wilcoxon Rank Sum Test was employed five times, one for each studied regression algorithm, with the purpose to compare the errors of the within- and cross-cluster experiments. The results of the tests are presented in Table 30.

Table 30: The results of the Wilcoxon Rank Sum Test for each one of the five machine learning algorithms that were tested

Algorithm Name	p-value[50]
SVR RBF	< 2.2e-16
Decision Trees	< 2.2e-16
Linear Regression	< 2.2e-16
Lasso Regression	< 2.2e-16
Ridge Regression	< 2.2e-16

As can be seen in Table 30, in all the studied cases the *p-value* was found to be lower than the threshold of 0.05. As a result, in all the studied cases the null hypothesis is rejected, which leads to the acceptance of the alternative hypothesis. This suggests that a statistical significant difference is observed between the errors of within-cluster and cross-cluster TD forecasting. Thus, this provides further support to our observation that the similarity of software projects may play an important role in the predictive performance of TD forecasting models in cross-project forecasting. In other words, when a TD forecasting model is applied on a software project that is highly similar to the project (or projects) that was used for its training, it is highly likely to provide satisfactory TD forecasts.

3.2 Selection of the Kernel Model per Cluster

In the previous section, we have shown that training and testing cross-project TD forecasting models within a specific cluster offers lower errors compared to cross-cluster predictions. The next important step of the overall approach is the selection of the best performing TD forecasting model for each one of the constructed clusters, in terms of both the forecasting algorithm and the dataset that it is trained on. The best model of each cluster will actually be used as the representative (or kernel) model of this cluster, i.e., it will be applied for providing TD forecasts on any new project that is assigned to this cluster afterwards.

For the purpose of identifying the best performing TD forecasting model for each cluster, we will exploit the results obtained through the exhaustive experiments performed within Section 3.1 for all combinations of the 27 projects. However, we will focus exclusively on within-cluster forecasting, since we are willing to find the best model for each cluster. Similarly to the cross-project MAPE values reported in Section 3.1, in this section, each time a specific project is used for training, we calculate the mean value of all the MAPE values generated when testing is performed on the rest of the projects of the same cluster. This way, we have a single value representing the performance of every project when used for the training process. Afterwards, we extend this approach by calculating also the mean value of these mean values, in order to have one single value representing the cross-project performance of each algorithm on each cluster.

As an example, Table 31 through Table 35 report the within-cluster MAPE values obtained by using the five selected forecasting algorithms for five steps (commits) ahead for Cluster

As reported by R version 3.3.3.

0. For reasons of brevity, we do not provide similar tables for the rest of the clusters. However, this information can be easily retrieved from the detailed tables found in the online Appendix [178] (Table 24 to Table 58), by focusing only on the cross-project results referring to a specific cluster. By inspecting Table 31 through Table 35, we can see that the last row consists of the mean of the MAPE values when a specific project is used for training and the remaining projects of the cluster are used for testing. The bottom right cell represents the mean value of all the aforementioned mean values, which as mentioned earlier represents the within-cluster cross-project performance of every project when used for the training process. It should be noted that the diagonal is empty, since at this point we do not care about within-project prediction. What we are looking for is the model that performs best in cross-project TD forecasting, within a given cluster.

Table 31: Within-cluster cross-project MAPE values obtained for Cluster 0 using SVR for 5 steps ahead

Train on: / Test on:	*Felix*	*Zookeeper*	*Httpcomponents-client*	*Commons-configuration*	*Mina-sshd*	*Commons-collections*	*Httpcomponents-core*	
Felix		74%*	58%*	44%**	47%**	50%**	44%**	
Zookeeper	16%***		19%***	57%*	44%**	32%**	55%*	
Httpcomponents-client	149%	183%		89%	104%	122%	91%	
Commons-configuration	161%	213%	151%		72%*	111%	52%*	
Mina-sshd	117%	155%	111%	55%*		88%	57%*	
Commons-collections	78%*	113%	72%*	56%*	54%*		55%*	
Httpcomponents-core	205%	257%	196%	88%	118%	157%		
Mean Value:	***121.5%***	***166.3%***	***101.5%***	***65.2%***	***73.7%***	***93.9%***	***59.6%***	***97.4%***

Table 32: Within-cluster cross-project MAPE values obtained for Cluster 0 using Random Forest for 5 steps ahead

Train on: / Test on:	*Felix*	*Zookeeper*	*Httpcomponents-client*	*Commons-configuration*	*Mina-sshd*	*Commons-collections*	*Httpcomponents-core*	
Felix		70%*	14%***	37%**	27%**	18%***	17%***	
Zookeeper	11%***		3%***	46%**	23%**	19%***	7%***	
Httpcomponents-client	50%**	163%		40%**	34%**	27%**	23%**	
Commons-configuration	81%	197%	55%*		36%**	32%**	103%	
Mina-sshd	53%*	141%	19%***	22%**		20%**	29%**	
Commons-collections	33%**	102%	14%***	24%**	19%***		38%**	
Httpcomponents-core	78%*	232%	29%**	25%**	27%**	24%**		
Mean Value:	***51.6%***	***151.4%***	***22.8%***	***32.9%***	***28.1%***	***23.9%***	***36.8%***	***49.6%***

Table 33: Within-cluster cross-project MAPE values obtained for Cluster 0 using Linear Regression for 5 steps ahead

Train on: / Test on:	*Felix*	*Zookeeper*	*Httpcomponents-client*	*Commons-configuration*	*Mina-sshd*	*Commons-collections*	*Httpcomponents-core*	
Felix		45%**	3%***	5%***	5%***	3%***	22%**	
Zookeeper	69%*		0.6%***	1%***	0.4%***	5%***	5%***	
Httpcomponents-client	78%*	28%**		1%***	2%***	6%***	2%***	
Commons-configuration	136%	52%*	2%***		7%***	6%***	3%***	
Mina-sshd	88%	31%**	1%***	2%***		3%***	5%***	
Commons-collections	67%*	16%***	2%***	3%***	7%***		8%***	
Httpcomponents-core	79%*	42%**	2%***	2%***	3%***	7%***		
Mean Value:	***86.6%***	***36.1%***	***2.3%***	***3.0%***	***4.3%***	***5.5%***	***7.9%***	***20.8%***

Table 34: Within-cluster cross-project MAPE values obtained for Cluster 0 using Lasso Regression for 5 steps ahead

Train on: / Test on:	*Felix*	*Zookeeper*	*Httpcomponents-client*	*Commons-configuration*	*Mina-sshd*	*Commons-collections*	*Httpcomponents-core*	
Felix		42%**	3%***	5%***	13%***	3%***	29%**	
Zookeeper	46%**		0.6%***	1%***	0.6%***	5%***	7%***	
Httpcomponents-client	59%*	28%**		1%***	4%***	6%***	2%***	
Commons-configuration	91%	57%*	3%***		20%***	6%***	7%***	
Mina-sshd	62%*	32%**	1%***	2%***		3%***	7%***	
Commons-collections	46%**	16%***	2%***	3%***	20%***		9%***	
Httpcomponents-core	68%*	42%**	2%***	2%***	7%***	7%***		
Mean Value:	***62.5%***	***36.6%***	***2.5%***	***2.9%***	***11.2%***	***5.5%***	***10.5%***	***18.8%***

Table 35: Within-cluster cross-project MAPE values obtained for Cluster 0 using Ridge Regression for 5 steps ahead

Train on: / Test on:	*Felix*	*Zookeeper*	*Httpcomponents-client*	*Commons-configuration*	*Mina-sshd*	*Commons-collections*	*Httpcomponents-core*
Felix		45%**	3%***	5%***	5%***	3%***	22%**
Zookeeper	69%*		0.6%***	1%***	0.4%***	5%***	5%***
Httpcomponents-client	78%*	28%**		1%***	2%***	6%***	2%***
Commons-configuration	136%	52%*	2%***		7%***	6%***	3%***
Mina-sshd	88%	31%**	1%***	2%***		3%***	5%***
Commons-collections	67%*	16%***	2%***	3%***	7%***		8%***
Httpcomponents-core	79%*	42%**	2%***	2%***	3%***	7%***	

Mean Value:	***86.6%***	***36.0%***	***2.3%***	***3.0%***	***4.3%***	***5.5%***	***7.9%***	***20.8%***

As mentioned earlier, in this section we aim to identify which is the best forecasting model for each cluster, so it can actually be used as the representative (or kernel) model of this cluster in order to provide TD forecasts on any new project that is assigned to this cluster afterwards. In what follows, we explore the performance of each model per cluster. To do that, we aggregate the values representing the cross-project performance of each algorithm trained on each project of each cluster (last row in Table 31 through Table 35) on graphs so that we can visualize the results that finally will lead to a general conclusion. There are two graphs for each cluster for 5 and 10 steps (commits) ahead respectively, thus, there are 12 graphs in total.

In Figure 35 and Figure 36, the results for Cluster 0 for 5 and 10 steps ahead respectively are illustrated for all five algorithms. X-axis represents the project on which the within-cluster cross-project model is trained, while y-axis represents the mean value of MAPE values for that model.

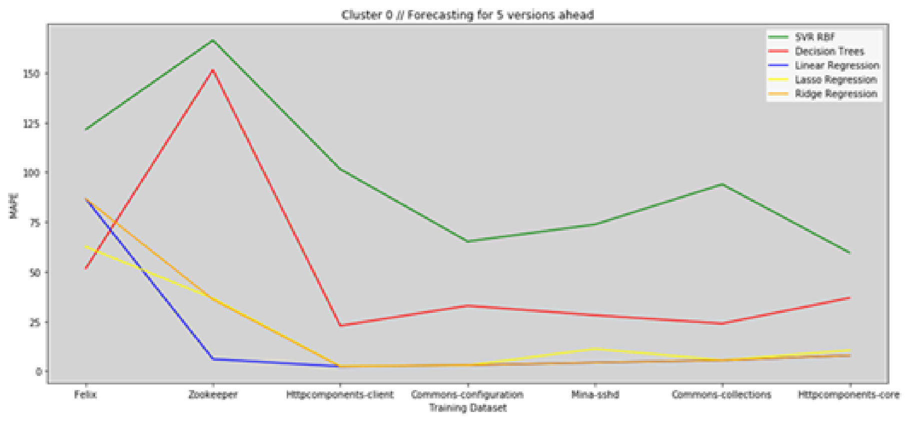


Figure 35: Within-cluster mean values of MAPE values obtained for Cluster 0 for 5 steps ahead

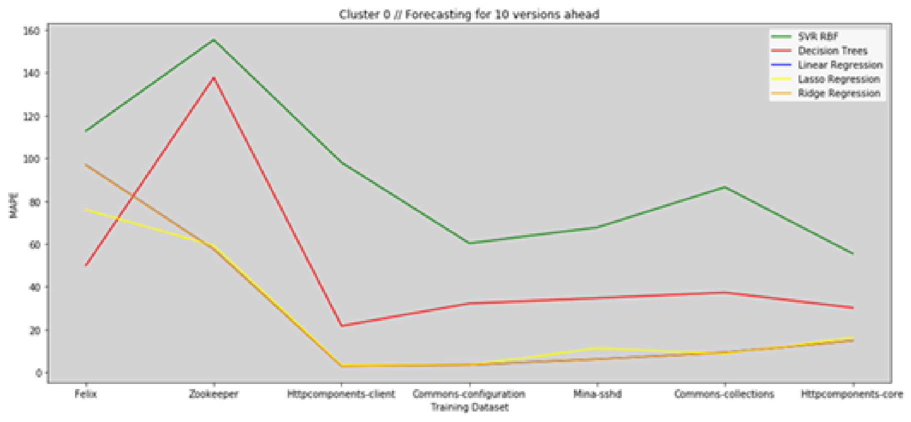


Figure 36: Within-cluster mean values of MAPE values obtained for Cluster 0 for 10 steps ahead

By inspecting Figure 35 and Figure 36, we can see that the algorithm that gave the highest MAPE value is SVR for all different training datasets. Random Forest provided us with the lowest errors when the model was trained on *Felix* both for 5 and 10 steps ahead. For

5 steps ahead, Linear Regression gave the lowest MAPE value when the model was trained on *Zookeeper*, whereas both Linear and Ridge Regression offered the lowest MAPE values when the model was trained on *Httpcomponents-client*, *Commons-configuration*, *Mina-sshd*, *Commons-collections* and *Httpcomponents-core*. For 10 steps ahead, the performance of Linear and Ridge Regression is exactly the same providing us with the lowest errors except for the case when training was performed on *Felix*. It is also obvious from the graphs that the datasets that are the most suitable to use for training are *Httpcomponents-client* and *Commons-configuration*.

On Figure 37 and Figure 38 the reader can see the results for Cluster 1 for 5 and 10 steps ahead respectively for all 5 algorithms.

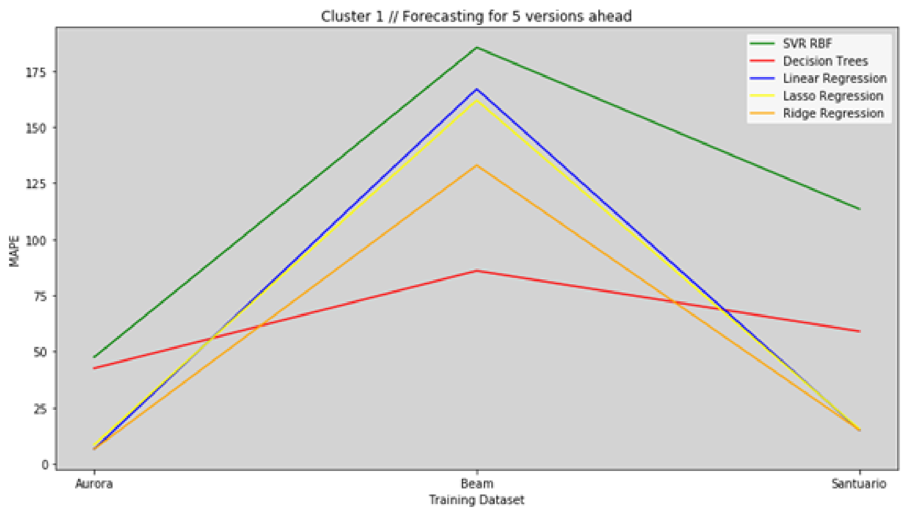


Figure 37: Within-cluster mean values of MAPE values obtained for Cluster 1 for 5 steps ahead

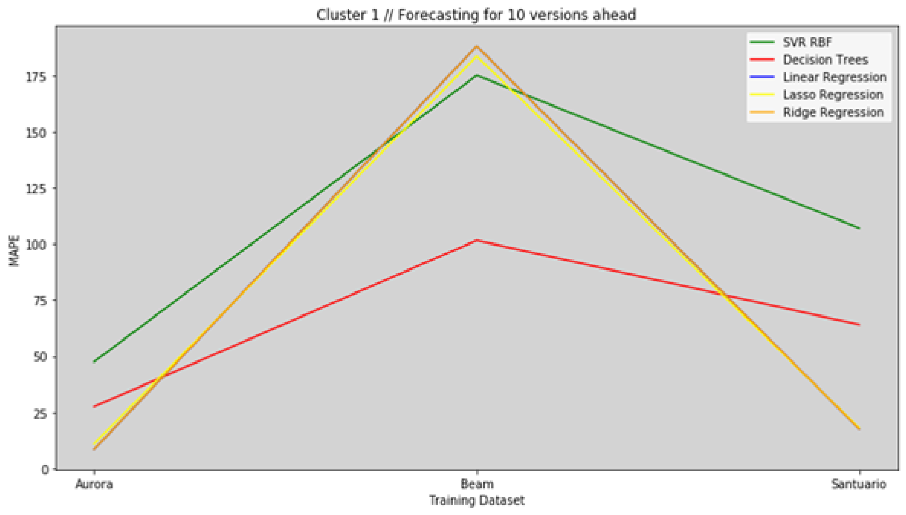


Figure 38: Within-cluster mean values of MAPE values obtained for Cluster 1 for 10 steps ahead

There are a few differences spotted between the two graphs. When the model is trained on *Beam*, SVR produces the highest errors followed by Linear, Lasso and Ridge Regression for 5 steps ahead whereas, for 10 steps ahead, Linear and Ridge Regression offer slightly higher errors than Lasso Regression followed by SVR. For both 5 and 10 steps ahead Random Forest gave us the lowest errors. In addition, when the model is trained on *Aurora* and *Santuario*, for 5 and 10 steps ahead, all algorithms have almost identical performance. For these cases Lasso, Ridge and Linear Regression produce the lowest errors and SVR the highest.

Overall, SVR produced the highest MAPE values except for the case of training on *Beam* for 10 steps ahead. On the other hand, Ridge and Lasso Regression generated the lowest MAPE values except when we train the model on *Beam* where the lowest errors are generated when Random Forest is used. As the reader can see, *Aurora* is the dataset that is the best out of the three of Cluster 1 to use for training.

On Figure 39 and Figure 40 the reader can see the results for Cluster 2 for 5 and 10 steps ahead respectively for all algorithms.

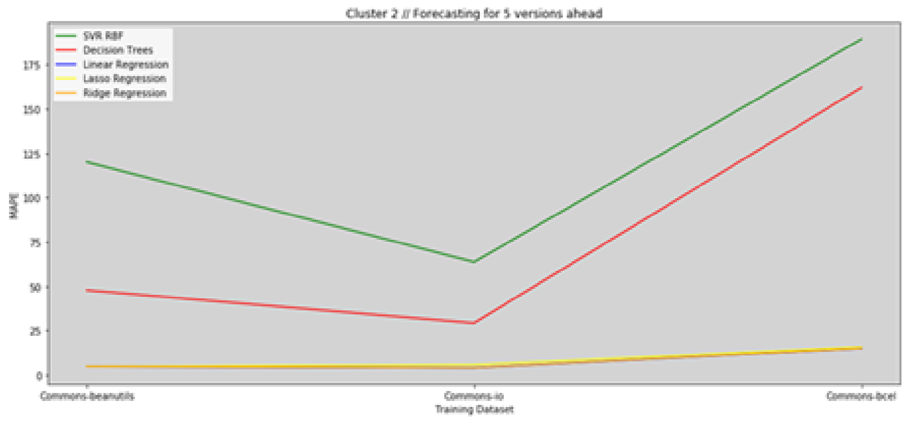


Figure 39: Within-cluster mean values of MAPE values obtained for Cluster 2 for 5 steps ahead

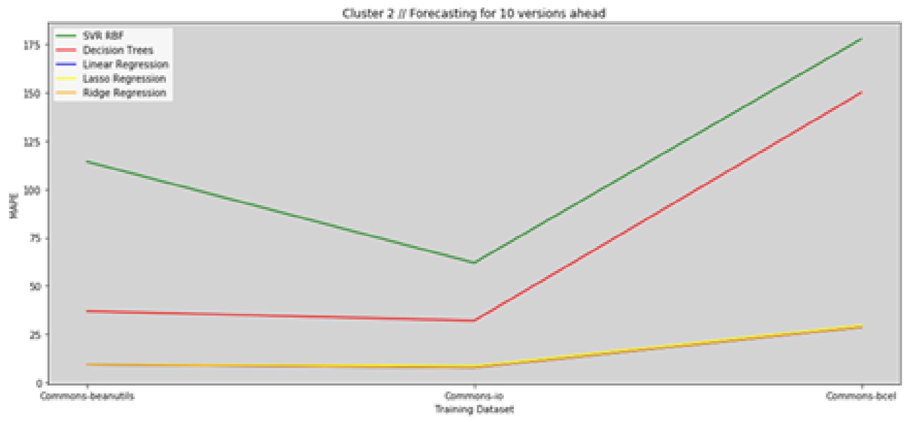


Figure 40: Within-cluster mean values of MAPE values obtained for Cluster 2 for 10 steps ahead

For Cluster 2, for 5 and 10 steps ahead, the two graphs are almost identical. When the model is trained on *Commons-beanutils*, *Commons-io* and *Commons-bcel* the lowest MAPE values were produced when using Linear, Ridge and Lasso Regression, while the highest MAPE values were produced when using SVR.

Overall, Linear and Ridge Regression provide us with the best results and the dataset that performs best as training dataset is *Commons-io*.

On Figure 41 and Figure 42 the reader can see the results for Cluster 3 for 5 and 10 steps ahead respectively for all algorithms.

