Technical Debt Forecasting: An empirical study on open-source repositories

Dimitrios Tsoukalas[a,b,*], Dionysios Kehagias[a], Miltiadis Siavvas[a], Alexander Chatzigeorgiou[b]

[a] *Information Technologies Institute, Centre for Research and Technology Hellas, Thessaloniki 57001, Greece*
[b] *Department of Applied Informatics, University of Macedonia, Thessaloniki 54643, Greece*

Abstract
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 paper, 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 paper 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.

Keywords: technical debt, technical debt forecasting, machine learning, empirical study

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 [3].

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 [4]. 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 [5]. System engineers and project managers need the right tools and appropriate training support to be able to perform long-term effective software maintenance [5]. 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 [6], fault-proneness [7] and general software evolution trends [8], 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 [9]. 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 [10]. 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 [8], [11]–[14]. 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 [15].

The work presented in this paper is a logical continuation and extension of our previous efforts [10], 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 [10] the main focus lied on univariate time series forecasting methods, in the present paper 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,

* *Corresponding author*
E-mail address: tsoukj@iti.gr (D. Tsoukalas), diok@iti.gr (D. Kehagias), siavvasm@iti.gr (M. Siavvas), achat@uom.gr (A. Chatzigeorgiou)

learn also from the evolution of additional features related to the target variable is a widely-used strategy [16], [17], 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 paper, 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 1. 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[1] online repository. For each application, we collected a subsequent number of snapshots (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[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, 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 1.

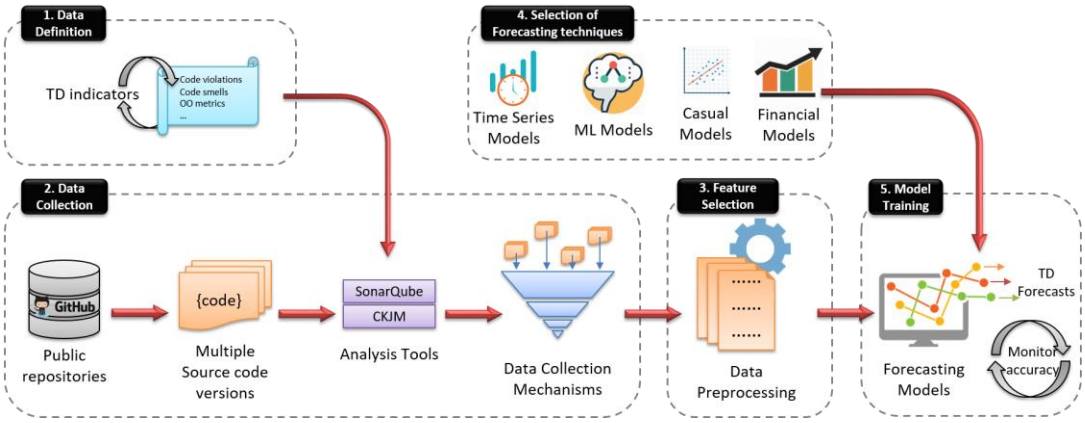


Figure 1: Paper roadmap

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 paper is structured as follows: Section 2 presents the background concepts in the field of forecasting and TD. Section 3 presents the related work in the field of forecasting models and more specifically in their applicability for TD forecasting. Section 4 thoroughly describes data definition, collection and pre-processing steps. Section 5 describes forecasting model training, testing and benchmarking, as well as the current state of technical implementation of the proposed approach.

[1] https://github.com/
[2] https://www.sonarqube.org/
[3] http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

Section 6 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 7 reports the limitations and validity threats of this empirical study, while section 8 discusses significant implications for both research and practice. Finally, Section 9 concludes the paper and discusses ideas for future work.

2 Background

This section provides an overview of relevant background on Forecasting and TD, in order to introduce unfamiliar readers with the main concepts of this paper.

2.1 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 [18]. 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) treat the examined system as a black box and assume that information needed to forecast is contained in a set of time-dependent data that will continue to follow same patterns as in the past [19]. 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 [20]. 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 [21], 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 paper 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 [22], [23]. A similar study by Lapedes and Farber [24] 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 [25], [26].

2.1.1 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 [27] and Lasso [28] 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.

2.1.2 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 regression, K-Nearest Neighbor regression, Decision Trees, Random Forest regression and various ANN variants, such as Multi-Layer Perceptron, Bayesian and Generalized Regression Neural Networks [21].

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) [29] 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 [30] 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) [31] 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 [32]. RF is an ensemble of Decision Trees trained with the "bagging" method [33]. 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.

2.2 Technical Debt Concepts

Nowadays, TD is seen as an important part of software management, as many studies have identified several causes for its creation. Fowler [34], [35] states that software development debt is usually a consequence of time pressure. Kruchten et al. [36] 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 [35] proposes TD quadrant, a 2×2 matrix (Intentionality x Awareness), to visualize four different pathways that lead to TD. McConnell [37] 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. Suryanarayana et al. [38] point out that extreme situation when accumulated TD is enormous and cannot be paid off could lead to technical bankruptcy. Moreover, several recent studies have highlighted the need to analyze TD from SDLC point of view. Li et al. [3] classify different TD types into ten levels based on their occurrence during the main phases of a software development process (i.e., requirements, design and architecture, implementation, testing, building, documentation, infrastructure, versioning, and defects).

2.2.1 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 [3]. 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 [39], 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 [40].

2.2.2 Technical Debt Indexes

In an attempt to provide an empirical TD quantification 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 [41]. 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 [42], and the Software Quality Assessment based on Lifecycle Expectations (SQALE) [43] methodology among others [44]–[46].