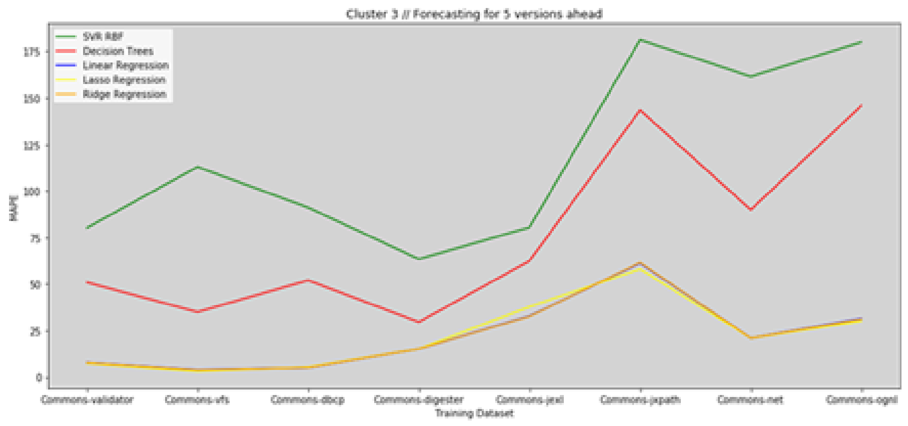


Figure 41: Within-cluster mean values of MAPE values obtained for Cluster 3 for 5 steps ahead

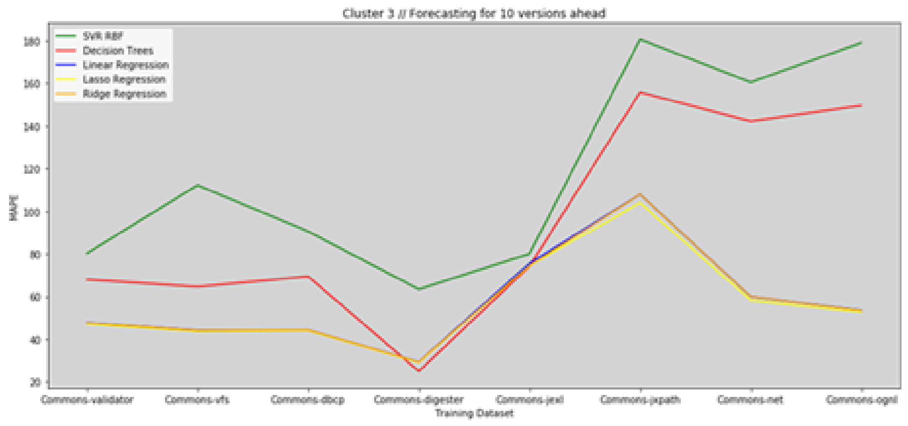


Figure 42: Within-cluster mean values of MAPE values obtained for Cluster 3 for 10 steps ahead

Comparing the two graphs we can verify that 5 and 10 steps ahead results are very much alike with a few differences. An exception is the case of training on *Commons-digester* where, for 10 steps ahead, errors obtained when using Random Forest are slightly lower than the errors obtained using the linear algorithms, whereas for 5 steps ahead the exact opposite is observed. Also, another exception is in the case of *Commons-jexl* where the performance of Random Forest, Linear, Lasso and Ridge Regression for 10 steps ahead is very close whereas for 5 steps ahead Linear, Lasso and Ridge Regression produced way lower errors than Random Forest.

Overall, SVR provided us with the highest errors. As we can easily detect from above, there is not a clear conclusion as to which algorithm or which training dataset might offer us the lowest errors both for 5 and 10 steps ahead. However, for 5 steps ahead, it is clear that Linear, Lasso and Ridge Regression display the best performance and the optimal datasets to use as training datasets are *Commons-vfs*, *Commons-dbcp* and *Commons-validator*.

On Figure 43 and Figure 44 the reader can see the results for Cluster 4 for 5 and 10 steps ahead respectively for all algorithms.

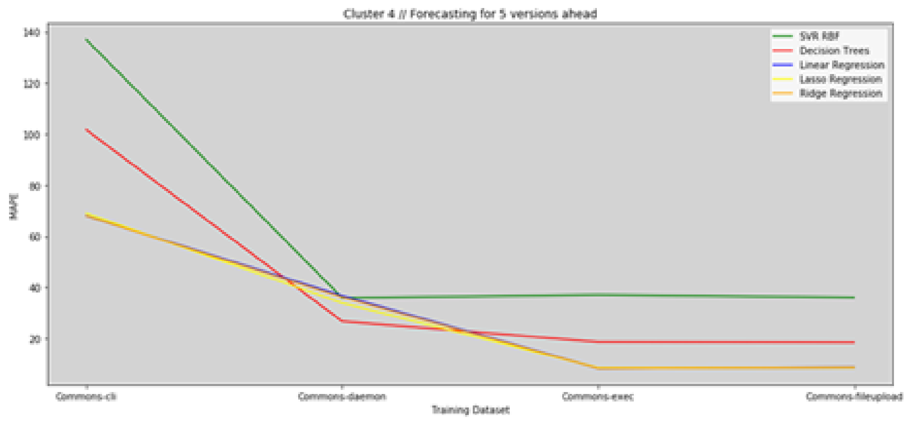


Figure 43: Within-cluster mean values of MAPE values obtained for Cluster 4 for 5 steps ahead

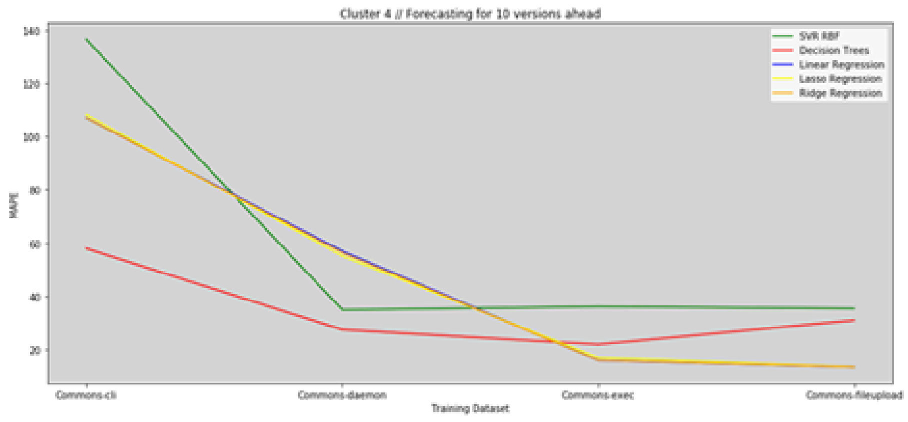


Figure 44: Within-cluster mean values of MAPE values obtained for Cluster 4 for 10 steps ahead

Both graphs look alike except for the cases where we use *Commons-cli* and *Commons-daemon* as training datasets. In the first case, Linear, Lasso and Ridge Regression produce better results than Random Forest for 5 steps ahead whereas for 10 steps ahead MAPE values produced by Linear, Lasso and Ridge Regression exceed by far those produced by Random Forest. In the case where training was performed on *Commons-daemon*, for 5 steps ahead, Linear, Lasso, Ridge Regression and SVR present the same performance which is worse than Random Forest performance, whereas for 10 steps ahead, again the lowest errors are provided by Random Forest, but this time, SVR follows with Linear, Lasso and Ridge Regression producing the highest errors.

For 5 and 10 steps ahead, there is not a general conclusion that can apply to both cases. For 5 steps ahead, for *Commons-cli*, *Commons-exec* and *Commons-fileupload*, the linear algorithms provide us with the best results whereas for *Commons-daemon*, Random Forest provides us with the best results. The datasets that generate the lowest errors when used for training are *Commons-exec* and *Commons-fileupload*.

On Figure 45 and Figure 46 the reader can see the results for Cluster 5 for 5 and 10 steps ahead respectively for all algorithms.

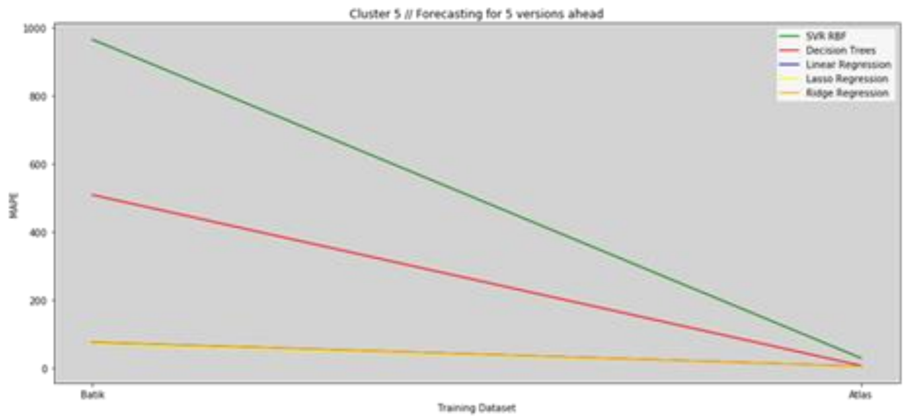


Figure 45: Within-cluster mean values of MAPE values obtained for Cluster 5 for 5 steps ahead

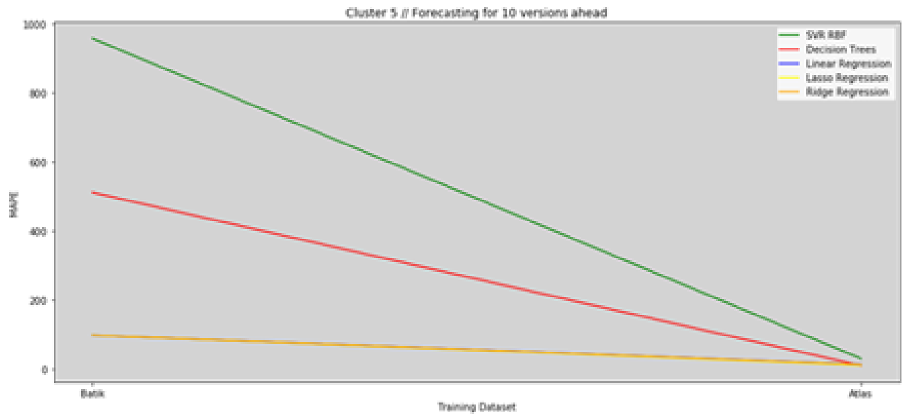


Figure 46: Within-cluster mean values of MAPE values obtained for Cluster 5 for 10 steps ahead

Cluster 5 only consists of two datasets. Both graphs representing 5 and 10 steps ahead are once more indistinguishable. When training was performed on *Batik*, SVR produce the highest MAPE values, followed by Random Forest followed by the linear algorithms producing by far the lowest errors. When we train the model on *Atlas* the lowest MAPE value is given by Lasso Regression and the highest MAPE value is given by SVR. To sum up, *Atlas* is the most suitable dataset for training using Lasso Regression.

After providing detailed results on the performance obtained by each algorithm when executing experiments by training and testing on every dataset of each cluster (i.e., within-cluster cross-project predictions), in the part that follows, we aim to aggregate these results in order to conclude on which algorithm is the optimal choice for each cluster. To do so, we obtain the mean of the mean values of the errors produced when training on each dataset of each cluster. In order to make it easier for the reader to understand, we revisit the bottom right cell of Table 31 through Table 35. The reader may assume that since we have 6 clusters and 5 different algorithms, we obtain 30 such values which are presented on Table 36. Through this process we can observe which of the algorithms offers the lowest mean value for each cluster, and therefore to determine the best-performing model (or kernel model) for each cluster.

Table 36: Performance of each algorithm for each cluster. The lowest values for each cluster have been marked with an asterisk

Cluster: / Algorithm:	*Cluster 0*	*Cluster 1*	*Cluster 2*	*Cluster 3*	*Cluster 4*	*Cluster 5*
SVR	97.4%	115.5%	140.3%	112.5%	69.0%	497.5%
Random Forest	49.6%	62.5%	85.3%	74.8%	46.9%	258.5%
Linear Regression	20.8%	62.8%	**9.3%***	**21.5%***	31.0%	41.0%
Lasso Regression	**18.8%***	62.0%	10.2%	21.7%	**30.4%**	**38.6%***
Ridge Regression	20.8%	**51.5%***	**9.3%***	**21.5%***	30.8%	41.0%

As a reminder, in this section we try to identify the best performing TD forecasting model for each one of the constructed clusters, in terms of both the forecasting algorithm and the dataset that it is trained on. At this point, we summarize the results that will provide an answer to the question we initially set in this section. In order to make a statement on which of the projects of each cluster are the optimal to use for training, we carefully examine the tables of the results of the corresponding algorithm that we just stated as optimal for each cluster. This means that we inspect Figure 35 to Figure 46, seeking for the curve that represents the performance of the optimal algorithm for each cluster. Then, we choose the projects that provide the lowest errors when used for training with the optimal algorithm. On Table 37, the reader may find the best training projects and the optimal algorithm for each cluster. Out of the total number of projects lying within each cluster, we chose to present the top 30% of them that exhibit the best performance. Hence, for Cluster 0, 1, 2, 3, 4 and 5 we choose two, one, one, three, two, and one projects respectively.

Table 37: Optimal training projects for each cluster and the corresponding selected algorithm that provided the lowest errors

Cluster 0	***Cluster 1***	***Cluster 2***	***Cluster 3***	***Cluster 4***	***Cluster 5***
Optimal Datasets for Training					
Httpcomponents-client	*Aurora*	*Commons-io*	*Commons-vfs*	*Commons-exec*	*Atlas*
Commons-configuration			*Commons-dbcp*	*Commons-fileupload*	
			Commons-validator		
Selected Algorithm					
Lasso Regression	Ridge Regression	Linear Regression	Linear Regression	Lasso Regression	Lasso Regression

By inspecting Table 37, we are now able to clearly state that for Cluster 0 the optimal algorithm is Lasso Regression, for Cluster 1 Ridge Regression, for Cluster 2 and Cluster 3 Linear Regression and finally, for Cluster 4 and Cluster 5 Lasso Regression. Lastly, we are also able to state that the optimal projects to be used for training for Cluster 0 are *Httpcomponents-client* and *Commons-configuration*, for Cluster 1 *Aurora*, for Cluster 2

Commons-io, for Cluster 3 *Commons-vfs*, *Commons-dbcp* and *Commons-validator*, for Cluster 4 *Commons-exec* and *Commons-fileupload* and finally, for Cluster 5 *Atlas*.

3.3 Merging Datasets

In the previous section, we tried to identify the best performing TD forecasting model for each one of the constructed clusters, in terms of both the forecasting algorithm and the dataset that it is trained on. While the choice of the optimal forecasting algorithm per cluster was clear, we noticed that there were cases where for some specific clusters more than one projects provided good cross-project results. In a real scenario where a new project arrives and we need to perform forecasts for, it would be difficult to find the optimal project and "borrow" its pre-trained model. This would require a lot of experiments based on trial and error. However, a merged dataset possibly allows for more generalizability. Thus, as our next step, we aim to examine whether merging project datasets within a cluster would improve the cross-project TD forecasting model performance. For this purpose, we chose to merge the best-performing projects (as reported in Table 37 of Section 3.2) which correspond to the proportion of 30% of the total number of datasets within each cluster, train cross-project models on the merged datasets and then test the performance of these models on the remaining datasets of the same cluster. For those clusters that one dataset corresponds to the proportion of 30%, no merging took place. We remind the reader that for Cluster 0, 1, 2, 3, 4 and 5 we chose two, one, one, three, two, and one projects respectively.

We ran the experiments for all the clusters for 5 steps ahead using the algorithms that were found to be optimal for each cluster as reported in Table 37. For the cases where a merged dataset is formed, we also present the results that were obtained before merging, so that we can gain a deeper understanding on whether the merged datasets demonstrate better cross-project forecasting performance or not. In Table 38 through Table 43, the errors obtained for each cluster are presented.

Table 38: Within-cluster cross-project MAPE values obtained for Cluster 0 using a merged dataset. Results corresponding to training on the datasets that the merged dataset consists of are also presented for comparison reasons

Train on: / Test on:	*Merged Dataset (0)*	*Httpcomponents-client (0)*	*Commons-configuration (0)*
Felix (0)	4%***	3%***	5%***
Zookeeper (0)	1%***	0.7%***	1%***
Mina-sshd (0)	1%***	1%***	2%***
Commons-collections (0)	3%***	2%***	3%***
Httpcomponents-core (0)	4%***	2%***	2%***
Mean Value:	***2.6%***	***1.6%***	***2.6%***

Table 39: For Cluster 1, 30% of the cluster corresponds to 1 dataset thus we present again the results when Aurora was used as a training dataset

Train on: / Test on:	*Aurora (1)*
Beam	7%***
Santuario	6%***
Mean Value:	***6.5%***

Table 40: For Cluster 2, 30% of the cluster corresponds to 1 dataset thus we present again the results when Commons-io was used as a training dataset

Train on: / Test on:	*Commons-io (2)*
Commons-beanutils (2)	8%***
Commons-bcel (2)	3%***
Mean Value:	***5.5%***

Table 41: Within-cluster cross-project MAPE values obtained for Cluster 3 using a merged dataset. Results corresponding to training on the datasets that the merged dataset consists of are also presented for comparison reasons

Train on: / Test on:	*Merged Dataset (3)*	*Commons-vfs (3)*	*Commons-dbcp (3)*	*Commons-validator (3)*
Commons-digester (3)	16%***	15%***	11%***	18%***
Commons-jexl (3)	7%***	7%***	9%***	14%***
Commons-jxpath (3)	1%***	1%***	2%***	8%***
Commons-net (3)	2%***	2%***	3%***	2%***
Commons-ognl (3)	1%***	1%***	4%***	4%***
Mean Value:	***5.4%***	***5.2%***	***5.8%***	***9.2%***

Table 42: Within-cluster cross-project MAPE values obtained for Cluster 4 using a merged dataset. Results corresponding to training on the datasets that the merged dataset consists of are also presented for comparison reasons

Train on: / Test on:	*Merged Dataset (4)*	*Commons-exec (4)*	*Commons-fileupload (4)*
Commons-cli (4)	20%***	15%***	19%***
Commons-daemon (4)	2%***	8%***	6%***
Mean Value:	***11.0%***	***11.5%***	***12.5%***

Table 43: For Cluster 5, 30% of the cluster corresponds to 1 dataset thus we present again the results when Atlas was used as a training dataset

Train on: / Test on:	*Atlas (5)*
Batik (5)	5%***

Mean Value:	***5.0%***

By inspecting Table 38 through Table 43, we can see that for all 6 clusters the interpretation of the results is somewhat similar. More specifically, when training is performed on the merged datasets, the obtained cross-project performance is equal or in some cases higher than the performance obtained when training is performed on the individual projects of which the merged datasets consist.

3.4 Case Study

The last step of our approach is the actual execution of the pre-trained models on previously unknown real-world software projects (i.e., software projects that have not been used for the previous analysis). According to our approach, when a new project arrives, it is initially assigned to one of the six clusters, based on its similarity, and, subsequently, the representative (i.e., kernel) TD forecasting model of this cluster is applied to the project. In order for our approach to be valid, the TD forecasting model of the cluster to which the new project is assigned should provide better forecasts (i.e., lower errors) compared to the TD forecasting models of the other clusters.

To this end, in the present section, we use three real-world software projects that were not used neither for the construction of the clusters, nor for the construction of the representative TD forecasting models of the clusters. The new testing projects are *Commons-jelly*, *Commons-codec* and *Commons-dbutils* and were also retrieved from the TD dataset provided by Lenarduzzi et al. [177], as presented in Section 2.1. After employing the K-means algorithm presented in Section 2.3, these projects have been assigned to Cluster 2, 3 and 4 respectively. Therefore, kernel models of the three dedicated clusters (i.e., Cluster 2, 3 and 4) will be used to make cross-project predictions. Our expectation is that these pre-trained kernel models will offer lower errors compared to the pre-trained models of other clusters.

Having assigned the new datasets to our clusters, we test the pre-trained models by referring to Table 37 that consists of our results. The testing process is similar to the testing process we followed in the previous section. This time however, we will not test all possible combinations of the datasets for 5 different algorithms, but instead we will test the kernel pre-trained models on the entire new testing dataset of each project assigned to the corresponding cluster. For Cluster 0, the pre-trained model is trained on the merged dataset that consists of *Httpcomponents-client* and *Commons-configuration* using Lasso Regression. For Cluster 1, the pre-trained model is trained on *Aurora* using Ridge Regression. For Cluster 2, the pre-trained model is trained on *Commons-io* using Linear Regression. For Cluster 3, the pre-trained model is trained on the merged dataset that consists of *Commons-vfs*, *Commons-dbcp* and *Commons-validator* using Linear Regression. For Cluster 4, the pre-trained model is trained on the merged dataset that consists of *Commons-exec* and *Commons-fileupload* using Lasso Regression. Finally, for Cluster 5 the pre-trained model is trained on *Atlas* using Lasso Regression. The results that were obtained when testing was performed on the new projects are presented on Table 44. Each row represents the performance of the pre-trained models on the new testing datasets.

The errors that have been obtained when training and testing took place within the same cluster have been marked with an asterisk.

Table 44: MAPEs obtained for the use cases of 4 new projects using all of the 6 pre-trained models

Train on: / Test on:	*Merged_Data set (0)*	*Aurora (1)*	*Commons-io (2)*	*Merged_Data set (3)*	*Merged_Data set (4)*	*Atlas (5)*
Commons-jelly (2)	2%	2%	**1%***	1%	7%	2%
Commons-codec (3)	4%	5%	2%	**1%***	7%	14%
Commons-dbutils (4)	2%	7%	5%	4%	**2%***	9%

The results have met the expectations. Indeed, as can be seen by inspecting Table 44, the pre-trained kernel models of each of Clusters 2, 3 and 4 produced errors lower or equal for each of the new testing projects compared to the pre-trained models of other clusters.

4 Conclusions and Future Work

TD refers to deliberate or inadvertent non-optimal design decisions made during the software development lifecycle that lead to poorly designed systems and therefore additional maintenance effort. Due to its interdisciplinary nature, TD has attracted the attention of both academia and industry over the last years, resulting in a considerable increase in the number of methods and accompanying tools that support TD management activities, such as TD identification, quantification or repayment. Besides standard TD management activities, predicting the accumulated TD during the evolution of a software application is considered crucial, since such an action would allow project managers and developers perform long-term effective software maintenance. However, current TD forecasting approaches build on the assumption that reliable historic data are available in order for the models to be applied to a specific software project and provide accurate forecasts. Under those circumstances, a method that enables the provision of TD forecasts for software projects that do not exhibit a long commit history could provide practical decision-making mechanisms from the early stages of software development. Cross-project TD forecasting, that is, building a forecasting model based on data retrieved from one project and using it to get reliable forecasts for a new, previously unknown software project, would allow project managers and developers leverage the benefits of TD forecasting in cases where a long history of commits is not available, e.g., from the very early stages of the development.

The purpose of this chapter is to examine whether the adoption of clustering is a promising solution for enhancing the accuracy of cross-project TD forecasting. In other words, we investigate whether the consideration of TD-related similarities between software projects could be the key for more accurate cross-project forecasting. For this purpose, a large dataset was utilized, comprising 27 real-world open-source Java applications. These applications were then fed into a clustering algorithm and divided into clusters of similar projects with respect to their TD aspects. Subsequently, cluster-representative TD forecasting models were constructed through several experiments, using five regression algorithms for forecasting horizons between 1 and 10 steps ahead. We observed that the cross-project prediction accuracy of the cluster-representative models was higher when forecasting for projects assigned to the same cluster (i.e., within-cluster forecasting), compared to forecasting for projects assigned to a different cluster (i.e., cross-cluster forecasting), while in most of the cases, the future TD value was captured with a sufficient level of accuracy. Furthermore, we investigated whether merging projects would further improve the cross-project forecasting performance of the cluster-representative models. Finally, to evaluate the usefulness of the proposed approach in practice, various previously unknown real-world applications were assigned to the defined clusters and the TD forecasting models associated to these clusters were applied to predict their future TD evolution. The forecasting error was observed to be smaller when the kernel model of the cluster to which the previously unknown project was assigned was used, compared to the error of the other kernel models. This suggests that the similarity of the software projects may be a key factor for enhancing cross-project TD forecasting, and, in turn, that clustering may be a viable solution for achieving better results in cross-project forecasting. In brief, the results of the analysis were encouraging and suggest that the proposed approach is a promising solution for more accurate cross-project TD forecasting.

Several directions for future work can be identified. First of all, the present study was based solely on open-source Java applications retrieved from the Apache Software Foundation. In order to investigate the generalizability of the produced results, we plan to replicate our study on a broader spectrum of real-world software applications that are written in other programming languages and that belong to different domains. Secondly, in this study, the consideration of TD-related similarities between software projects, as well as the selection of features to be used as predictors for the construction of forecasting models was based only on TD indicators retrieved from SonarQube. We consider to extend this study by considering also TD indicators retrieved from other sources, such as different ASA tools and OO metric suits. We also plan to investigate the extension of cross-project TD forecasting to lower levels of granularity of a software project, such as package, class or function level. Finally, future work includes improving the proposed approach by considering more sophisticated clustering methods, as well as the possibility that the clustering process dynamically repeats the training process once a new project arrives, in order to update the clusters and retrain the dedicated forecasting models taking into account the newly acquired data.

Chapter VIII Machine Learning Techniques for Technical Debt Identification

"Left unchecked, technical debt will ensure that the only work that gets done is unplanned work!"

Gene Kim – Author and IT researcher

Published as:

D. Tsoukalas, N. Mittas, A. Chatzigeorgiou, D. Kehagias, A. Ampatzoglou, T. Amanatidis, and L. Angelis, "Machine Learning for Technical Debt Identification," in *IEEE Transaction on Software Engineering (TSE)*, 2021. doi: 10.1109/TSE.2021.3129355.

Chapter Summary

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

1 Introduction

Technical Debt (TD) is a metaphor facilitating the discussion among technical and non-technical stakeholders when it comes to investments on improving software quality [38]. As with financial debt, TD that is accumulated due to problematic design and implementation choices needs to be repaid early enough in the software development life cycle. If not done so, then it can generate interest payments in the form of increased future costs that would be difficult to be paid off. In this context, the development of software projects can be significantly impeded by the presence of TD: wasted effort due to TD in software companies can reach up to 23% of total developers' time [31] and in extreme situations may even lead to an unmaintainable software product and thus, to technical bankruptcy [9]. As a result, proper TD Management should be applied so as to identify, quantify, prioritize and repay TD issues. However, managing TD across the entire software development lifecycle is a challenging task that calls for appropriate tooling. A recent study

[32] revealed 26 tools that can help towards the identification, quantification and repayment of TD. The strategies employed to identify TD issues differ, but the most frequently used approach relies on predefined rules that can be asserted by static source code analysis: any rule violation yields a TD issue while the estimated time to address the problem contributes to the overall TD principal. Due to their different rulesets, none of these tools results can be considered as an ultimate oracle [7], while using multiple tools to aggregate their results is not a feasible solution, since: (a) executing multiple tools for the same reason can be considered as a waste of resources; (b) there is no clear synthesis method; (c) acquiring multiple proprietary tools is not cost-effective.

The aforementioned shortcoming leads to important problems: On the one hand, academic endeavors face serious construct validity issues, because regardless of the tool used to identify Technical Debt Items (TDIs), there is always a doubt on if the results would be the same if a different tool infrastructure was used. On the other hand, regarding industry, usually the list of identified issues is very long, and the practitioners get lost in the numerous suggestions. On top of that, they face a decision-making problem: "*which tool shall I trust for TD identification/quantification?*". As a first step to alleviate the aforementioned problems, in a previous study [7], we analyzed the TD assessment by three widely-adopted tools, namely *SonarQube* [179], *CAST* [85] and *Squore* [180] with the goal of evaluating the degree of agreement among them and building an empirical benchmark (TD Benchmarker[51]) of classes/files sharing similar levels of TD. The outcome of that study (apart from the benchmark per se) confirmed that different tools end-up in diverse assessments of TD, but to some extent they converge on the identification of classes that exhibit high-levels of TD.

Given the aforementioned result, in this chapter, we propose the use of *statistical* and *Machine Learning* (ML) models for classifying TD classes (High/Not-High TD). As ground truth for the proposed classification framework, we use the dataset obtained from applying the TD-Benchmarker on 25 OSS Java projects, and we classify as high-TD the classes that all tools identify as TDIs. Then, supposing that the use of multiple sources of information will result in more accurate models, we built a set of independent variables based on a wide spectrum of software characteristics spanning from code-related metrics to metrics that capture aspects of the development process, retrieved by open-source tools. Finally, we apply various statistical and ML algorithms, so as to select the most fitting one for the given problem. Ultimately, the derived models subsume the collective knowledge that would be extracted by combining the results of three TD tools, exploiting a "commonly agreed TD knowledge base" [7]. To facilitate their adoption in practice, the models have been implemented as a tool prototype that relies on other open-source tools to automatically retrieve all independent variables and identify high-TD classes for any software project.

The main contributions of this chapter are summarized as follows: i) An empirical evaluation of various statistical and Machine Learning algorithms on their ability to detect high-TD software classes; ii) A set of factors to be considered while performing the

[51] http://195.251.210.147:3838

classification, spanning from code metrics to repository activity, retrieved by open-source tools; iii) A tool prototype that yields the identified high-TD classes for any arbitrary Java project by pointing to its git repository.

2 Methodology

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

An overview of the methodology is presented in Figure 47.

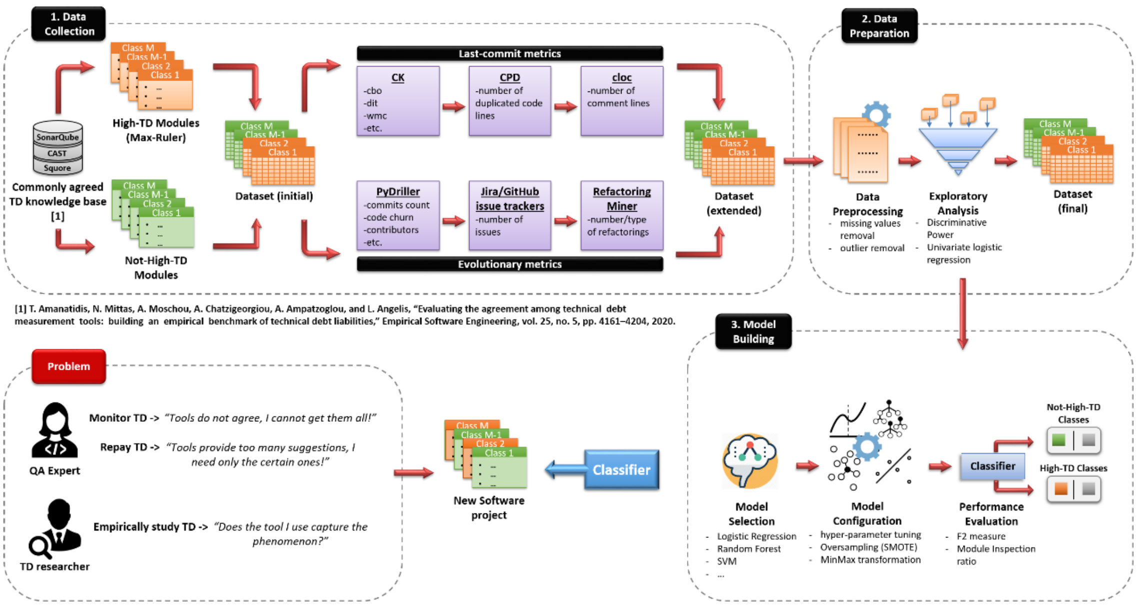


Figure 47: Chapter VIII roadmap

2.1 *Data Collection*

2.1.1 *Project Selection and Dependent Variable*

The dataset of this study is based on an empirical benchmark that was constructed within the context of a study by Amanatidis et al. [7]. The aim of that study was to evaluate the degree of agreement among leading TD assessment tools by proposing a framework to capture the diversity of the examined tools and thus, to identify profiles representing characteristic cases of modules with respect to their level of TD. For this purpose, they have constructed a benchmark dataset by using three popular TD assessment tools, i.e., SonarQube (v7.9, 2019), CAST (v8.3, 2018), and Squore (v19.0, 2019), to analyze 25 Java and 25 JavaScript projects and subsequently extract sets of modules exhibiting similarity to a selected profile (e.g., that of high TD levels in all employed tools).

The primary goal of this study is to train effective statistical and ML techniques to identify high-TD level modules, that is, to produce models that predict modules belonging to the *Max-Ruler* class profile. The Max-Ruler class profile, as defined by Amanatidis et al. [7], refers to modules whose reference assessment type indicates a high amount of TD based on the results of the tree applied tools. We focus exclusively on projects developed using Java, since most of the tools that we used to extend the dataset can be applied only for Java applications. As in the case of the work by Amanatidis et al. [7], we have analyzed the same 25 projects, considering their modules as units of analysis. These projects are presented in detail in Table 45. According to the authors [7], the criteria for selecting them were the programming language (i.e., Java), their public accessibility in GitHub, their popularity (more than 3 K stars) and finally, their active maintenance till the time of the study.

Table 45: Selected Projects

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

The dataset containing TD assessment of the three tools for each module of the 25 Java projects is publicly available at Zenodo[52] as an Excel file. The Max-Ruler (i.e., high-TD) modules of each project can be obtained in the form of csv files from the TD Benchmarker[53]. To begin the construction of the dataset used throughout this study, we initially downloaded the Excel file containing all the modules of the 25 Java projects under

[52] https://zenodo.org/record/3951041\#.X5ApmND7SUk

[53] http://195.251.210.147:3838/

examination and then, for each project, we downloaded the csv file containing only the high-TD modules. Subsequently, we merged high-TD module instances with the initial Excel file (containing all modules) and we labeled high-TD modules with "1". The rest of the modules were labeled as "0". This process resulted in a dataset containing 18,857 modules, out of which 1,283 belong to the Max-Ruler profile (i.e., high-TD modules).

2.1.2 Analysis Tools and Independent Variables

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

Initially, evolutionary metrics for each module of the 25 selected Java projects were computed by employing two widely used OSS tools, namely *PyDriller* [181] and *RefactoringMiner* [182]. More specifically, PyDriller (v1.15.5, 2021), i.e., a Python framework meant for mining Git repositories, was used to compute module-level Git-related metrics, such as commits count, code churn, and contributors experience across the whole evolution of the projects. Subsequently, RefactoringMiner (v2.0.3. 2020), a Java-based tool able to detect 55 different types of refactorings, was employed to compute the total number of refactorings for each module across their whole evolution. Apart from the two aforementioned tools, *Jira* and *GitHub* issue tracker APIs (depending on the project) were also used to fetch the total number of issues related to each module of the selected projects, across its entire evolution.

Besides the tools used to compute evolutionary metrics, three additional tools, namely *CK* [183], *PMD's Copy/Paste Detector (CPD)*[54], and *cloc*[55], were considered for computing metrics related to the latest commit of each module. More specifically, CK (v0.6.3, 2020), a tool that calculates class-level metrics in Java projects by means of static analysis was used to compute various OO metrics, such as CBO, DIT, and LCOM for each module. Subsequently, CPD (v6.30.0, 2020), a tool able to locate duplicate code in various programming languages, including Java, was employed to compute the total number of duplicated lines for each module. Finally, cloc (v1.88, 2020), an open-source tool able to count comment lines and source code lines in many programming languages, was used to compute the total number of code and comment lines for each module.

To train effective classification models able to identify high-TD modules, we need to investigate what kind of metrics (or sets of metrics) are closely related to whether a module is of high-TD or not. However, as stated in many studies [184], the growth of TD in a software system is related to the growth of the system itself in such a way that the larger a

[54] https://pmd.github.io/latest/pmd_userdocs_cpd.html

[55] https://github.com/AlDanial/cloc#quick-start-

system is in size, the larger its TD value will be. Therefore, to exclude the possibility that the TD level of a module is correlated with a metric only because of the size of the module, we normalized two of the metrics to control for the effect of size. The first metric that went through this transformation is *duplicated_lines*, which after a division by the *ncloc* metric of each module, was renamed to *duplicated_lines_density*. In addition, the *comment_lines* metric was also normalized but instead of dividing by the lines of code, we divided by the sum of code and comment lines (*ncloc+comment_lines*) of each module. Subsequently, *comment_lines* metric was renamed to *comment_lines_density*.

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

In Table 46, we present the metrics that were evaluated during the data collection step for each module of the 25 Java projects, along with a short description.

Table 46: Selected Metrics and Descriptive Statistics

Metrics	**Description**	**M**	**SD**	**min**	**Q_1**	**Mdn**	**Q_3**	**max**
PyDriller								
commits_count	Total number of commits made to a file in the evolution period.	11.11	15.92	1	3	7	13	317
code_churn_avg	Average size of a code churn of a file in the evolution period.	27.31	44.89	-2	9	16	30	1543
contributors_count	Total number of contributors who modified a file in the evolution period.	3.50	2.70	1	2	3	4	45
contributors_experience	Percentage of the lines authored by the highest contributor of a file in the evolution period.	77.91	20.99	17.64	60.49	82.95	98.32	100
hunks_count	Median number of hunks made to a file in the evolution period. A hunk is a continuous block of changes in a diff. This number assesses how fragmented the commit file is (i.e., lots of changes all over the file versus one big change).	1.76	1.27	0	1	1.50	2	26.50
Jira/GitHub Issue Tracker								
issue_tracker_issues	Total number of times a file name has been reported in the project’s Jira or GitHub issue tracker (mentioned within either the title or the body of the registered issue) in the evolution period.	10.08	53.05	0	0	0	3	1187
CK								
cbo	Coupling between objects. This metric counts the number of dependencies a file has.	7.44	7.91	0	2	5	10	109
wmc	Weight Method Class or McCabe's complexity. This metric counts the number of branch instructions in a file.	17.50	28.96	0	3	8	19	453

Metrics	**Description**	**M**	**SD**	**min**	**Q_1**	**Mdn**	**Q_3**	**max**
dit	Depth Inheritance Tree. This metric counts the number of "fathers" a file has. All classes have DIT at least 1.	2.04	1.84	1	1	1	2	52
rfc	Response for a Class. This metric counts the number of unique method invocations in a file.	14.15	22.58	0	1	7	17	293
lcom	Lack of Cohesion in Methods. This metric counts the sets of methods in a file that are not related through the sharing of some of the file's fields.	63.58	331.96	0	0	2	16	7503
max_nested_blocks	Highest number of code blocks nested together in a file.	1.31	1.49	0	0	1	2	21
total_methods	Total number of methods in a file.	8.68	12.04	0	2	5	10	256
total_variables	Total number of declared variables in a file.	9.54	18.30	0	1	4	10	305
RefactoringMiner								
total_refactorings	Total number of refactorings for a file in the evolution period.	15.66	52.23	0	1	1	9	1024
Copy-Paste Detector (PMD-CPD)								
duplicated_lines_density	Percentage of lines in a file involved in duplications (100 * duplicated_lines / lines of code). The minimum token length which should be reported as a duplicate is set to 100.	0.05	0.20	0	0	0	0	0.98
cloc								
comment_lines_density	Percentage of lines in a file containing either comment or commented-out code (100 * comment_lines / total number of physical lines).	0.39	0.22	0	0.22	0.36	0.54	0.94

Metrics	**Description**	**M**	**SD**	**min**	**Q_1**	**Mdn**	**Q_3**	**max**
ncloc	Total number of lines of code in a file, ignoring empty lines and comments.	97.53	142.38	2	23	51	110	1903
Note: M, SD, min, Q_1, Mdn, Q_3, max represent the mean, standard deviation, minimum, first quartile, median, third quartile, maximum values								

Therefore, the final dataset comprises a table with 18,857 rows (the number of analyzed modules) and 19 columns, where each one of the first 18 columns holds the value of a specific metric, while an extra column at the end of the table holds the value (class) of the Max-Ruler (i.e., whether a module is of high-TD or not). Since the goal of this study is to investigate the relationship of various metrics (code, people or processes-related) to whether a module is of high-TD or not and subsequently train effective ML models to identify high-TD modules, the columns that refer to metrics will play the role of independent variables, while the last column that refers to Max-Ruler class will play the role of the dependent variable, i.e., the module TD level that we try to predict. This format helped us during the classification model building phase described in Section 2.3.

2.2 Data Preparation

2.2.1 Data Pre-processing

After extracting the module-level metrics of each project (using the six tools) and merging them into a common dataset as described in Section 2.1, we proceed with appropriate data pre-processing tasks, which include missing values handling and outlier detection techniques.

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

By quickly inspecting our data, we noticed that all independent variables presented skewed distributions, with a number of extreme values. Therefore, after performing missing values removal, we proceeded with outlier detection. Outliers are extreme values that may often lead to measurement error during the data exploration process or to poor predictive performance during ML model training. ML modeling and model skill in general can be improved by understanding and even removing these outlier values [185], [186]. In cases where the distribution of values in the sample is Gaussian (or Gaussian-like), the standard deviation of the sample can be used as a cut-off for identifying outliers. In our case however, the distributions of the metrics are skewed, meaning that none of the metrics follows a normal (Gaussian) distribution. Therefore, we used an automatic outlier detection technique known as the *Local Outlier Factor (LOF)*. LOF is a technique that attempts to harness the idea of nearest neighbors for outlier detection. Each sample is assigned a scoring of how isolated it is or how likely it is to be an outlier based on the size of its local neighborhood. The samples with the largest score are more likely to be outliers. Applying LOF to the dataset resulted in a new dataset containing 17,797 modules, meaning that 812 modules were removed as the algorithm labeled them as outliers.

The descriptive statistics of the variables of the final dataset are presented in Table 46. It should be noted at this point that the performance evaluation of the selected classifiers, as will be presented later in Section 3, was also investigated without applying the aforementioned outlier detection and removal step. As expected, however, the non-removal of extreme values resulted in worse performance metrics and, therefore, we proceeded with the outlier removal step as described in this section.

2.2.2 Exploratory Analysis

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

To determine the ability of the selected 18 metrics to discriminate between high-TD and not-high-TD modules, we tested the null hypothesis that the distributions of each metric for high-TD and not-high-TD modules are equal. Since the metrics are skewed and have unequal variances (see Table 46), we used the non-parametric *Mann-Whitney U test* [187] and tested our hypothesis at the 95% confidence level ($a = 0.05$). In the first part of Table 47, we report the *p-values* of the individual Mann-Whitney U tests. As can be seen, the *p-values* of all metrics were found to be lower than the alpha level. Hence, in all cases the null hypothesis is rejected. This suggests that a statistically significant difference is observed between the values of the metrics of high-TD and not-high-TD modules, which indicates that all of these metrics can discriminate and potentially be used as predictors of high-TD software modules.