Regarding the various TD indexes and corresponding tools that have been proposed, in their study, Curtis et al. [44] are based on Software Economics theories and quantify TD as the cost of violating architectural and code rules, giving three levels of severity to violations: high, medium and low. To achieve that, they introduce a cost function that quantifies principal and interest taking as input the number of must-fix violations, the time required to fix each violation, and the cost for fixing a violation. To further support their findings, they integrate their formula into CAST[4], a tool that quantifies TD by identifying architectural and code violations and categorizing them by quality attributes. In another work, Marinescu [45] 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[5], a tool that evaluates software quality by providing a global score known as the Quality Deficit Index (QDI). In a related study, Letouzey [46] 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[6], 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. Finally, SonarQube[7] is a widely-used open source platform for continuous inspection of code quality that provides analysis functionalities and a wide range of metrics for measuring code quality attributes. During the previous years, SonarQube used the SQALE method to assess the TD of a software product but has now switched to a different method. More specifically, it 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. Other popular quality assessment tools that worth mentioning are Sigrid[8], Structure101[9], NDepend[10], and Teamscale[11].

---

[4] https://www.castsoftware.com/
[5] inFusion tool is no longer supported and has been evolved into http://www.aireviewer.com/
[6] https://www.vector.com/int/en/products/products-a-z/software/squore/
[7] https://www.sonarqube.org/
[8] https://www.softwareimprovementgroup.com/solutions/sigrid-software-assurance-platform/

3 Related Work

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. In this section, we investigate the state-of-the-art and examine the major contributions that have been made until today in the field of TD 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 [47]. 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 [48]. For this reason, the attempts to analyze, understand and predict the evolution of a software system have increased considerably in the last years [49], and nowadays, the terms software evolution and software maintenance are often used as synonyms [50]. In his work, Mens [50] 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 [42], 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 [51]–[53]. For instance, Wagner [51] 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. [52] try to predict object-oriented software maintainability by applying a Bayesian network, while Zhou et al. [53] 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 [42] further 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 [11], [14], [54], [55], software defects [12], [56], fault-proneness [7], [13], [57], [58], code smells [6], and vulnerabilities [59]. The majority of these methods try to approach the subject by applying time series or ML prediction models on individual software properties based on the analysis of available information (historical data, trends, source code metrics, etc.).

A commonly used technique to analyze the evolution of software systems is time series analysis. In their study, Yazdi et al. [11] 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. [14] 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. [12] 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. Finally, Goulo et al. [13] 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.

In addition to time series analysis, multiple studies address the problem of forecasting the evolution of various aspects of software quality by employing ML techniques. In their study, Chug and Malhotra [54] 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 [55] 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. In the same way, Fontana et al. [6] compare 16 different supervised ML techniques for code smell detection using 74 software systems. They report that the highest performance is obtained by using J48 and Random Forest algorithms, while code smells can be detected with very high accuracy. Regarding fault-proneness prediction, in a study conducted by Arisholm et al. [7], the authors propose a multivariate regression model for predicting fault-prone components of object-oriented legacy systems by using history change and fault data from previous releases. Moreover, in [57], Gondra et al. propose the use of ML to predict software fault-proneness. Their approach first employs sensitivity analysis to select software metrics that are more likely to indicate the existence of errors, and afterward, trains an ANN to predict future fault-proneness. In a relative study, Nagappan et al. [56] use principal component analysis on code metrics to build regression models that accurately predict the likelihood of post-release defects. Finally, Khoshgoftaar et al. [58] use regression and

[9] http://structure101.com/products/workspace/
[10] https://www.ndepend.com/
[11] https://www.cqse.eu/en/products/teamscale/landing/

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.

The multitude of models that are available in the literature for predicting the 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 [4], [40], [60]–[63]. In their study, Ampatzoglou et al. [62] 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. [63] 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. [40] 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, only a few contributions exist so far regarding TD forecasting [10], [64], indicating that it is a scarcely investigated field. The need for forecasting the evolution of TD has been highlighted by a recent study by Tsoukalas et al. [9], in which the authors raise the awareness of the gap in the field of TD. They claim that an 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, they stress that it would be useful to examine if TD forecasting could foster the development of high-quality software products. In a first attempt towards this issue, Skourletopoulos et al. [64] introduce the concept of predicting TD for Software as a Service (SaaS) systems, by exploiting COCOMO, a software cost model proposed by Boehm [65]. However, their study is limited only to cloud computing systems. In another study [10], the authors empirically evaluate the ability of time series analysis to model and predict TD evolution in long-lived, open-source software projects. They find that the autoregressive integrated moving average model (ARIMA) can provide accurate predictions over a fairly long period of up to 8 weeks. However, they observe that predictive power decreases considerably for longer forecasting horizons.

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, 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.

4 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[12] and CKJM Extended[13] 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.

4.1 TD Indicator Definition

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 [3], [66] 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) [67] or the Quality Model for Object Oriented Design (QMOOD) [68], 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 [69], [70], 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 [66], [71]. 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 [72]. These problems may impede the software maintenance process and impose the need for code refactoring [73]. In addition, Automatic Static Analysis (ASA) tools, such as FindBugs[14] or Checkstyle[15], are also widely used to indicate TD [74]–[76]. ASA tools allow the analysis of source code in search for bugs or violations of good programming

[12] https://www.sonarqube.org/
[13] http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/
[14] http://findbugs.sourceforge.net/
[15] https://checkstyle.sourceforge.io/

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 [77].

As discussed above, 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 paper, 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[16] including nearly more than 100K public open-source projects[17]. 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, 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 [78], 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[18] 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 [67], [68]. 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 [69], [70] (see "Studies" column in Table 1). 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 [79], an extended version of the CKJM open-source tool able to calculate a wide range of metrics[19] (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 1, 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.