To complement the Mann-Whitney U test results, Table 47 also presents the median values of the computed metrics both for the high-TD and for the not-high-TD modules of the dataset. We chose to present the medians instead of the mean values since normal distributions of the metrics cannot be assumed. As can be seen, the median values tend to be different in each metric. In all the cases, the metrics seem to receive a higher value at high-TD modules, with the only exception of the *contributors_experience* and *comment_lines_density* metrics.

By examining the discriminative power of the studied metrics, we observed that their values demonstrate a statistically significant difference between high-TD and not-high-TD modules. In order to reach safer conclusions regarding the relationships between the selected metrics and the TD class of a module (high-TD / not-high-TD), we applied *univariate logistic regression* analysis. Univariate logistic regression focuses on determining the relationship between one independent variable (i.e., each metric) and the dependent variable (i.e., high-TD class) and has been widely used in software engineering studies to examine the effect of each metric separately [11], [143]. Thus, we used this method to help us with the process of identifying if underlying relationships between high-TD class and the selected metrics are statistically significant and in what magnitude.

Table 47 summarizes the results of the univariate logistic regression analysis for each metric, applied on our dataset. Column "*Pseudo R^2*" gives goodness-of-fit index pseudo R-squared (McFadden's R^2 index [188]), which measures improvement in model likelihood over the null model. Columns "*p-value*" and "*SE*" show the statistical significance and the standard error for the independent variables respectively. We set the significance level at *a = 0.05*. Metrics with *p-values* lower than 0.05 are considered statistically significant to high-TD class. On the contrary, metrics with *p-values* greater than 0.05 can be removed from further analysis, since they are not considered statistically significant. In our case, all 18 metrics are significantly related to the probability of high TD (*p-value≤0.05*).

Table 47: Discriminative Power and Univariate Logistic Regression Results

Metrics	**Discriminative Power**			**Univariate logistic regression**			
	Median		**Mann-Whitney U test (p-value)**	**Pseudo R^2**	**p-value**	**SE**	**OR (95% CI)**
	not-high-TD	**high-TD**					
commits_count	6	21	<0.001	0.167	<0.001	0.002	1.056 (1.053 - 1.060)
code_churn_avg	15	24	<0.001	0.038	<0.001	0.001	1.010 (1.009 - 1.011)
contributors_count	3	5	<0.001	0.120	<0.001	0.009	1.315 (1.293 - 1.339)
contributors_experience	84	72	<0.001	0.016	<0.001	0.001	0.984 (0.981 - 0.987)
hunks_count	1	2	<0.001	0.035	<0.001	0.017	1.358 (1.313 - 1.405)
issue_tracker_issues	0	5	<0.001	0.006	<0.001	0.000	1.003 (1.002 - 1.003)
cbo	5	15	<0.001	0.201	<0.001	0.003	1.130 (1.123 - 1.137)
wmc	7	64	<0.001	0.389	<0.001	0.001	1.055 (1.053 - 1.058)
dit	1	2	<0.001	0.018	<0.001	0.012	1.168 (1.141 - 1.196)
rfc	6	49	<0.001	0.324	<0.001	0.001	1.059 (1.056 - 1.062)
lcom	1	92	<0.001	0.064	<0.001	0.001	1.002 (1.001 - 1.002)
max_nested_blocks	1	3	<0.001	0.233	<0.001	0.020	2.224 (2.139 - 2.312)

Metrics	Discriminative Power			Univariate logistic regression			
	Median		Mann-Whitney U test (p-value)	Pseudo R^2	p-value	SE	OR (95% CI)
	not-high-TD	high-TD					
total_methods	4	21	<0.001	0.175	<0.001	0.002	1.073 (1.069 - 1.077)
total_variables	3	38	<0.001	0.371	<0.001	0.002	1.085 (1.081 - 1.089)
total_refactorings	1	22	<0.001	0.124	<0.001	0.000	1.014 (1.013 - 1.015)
duplicated_lines_density	5	14	<0.001	0.022	<0.001	0.001	1.016 (1.014 - 1.018)
comment_lines_density	38	17	<0.001	0.169	<0.001	0.002	0.929 (0.926 - 0.935)
ncloc	47	366	<0.001	0.531	<0.001	0.000	1.014 (1.014 - 1.015)

In the last column of Table 47, we present the *Odds Ratio (OR)* of each metric. In short, the *OR* is the ratio of the probability of an event occurring to the event not occurring. Mathematically, it can be computed by taking the exponent of the estimated coefficients, as derived by applying Logistic Regression. The reason for considering the exponent of the coefficients is that, since the model is Logit, we cannot directly draw useful conclusions by inspecting the initial coefficient values (the effect will be exponential). Thus, converting to *OR* is more intuitive in the interpretation, as follows: *OR=1* means same odds, *OR<1* means fewer odds, and *OR>1* means greater odds. For example, by inspecting the "*OR*" column in Table 47 we observe that the metric *contributors_count* has an *OR* of 1.31, suggesting that for one unit increase in the number of contributors (i.e., an additional contributor) we expect the odds of a module being labeled as high-TD to be increased by a factor of 1.31 (i.e., a 31% increase). On the other hand, the metric *comment_lines_density* has an *OR* of 0.92 suggesting that for one unit increase in the percentage of the lines containing a comment we expect about 0.08 times decrease (i.e., a 8% decrease) in the odds of a module being labeled as high-TD.

To conclude the exploratory analysis, the Mann-Whitney U test revealed that all metrics can discriminate and potentially be used as predictors of high-TD software modules. Furthermore, the univariate logistic regression analysis suggested that all of the metrics were found to be significantly related to the probability of high TD (*p-value≤0.05*). Therefore, the entire set of metrics presented in Table 46 will be considered as input during the construction of the models described in Section 2.3. After all, some of the ML models used in this study are known to eliminate variables that they consider insignificant along their iterations based on embedded feature selection methods.

2.3 Model Building

This section provides details regarding the model building process that involves specific steps that are (i) *model selection*, (ii) *model configuration* and (iii) *performance evaluation*. Generally speaking, the design of our experimental study involves a candidate set of classification learning algorithms $A = \{A_1, \ldots, A_K\}$, where the objective of each candidate A_K is to learn a mapping function from input variables (or predictors) x_i to output variable (or response) y_i, where $y_i \in \{0,1\}$ given a set of *n* input-output observations of the form $D = \{(x_i, y_i)\}_{(i=1)}^{n}$. In our case, the output variable is the characterization of TD into *not-High-TD* (denoted by 0) and *high-TD* (denoted by 1) and the output variables that are derived through the process described in Section 2.1.

2.3.1 Model Selection

Having in mind that there is a plethora of prediction candidates that can be used for classification purposes, we decided to explore a specific set of well-established statistical and ML algorithms that have been extensively applied in other similar experimental studies for code smell or bug prediction [106], [110]. More specifically, we used seven different classifiers that are summarized in Table 48: two are simple statistical/probabilistic models (*Logistic Regression* (LR), *Naïve Bayes* (NB)), and five are more sophisticated single (*Decision Trees* (DT), *K-Nearest Neighbor* (KNN), *Support Vector Machines* (SVM)) or ensemble (*Random Forest* (RF) and *eXtreme Gradient Boosting* (XGBoost)) ML models.

Table 48: Classification Models of the Experimental Setup

Classification Model	**General Idea**	**Best Tuning Parameters**
Logistic Regression (LR)	Employs a logit function to predict the probability of a categorical target variable belonging to a certain class.	penalty='none', solver='lbfgs'
Naive Bayes Classifier (NB)	Applies Bayes' theorem to construct a probabilistic classifier based on the independence assumption between variables.	N/A
Decision Tree (DT)	Constructs hierarchical models composed of decision nodes and leaves to predict the class of the target variable.	criterion='gini', max_depth='none'
k-Nearest Neighbor (kNN)	Uses a distance function to predict the class of a new data point based on the majority label of the k data points closest to it.	n_neighbors=8
Support Vector Machine (SVM)	Tries to find the optimal N-dimensional hyperplane that maximises the margin between	kernel='rbf', C=1

Classification Model	**General Idea**	**Best Tuning Parameters**
	the data points to classify them into predefined classes.	
Random Forest (RF)	A decision-tree-based ensemble algorithm that collects all the votes that are produced by its decision trees and provides a final classification result.	n_estimators=100, criterion='gini', max_depth='none'
XGBoost (XGB)	A decision-tree-based ensemble algorithm that uses multiple decision trees to predict an outcome based on a gradient boosting framework.	n_estimators=50, booster ='gbtree'

Although the candidate classifiers have shown to be effective in many application domains, their performances are significantly affected by the *class imbalance problem* [189]. This challenging issue occurs in classification tasks, when the class distributions of the response variable are highly imbalanced, which practically means that the one class is underrepresented (minority class) compared to the other (majority class). In this study, the response variable suffers from *intrinsic* imbalance [189] (ratio ~1:15), due to the nature of the problem and thus, it presents a highly-skewed class distribution. This fact causes problems to the learning process, which leads to poor generalization ability of classifiers, since most of these algorithms assume balanced class distributions. Moreover, the majority of these learners aims to maximize the overall accuracy (or total error rate) of the fitted model, which results in turn, to classifiers that tend to be biased with models presenting an excellent prediction performance for the majority class but at the same time, extremely poor performance for the minority class.

The class imbalance problem is usually addressed by employing two widely-known techniques, namely *oversampling* and *undersampling* [190]. Oversampling achieves the desired balance between classes by duplicating the minority class data, while undersampling does so by removing randomly chosen data from the majority class. There are cases where a combination of over-and under-sampling works better [190], i.e., by using specific ratios of both techniques until the desired trade-off between precision and recall is achieved.

In the current study, we opted for oversampling rather than undersampling, as we wanted to avoid removing valuable data from our dataset (and therefore avoid a possible drop in the models' performance). Furthermore, as regards the oversampling ratio, we chose to augment the minority class until its data instances become equal to those of the majority class (ratio 1:1). The reasoning behind this decision is twofold. First, a 1:1 class ratio guarantees the best possible representation of the minority class, which in this study we consider more important than the majority class. Second, during the evaluation process of ML models (described in Section 2.3.2), we noticed that having a 1:1 class ratio resulted in better performance metrics compared to the classifiers' performance when trying different ratios of oversampling, undersampling, or combinations of both.

To overcome this inherent limitation of classification learners to provide accurate predictions for both minority and majority classes in the presence of class imbalanced input variable, we made use of a well-known synthetic sampling method, namely the *Synthetic Minority Oversampling Technique (SMOTE)* [190]. *SMOTE* performs oversampling of the minority class by generating new synthetic instances based on the nearest neighbors belonging to that class. More details on where and how we integrated SMOTE into our experimental setup are presented in Section 2.3.2.

2.3.2 Model Configuration

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

Starting with the validation phase, we performed *3×10 repeated stratified cross-validation* [191]. The dataset was randomly split into 10 folds, from which nine participate in training and the remaining one participates in testing, rotating each time the test fold until every fold serves as a test set. Moreover, each fold was initially stratified to properly preserve the initial high-TD – not-high-TD class ratio. Subsequently, oversampling (using SMOTE) was integrated into the cross-validation process. Despite the fact that the training folds undergo oversampling, the test fold always retains the initial class ratio for a proper model validation. The entire 10-fold cross-validation process was then repeated 3 times to account for possible sampling bias in random splits. Hence, the computed performance metrics of each classifier produced during the cross-validation are the averaged values of 30 (*3×10*) models trained and evaluated on the same dataset. In that way, we reduce the possibility of having bias introduced by the selection of non-representative subsets of the broader dataset.

As a common practice during the validation phase, we employed hyper-parameter tuning to determine the optimal parameters for each classifier and therefore increase its predictive power. To do so, we used the *Grid-search* method [168], which is commonly used to find the optimal hyper-parameters, by performing an exhaustive search over specified parameter values for an estimator. We chose the *F_2-measure* as the objective function of the estimator to evaluate a parameter setting (further details regarding the selection of *F_2-measure* are presented in Section 2.3.3). Hyper-parameter selection was performed using the stratified *3×10*-fold cross validation to avoid overfitting and ensure that the fine-tuned classifiers have a good degree of generalization before assessing their performance on the test set. In Table 48, we report the optimal hyper-parameters as they were adjusted during the tuning process.

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

Finally, after fine-tuning and evaluating our classifiers through the validation phase described above, we proceeded with the final test phase. More specifically, we re-trained the fine-tuned classifiers using the training/validation set and applied them on the test set. Oversampling (using SMOTE) and MinMax transformation were used again, but only on the training/validation set, which at this point is used solely for model training. The data samples comprising the test set were never seen by the models during the previous phase (training/validation). Our goal was to create a hypothetical, but practical situation, where a new system or a set of systems is available, and the proposed classification framework must be applied to predict the presence of high-TD modules in the new (set of) system(s). In contrast to the cross-validation process where the performance metrics are averaged among the 30 (3×10) iterations, in this case, the computed metrics of each classifier comprise a single value, since each model was executed on the entire test set once. For our experiments, we used the Python language and more specifically the *scikit-learn*[56] ML library.

2.3.3 Performance Evaluation

In practice, the performance evaluation of a binary classifier is usually assessed through alternative metrics based on the construction of a *confusion matrix* (Table 49). Typically, in a class-imbalanced learning process, the minority and majority classes are considered to represent the positive (+) and negative (-) outputs, respectively [192].

Table 49: Confusion Matrix

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

[56] https://scikit-learn.org/stable/

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

Based on the previous considerations, we made use of appropriate evaluation metrics that have been proposed for dealing with the class imbalanced problem [189]. The *F-measure* is a widely used metric combining *precision* and *recall*, which are defined as follows:

$$Precision = \frac{\text{TP}}{\text{TP+FP}}$$

$$Recall = \frac{\text{TP}}{\text{TP+FN}}$$

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

where β is a coefficient for adjusting the relative importance of precision with respect to recall. Each of the above metrics represents different aspects of prediction performance providing straightforward directions about the quality of a classifier. Both precision and recall focus on the minority (positive) class, while the former is known a measure of *exactness* and the latter as a measure of *completeness* [192]. These two metrics are combined in order to provide an overall evaluation metric related to the *effectiveness* of a classifier on predicting correctly the minority cases, that is, the class of great importance in our experimental setup. More importantly, the *F-measure* provides a straightforward manner to place more importance to *FN* compared to *FP* misclassified cases by setting $\beta = 2$ leading to a special case of F-measure, known as *F_2-measure*. In other words, we consider it more risky for a development team to ignore modules that might have high TD (i.e., to suffer from the presence of many *FNs* which might lead to inappropriate decisions with respect to maintenance) than to go through many modules which are marked as problematic whereas they aren't (i.e., *FPs*). Therefore, for evaluating the selected set of binary classification models, we chose the *F_2-measure* as one of our main performance indicators.

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

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

Following a similar approach to other studies [11], [164], [166], we use the number of modules to inspect as an estimator of the inspection effort. We define the *Module Inspection (MI)* ratio as the ratio of modules (software classes) predicted as high-TD to the total number of modules:

$$MI = \frac{(\mathrm{TP}+\mathrm{FP})}{(\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN})}$$

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

To conclude, the *F_2-measure* and *MI* performance metrics are used as the basis for the comparison and selection of the best model(s), as their combination provides a complete picture of the model performance in predicting high-TD modules. In fact, *F_2-measure* indicates how effectively the produced models detect *TPs* but without ignoring *FPs* (i.e., by weighting recall higher than precision), whereas the *MI* indicates how efficient the models are in predicting *TPs*, based on how many *FPs* have to be triaged by the developers until a *TP* is detected. For completeness, in the experimental results presented in Section 3 we report other indicators like *precision*, *recall* and *Precision-Recall curves*.

3 Experimental Results

3.1 Quantitative Analysis

Table 50 summarizes the performance metrics for the set of examined classifiers validated and tested through the *training-validation-test* approach introduced in Section 2.3.2. More specifically, "Validation" column presents the results obtained for each classifier through the repeated stratified cross-validation process (validation phase). To this regard, the results present an overall indicator computed by averaging the performance metrics of *$k=10$* folds over three repeated executions. Additionally, the standard deviation is also reported. On the other hand, "Test" column presents the performance metrics of each examined classifier on the test (holdout) set, i.e., the 20% of the dataset that has not been used during training/validation. In both cases, the best classifier in terms of each performance measure is denoted in bold font. As mentioned in Section 2.3.3, we considered *F_2-measure* and *MI* ratio as the main performance indicators, since they cover both accuracy and practicality of the produced models.

Table 50: Evaluation Results for All Classifiers

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

The findings of Table 50 indicate that there is a subset of superior classifiers presenting small divergences in terms of *F_2-measure*. More specifically, as regards the validation phase, LR achieves the highest *F_2-measure* score with a value of 0.764. However, XGB, RF and SVM classifiers are following closely with *F_2-measure* scores ranging between 0.758 and 0.761. On the other hand, KNN, NB and DT seem to have noticeably lower *F_2-measure* scores. In terms of results in the test (holdout) set, we notice that the performance of the classifiers is not only preserved compared to the validation phase results, but is also slightly higher. More specifically, RF achieves the highest *F_2-measure* score with a value of 0.790, followed closely by XGB, LR and SVM with *F_2-measure* scores ranging between 0.781 and 0.788. Certainly, the overall performance evaluation is totally based on statistical measures evaluated from samples and for this reason, they contain significant variability that could lead to erroneous decision-making regarding the superiority of a subset of classifiers against competing ones. Thus, identifying a subset of superior models should be based on statistical hypothesis testing [193].

To this regard, within the context of evaluating the examined classifiers through the validation phase, we made use of a multiple hypothesis testing procedure, namely the *Scott-Knott (SK)* algorithm [194] that takes into account the error inflation problem caused by the simultaneous comparison of multiple prediction models [195]. A major advantage compared to other traditional hypothesis procedures is the fact that the algorithm results into mutually exclusive clusters of classifiers with similar performance and thus, the interpretation and decision-making is a straightforward process. Finally, the algorithm is totally based on well-established statistical concepts of *Design of Experiments (DoE)* taking into account both *treatment* and *blocking* factors. In our case, the validation phase consists of 30 repeated performance measurements (i.e., *F_2-measure*) evaluated from each classifier (*treatment effect*) on *$k=10$* folds after three repeated executions (*blocking effect*). Describing briefly, the treatment effect takes into account the differences between the set of candidate classifiers, whereas the blocking effect is an additional factor that should be taken into account but we are not directly interested in and it is related to the splitting of dataset into different test sets on each execution of the stratified cross-validation process.

The findings of the SK algorithm indicated a statistically significant treatment effect (*$p<0.001$*) on *F_2-measure*, which means that the observed differences among the seven classifiers cannot have occurred just by chance and there is indeed, a subset of superior classifiers in terms of *F_2-measure*. To this regard, the SK algorithm resulted into four homogenous clusters of classifiers that can be found in the last column of Table 50. The classifiers are ranked starting from the best (denoted by A) to the worst (denoted by D) clusters, whereas classifiers that do not present statistically significant differences are grouped into the same cluster. The best cluster encompasses four classifiers (RF, XGB, LR, and SVM) that present similar prediction performances. On the other hand, DT can be considered as the worst choice for identifying if a module is a high-TD or not, based on metrics.

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

While we consider it riskier for a development team to ignore actual high-TD modules than inspect many modules which have been falsely reported as high-TD, a statement that a model showing high recall and low precision is better than a model showing high precision and low recall is bold and arguable. Developers may prefer to inspect a large amount of potentially problematic modules because they can afford a lot of resources for refactoring activities. On the other hand, another development team may prefer a model that reduces the waste of effort even at the cost of missing some cases of high-TD modules. After all, TD management is about reducing the wasted effort during development [31]. We can argue that the RF classifier strikes a balance between these two cases, since it provides a quite satisfactory recall score, while at the same it preserves a lower (but significantly higher compared to the other models) precision score.

To complement the comparison of classification performance metrics presented above, we also present *Precision-Recall curves* [196] of each classifier, averaging the results over each fold of the *3×10* repeated stratified cross-validation process followed within the validation phase. A Precision-Recall curve is a plot that shows the tradeoff between precision (*y-axis*) and recall (*x-axis*) for different probability cutoffs, thus providing a graphical representation of a classifier's performance across many thresholds, rather than a single value (e.g., F_2*-measure*). In contrast to the *ROC* curves that plot the *FP* vs *TP* rate, Precision-Recall curves are a better choice for imbalanced datasets, since the number of *TNs* is not taken into account [196]. We also provide values for the *area under curve* (AUC) for each classifier, summarizing their skill across different thresholds. A high AUC represents both high recall and high precision. As can be seen by inspecting Figure 48, the curve of RF classifier is closer to the optimal top-right corner, which practically means that it can achieve similar recall scores by sacrificing less precision compared to the other classifiers. In addition, RF presents the highest AUC, with a value of 0.77, followed by LR (*AUC=0.75*) and XGB (*AUC=0.74*). The dashed horizontal line depicts a "no-skill" classifier, i.e., a classifier that predicts that all instances belong to the positive class. Its *y-value* is 0.067, equal to the ratio of positive cases in the dataset.