Table 1 - 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.	[80] [81] [4]
Security Metrics		
vulnerabilities	Total number of vulnerability issues of a project.	[81] [4] [82]
Maintainability Metrics		

[16] https://www.sonarqube.org/
[17] https://sonarcloud.io/explore/projects
[18] https://docs.sonarqube.org/latest/user-guide/metric-definitions/
[19] http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/metric.html

code_smells	Total number of code smell issues of a project.	[4] [73] [83] [84] [85] [86] [87]
Size metrics		
comment_lines	Number of lines containing either comment or commented-out code of a project.	[81] [88]
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.	[4] [87] [89] [90] [55]
Coverage Metrics		
uncovered_lines	Number of lines of code of a project, which are not covered by unit tests.	[81]
Duplication Metrics		
duplicated_blocks	Number of duplicated blocks of lines of a project.	[81] [91] [45]
Complexity Metrics		
complexity	The Cyclomatic Complexity of a project calculated based on the number of paths through the code.	[92] [93] [94]
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.	[88] [95]
WMC	Weighted Methods per Class: The total number of methods that a class contains weighted by their complexity values, averaged for all project classes.	[90] [92] [94] [52] [96] [53] [55]
DIT	Depth of Inheritance Tree: The depth of inheritance tree for each class from the object hierarchy top, averaged for all project classes.	[92] [94] [52] [96] [53] [97] [55]
NOC	Number of Children: The number of immediate descendants (i.e., children) of a class, averaged for all project classes.	[92] [93] [96] [53] [97] [55]
RFC	Response for a Class: The number of local methods plus the number of methods called by class methods, averaged for all project classes.	[90] [92] [93] [94] [52] [96] [53] [97]
Coupling Metrics		
CBO	Coupling Between Objects: The total number of classes coupled to a given class, averaged for all project classes.	[90] [92] [94] [96] [97]
Ca	Afferent Coupling: The total number of other classes that call methods of the given class, averaged for all project classes.	[88] [97] [98]
Ce	Efferent Coupling: The total number of other classes that their methods are called by the given class, averaged for all project classes.	[88] [97] [98]
CBM	Coupling Between Methods: The total number of parent classes to which a given class is coupled, averaged for all project classes.	[90] [92] [94]
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.	[90] [92]
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.	[90] [92] [52] [96] [53] [97] [55]
LCOM3	Lack of Cohesion in Methods: Similar to LCOM but ranging from 0 to 2.	[94] [97]
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).	[90]
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.	[93] [94] [52] [53] [55]
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).	[99] [100]
MOA	Measure of Aggregation: The total number of data declarations (class fields) whose types are user defined classes, averaged for all project classes.	[99] [100]

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 1), 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 **Error! Reference source not found.** will allow us to come up with the best predictors set, tailored to our dataset and experimental setup.

4.2 Collection of Data

To start the dataset construction process, we initially selected 15 popular open-source applications from the GitHub[20] 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

[20] https://github.com

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 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[21]. In Table 2, the applications that were selected for constructing the codebase are presented in detail.

Table 2: 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.

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

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 4.1) in order to analyze each snapshot and build 15 application-specific datasets consisting of the TD indicators described in Table 1. 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 paper 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. [101] and Bagen et al. [102], but also in some of our previous studies as well [103], [104].

4.3 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" [105], 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 1 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 4.2 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[22]. The same dataset was used in the study by Siavvas et al. [103] for calibrating a Quality Assessment Model, as well as in a similar study [104] 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 4.2. Finally, we merged these metrics with the metrics obtained from the last snapshot of each of our analyzed applications presented in Table 2, 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 2 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[23].

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.

4.3.1 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 3. 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 2) 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 3 we have included all 225 applications. For the conduction of our experiments, we used the Python programming language and more specifically the Pandas[24] data analysis library.

[22] https://mvnrepository.com/
[23] https://sites.google.com/view/technical-debt-forecasting/main
[24] https://pandas.pydata.org/

Table 3: Descriptive statistics of TD indicators (extended dataset)

Metric	*Mean value*	*Standard deviation*	*Minimum value*	*Lower quartile*	*Median value*	*Upper quartile*	*Maximum value*	*Skewness*	*Kurtosis*
bugs	29.665	61.148	0	3	9	30	585	5.286	37.601
vulnerabilities	38.818	125.935	0	1	7	32	1598	9.675	114.507
code_smells	858.660	2095.092	2	94	235	732	16442	5.551	35.262
comment_lines	5113.780	15587.257	1	387	1381	3889	170698	8.203	77.661
ncloc	19524.278	37253.222	175	2655	7068	20438	357664	5.262	38.091
uncovered_lines	9815.507	17856.102	32	1302	3297	10288	137326	3.905	19.212
duplicated_blocks	110.177	429.327	0	0	12	47	4219	7.771	67.183
complexity	3914.024	7420.959	11	421	1352	4284	61862	4.275	23.544
AMC	53.370	74.382	9.407	25.374	36.565	52.876	819.191	6.697	58.288
WMC	36.399	82.087	2.085	12.638	18.921	31.524	814.007	7.136	57.322
DIT	0.760	0.309	0	0.655	0.813	0.964	1.523	-0.936	0.977
NOC	0.527	1.900	0	0.080	0.261	0.481	26.770	12.828	176.915
RFC	71.453	90.004	11.618	39.508	54.295	78.112	993.299	7.631	69.342
CBO	17.177	21.071	0	9.056	13.554	18.983	245.469	7.139	68.779
Ca	6.243	8.145	0	2.463	3.850	7.399	73.761	4.974	31.903
Ce	12.425	18.100	0	5.922	9.664	13.024	210.225	7.710	73.955
CBM	0.143	0.231	0	0.026	0.085	0.164	2.340	5.366	41.706
IC	0.098	0.106	0	0.025	0.071	0.137	0.595	2.054	5.452
LCOM	442.525	1552.824	-12160.67	61.584	153.151	503.916	13693.182	0.885	46.386
LCOM3	0.863	0.151	0.435	0.776	0.840	0.916	1.648	1.377	5.060
CAM	0.337	0.081	0.050	0.278	0.333	0.377	0.683	0.858	3.011
NPM	23.251	51.913	0.250	8.009	12.287	21.058	655.642	9.550	109.341
DAM	0.661	0.178	0.024	0.574	0.682	0.779	1.000	-0.952	1.526
MOA	3.579	7.849	0	0.761	1.591	3.263	81.424	6.701	56.123

Metrics that vary little are not likely to be useful predictors. In our case, from Table 3 we observed that for all metrics there are significant differences between the lower 25^{th} (lower) percentile, the median, and the 75^{th} (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 [67], 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 2, histograms of each metric are presented to further complement our initial analysis. We used histograms to further examine the normality and skewness of each metric. A normal distribution is symmetric and bell-shaped. We observed that most of the metrics are not normally distributed.