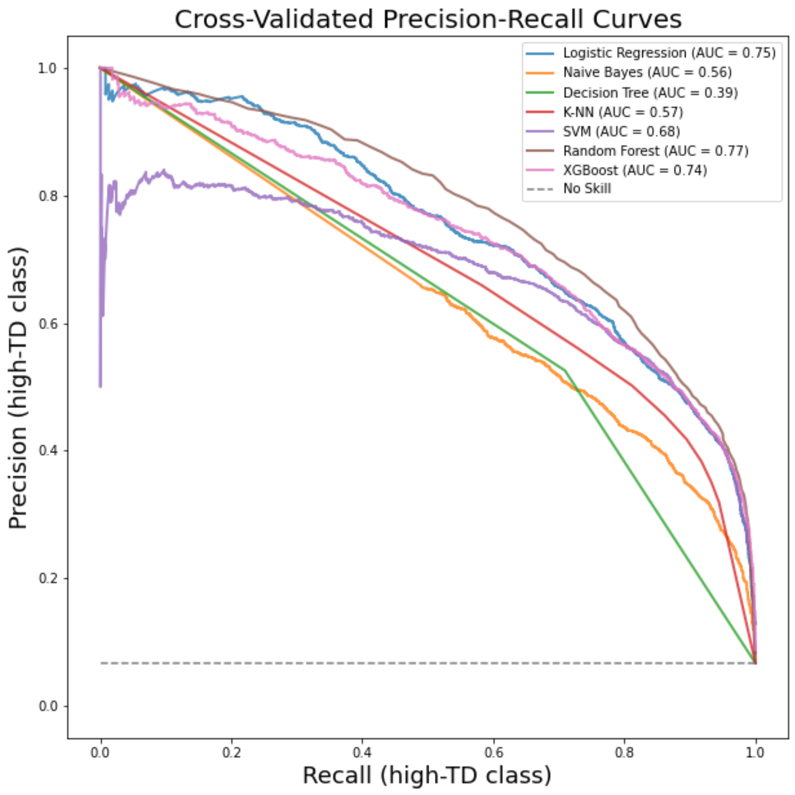


Figure 48: Cross-validated Precision-Recall Curves

Regarding the second main performance indicator, i.e., the *MI* ratio, by inspecting the test phase metrics in Table 50 we can see that RF has the best (i.e., lowest) score with a value of 0.092. This can be seen as a logical outcome considering that RF achieved by far the best precision score among the examined classifiers. Possible tradeoffs between high recall and low precision have also implications in terms of cost effectiveness. As an example, let's consider a project coming from our dataset. The JClouds application has around 5,000 modules, among which 126 are labeled as high-TD. The SVM classifier provided the highest recall score (0.954) with a 14% *MI* ratio. This means that a development team would have to inspect *5,000 × 14% = 700* potentially high-TD modules in order to identify the 95% of real high-TD cases, that is, 120 modules (and miss 6 high-TD modules). On the other hand, the RF classifier provided a recall score of 0.858 with a 9% *MI* ratio. This means that a development team would have to inspect *5,000 × 9% = 450* potentially high-TD modules to identify the 86% of real high-TD cases, that is, 108 modules (and miss 18 high-TD modules). Similarly, if the additional 250 modules that need inspection if adopting the SVM over the RF model are worth the additional 12 high-TD modules identified by the SVM model is a company-specific decision. However, we believe that pursuing moderate precision and high recall might be a more cost-effective approach than pursing the highest recall at the cost of a very low precision.

3.2 Sensitivity Analysis

To investigate potential variations and ranking instabilities concerning the prediction capabilities of the examined classifiers per project, we decided to perform *sensitivity analysis*. Regarding the experimental setup, we performed a *3×10* repeated stratified cross-validation process twenty one times, using at each of the 21 iterations a single project as the dataset for both training and testing each examined classifier. To evaluate the classifiers' performance we considered again the *F_2-measure*. We have to clarify that we were not able to conduct the above analysis for four projects (i.e., gson, javacv, vassonic and xxl-job), due to the limited number of modules (min=64, max=112) that made the cross-validation process impossible to execute. The results of the sensitivity analysis are summarized in the Table 51.

Table 51: Sensitivity Analysis Results

Dataset	**#modules**	**RF**	**LR**	**SVM**	**XGB**	**KNN**	**NB**	**DT**
merged	17797	**A (0.760)**	**A (0.764)**	**A (0.758)**	**A (0.761)**	B (0.731)	C (0.684)	D (0.663)
arduino	231	**A (0.680)**	**A (0.702)**	**A (0.641)**	**A (0.728)**	**A (0.770)**	**A (0.774)**	**A (0.635)**
arthas	279	**A (0.641)**	B (0.554)	B (0.522)	**A (0.666)**	**A (0.707)**	**A (0.623)**	B (0.536)
azkaban	510	**A (0.793)**	B (0.697)	B (0.682)	**A (0.847)**	B (0.722)	**A (0.775)**	B (0.748)
cayenne	1508	**A (0.751)**	**A (0.758)**	B (0.699)	**A (0.768)**	B (0.710)	B (0.730)	C (0.623)
deltaspike	668	B (0.561)	**A (0.659)**	**A (0.672)**	**A (0.612)**	**A (0.743)**	**A (0.688)**	C (0.433)

Dataset	**#modules**	**RF**	**LR**	**SVM**	**XGB**	**KNN**	**NB**	**DT**
exoplayer	636	**A (0.878)**	B (0.813)	**A (0.857)**	B (0.812)	B (0.824)	B (0.819)	B (0.780)
fop	1535	**A (0.792)**	**A (0.816)**	**A (0.824)**	**A (0.797)**	**A (0.821)**	**A (0.802)**	B (0.664)
jclouds	2889	**A (0.769)**	**A (0.791)**	**A (0.762)**	**A (0.805)**	B (0.699)	B (0.668)	B (0.692)
joda-time	165	B (0.640)	B (0.527)	B (0.557)	**A (0.840)**	B (0.674)	B (0.581)	**A (0.823)**
libgdx	1855	**A (0.805)**	**A (0.814)**	**A (0.812)**	**A (0.795)**	**A (0.772)**	**A (0.785)**	B (0.712)
maven	621	**A (0.716)**	**A (0.736)**	**A (0.723)**	**A (0.769)**	**A (0.777)**	**A (0.699)**	**A (0.670)**
mina	436	B (0.504)	**A (0.633)**	**A (0.545)**	**A (0.627)**	**A (0.576)**	**A (0.638)**	C (0.375)
nacos	390	**A (0.804)**	B (0.663)	**A (0.759)**	**A (0.758)**	**A (0.781)**	**A (0.747)**	B (0.667)
opennlp	670	**A (0.846)**	**A (0.826)**	**A (0.867)**	**A (0.829)**	**A (0.806)**	**A (0.844)**	**A (0.855)**
openrefine	593	**A (0.812)**	**A (0.784)**	**A (0.763)**	**A (0.818)**	B (0.734)	**A (0.805)**	B (0.674)
pdfbox	983	**A (0.662)**	**A (0.715)**	**A (0.692)**	**A (0.703)**	**A (0.711)**	**A (0.751)**	B (0.508)
redisson	823	**A (0.843)**	**A (0.830)**	**A (0.828)**	**A (0.835)**	**A (0.852)**	**A (0.817)**	**A (0.784)**
RxJava	771	**A (0.833)**	**A (0.850)**	**A (0.859)**	**A (0.864)**	B (0.787)	B (0.805)	B (0.782)
testng	343	**A (0.665)**	**A (0.653)**	**A (0.710)**	**A (0.700)**	**A (0.750)**	**A (0.670)**	**A (0.619)**
wss4j	486	**A (0.729)**	**A (0.757)**	**A (0.801)**	**A (0.721)**	**A (0.744)**	**A (0.752)**	B (0.653)
zaproxy	1121	**A (0.776)**	**A (0.776)**	**A (0.773)**	**A (0.809)**	**A (0.799)**	**A (0.769)**	B (0.701)

Dataset	**#modules**	**RF**	**LR**	**SVM**	**XGB**	**KNN**	**NB**	**DT**
% of experiments belonging to best cluster A		86.36%	77.27%	81.82%	95.45%	63.64%	72.73%	27.27%

Each row of the table presents the findings derived from the analysis conducted on each single dataset accompanied by the results of the SK algorithm regarding the statistical hypothesis testing procedure based on *F_2-measure*. Classifiers belonging to the best cluster are denoted in bold, whereas the last row provides an overview of classifiers' performance, that is, the percentage of experiments in which the classifiers were categorized into the set of superior models (denoted by the letter A). A first interesting finding concerns the ranking stability of classifiers characterized as superiors based on our previous cross-validation experimentation on the merged dataset (see Section 3.1), depicted also in the first row of Table 51. More specifically, XGB presents consistently outstanding prediction capabilities, since this classifier was grouped into the best cluster in 22 out of 23 experiments (95.45%). Furthermore, XGB exhibits even better *F_2-measure* scores on 13 datasets compared to the corresponding metric evaluated on the merged set of projects. RF and SVM can be also considered as a good alternative choice with noteworthy performances in the majority of the datasets. On the other hand, there is a strong indication of classifiers that consistently present moderate (KNN and NB) and poor (DT) predictive power, a finding that is aligned to the results obtained from the experimentation on the set of all projects.

Based on the results of both Section 3.1 and Section 3.2, it is of no surprise that more sophisticated algorithms, such as RF, XGB, and SVM, are performing better than other, more simple algorithms, such as DT or NB. This is a general remark, often encountered in various classification, or even regression tasks. In our case, the finding that XGB, RF and SVM are demonstrating a significantly higher performance compared to the rest of the examined classifiers might be attributed to the fact that these three algorithms stand out when non-linear underlying relationships exist in the data, which also applies in our case. Furthermore, the fact that XGB and RF, that is, two ensemble decision-tree-based algorithms, are ranked as the top 2 best-performing algorithms is also in line with the findings of similar studies (as presented in Chapter III), where tree-based ensemble algorithms (e.g., RF) have been proved to be among the most popular and effective classifiers in related empirical SE studies.

3.3 Illustrative Examples of Identifying High-TD Classes

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

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

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

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

4 Tool Implementation

4.1 *Implementation*

As a proof of concept, the proposed classification framework described in this chapter has been implemented in the form of a tool. This tool, named "TD Classifier", is implemented as a web application, including both a backend and its associated frontend. It offers interactive visualizations that enable the prompt identification of classes that are more likely to be problematic. A running instance of the tool is available online[57], enabling in that way its adoption by developers in practice, and, in turn, its further quantitative and qualitative evaluation by the community.

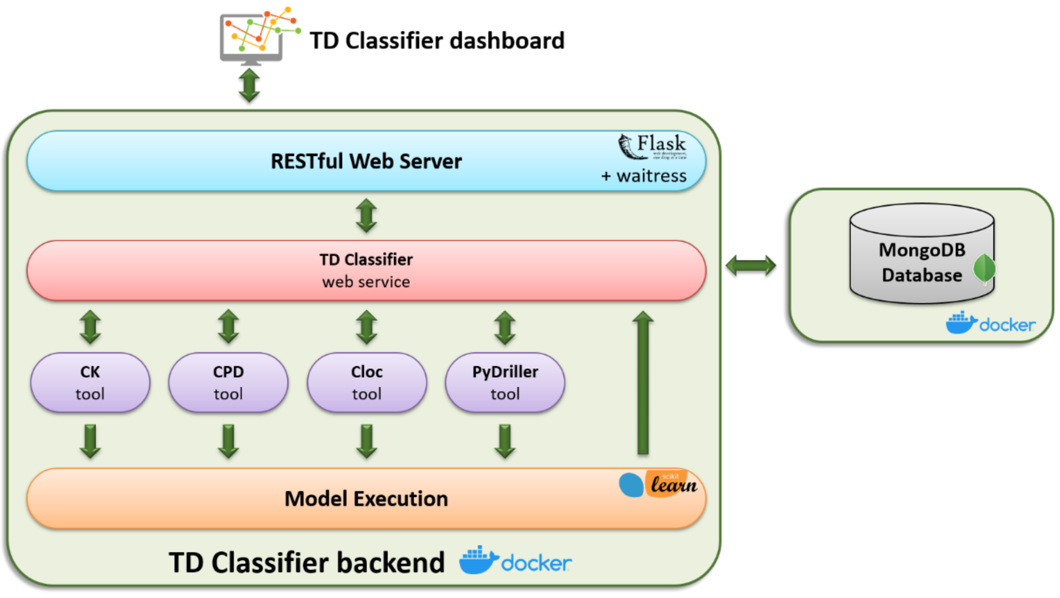


Figure 49: Overall Architecture of the TD Classifier

Figure 49 depicts the overall architecture of the TD Classifier tool. The tool is implemented in the form of a web application, including both a backend and its associated frontend. The backend of the tool, developed in Python, is actually a Microservice, making it easily accessible to the software engineering community and facilitating its integration into third-party software.

As can be seen by Figure 49, the entry point of the TD Classifier backend is a RESTful web server that uses the Flask web framework wrapped inside Waitress, a Python WSGI production-ready server. At a lower level, the server exposes the TD Classifier API, implemented as an individual web service. This web service plays the role of an orchestrator that is responsible for: i) cloning a project, ii) invoking the analysis tools described in Section 2.1.2 (i.e., PyDriller, CPD, etc.) for the collection of the required metrics, iii) executing the pre-trained classifier and finally returning the results.

[57] http://160.40.52.130:3000/tdclassifier

To facilitate the building and deployment process, Docker technology has been considered. More specifically, the tool's backend has been implemented as an individual Docker Image and deployed as an individual Docker Container. For this purpose, a Docker File, i.e., a "recipe" that describes what tools should be bundled inside the container, has been created and is available online in the repository of the tool. In that way, potential users can generate their own TD Classifier backend container easily, by building the Image from scratch and hosting it locally. In addition, apart from the scripts of the TD Classifier backend per se, all of the third-party analysis tools that are responsible for gathering the required model input are also bundled into the Docker Image as standalone executables (in the form of either jar files or shell scripts natively provided by the developers of the tool). This setup not only enhances portability by making the tool easy to install but also speeds up execution time as no external calls are required for their execution. It is worth mentioning that the analysis tools run in parallel, in order to reduce the tool's overall execution time.

Finally, a MongoDB database dedicated to storing the output of the TD Classifier web service allows the tool to quickly retrieve past results upon demand, without having to go through the time-consuming process of re-executing the analysis tools and the dedicated classifier. The database is optional and is also "dockerized" within its own container.

Apart from the tool's backend, an intuitive frontend (i.e., user interface) has been also implemented in order to facilitate its adoption in practice. The TD Classifier frontend has been integrated into the SDK4ED platform, which is the main outcome of the successful culmination of the SDK4ED[58] European project. The frontend of the tool, developed using the React framework, communicates seamlessly with the backend, allowing the easy invocation of the main functionalities (i.e., web services) that the tool provides, and the visualization of the produced results. Additional information regarding the TD Classifier frontend is presented in Section 4.2, where we provide a case study on a real-world open-source software application that evaluates the usefulness of the proposed tool in practice.

4.2 Evaluation

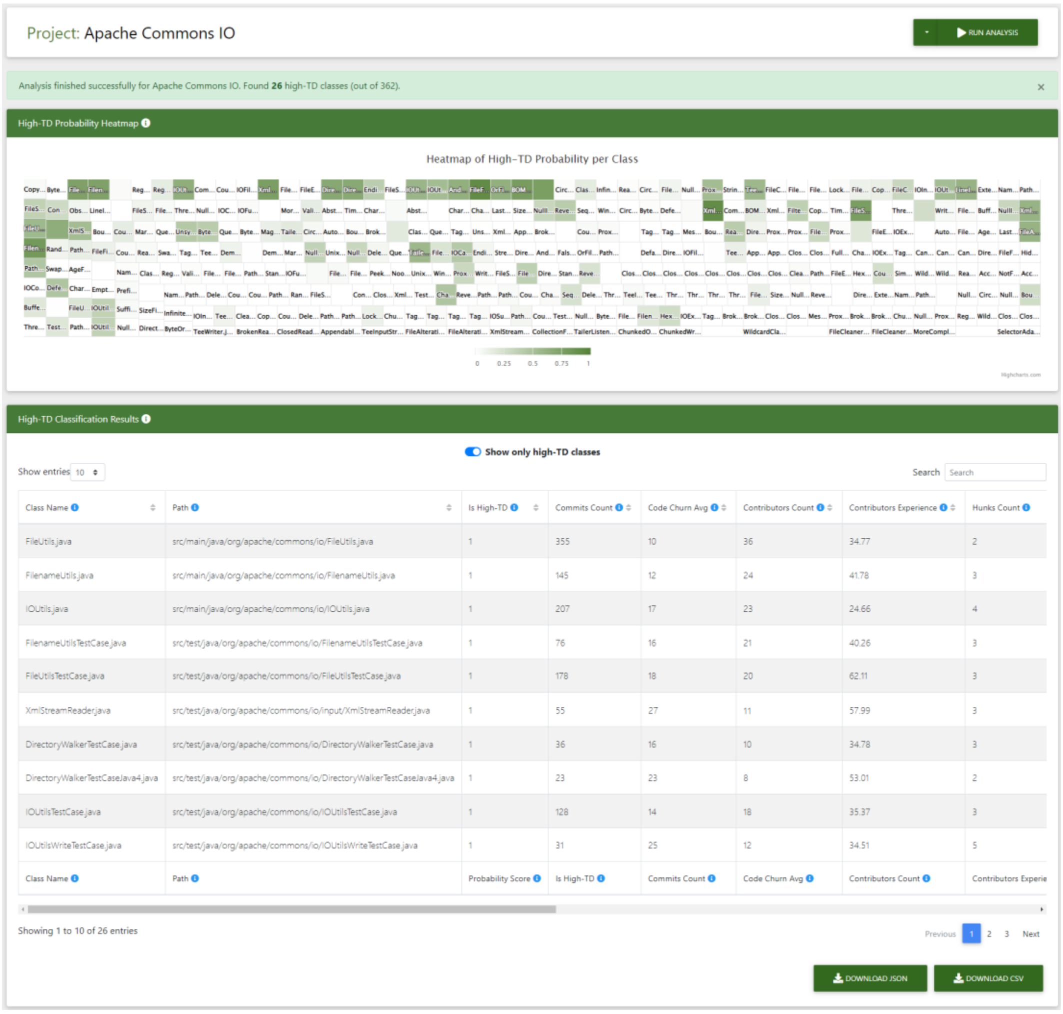
In this section, the proposed tool is demonstrated through a case study on a real-world open source software application. This case study also acts as a testbed for evaluating the ability of the proposed approach to identify candidate high-TD items. To evaluate the effectiveness of the TD Classifier tool, we use a popular open source Java project, namely Apache Commons IO[59]. Apache Commons IO is a library of utilities to assist with developing IO functionality, whose code is hosted on GitHub[60] with more than 3.000 commits. It should be mentioned that this project has not been used in our research study for model training or evaluation.

[58] https://sdk4ed.eu/

[59] https://commons.apache.org/proper/commons-io/

[60] https://github.com/apache/commons-io

Since TD Classifier is part of the overall SDK4ED Dashboard, the user must initially navigate to the SDK4ED Dashboard home page[61] and select an existing project, or create a new one. Then, they can navigate to the "TD Classifier" panel (located under the "Technical Debt" drop-down button on the top navigation menu), where they can select the type of analysis they would like to execute and click on the "Run Analysis" button to start the process. Currently, the tool supports three types of analysis: A *Fast* analysis will take into account only the software classes that were modified during the last 100 commits, a *Normal* analysis the classes that were modified during the last 1000 commits, whereas a *Full* analysis will take into account the whole project history.


Project: Apache Commons IO
RUN ANALYSIS
High-TD Probability Heatmap
Heatmap of High–TD Probability per Class
0
0.25
0.5
0.75
1
Highcharts.com
High-TD Classification Results
Show only high-TD classes
Show entries 10
Search
Previous
1
2
3
Next
DOWNLOAD JSON
DOWNLOAD CSV

Analysis finished successfully for Apache Commons IO. Found **26** high-TD classes (out of 362).