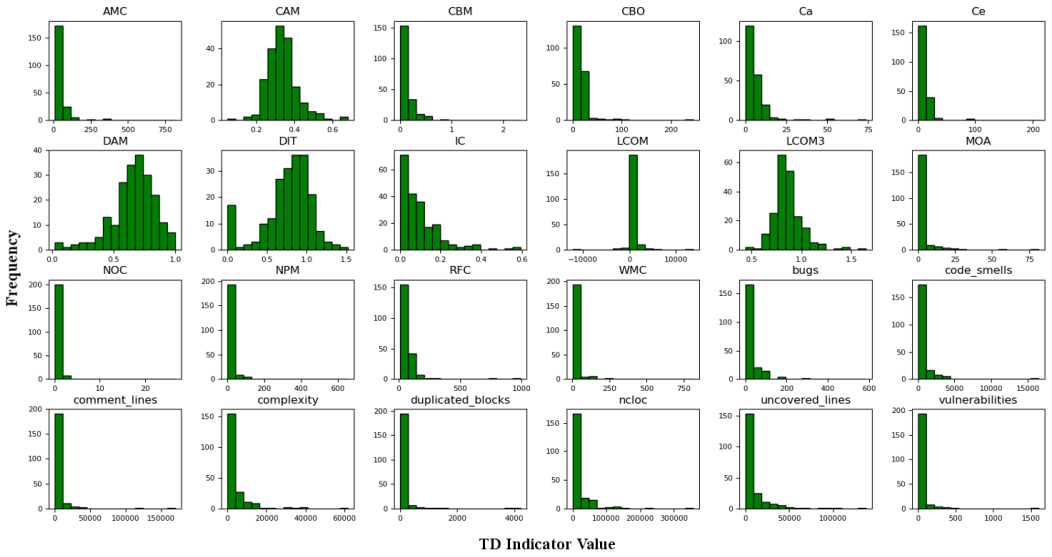


Figure 2: Histograms of TD indicators (extended dataset)

To further validate this finding we used boxplots, which can be found online[25] as supportive 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.

4.3.2 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) [106] analysis between TD Principal as computed by SonarQube and each TD indicator described in Table 3 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's et al. [107] 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 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 3, 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.

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

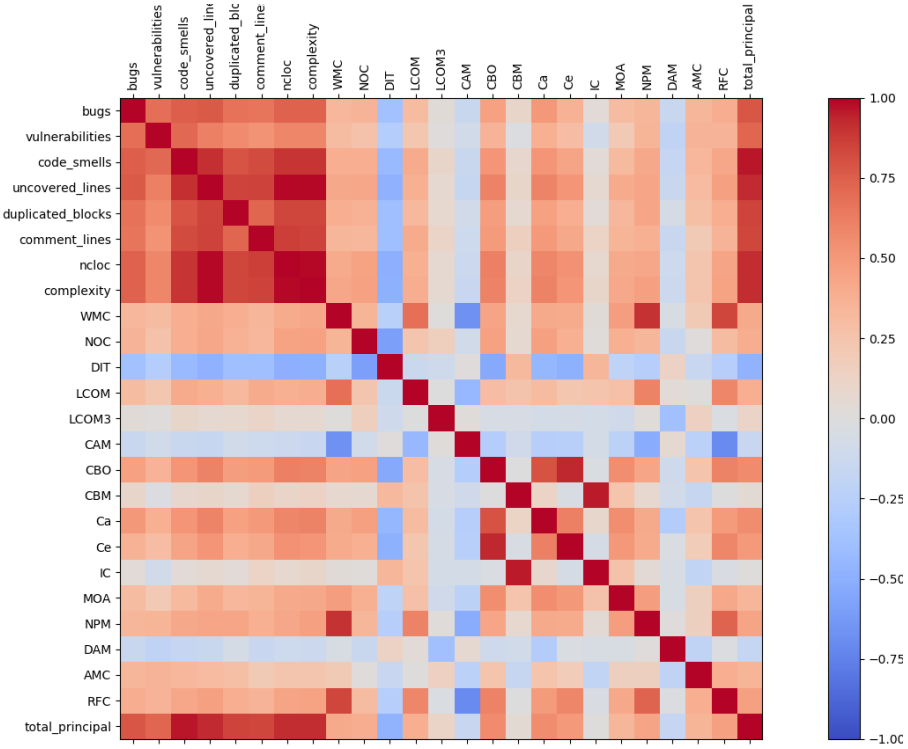


Figure 3: Spearman's rank correlation of TD indicators (extended dataset)

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 4. 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 4: 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.

4.3.3 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. [108]. 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 [7], [95], [108]. 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 [21]. Using the natural logarithm reduces the skew of the response and predictors (linear regression assumptions include normal distribution of the residuals).

Table 5 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 5: 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 α = 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.

4.3.4 Multivariate Analysis

Since univariate analysis did not filter out any metrics, we proceeded with applying multivariate regression analysis [109] 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 [109], 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 [7], [110]–[113]. During this step, all metrics reported in Table 5 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 4.4 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 * 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 6. 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[26] and provide information regarding the metric that was eliminated at each iteration until reaching the final set of TD indicators shown in Table 6. 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*) [114]. 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:

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

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

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 6. 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 6: 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[27] 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 5.

4.4 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 7. 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" [115] 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 past information and future information simultaneously into a single row. This approach is described in more detail below.