Class Name	Path	Is High-TD	Commits Count	Code Churn Avg	Contributors Count	Contributors Experience	Hunks Count
FileUtils.java	src/main/java/org/apache/commons/io/FileUtils.java	1	355	10	36	34.77	2
FilenameUtils.java	src/main/java/org/apache/commons/io/FilenameUtils.java	1	145	12	24	41.78	3
IOUtils.java	src/main/java/org/apache/commons/io/IOUtils.java	1	207	17	23	24.66	4
FilenameUtilsTestCase.java	src/test/java/org/apache/commons/io/FilenameUtilsTestCase.java	1	76	16	21	40.26	3
FileUtilsTestCase.java	src/test/java/org/apache/commons/io/FileUtilsTestCase.java	1	178	18	20	62.11	3
XmlStreamReader.java	src/test/java/org/apache/commons/io/input/XmlStreamReader.java	1	55	27	11	57.99	3
DirectoryWalkerTestCase.java	src/test/java/org/apache/commons/io/DirectoryWalkerTestCase.java	1	36	16	10	34.78	3
DirectoryWalkerTestCaseJava4.java	src/test/java/org/apache/commons/io/DirectoryWalkerTestCaseJava4.java	1	23	23	8	53.01	2
IOUtilsTestCase.java	src/test/java/org/apache/commons/io/IOUtilsTestCase.java	1	128	14	18	35.37	3
IOUtilsWriteTestCase.java	src/test/java/org/apache/commons/io/IOUtilsWriteTestCase.java	1	31	25	12	34.51	5
Class Name	Path	Probability Score	Is High-TD	Commits Count	Code Churn Avg	Contributors Count	Contributors Experie

Showing 1 to 10 of 26 entries

Figure 50: Heat map and complementary table visualizing the TD Classifier results for the Apache Commons IO project

For the sake of demonstrating the TD Classifier tool on the Apache Commons IO project, a Full analysis is selected. Once the process finishes, the user is presented with a screen that visualizes the results, as depicted in Figure 50. On the upper part of the panel, a

[61] http://160.40.52.130:3000/

notification informs the user that the tool has identified 26 high-TD classes, out of the total 362 analyzed classes.

To effectively convey the output of the TD Classifier to the developers and project managers of the software application, a heat map has been selected as a means of visualization. As can be seen by inspecting Figure 50, the middle panel contains a heat map that presents the classification results retrieved from the analysis of the Apache Commons IO project. In particular, the rectangles correspond to the classes of the selected software project, as identified and analyzed by the data collection tools. The color of each rectangle denotes the probability of the corresponding class to be problematic (i.e., have high-TD), as calculated by the dedicated pre-trained classifier. More specifically, the greener the rectangle, the higher the probability that a class is problematic. In that way, the tool enables practitioners to promptly identify candidate TD items and therefore, plan more targeted refactoring activities.

Apart from the heat map, a complementary table comprising the detailed results of the analysis is presented at the bottom panel of Figure 50. This table contains supplementary information that, in addition to the information of whether a class is of high TD or not, includes also all of the 18 development process metrics (e.g., commits count, code churn, and contributors experience) and code metrics (e.g., CBO, DIT, and LCOM) that were calculated during the data collection process of the analysis. A toggle button at the top of the table allows the user to focus only on the classes that were identified as problematic. Moreover, through the table's sorting functionality, the user can rank the results based on any characteristic of interest, while a search field allows the easy retrieval of information for any specific class. Finally, two dedicated buttons at the bottom of the table allow the user to download the analysis results in JSON or CSV format, for further processing.

To perform a preliminary comparative analysis between the TD Classifier tool and other well-established TD assessment tools, we analyzed the Apache Commons IO project using SonarQube. SonarQube is one of the three tools that helped build the ground truth [7] that was used for the construction of our model and the only one among the three tools that does not require a commercial licence. It should also be noted that SonarQube metrics are not part of our classifier's features.

As a first example, let us consider *FileUtils.java*, i.e., the first class in our list of identified high-TD classes, as presented in Figure 50. To have an indication of whether this class was correctly labeled as high-TD in the first place, we took a closer look at the analysis results produced by SonarQube. More specifically, SonarQube has ranked this class 3rd in terms of issues (22 identified), 2nd in terms of cyclomatic complexity (value of 265), and 5th in terms of TD accumulation (3 hours). In another example, let us consider the second entry in our high-TD classes list, i.e., *FilenameUtils.java*. By inspecting SonarQube results, we observed that the tool has ranked this class 6th in terms of issues (14 identified), 3rd in terms of cyclomatic complexity (value of 226), and 2nd in terms of TD accumulation (4.5 hours). Finally, let us consider the third entry in our high-TD classes list, i.e., *IOUtils.java*. By revisiting SonarQube results, we observed that it has ranked this class 2nd in terms of issues (29 identified), 1st in terms of cyclomatic complexity (value of 283), and 1st in terms of TD accumulation (5 hours). Similar observations can be also made for the rest of the classes identified as high-TD by our tool. The above comparison results provide us with preliminary evidence that the classes identified as high-TD by the TD Classifier are indeed problematic.

On the other hand, we identified cases of classes that were labeled as problematic by our tool, but at the same time their TD-related importance was probably underestimated by SonarQube. As an example, let us consider *XmlStreamReader.java*. As can be seen by inspecting the list of identified high-TD classes in Figure 50, the relatively high complexity (wmc=131), low cohesion (lcom=147), high code churn average (27), or high number of contributors (11) make this class a good high-TD candidate. However, SonarQube has labeled this class as having no TD (0 minutes), probably because it only considers code smell issues to calculate TD remediation effort. While large-scale analysis is required to further evaluate the validity and generalizability of the above findings, our preliminary comparative analysis combined with the relatively high performance obtained through our related research work presented in this chapter highlights the practical importance of TD Classifier. Ultimately, the derived tool subsumes the collective knowledge that would be extracted by combining the results of various well-established TD tools, therefore increasing the chances that the identified classes suffer indeed from high-TD.

TD Classifier will continue to evolve to meet the challenges posed by its use in both academia and practice. We plan to improve the tool's performance and scalability. We also plan to extend our classification framework in other programming languages (e.g., C/C++, python, JavaScript, etc.), by incorporating additional analysis tools into the analysis pipeline.

5 Implications to Researchers and Practitioners

In this study, we made an attempt to leverage the knowledge acquired by the application of leading TD assessment tools in the form of a benchmark of high-TD modules. Considering 18 metrics as features, the benchmark allowed the construction of ML models that can accurately classify modules as high-TD or not. The relatively high performance of the best classifiers enables practitioners to identify candidate TD items in their own systems with a high degree of certainty that these items are indeed problematic. The same models provide the opportunity to researchers for further experimentation and analysis of high-TD modules, without having to resort to a multitude of commercial and open-source tools for establishing the ground truth. A prototype tool that is able to classify software modules as high/not-high TD for any arbitrary Java project is available online[62].

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

[62] https://sites.google.com/view/ml-td-identification/home

6 Threats to Validity

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

Threats to *internal validity* concern unanticipated relationships or external factors that could affect the variables being investigated. Regarding the selection of the independent variables, it is reasonable to assume that numerous other software-related metrics that affect TD might have not been taken into account. However, the fact that we constructed our TD predictor set based on metrics that have been widely used in the literature, limits this threat. Regarding our decision not to perform feature selection and consider the entire set of metrics (presented in Table 46) as input to the classification models, we avoided selection bias by using hypothesis testing and univariate logistic regression analysis to investigate their discriminative and predictive power. Subsequently, all metrics were found to be significant towards predicting high-TD software modules. Finally, as explained in Section 2.3, we employed oversampling to increase classifiers' effectiveness in predicting the minority class. Nevertheless, we acknowledge that balancing a dataset may lead to distributions far different from those expected in real life and may therefore result in less accurate models in practice. Investigating whether it is feasible to create equally accurate models trained on data reflecting real distributions will be the subject of future work.

Threats to *construct validity* concern the relation between the theory and the observation. In this work, construct validity threats mainly stem from possible inaccuracies in the measurements performed for collecting the software-related metrics (i.e., the independent variables), but also from inaccuracies in the measurements performed for the construction of the TD Benchmarker [7] (i.e., the dependent variable). To mitigate the risks related to the data collection process, we decided to use five well-known tools, which have been widely used in similar software engineering studies to extract software-related metrics [103], [104], [113], [177], [197]. As regards the dependent variable, i.e., the TD Benchmarker created within the context of Amanatidis et al. [7], construct validity threats are mainly due to possible inaccuracies in the static analysis measurements performed by the three employed tools, namely SonarQube, CAST and Squore. However, all three platforms are major TD tools, widely adopted by software industries and researchers [32].

Regarding our decision to consider as high-TD only the modules that have been identified as such by all three TD tools (i.e., belonging to the Max-Ruler profile), we believe that while modules recognized as high-TD by multiple tools warrant more attention by the development team, ML models can be trained on any provided dataset. If a development team wishes to tag as high-TD modules even the ones identified as such by only a fraction of the three tools (e.g., two out of three), a different archetype can be selected (e.g., the Partner profile [7]) for extracting the training dataset. In an industrial context, the training dataset could also result from actual labeling of problematic classes so as to reflect the developers' perception of high TD. As for the experimented classification models, we exploited the ML algorithms implementation provided by the scikit-learn library, which is widely considered as a reliable tool.

Reliability validity threats concern the possibility of replicating this study. To facilitate replication, we provide an experimental package containing the dataset that was constructed, as well as the scripts used for data collection, data preparation and classification model construction. This material can be found online[63]. Moreover, the source code of the 25 examined projects is publicly available on GitHub to obtain the same data. Finally, as a means of practically validating our approach, we provide in the supporting material a prototype tool able to classify software modules as high-TD/not-high-TD for any arbitrary Java project. We believe that this tool will enable further feature experimentation through its use in academic or industrial setting and will pave the way for more data-driven TD management tools.

7 Conclusion and Future Work

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

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

[63] https://sites.google.com/view/ml-td-identification/home

Chapter IX Conclusions and Future Work

"*In literature and in life we ultimately pursue, not conclusions, but beginnings.*"

Sam Tanenhaus - American historian and journalist

The purpose of the present dissertation is to advance the state of the art in the fields of TD estimation and TD forecasting, by investigating novel approaches based mainly on ML techniques that facilitate TD management activities with respect to decision-making under uncertainty. More specifically, considering the lack of concrete approaches regarding the TD Forecasting concept, the present dissertation examined a multitude of different forecasting approaches, ranging from simple time series to more sophisticated ML models, for their ability to accurately predict the future TD evolution of long-lived, open-source software systems. It also investigated the ability of data clustering techniques to improve the accuracy of cross-project TD forecasting, allowing in that way the provision of reliable forecasts for projects with insufficient historical data, such as new projects or projects whose past evolution cannot be retrieved. On the other hand, by acknowledging the lack of a commonly accepted basis regarding the detection and estimation of TD, this dissertation investigated the ability of ML to exploit the collective knowledge extracted by combining the results of multiple leading TD tools. The results showed that various classification models are able to accurately identify candidate high-TD software classes in software systems, eliminating in that way the need to resort to a multitude of tools for establishing the ground truth. Section 1 provides a summary of the present dissertation and its main contributions through the lens of the six goals described in the introductory Chapter I, while Section 2 presents some interesting directions for future work.

1 Summary and Achievements

In Chapter III, we investigated the state-of-the-art and examined the major contributions that have been made until today in the field of TD estimation and forecasting. Through our study, we identified some interesting open issues that should be addressed through further research. In particular, already existing methods and tools for TD estimation have not reached a satisfactory level of maturity yet, while there is still a large volume of potential metrics and techniques that have not been used and that could potentially increase the completeness of the TD estimation concept. In addition, although there has been extensive research with respect to predicting the evolution of individual software features, quality attributes, and quality properties that are directly or indirectly related to the TD of a software project, no concrete contributions exist in the related literature regarding TD forecasting.

Considering the lack of concrete approaches regarding the TD Forecasting concept, in Chapter IV, we examined the usage of time series models as a valid and accurate approach to forecasting TD in long-lived, open-source, software projects. To the best of our knowledge, this is the first time that time series forecasting, and more specifically ARIMA methodology has been employed for this purpose. For the purpose of our study we obtained 3 years of TD evolution history, computed from the static code analysis of five independently developed, maintained, and managed open-source software systems. Based on this data, we showed that a single time series model, ARIMA(0,1,1), can accurately predict the short-term pattern of TD evolution for each of five projects. The results indicated that ARIMA time series models outperform purely random processes and random walks up to 8 weeks into the future. However, we observed that predictive power decreases considerably for longer forecasting horizons. This led us to the conclusion that we should also consider other forecasting techniques, such as more advanced ML algorithms, able to handle irrelevant features, as well as to support complex relationships and tolerance to noise between variables.

Trying to overcome the challenges of ARIMA models, in Chapter V, we examined whether and to what extent the usage of ML models constitutes a meaningful and accurate approach to forecasting TD in long-lived, open-source software applications. Across the 15 independently developed, maintained, and managed open source projects, we showed that TD Principal patterns can be modeled adequately by ML techniques. More specifically, for forecasting horizons between 1 and 20 weeks ahead, Regularization models (i.e., Lasso and Ridge regression) were able to fit and provide meaningful forecasts of TD Principal evolution. Trying to forecast longer into the future however, we noticed that their predictive power dropped significantly. On the contrary, the non-linear Random Forest regression demonstrated an almost stable performance over the holdout sample for all examined (up to 40) steps ahead. In fact, for forecasting horizons longer than 20 weeks ahead, Random Forest regression was able to capture the trend of the future evolution of the TD Principal with higher predictive power compared to the linear models. This indicates that a more complex model performs better when the forecast horizon is longer. From the above analysis, we concluded that ML models constitute a suitable and effective approach for TD Principal forecasting, as they were able to fit and provide meaningful estimates of TD evolution over a relatively long period, while in most of the cases, the future TD value was captured with a sufficient level of accuracy. To the best of our knowledge, this is the first study in the field of TD that examines the applicability of ML models for TD forecasting and therefore, it constitutes a good basis for future research and experimentation.

Moving on to Chapter VI, we leveraged the models developed in our previous study (presented in Chapter V) to introduce a practical approach for a more granular prioritization of TD liabilities that incorporates information retrieved from static analysis, change-proneness analysis and TD forecasting, emphasizing on the class-level of granularity to provide highly actionable results. Through our case study across the 10 most change-prone investigated classes of the Apache Kafka application, we have shown that taking into account the future evolution of TD and combining it with proper change-proneness analysis and visualization techniques can enable the early identification of classes that are more likely to become unmaintainable in the future, and therefore allow project managers and developers plan effective TD repayment strategies.

Chapter IV through Chapter VI have focused on statistical and ML techniques, able to accurately predict the future TD evolution of long-lived, open-source software systems. However, these approaches presuppose that reliable and sufficient past data (in the form of past commits) are available for a particular software project, in order to initiate a proper tailored model-training process. This requirement affects the practicality of the produced models.

To complement our previous efforts towards the TD Forecasting concept, in Chapter VII, we examined whether the adoption of clustering is a promising solution for enhancing the accuracy of cross-project TD forecasting. In other words, we investigated whether the consideration of TD-related similarities between software projects could be the key for building a forecasting model based on data retrieved from one project and using it to get reliable forecasts for a new, previously unknown software project. For this purpose, a large dataset was utilized, comprising 27 real-world open-source Java projects. These projects were then fed into a clustering algorithm and divided into clusters of similar projects with respect to their code metrics. Subsequently, cluster-representative TD forecasting models were constructed through several experiments, using five regression algorithms for forecasting horizons between 1 and 10 steps ahead. Finally, by assigning various previously unknown projects to the defined clusters, we observed that the cross-project accuracy of the cluster-representative models was higher when applied to projects assigned to the same cluster (i.e., within-cluster forecasting), compared to their accuracy when applied to projects assigned to a different cluster (i.e., cross-cluster forecasting). Furthermore, in most of the cases the future TD value was captured with a sufficient level of accuracy. This suggests that the similarity of the software projects may be a key factor for enhancing cross-project TD forecasting, and, in turn, that clustering may be a viable solution for achieving better results in cross-project forecasting.

Finally, by acknowledging the lack of a commonly accepted basis regarding the detection and estimation of TD among the existing TD tools, in Chapter VIII, we investigated the ability of well-known ML algorithms to leverage the collective knowledge extracted by different leading TD tools with the purpose to create a classification framework able to identify high-TD software classes. As ground truth, we considered a benchmark of 18,857 high-TD classes extracted from the application of three leading TD tools on 25 Java open-source projects. As model features, we considered 18 metrics covering a wide spectrum of software characteristics, spanning from code metrics to refactorings and repository activity. The findings revealed that a subset of superior classifiers can effectively identify high-TD software classes, achieving an *F_2-measure* score of approximately 0.79 on the test set. The associated *Module Inspection ratio* was found to be approximately 0.10 while the recall is close to 0.86, implying that one tenth of the system classes would have to be inspected by the development team for identifying 86% of all true high-TD classes. Through the application of the Scott-Knott algorithm it was found that Random Forest, Logistic Regression, Support Vector Machines and XGBoost presented similar prediction performance. Such models encompass the aggregate knowledge of multiple TD identification tools thereby increasing the certainty that the identified classes suffer indeed from high-TD.

2 Future Work

Based on the outcomes of the present thesis, several directions for future work can be identified. These directions are described below:

- The empirical studies presented in Chapter IV through Chapter VIII were based on open source software applications written in Java programming language. Hence, an interesting direction for future work would be to replicate these studies using commercial software, as well as software applications written in other programming languages, in order to investigate the generalizability of the obtained results.

- The large-scale empirical study presented in Chapter V has focused on examining the ability of various ML forecasting algorithms to predict the future TD evolution of a software application at the system level, thus allowing project managers to efficiently prioritize TD activities when dealing with different software applications. When it comes to a specific application however, the developers are often overwhelmed with a large volume of TD liabilities (e.g., code smells, bugs, vulnerabilities, etc.) that they need to fix. This renders the TD repayment procedure tedious, time consuming and effort demanding. While Chapter VI introduced an initial attempt to leverage class-level TD forecasting for more effective TD prioritization, an interesting direction for future work would be to investigate the possibility of extending TD forecasting techniques to all levels of granularity of a software project (i.e., package, class and function level). This would provide a highly granular prioritization of TD liabilities, allowing for a ranking of a software project's artifacts based on predictions of their long-term accumulated TD values.

- The TD indicators (i.e., predictors) that were used as input to the ML models built in the empirical studies presented in Chapter V through Chapter VIII were mainly based on software-related metrics extracted by popular static analysis tools, such as SonarQube, CKJM, CK, and PMD. To this end, it would be useful to investigate different efficient ways to produce models for accurate TD identification and forecasting, by bringing into the equation other types of software repositories that could be a potential source of TD related data, such as project management and issue-tracking systems, as well as archived communication between project personnel. More specifically, mining TD related data from project-and issue-tracking systems, as well as analyzing the communication between project personnel could reveal indications of high-TD artifacts that concentrate a large part of maintenance effort. These indications could then be factored in TD forecasting and identification techniques to target these critical - from a maintenance point of view – artifacts and pave the way for the advance in the state of the art in this domain.

- The empirical studies presented in Chapter IV through Chapter VII have contributed to the TD forecasting challenge by investigating the performance of statistical time series and ML models. The outcome of this work indicated that these methods are indeed capable of providing accurate and useful results. However, despite the fact that these approaches were proven able to produce reliable TD predictions for short- and long-term forecasting horizons, their predictive performance dropped noticeably as forecasting horizons increased. More specifically, the performance of statistical time series models was found to be satisfactory for forecasts up to 8 commits ahead, while the ML models were able to provide reliable results up to 40 commits ahead. This partially limits the practicality of the produced models, as they cannot be applied to cases where a software company requires a long-term TD management plan. To this end, an interesting direction for future work would be to investigate whether the adoption of more sophisticated forecasting techniques, such as Deep Learning models, may lead to more accurate and practical TD forecasts, giving particular emphasis on the long-term predictions.

- As far as the concept of TD forecasting is concerned, the work conducted within the context of this dissertation has focused on predicting the future evolution of TD Principal, i.e., the effort required to address the difference between the current and the optimal level of design-time quality. However, predicting the future value of TD Interest would be also critical for decision making, as it can be used to timely determine the point at which a software product would become unmaintainable, and therefore to respond promptly through appropriate refactoring activities in order to prevent this situation. An interesting way of approaching this challenge would be to examine whether forecasting techniques could contribute towards enhancing the process of identifying the "breaking point" of an application, a term introduced by Chatzigeorgiou et al. [116] that refers to the point in time where the accumulated interest is equal to the TD principal and, thus, the cost becomes higher than the benefit.
- Finally, in some of our previous studies (not included in the present dissertation) we highlighted that TD may be closely related to software security [131], [198]. Hence, another interesting direction would be to examine whether TD forecasting can be used as a means to project the future evolution of important security-related aspects of a software under development, such as the manifestation of vulnerabilities.

Publications

Parts of this dissertation have been published or submitted for publication in a number of peer-reviewed conferences and journals. A list of these articles can be found below:

Journals

- D. Tsoukalas, N. Mittas, A. Chatzigeorgiou, D. Kehagias, A. Ampatzoglou, T. Amanatidis, and L. Angelis, “Machine Learning for Technical Debt Identification,” in *IEEE Transaction on Software Engineering*, 2021. doi: 10.1109/TSE.2021.3129355.
- D. Tsoukalas, M. Mathioudaki, M. Siavvas, D. Kehagias, and A. Chatzigeorgiou, “A Clustering Approach Towards Cross-Project Technical Debt Forecasting,” in *SN Computer Science*, 2021, vol. 2, pp. 1-30. doi: 10.1007/s42979-020-00408-4.
- D. Tsoukalas, D. Kehagias, M. Siavvas, and A. Chatzigeorgiou, “Technical Debt Forecasting: An empirical study on open-source repositories,” in *Journal of Systems and Software*, 2020, vol. 170, p. 110777. doi: 10.1016/j.jss.2020.110777.