Table 7: 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 7, 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 7 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 7, 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 8. 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 8.

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

Table 8: 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 9.

Table 9: 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.

5 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 **Error! Reference source not found.**, 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.

5.1 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 [69], [54], [116], [117], [55] and Security [118]–[120], or lower-level software properties, such as code smells [6] and defects [111]. 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 paper, 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*[28] 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 [121], 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 4 below provides a visualization of the Walk-forward Train-Test validation behavior.

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

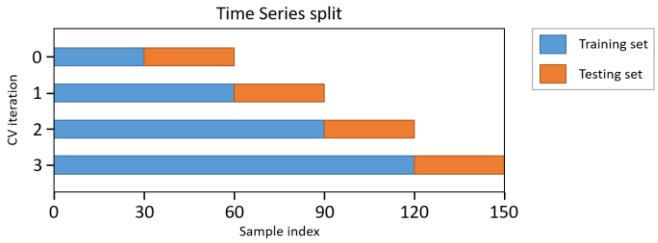


Figure 4: Walk-forward Train-Test validation

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 [122]. It is worth mentioning here that number of splits = 5 was chosen also for the other application datasets that contain 150 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[29], a python implementation of the Grid-search method [123]. 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:

$$MAPE = \frac{100}{n}\sum_{i=1}^{n}\frac{|y_i - \hat{y}_i|}{y_i} \tag{2}$$

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:

$$MAE = \frac{\sum_{i=1}^{n}|y_i - \hat{y}_i|}{\mathrm{n}} \tag{3}$$

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{4}$$

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

In Table 10, 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

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

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 4.4, 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 Section 5.2.

Table 10: 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 5 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 6, Figure 7, Figure 8, Figure 9 and Figure 10 illustrate the MAPE of the forecasting models for 1, 5, 10, 20 and 40 steps (weeks) ahead respectively.

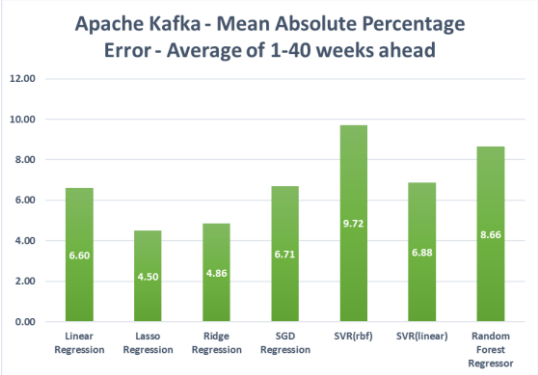


Figure 5: Apache Kafka TD predictions – MAPE averaged for all steps-ahead using Walk-forward Train-Test validation

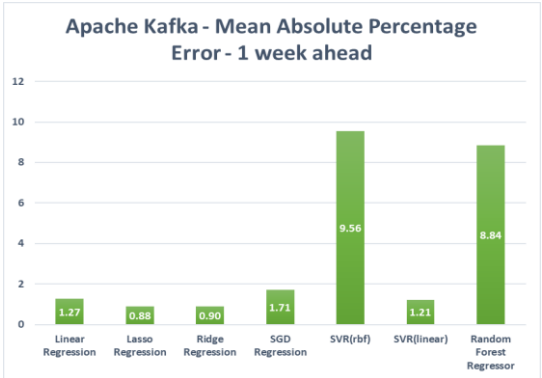


Figure 6: Apache Kafka TD predictions – MAPE for 1 step-ahead using Walk-forward Train-Test validation

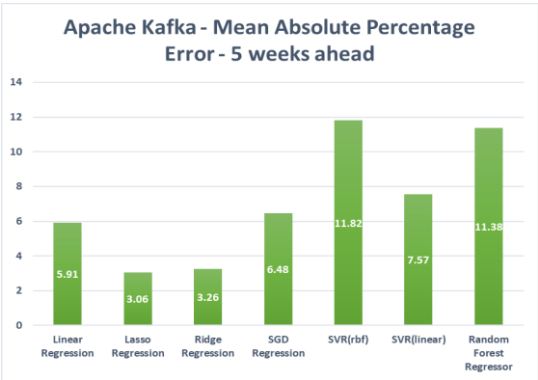


Figure 7: Apache Kafka TD predictions – MAPE for 5 steps-ahead using Walk-forward Train-Test validation

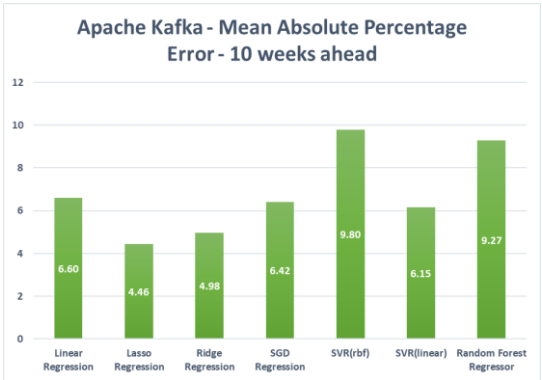


Figure 8: Apache Kafka TD predictions – MAPE for 10 steps-ahead using Walk-forward Train-Test validation

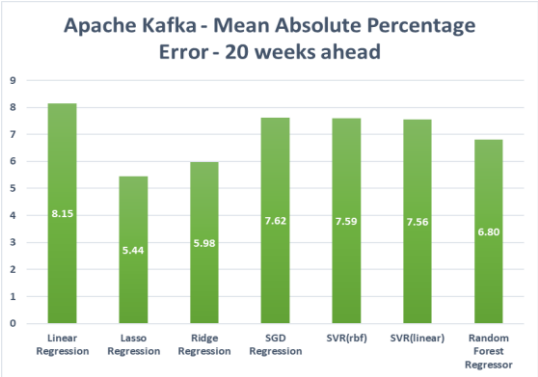


Figure 9: Apache Kafka TD predictions – MAPE for 20 steps-ahead using Walk-forward Train-Test validation

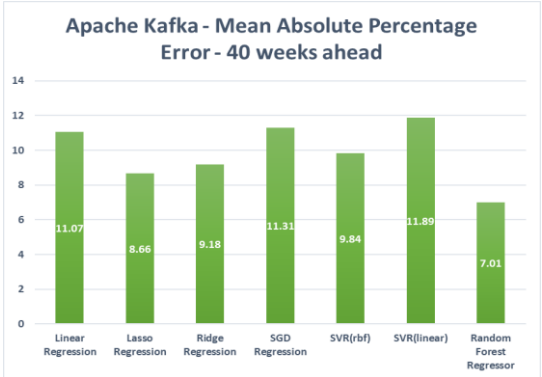


Figure 10: Apache Kafka TD predictions – MAPE for 40 steps-ahead using Walk-forward Train-Test validation

By observing Figure 5, 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 6, 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 7 and Figure 8 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 9. 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 10, 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 Appendix section (Table 14 to Table 27). Figure 11 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 12, Figure 13, Figure 14, Figure 15 and Figure 16 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.

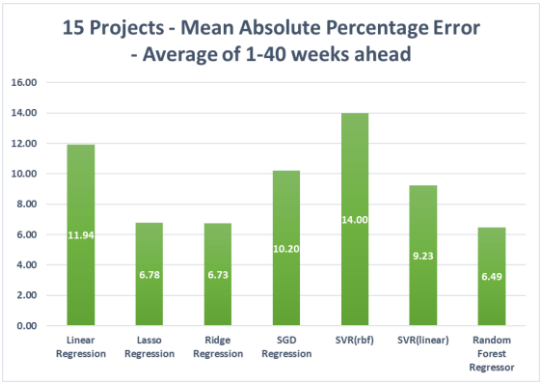


Figure 11: 15 projects TD predictions – MAPE averaged for all steps-ahead using Walk-forward Train-Test validation

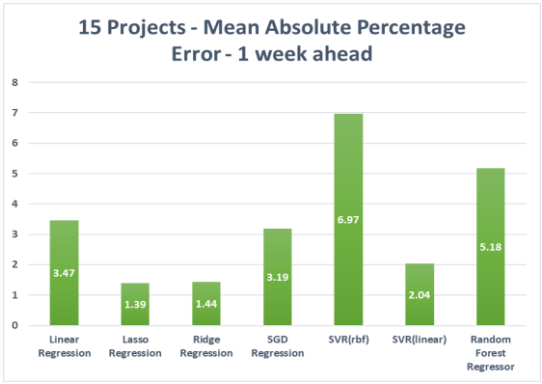


Figure 12: 15 projects TD predictions – MAPE for 1 step-ahead using Walk-forward Train-Test validation

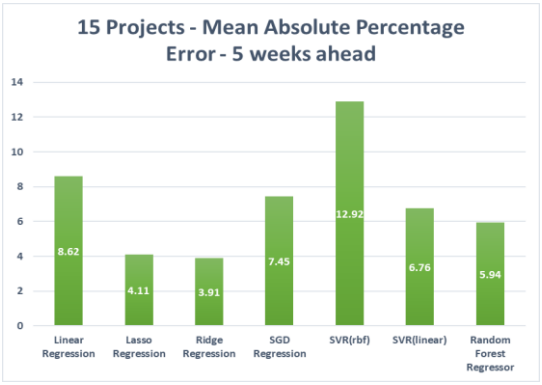


Figure 13: 15 projects TD predictions – MAPE for 5 steps-ahead using Walk-forward Train-Test validation

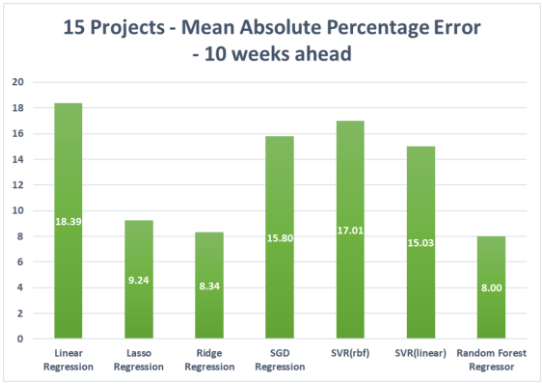


Figure 14: 15 projects TD predictions – MAPE for 10 steps-ahead using Walk-forward Train-Test validation

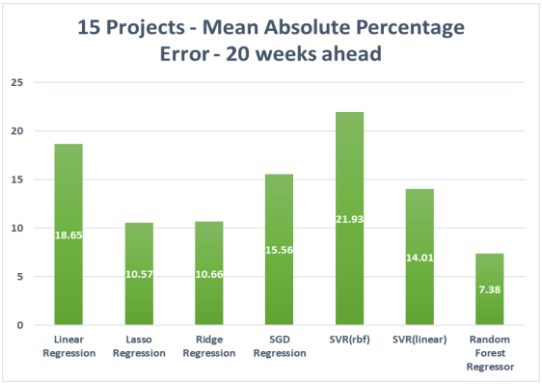


Figure 15: 15 projects TD predictions – MAPE for 20 steps-ahead using Walk-forward Train-Test validation

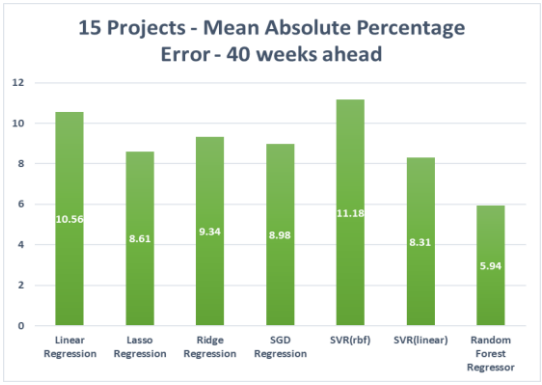


Figure 16: 15 projects TD predictions – MAPE for 40 steps-ahead using Walk-forward Train-Test validation

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 12 and Figure 13, 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 15) up to 40 steps ahead

(Figure 16), 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 the observation 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.