Conferences

- D. Tsoukalas, A. Chatzigeorgiou, A. Ampatzoglou, N. Mittas, and D. Kehagias, “TD Classifier: Automatic Identification of Java Classes with High Technical Debt” in *International Conference on Technical Debt 2022 (TechDebt22)* (submitted, under review)
- D. Tsoukalas, M. Jankovic, M. Siavvas, D. Kehagias, A. Chatzigeorgiou, and D. Tzovaras, “On the Applicability of Time Series Models for Technical Debt Forecasting,” in *15th China-Europe International Symposium on Software Engineering Education (CEISEE 2019)*, IEEE TEMS, 2019. doi: 10.13140/RG.2.2.33152.79367.
- D. Tsoukalas, M. Siavvas, M. Jankovic, D. Kehagias, A. Chatzigeorgiou, and D. Tzovaras, “Methods and Tools for TD Estimation and Forecasting: A State-of-the-art Survey,” in *IEEE International Conference on Intelligent Systems (IS 2018)*, 2018, pp. 698-705. doi: 10.1109/IS.2018.8710521.

Other publications

Other publications that have been produced and are directly or indirectly related to the research work conducted within the course of this PhD are listed below. It should be noted that the contents of these publications are not included in the present dissertation for reasons of brevity and cohesion.

- Siavvas, M., Tsoukalas, D., Jankovic, M., Kehagias, D., Tzovaras, D., “Technical debt as an indicator of software security risk: a machine learning approach for software development enterprises”, in *Enterprise Information Systems (TEIS)*, 2020, pp. 1-43. doi: 10.1080/17517575.2020.1824017

- M. Mathioudaki, D. Tsoukalas, M. Siavvas, and D. Kehagias, “Technical Debt Forecasting Based on Deep Learning Techniques”, in *21st International Conference on Computational Science and Its Applications (ICCSA 2021)*, 2021, pp. 306-322. doi: 10.1007/978-3-030-87007-2_22.

References

[1] W. Cunningham, "The WyCash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993, doi: http://dx.doi.org/10.1145/157710.157715.

[2] N. Brown *et al.*, "Managing technical debt in software-reliant systems," in *Proceedings of the workshop on Future of software engineering research (FSE/SDP)*, 2010, pp. 47–52. doi: http://dx.doi.org/10.1145/1882362.1882373.

[3] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Information and Software Technology*, vol. 64, pp. 52–73, 2015, doi: http://dx.doi.org/10.1016/j.infsof.2015.04.001.

[4] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," in *Advances in Computers*, vol. 82, Elsevier, 2011, pp. 25–46. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780123855121000025

[5] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, pp. 193–220, 2015, doi: http://dx.doi.org/10.1016/j.jss.2014.12.027.

[6] F. A. Fontana, R. Roveda, and M. Zanoni, "Technical debt indexes provided by tools: a preliminary discussion," in *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, 2016, pp. 28–31. doi: 10.1109/MTD.2016.11.

[7] T. Amanatidis, N. Mittas, A. Moschou, A. Chatzigeorgiou, A. Ampatzoglou, and L. Angelis, "Evaluating the agreement among technical debt measurement tools: building an empirical benchmark of technical debt liabilities," *Empirical Software Engineering*, vol. 25, no. 5, pp. 4161–4204, 2020, doi: https://doi.org/10.1007/s10664-020-09869-w.

[8] G. Digkas, M. Lungu, A. Chatzigeorgiou, and P. Avgeriou, "The evolution of technical debt in the apache ecosystem," in *European Conference on Software Architecture (ECSA)*, 2017, pp. 51–66. doi: http://dx.doi.org/10.1007/978-3-319-65831-5_4.

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

[10] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016, doi: https://doi.org/10.1007/s10664-015-9378-4.

[11] E. Arisholm and L. C. Briand, "Predicting fault-prone components in a java legacy system," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering and measurement (ESEM)*, 2006, pp. 8–17. doi: http://dx.doi.org/10.1145/1159733.1159738.

[12] T. Chaikalis and A. Chatzigeorgiou, "Forecasting java software evolution trends employing network models," *IEEE Transactions on Software Engineering*, vol. 41, no. 6, pp. 582–602, 2015, doi: http://dx.doi.org/10.1109/TSE.2014.2381249.

[13] D. Tsoukalas, M. Siavvas, M. Jankovic, D. Kehagias, A. Chatzigeorgiou, and D. Tzovaras, "Methods and Tools for TD Estimation and Forecasting: A State-of-the-art Survey," in *International Conference on Intelligent Systems (IS 2018)*, 2018, pp. 698–705. doi: http://dx.doi.org/10.1109/IS.2018.8710521.

[14] D. Tsoukalas, M. Jankovic, M. Siavvas, D. Kehagias, A. Chatzigeorgiou, and D. Tzovaras, "On the Applicability of Time Series Models for Technical Debt

Forecasting," in *15th China-Europe International Symposium on Software Engineering Education (CEISEE 2019),* 2019, pp. 1–10. doi: 10.13140/RG.2.2.33152.79367.

[15] D. Tsoukalas, D. Kehagias, M. Siavvas, and A. Chatzigeorgiou, "Technical Debt Forecasting: An empirical study on open-source repositories," in *Journal of Systems and Software,* 2020, vol. 170, p. 110777. doi: https://doi.org/10.1016/j.jss.2020.110777.

[16] D. Tsoukalas, M. Mathioudaki, M. Siavvas, D. Kehagias, and A. Chatzigeorgiou, "A Clustering Approach Towards Cross-Project Technical Debt Forecasting," *SN Computer Science*, vol. 2, no. 1, pp. 1–30, 2021, doi: 10.1007/s42979-020-00408-4.

[17] D. Tsoukalas *et al.*, "Machine Learning for Technical Debt Identification," *IEEE Transactions on Software Engineering*, no. 01, pp. 1–1, 2021, doi: 10.1109/TSE.2021.3129355.

[18] A. Ampatzoglou, A. Ampatzoglou, P. Avgeriou, and A. Chatzigeorgiou, "Establishing a framework for managing interest in technical debt," 2015. doi: 10.5220/0005885700750085.

[19] H. S. Yazdi, M. Mirbolouki, P. Pietsch, T. Kehrer, and U. Kelter, "Analysis and prediction of design model evolution using time series," in *International Conference on Advanced Information Systems Engineering (CAiSE),* 2014, pp. 1–15. doi: 10.1007/978-3-319-07869-4_1.

[20] U. Raja, D. P. Hale, and J. E. Hale, "Modeling software evolution defects: a time series approach," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 1, pp. 49–71, 2009, doi: http://dx.doi.org/10.1002/smr.398.

[21] M. Goulão, N. Fonte, M. Wermelinger, and F. B. e Abreu, "Software evolution prediction using seasonal time analysis: a comparative study," in *16th European Conference on Software Maintenance and Reengineering (CSMR),* 2012, pp. 213–222. doi: http://dx.doi.org/10.1109/CSMR.2012.30.

[22] B. Kenmei, G. Antoniol, and M. Di Penta, "Trend analysis and issue prediction in large-scale open source systems," in *12th European Conference on Software Maintenance and Reengineering (CSMR),* 2008, pp. 73–82. doi: http://dx.doi.org/10.1109/CSMR.2008.4493302.

[23] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time series analysis: forecasting and control,* 5th ed. John Wiley & Sons, 2015.

[24] A. Martini, J. Bosch, and M. Chaudron, "Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study," *Information and Software Technology,* vol. 67, pp. 237–253, 2015.

[25] Z. Li, P. Liang, and P. Avgeriou, "Architectural debt management in value-oriented architecting," in *Economics-Driven Software Architecture,* Elsevier, 2014, pp. 183–204.

[26] F. A. Fontana, V. Ferme, and S. Spinelli, "Investigating the impact of code smells debt on quality code evaluation," in *Proceedings of the Third International Workshop on Managing Technical Debt (MTD),* 2012, pp. 15–22. doi: http://dx.doi.org/10.1109/MTD.2012.6225993.

[27] A. Choudhary and P. Singh, "Minimizing Refactoring Effort through Prioritization of Classes based on Historical, Architectural and Code Smell Information.," in *QuASoQ/TDA@ APSEC*, 2016, pp. 76–79.

[28] S. Akbarinasaji, A. Bener, and A. Neal, "A heuristic for estimating the impact of lingering defects: can debt analogy be used as a metric?," in *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics*, 2017, pp. 36–42.

[29] A. Nugroho, J. Visser, and T. Kuipers, “An empirical model of technical debt and interest,” in *Proceedings of the 2nd Workshop on Managing Technical Debt*, 2011, pp. 1–8. doi: 10.1145/1985362.1985364.

[30] D. Falessi and A. Voegele, “Validating and prioritizing quality rules for managing technical debt: An industrial case study,” in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 2015, pp. 41–48.

[31] T. Besker, A. Martini, and J. Bosch, “Software developer productivity loss due to technical debt—A replication and extension study examining developers' development work,” *Journal of Systems and Software*, vol. 156, pp. 41–61, 2019, doi: https://doi.org/10.1016/j.jss.2019.06.004.

[32] P. Avgeriou *et al.*, “An Overview and Comparison of Technical Debt Measurement Tools,” *IEEE Software, accepted for publication*, 2021.

[33] M. Fowler, “Technical Debt,” 2003. http://www.martinfowler.com/bliki/TechnicalDebt.html (accessed Jul. 30, 2018).

[34] M. Fowler, “Technical Debt Quadrant,” 2009. http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html (accessed Jul. 30, 2018).

[35] P. Kruchten, R. L. Nord, and I. Ozkaya, “Technical debt: From metaphor to theory and practice,” *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012, doi: 10.1109/MS.2012.167.

[36] S. McConnell, “How to Categorize and Communicate Technical Debt,” 2012. https://www.castsoftware.com/blog/steve-mcconnell-on-categorizing-managing-technical-debt (accessed Jul. 30, 2018).

[37] A. Ampatzoglou, A. Michailidis, C. Sarikyriakidis, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, “A Framework for Managing Interest in Technical Debt: An Industrial Validation,” 2018. doi: http://dx.doi.org/10.1145/3194164.3194175.

[38] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, “Managing technical debt in software engineering (dagstuhl seminar 16162),” in *Dagstuhl Reports*, 2016, vol. 6.

[39] C. Sterling, *Managing Software Debt: Building for Inevitable Change (Adobe Reader)*. Addison-Wesley Professional, 2010.

[40] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, “Identification and management of technical debt: A systematic mapping study,” *Information and Software Technology*, vol. 70, pp. 100–121, 2016, doi: http://dx.doi.org/10.1016/j.infsof.2015.10.008.

[41] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994, doi: http://dx.doi.org/10.1109/32.295895.

[42] J. Bansiya and 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, doi: 10.1109/32.979986.

[43] M. Riaz, E. Mendes, and E. Tempero, “A systematic review of software maintainability prediction and metrics,” in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2009, pp. 367–377. doi: http://dx.doi.org/10.1109/ESEM.2009.5314233.

[44] F. Fioravanti and P. Nesi, “Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems,” *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1062–1084, 2001, doi: http://dx.doi.org/10.1109/32.988708.

[45] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, “On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation,” *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018, doi: http://dx.doi.org/10.1007/s10664-017-9535-z.

[46] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[47] A. Vetro’, “Using Automatic Static Analysis to Identify Technical Debt,” in *Proceedings of the 34th International Conference on Software Engineering (ICSE),* Zurich, Switzerland, 2012, pp. 1613–1615. doi: 10.5555/2337223.2337499.

[48] N. Zazworka *et al.*, “Comparing four approaches for technical debt identification,” *Software Quality Journal*, vol. 22, no. 3, pp. 403–426, 2014, doi: https://doi.org/10.1007/s11219-013-9200-8.

[49] C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman, and F. Shull, “Organizing the Technical Debt Landscape,” in *Proceedings of the Third International Workshop on Managing Technical Debt (MTD),* Zurich, Switzerland, 2012, pp. 23–26. doi: 10.5555/2666036.2666040.

[50] N. Zazworka, R. O. Spínola, A. Vetro, F. Shull, and C. Seaman, “A case study on effectively identifying technical debt,” in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering (EASE),* 2013, pp. 42–47. doi: http://dx.doi.org/10.1145/2460999.2461005.

[51] ISO/IEC, “ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models,” ISO/IEC, 2011.

[52] J.-L. Letouzey and M. Ilkiewicz, “Managing technical debt with the sqale method,” *IEEE software,* vol. 29, no. 6, pp. 44–51, 2012, doi: http://dx.doi.org/10.1109/MS.2012.129.

[53] B. Curtis, J. Sappidi, and A. Szynkarski, “Estimating the size, cost, and types of technical debt,” in *Proceedings of the Third International Workshop on Managing Technical Debt (MTD),* 2012, pp. 49–53.

[54] R. Marinescu, “Assessing technical debt by identifying design flaws in software systems,” *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9–1, 2012, doi: 10.1147/JRD.2012.2204512.

[55] J.-L. Letouzey, “The SQALE method for evaluating technical debt,” in *Third International Workshop on Managing Technical Debt (MTD),* 2012, pp. 31–36. doi: 10.1109/MTD.2012.6225997.

[56] C. Fernández-Sánchez, J. Garbajosa, C. Vidal, and A. Yagüe, “An analysis of techniques and methods for technical debt management: a reflection from the architecture perspective,” in *Software Architecture and Metrics (SAM), 2015 IEEE/ACM 2nd International Workshop on*, 2015, pp. 22–28.

[57] C. Seaman *et al.*, “Using technical debt data in decision making: Potential decision approaches,” in *Proceedings of the Third International Workshop on Managing Technical Debt*, 2012, pp. 45–48.

[58] A. K. Palit and D. Popovic, *Computational intelligence in time series forecasting: theory and engineering applications*. Springer Science & Business Media, 2006. [Online]. Available: https://www.springer.com/gp/book/9781852339487

[59] J. Das, *Statistics for Business Decisions*. Academic Publishers, 2012.

[60] G. Bontempi, S. B. Taieb, and Y.-A. Le Borgne, “Machine Learning Strategies for Time Series Forecasting,” in *Machine learning strategies for time series*

forecasting, Springer Berlin Heidelberg, 2013. [Online]. Available: https://doi.org/10.1007/978-3-642-36318-4_3

[61] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, "Statistical and Machine Learning forecasting methods: Concerns and ways forward," *PloS one*, vol. 13, no. 3, p. e0194889, 2018, doi: http://dx.doi.org/10.1371/journal.pone.0194889.

[62] P. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," Harvard University, Cambridge, 1974. [Online]. Available: https://books.google.gr/books?id=z81XmgEACAAJ

[63] P. J. Werbos, "Generalization of backpropagation with application to a recurrent gas market model," *Neural networks*, vol. 1, no. 4, pp. 339–356, 1988, doi: http://dx.doi.org/10.1016/0893-6080(88)90007-X.

[64] A. Lapedes and R. Farber, "Nonlinear signal processing using neural networks: Prediction and system modelling," United States, 1987.

[65] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*, vol. 1. Springer series in statistics New York, 2001.

[66] E. Alpaydin, *Introduction to Machine Learning, 2nd edn.*, 2nd ed. The MIT Press (February 2010), 2010.

[67] A. E. Hoerl and R. W. Kennard, "Ridge regression: Biased estimation for nonorthogonal problems," *Technometrics*, vol. 12, no. 1, pp. 55–67, 2000, doi: http://dx.doi.org/10.1080/00401706.2000.10485983.

[68] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996, doi: http://dx.doi.org/10.1111/j.2517-6161.1996.tb02080.x.

[69] H. Drucker, C. J. Burges, L. Kaufman, A. J. Smola, and V. Vapnik, "Support vector regression machines," in *Proceedings of the 9th International Conference on Neural Information Processing Systems (NIPS)*, 1997, pp. 155–161.

[70] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992, doi: http://dx.doi.org/10.2307/2685209.

[71] L. Breiman, *Classification and Regression Trees*. Routledge, 2017. [Online]. Available: https://books.google.gr/books?id=JwQx-WOmSyQC

[72] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, pp. 832–844, 1998, doi: 10.1109/34.709601.

[73] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996, doi: http://dx.doi.org/10.1007/BF00058655.

[74] J. Highsmith, "The Financial Implications of Technical Debt," 2010. http://jimhighsmith.com/the-financial-implications-of-technical-debt/ (accessed Jul. 30, 2018).

[75] Y. Guo and C. Seaman, "A portfolio approach to technical debt management," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, 2011, pp. 31–34.

[76] J. Holvitie and V. Leppänen, "DebtFlag: Technical debt management with a development environment integrated tool," in *Proceedings of the 4th International Workshop on Managing Technical Debt*, 2013, pp. 20–27.

[77] E. Alzaghoul and R. Bahsoon, "CloudMTD: Using real options to manage technical debt in cloud-based service selection," in *Managing Technical Debt (MTD), 2013 4th International Workshop on*, 2013, pp. 55–62.

[78] "Automated Technical Debt Measure V1.0," Object Management Group (OMG), 2018. [Online]. Available: https://www.omg.org/spec/ATDM/1.0/PDF

[79] J. Shore, “Quality with a Name,” 2006. http://jamesshore.com/Articles/Quality-With-a-Name.html (accessed Jul. 30, 2018).

[80] O. Gaudin, “Evaluate your technical debt with Sonar,” *Sonar, Jun,* 2009.

[81] J. Bohnet and J. Döllner, “Monitoring code quality and development activity by software maps,” in *Proceedings of the 2nd Workshop on Managing Technical Debt,* 2011, pp. 9–16.

[82] J. de Groot, A. Nugroho, T. Bäck, and J. Visser, “What is the value of your software?,” in *Proceedings of the Third International Workshop on Managing Technical Debt,* 2012, pp. 37–44.

[83] N. A. Ernst, “On the role of requirements in understanding and managing technical debt,” in *Proceedings of the Third International Workshop on Managing Technical Debt,* 2012, pp. 61–64.

[84] S. Strasser, C. Frederickson, K. Fenger, and C. Izurieta, “An automated software tool for validating design patterns,” in *ISCA 24th International Conference on Computer Applications in Industry and Engineering. CAINE,* 2011, vol. 11.

[85] B. Curtis, J. Sappidi, and A. Szynkarski, “Estimating the principal of an application’s technical debt,” *IEEE software,* vol. 29, no. 6, pp. 34–42, 2012, doi: 10.1109/MS.2012.156.

[86] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, “In search of a metric for managing architectural technical debt,” in *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on,* 2012, pp. 91–100.

[87] C. Fernández-Sánchez, H. Humanes, J. Garbajosa, and J. Díaz, “An Open Tool for Assisting in Technical Debt Management,” in *Software Engineering and Advanced Applications (SEAA), 2017 43rd Euromicro Conference on,* 2017, pp. 400–403.

[88] M. M. Lehman, “Programs, life cycles, and laws of software evolution,” *Proceedings of the IEEE,* vol. 68, no. 9, pp. 1060–1076, 1980, doi: http://dx.doi.org/10.1109/PROC.1980.11805.

[89] H. C. Gall and M. Lanza, “Software evolution: analysis and visualization,” in *Proceedings of the 28th international conference on Software engineering (ICSE),* 2006, pp. 1055–1056. doi: http://dx.doi.org/10.1145/1134285.1134502.

[90] M. W. Godfrey and D. M. German, “The past, present, and future of software evolution,” in *Frontiers of Software Maintenance (FoSM),* 2008, pp. 129–138. doi: http://dx.doi.org/10.1109/FOSM.2008.4659256.

[91] T. Mens, “Introduction and roadmap: History and challenges of software evolution,” in *Introduction and Roadmap: History and Challenges of Software Evolution, chapter 1. Software Evolution,* Springer, 2008. [Online]. Available: https://doi.org/10.1007/978-3-540-76440-3_1

[92] S. Wagner, “A Bayesian network approach to assess and predict software quality using activity-based quality models,” *Proceedings of the 5th International Conference on Predictor Models in Software Engineering (PROMISE),* p. 1, 2009, doi: 10.1145/1540438.1540447.

[93] C. Van Koten and A. Gray, “An application of Bayesian network for predicting object-oriented software maintainability,” *Information and Software Technology,* vol. 48, no. 1, pp. 59–67, 2006, doi: http://dx.doi.org/10.1016/j.infsof.2005.03.002.

[94] Y. Zhou and H. Leung, “Predicting object-oriented software maintainability using multivariate adaptive regression splines,” *Journal of Systems and Software,* vol. 80, no. 8, pp. 1349–1361, 2007, doi: http://dx.doi.org/10.1016/j.jss.2006.10.049.

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

[96] M. O. Elish and K. O. Elish, “Application of TreeNet in Predicting Object-Oriented Software Maintainability: A Comparative Study,” in *2009 13th European Conference on Software Maintenance and Reengineering (CSMR),* Mar. 2009, pp. 69–78. doi: 10.1109/CSMR.2009.57.

[97] N. Nagappan, T. Ball, and A. Zeller, “Mining metrics to predict component failures,” in *Proceedings of the 28th international conference on Software engineering (ICSE),* 2006, pp. 452–461. doi: http://dx.doi.org/10.1145/1134285.1134349.