5.2 Model Execution

Following the construction and benchmarking of our models described in Section 5.1, this section presents indicative examples of model execution as well as indicative visualizations of the forecasting results. As reported in Section 4.4, we decided to adopt the *Direct* approach, meaning that separate models were developed to forecast each forecast lead time. In Figure 17 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.

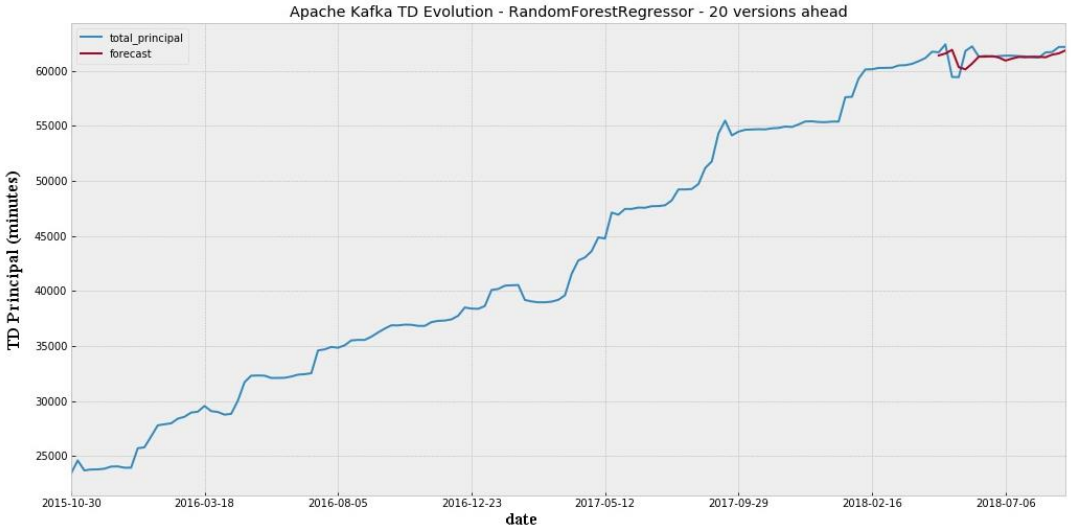


Figure 17: Apache Kafka TD Principal forecasting for 20 steps ahead using Random Forest and the Direct approach

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[30] supportive material. As can be seen in both Figure 17 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 5.1.

5.3 Technical Implementation

The work presented in this paper 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[31] 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 allows the invocation of forecasting models and displays the results, providing users with insightful information for the future evolution of

[30] https://sites.google.com/view/technical-debt-forecasting/main
[31] https://sdk4ed.eu/

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[32] (currently being used for development purposes). The main screen of the tool is provided in Figure 18.

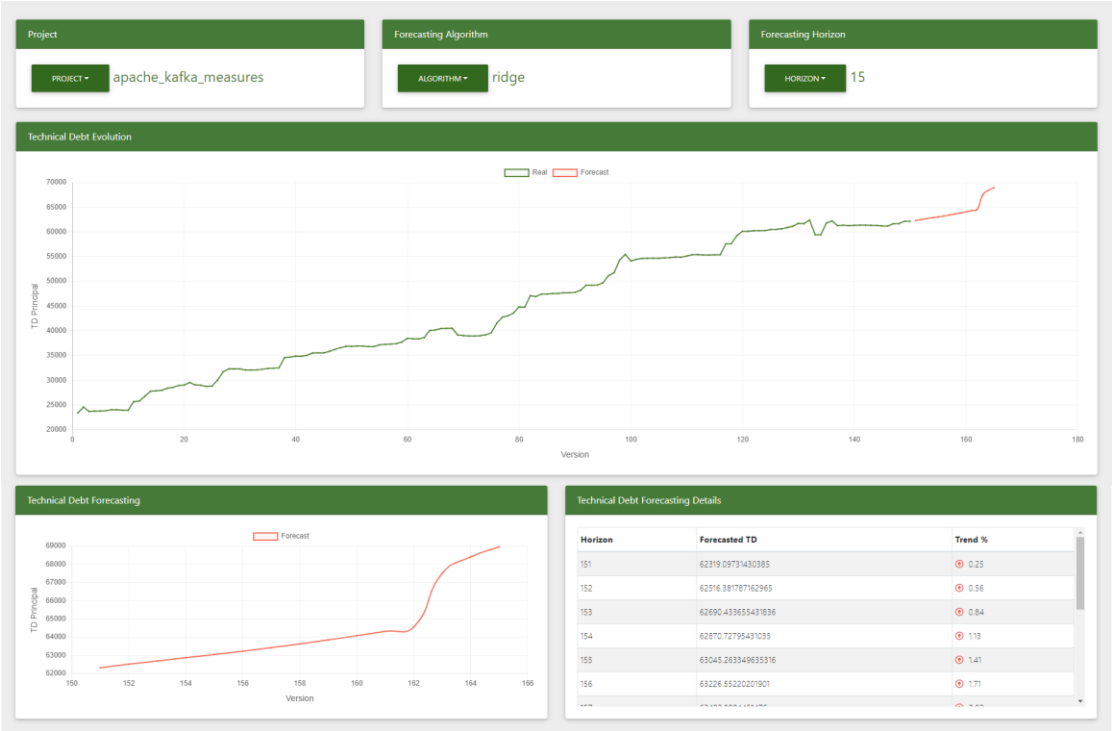


Figure 18: Main screen of the TD Forecasting tool

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.

6 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.

6.1 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

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

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[33]. 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 [124]: (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 11), and an introductory part (2 questions). The questionnaire begins with the introductory part (i.e., 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 4. 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 6.