[98] I. Gondra, “Applying machine learning to software fault-proneness prediction,” *Journal of Systems and Software*, vol. 81, no. 2, pp. 186–195, 2008, doi: http://dx.doi.org/10.1016/j.jss.2007.05.035.

[99] T. M. Khoshgoftaar, E. B. Allen, and J. Deng, “Using regression trees to classify fault-prone software modules,” *IEEE Transactions on Reliability,* vol. 51, no. 4, pp. 455–462, 2002, doi: http://dx.doi.org/10.1109/TR.2002.804488.

[100] Y. Roumani, J. K. Nwankpa, and Y. F. Roumani, “Time series modeling of vulnerabilities,” *Computers & Security,* vol. 51, pp. 32–40, 2015, doi: http://dx.doi.org/10.1016/j.cose.2015.03.003.

[101] G. Antoniol, M. Di Penta, and S. Gradara, “Predicting Software Evolution: An Approach and a Case Study.”

[102] R. Malhotra and K. Lata, “On the Application of Cross-Project Validation for Predicting Maintainability of Open Source Software using Machine Learning Techniques,” in *2018 7th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO),* 2018, pp. 175–181. doi: 10.1109/ICRITO.2018.8748749.

[103] D. Cruz, A. Santana, and E. Figueiredo, “Detecting Bad Smells with Machine Learning Algorithms: An Empirical Study,” in *Proceedings of the 3rd International Conference on Technical Debt,* New York, NY, USA, 2020, pp. 31–40. doi: 10.1145/3387906.3388618.

[104] F. Pecorelli, F. Palomba, F. Khomh, and A. De Lucia, “Developer-Driven Code Smell Prioritization,” in *Proceedings of the 17th International Conference on Mining Software Repositories,* New York, NY, USA, 2020, pp. 220–231. doi: 10.1145/3379597.3387457.

[105] S. Lujan, F. Pecorelli, F. Palomba, A. De Lucia, and V. Lenarduzzi, “A Preliminary Study on the Adequacy of Static Analysis Warnings with Respect to Code Smell Prediction,” in *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, New York, NY, USA, 2020, pp. 1–6. doi: 10.1145/3416505.3423559.

[106] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, “Machine learning techniques for code smell detection: A systematic literature review and meta-analysis,” *Information and Software Technology,* vol. 108, pp. 115–138, 2019, doi: https://doi.org/10.1016/j.infsof.2018.12.009.

[107] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, “Code smell detection by deep direct-learning and transfer-learning,” *Journal of Systems and Software,* vol. 176, p. 110936, 2021, doi: https://doi.org/10.1016/j.jss.2021.110936.

[108] R. Abbas, F. A. Albalooshi, and M. Hammad, “Software Change Proneness Prediction Using Machine Learning,” in *2020 International Conference on*

Innovation and Intelligence for Informatics, Computing and Technologies (3ICT), 2020, pp. 1–7. doi: 10.1109/3ICT51146.2020.9311978.

[109] N. Pritam *et al.*, "Assessment of Code Smell for Predicting Class Change Proneness Using Machine Learning," *IEEE Access*, vol. 7, pp. 37414–37425, 2019.

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

[111] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, 2007, doi: 10.1109/TSE.2007.256941.

[112] V. U. B. Challagulla, F. B. Bastani, and and R. A. Paul, "Empirical assessment of machine learning based software defect prediction techniques," in *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, 2005, pp. 263–270. doi: http://dx.doi.org/10.1109/WORDS.2005.32.

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

[114] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou, "How do developers fix issues and pay back technical debt in the Apache ecosystem?," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 153–163. doi: 10.1109/SANER.2018.8330205.

[115] J. Tan, M. Lungu, and P. Avgeriou, "Towards Studying the Evolution of Technical Debt in the Python Projects from the Apache Software Ecosystem.," in *17th Belgium-Netherlands Software Evolution Workshop (BENEVOL)*, 2018, pp. 43–45.

[116] A. Chatzigeorgiou, A. Ampatzoglou, A. Ampatzoglou, and T. Amanatidis, "Estimating the breaking point for technical debt," in *IEEE 7th international workshop on Managing technical debt (MTD)*, 2015, pp. 53–56. doi: http://dx.doi.org/10.1109/MTD.2015.7332625.

[117] G. Skourletopoulos, C. X. Mavromoustakis, R. Bahsoon, G. Mastorakis, and E. Pallis, "Predicting and quantifying the technical debt in cloud software engineering.," in *19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, 2014, pp. 36–40. doi: http://dx.doi.org/10.1109/CAMAD.2014.7033201.

[118] B. W. Boehm and others, "Software engineering economics," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 1, pp. 4–21, 1984, doi: 10.1109/TSE.1984.5010193.

[119] C. Fernández-Sánchez, J. Garbajosa, A. Yagüe, and J. Perez, "Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study," *Journal of Systems and Software*, vol. 124, pp. 22–38, 2017.

[120] D. A. Dickey and W. A. Fuller, "Distribution of the estimators for autoregressive time series with a unit root," *Journal of the American statistical association*, vol. 74, no. 366a, pp. 427–431, 1979.

[121] R. McCleary and R. Hay, *Applied time series analysis for the social sciences*. Sage Publications, 1980. [Online]. Available: https://books.google.gr/books?id=D6-CAAAAIAAJ

[122] G. M. Ljung and G. E. Box, "On a measure of lack of fit in time series models," *Biometrika*, vol. 65, no. 2, pp. 297–303, 1978.

[123] H. Akaike, "Information theory and an extension of the maximum likelihood principle," in *Selected papers of hirotugu akaike*, Springer, 1998, pp. 199–213.

[124] M. Stone, "Cross-validatory choice and assessment of statistical predictions," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 36, no. 2, pp. 111–133, 1974, doi: http://dx.doi.org/10.1111/j.2517-6161.1974.tb00994.x.

[125] P. V. Bidarkota, "The comparative forecast performance of univariate and multivariate models: an application to real interest rate forecasting," *International Journal of Forecasting*, vol. 14, no. 4, pp. 457–468, 1998, doi: https://doi.org/10.1016/S0169-2070(98)00036-3.

[126] J. [du Preez and S. F. Witt, "Univariate versus multivariate time series forecasting: an application to international tourism demand," *International Journal of Forecasting*, vol. 19, no. 3, pp. 435–451, 2003, doi: https://doi.org/10.1016/S0169-2070(02)00057-2.

[127] V. Lenarduzzi, F. Lomio, D. Taibi, and H. Huttunen, "On the Fault Proneness of SonarQube Technical Debt Violations: A comparison of eight Machine Learning Techniques," *Computing Research Repository (CoRR)*, vol. abs/1907.00376, 2019, [Online]. Available: http://arxiv.org/abs/1907.00376

[128] M. Jureczko and D. Spinellis, "Using Object-Oriented Design Metrics to Predict Software Defects," vol. Models and Methodology of System Dependability, Wroclaw, Poland: Oficyna Wydawnicza Politechniki Wroclawskiej, 2010, pp. 69–81.

[129] J. Xuan, Y. Hu, and H. Jiang, "Debt-Prone Bugs: Technical Debt in Software Maintenance," *Computing Research Repository (CoRR)*, vol. abs/1704.04766, 2017, [Online]. Available: http://arxiv.org/abs/1704.04766

[130] I. Griffith, D. Reimanis, C. Izurieta, Z. Codabux, A. Deo, and B. Williams, "The correspondence between software quality models and technical debt estimation approaches," in *Sixth International Workshop on Managing Technical Debt (MTD)*, 2014, pp. 19–26. doi: 10.1109/MTD.2014.13.

[131] M. Siavvas *et al.*, "An Empirical Evaluation of the Relationship between Technical Debt and Software Security," *9th International Conference on Information Society and Technology (ICIST) 2019*, 2019.

[132] N. Zazworka, C. Seaman, and F. Shull, "Prioritizing design debt investment opportunities," in *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD)*, 2011, pp. 39–42. doi: 10.1145/1985362.1985372.

[133] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems," in *2010 IEEE International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10. doi: 10.1109/ICSM.2010.5609564.

[134] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "Assessing code smell interest probability: a case study," in *Proceedings of the XP2017 Scientific Workshops*, 2017, p. 5. doi: 10.1145/3120459.3120465.

[135] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dyb\a a, "Quantifying the effect of code smells on maintenance effort," *IEEE transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2012, doi: 10.1109/TSE.2012.89.

[136] M. A. A. Mamun, A. Martini, M. Staron, C. Berger, and J. Hansson, "Evolution of technical debt: An exploratory study," in *2019 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM)*, 2019, pp. 87–102.

[137] S. Karus and M. Dumas, "Code churn estimation using organisational and code metrics: An experimental comparison," *Information and Software Technology*, vol. 54, no. 2, pp. 203–211, 2012, doi: https://doi.org/10.1016/j.infsof.2011.09.004.

[138] G. A. D. Lucca, A. R. Fasolino, P. Tramontana, and C. A. Visaggio, “Towards the Definition of a Maintainability Model for Web Applications,” in *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR’04)*, USA, 2004, p. 279. doi: 10.1109/CSMR.2004.1281430.
[139] S. Eski and F. Buzluca, “An Empirical Study on Object-Oriented Metrics and Software Evolution in Order to Reduce Testing Costs by Predicting Change-Prone Classes,” in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*, Mar. 2011, pp. 566–571. doi: 10.1109/ICSTW.2011.43.
[140] E. Giger, M. Pinzger, and H. C. Gall, “Can we predict types of code changes? An empirical analysis,” in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, Jun. 2012, pp. 217–226. doi: 10.1109/MSR.2012.6224284.
[141] M. Bruntink and A. van Deursen, “An empirical study into class testability,” *Journal of Systems and Software*, vol. 79, no. 9, pp. 1219–1232, 2006, doi: https://doi.org/10.1016/j.jss.2006.02.036.
[142] Y. Singh and A. Saha, “Prediction of Testability Using the Design Metrics for Object-Oriented Software,” *International Journal of Computer Applications in Technology*, vol. 44, no. 1, pp. 12–22, Jul. 2012, doi: 10.1504/IJCAT.2012.048204.
[143] Y. Zhou and B. Xu, “Predicting the maintainability of open source software using design metrics,” *Wuhan University Journal of Natural Sciences*, vol. 13, no. 1, pp. 14–20, 2008, doi: http://dx.doi.org/10.1007/s11859-008-0104-6.
[144] R. Shatnawi and W. Li, “The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process,” *Journal of Systems and Software*, vol. 81, no. 11, pp. 1868–1882, 2008, doi: https://doi.org/10.1016/j.jss.2007.12.794.
[145] Y. Zhou *et al.*, “An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems,” *Science China Information Sciences*, vol. 55, pp. 2800–2815, 2012, doi: https://doi.org/10.1007/s11432-012-4745-x.
[146] M. O. Elish, “Exploring the Relationships between Design Metrics and Package Understandability: A Case Study,” in *2010 IEEE 18th International Conference on Program Comprehension (ICPC)*, Jun. 2010, pp. 144–147. doi: 10.1109/ICPC.2010.43.
[147] K. Kaur and S. Anand, “A Maintainability Estimation Model and Metrics for Object-Oriented Design (MOOD),” *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, vol. 2, no. 5, 2013.
[148] P. K. Goyal and G. Joshi, “QMOOD metric sets to assess quality of Java program,” in *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, Feb. 2014, pp. 520–533. doi: 10.1109/ICICICT.2014.6781337.
[149] S. Wagner *et al.*, “Operationalised product quality models and assessment: The Quamoco approach,” *Information and Software Technology*, vol. 62, pp. 101–123, 2015, doi: https://doi.org/10.1016/j.infsof.2015.02.009.
[150] R. Baggen, J. P. Correia, K. Schill, and J. Visser, “Standardized code quality benchmarking for improving software maintainability,” *Software Quality Journal*, vol. 20, no. 2, pp. 287–307, Jun. 2012, doi: 10.1007/s11219-011-9144-9.
[151] M. G. Siavvas, K. C. Chatzidimitriou, and A. L. Symeonidis, “QATCH-An adaptive framework for software product quality assessment,” *Expert Systems with*

Applications, vol. 86, pp. 350–366, 2017, doi: http://dx.doi.org/10.1016/j.eswa.2017.05.060.

[152] M. Siavvas, D. Kehagias, and D. Tzovaras, "A preliminary study on the relationship among software metrics and specific vulnerability types," in *2017 International Conference on Computational Science and Computational Intelligence (CSCI),* 2017, pp. 916–921. doi: http://dx.doi.org/10.1109/CSCI.2017.159.

[153] R. E. Bellman, *Dynamic Programming*. Dover Publications, 2003. [Online]. Available: https://books.google.gr/books?id=fyVtp3EMxasC

[154] C. Spearman, "The proof and measurement of association between two things," *The American journal of psychology,* vol. 100, no. 3/4, pp. 441–471, 1987, doi: http://dx.doi.org/10.2307/1422689.

[155] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic Press, 1977.

[156] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *IEEE Transactions on Software Engineering,* vol. 27, no. 7, pp. 630–650, 2001, doi: http://dx.doi.org/10.1109/32.935855.

[157] M. Efroymson, "Multiple regression analysis," *Mathematical methods for digital computers,* pp. 191–203, 1960.

[158] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th international conference on Software engineering (ICSE),* 2005, pp. 580–586. doi: http://dx.doi.org/10.1109/ICSE.2005.1553604.

[159] J. Munson and T. Khoshgoftaar, "Regression modelling of software quality: empirical investigation," *Information and Software Technology,* vol. 32, no. 2, pp. 106–114, 1990, doi: http://dx.doi.org/10.1016/0950-5849(90)90109-5.

[160] T. M. Khoshgoftaar and J. C. Munson, "Predicting software development errors using software complexity metrics," *IEEE Journal on Selected Areas in Communications,* vol. 8, no. 2, pp. 253–261, 1990, doi: http://dx.doi.org/10.1109/49.46879.

[161] D. W. Marquaridt, "Generalized inverses, ridge regression, biased linear estimation, and nonlinear estimation," *Technometrics,* vol. 12, no. 3, pp. 591–612, 1970, doi: http://dx.doi.org/10.1080/00401706.1970.10488699.

[162] T. G. Dietterich, "Machine learning for sequential data: A review," in *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR),* 2002, pp. 15–30. doi: http://dx.doi.org/10.1007/3-540-70659-3_2.

[163] C. Jin and J. Liu, "Applications of Support Vector Machine and Unsupervised Learning for Predicting Maintainability Using Object-Oriented Metrics," in *International Conference on Multimedia and Information Technology (MITA),* Apr. 2010, vol. 1, pp. 24–27. doi: 10.1109/MMIT.2010.10.

[164] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," *International Symposium on Software Reliability Engineering (ISSRE),* pp. 23–33, 2014, doi: 10.1109/ISSRE.2014.32.

[165] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture,* vol. 57, pp. 294–313, 2011, doi: 10.1016/j.sysarc.2010.06.003.

[166] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities,"

IEEE Transactions on Software Engineering, vol. 37, no. 6, pp. 772–787, 2011, doi: 10.1109/TSE.2010.81.

[167] R. Kohavi and others, “A study of cross-validation and bootstrap for accuracy estimation and model selection,” in *Proceedings of the 14th international joint conference on Artificial intelligence (IJCAI)*, 1995, vol. 2, pp. 1137–1145. doi: 10.5555/1643031.1643047.

[168] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, “Efficient and robust automated machine learning,” in *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS)*, 2015, vol. 2, pp. 2962–2970. doi: 10.5555/2969442.2969547.

[169] D. Tsoukalas, D. Kehagias, M. Siavvas, and A. Chatzigeorgiou, “Technical Debt Forecasting: An empirical study on open-source repositories - Supporting Material,” 2020. https://sites.google.com/view/technical-debt-forecasting/appendix

[170] B. A. Kitchenham and S. L. Pfleeger, “Principles of Survey Research: Part 3: Constructing a Survey Instrument,” *SIGSOFT Software Engineering Notes*, vol. 27, no. 2, pp. 20–24, Mar. 2002, doi: 10.1145/511152.511155.

[171] K. Schmid, “A Formal Approach to Technical Debt Decision Making,” in *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures (QoSA)*, New York, NY, USA, 2013, pp. 153–162. doi: 10.1145/2465478.2465492.

[172] D. Falessi, M. A. Shaw, F. Shull, K. Mullen, and M. S. Keymind, “Practical considerations, challenges, and requirements of tool-support for managing technical debt,” in *2013 4th International Workshop on Managing Technical Debt (MTD)*, 2013, pp. 16–19.

[173] Y. Guo, R. O. Spínola, and C. Seaman, “Exploring the costs of technical debt management–a case study,” *Empirical Software Engineering*, vol. 21, no. 1, pp. 159–182, 2016.

[174] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, “An exploratory study of the impact of code smells on software change-proneness,” in *Reverse Engineering, 2009. WCRE’09. 16th Working Conference on*, 2009, pp. 75–84.

[175] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, “Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study,” *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 257–268, 2014, doi: 10.1145/2635868.2635880.

[176] *Supporting Material*. 2020. [Online]. Available: https://sites.google.com/view/granular-td-forecasting/home

[177] V. Lenarduzzi, N. Saarimäki, and D. Taibi, “The technical debt dataset,” in *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, 2019, pp. 2–11.

[178] D. Tsoukalas, M. Mathioudaki, M. Siavvas, D. Kehagias, and A. Chatzigeorgiou, “A Clustering Approach towards Cross-project Technical Debt Forecasting - Supporting Material,” Oct. 01, 2020. https://sites.google.com/view/clustering-td-forecasting/appendix

[179] G. A. Campbell and P. P. Papapetrou, *SonarQube in action*, 1st edn. Greenwich: Manning Publications Co., 2013.

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

[181] D. Spadini, M. Aniche, and A. Bacchelli, “PyDriller: Python Framework for Mining Software Repositories,” in *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018. doi: 10.1145/3236024.3264598.

[182] N. Tsantalis, A. Ketkar, and D. Dig, “RefactoringMiner 2.0,” *IEEE Transactions on Software Engineering*, 2020, doi: 10.1109/TSE.2020.3007722.

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

[184] G. Digkas, A. N. Chatzigeorgiou, A. Ampatzoglou, and P. C. Avgeriou, “Can Clean New Code reduce Technical Debt Density,” *IEEE Transactions on Software Engineering*, 2020, doi: 10.1109/TSE.2020.3032557.

[185] M. R. Smith and T. Martinez, “Improving classification accuracy by identifying and removing instances that should be misclassified,” in *The 2011 International Joint Conference on Neural Networks*, 2011, pp. 2690–2697. doi: 10.1109/IJCNN.2011.6033571.

[186] M. Kuhn and K. Johnson, *Applied predictive modeling*, 1st edn. Springer, 2013.

[187] H. B. Mann and D. R. Whitney, “On a test of whether one of two random variables is stochastically larger than the other,” *The annals of mathematical statistics*, pp. 50–60, 1947.

[188] D. McFadden and others, “Conditional logit analysis of qualitative choice behavior,” 1973.

[189] H. He and E. A. Garcia, “Learning from Imbalanced Data,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263–1284, 2009, doi: 10.1109/TKDE.2008.239.

[190] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “SMOTE: synthetic minority over-sampling technique,” *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002, doi: https://doi.org/10.1613/jair.953.

[191] H. He and Y. Ma, “Imbalanced learning: foundations, algorithms, and applications,” 2013.

[192] P. Branco, L. Torgo, and R. P. Ribeiro, “A Survey of Predictive Modeling on Imbalanced Domains,” *ACM Comput. Surv.*, vol. 49, no. 2, Aug. 2016, doi: 10.1145/2907070.

[193] J. Demšar, “Statistical Comparisons of Classifiers over Multiple Data Sets,” *J. Mach. Learn. Res.*, vol. 7, pp. 1–30, Dec. 2006.

[194] A. J. Scott and M. Knott, “A Cluster Analysis Method for Grouping Means in the Analysis of Variance,” *Biometrics*, vol. 30, no. 3, pp. 507–512, 1974, doi: https://doi.org/10.2307/2529204.

[195] N. Mittas and L. Angelis, “Ranking and Clustering Software Cost Estimation Models through a Multiple Comparisons Algorithm,” *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 537–551, 2013, doi: 10.1109/TSE.2012.45.

[196] J. Davis and M. Goadrich, “The Relationship between Precision-Recall and ROC Curves,” in *Proceedings of the 23rd International Conference on Machine Learning*, New York, NY, USA, 2006, pp. 233–240. doi: 10.1145/1143844.1143874.

[197] Z. Codabux and C. Dutchyn, “Profiling Developers Through the Lens of Technical Debt,” New York, NY, USA, 2020. doi: 10.1145/3382494.3422172.

[198] M. Siavvas, D. Tsoukalas, M. Jankovic, D. Kehagias, and D. Tzovaras, “Technical debt as an indicator of software security risk: a machine learning approach for software development enterprises,” *Enterprise Information Systems*, vol. 0, no. 0, pp. 1–43, 2020, doi: 10.1080/17517575.2020.1824017.