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 4.2 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[34].

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 5.3 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.

[33] https://forms.gle/Jjg8RoA55m1EwMJ77
[34] https://sites.google.com/view/technical-debt-forecasting/main

Table 11: 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?
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.

6.2 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.

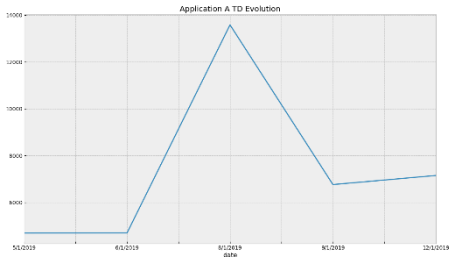
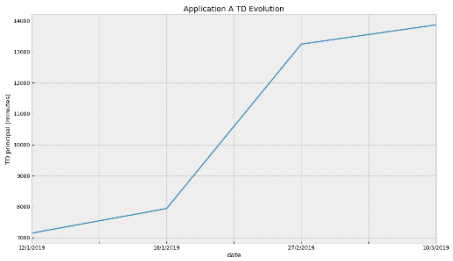
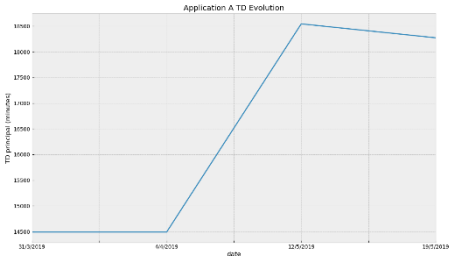
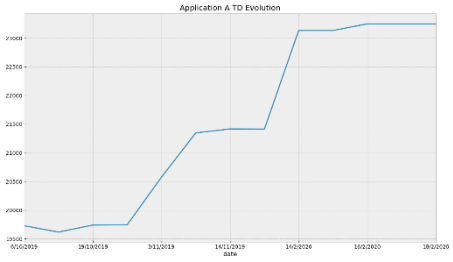
Regarding Q2.3 "To what extent would a forecast of the TD Principal make you consider changing the planned future development of a project?", on a Likert scale ranging from: 1 - "Not at all" to 5 - "To a great extent", three participants (P1, P2, and P4) chose "To a moderate extent" (option 4), while 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 [72], 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 12 and Table 13, 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[35].

Table 12: Participants comments on TD Principal trends of Project A

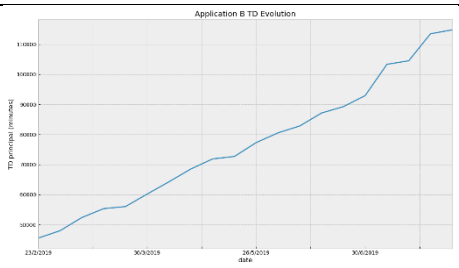
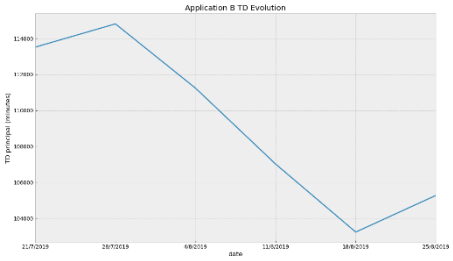
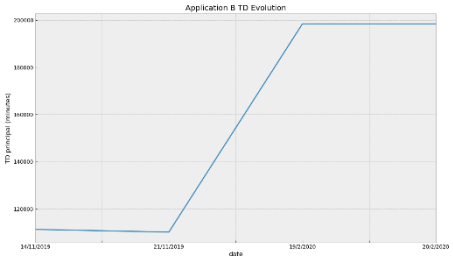
Cases	TD Principal Trend	Participants Comments (what is the cause of these TD changes?)
Q3.1	 Temporal increase (6/1/19 – 9/1/19)	**P3:** "*The project was new and a large amount of functionality was added for the first time from less experienced engineers. Then improvements were made to the initial 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	 Sharp increase (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	 Sharp increase (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*
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 12, 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

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

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 13: 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 13, 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 19 and Figure 20 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.

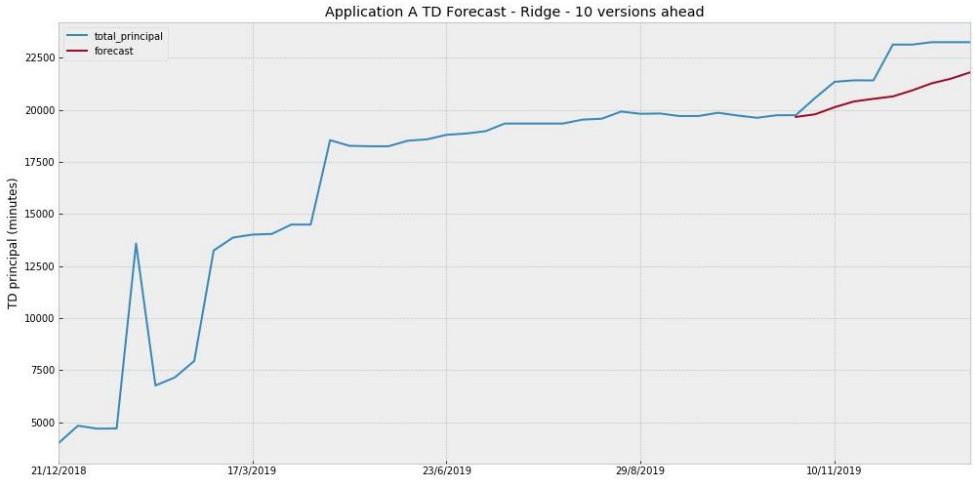


Figure 19: Project A TD Principal forecasting for 10 steps ahead using Ridge (the ground truth from the starting point of the forecasts and onwards was hidden from the participants)

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 19. 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.

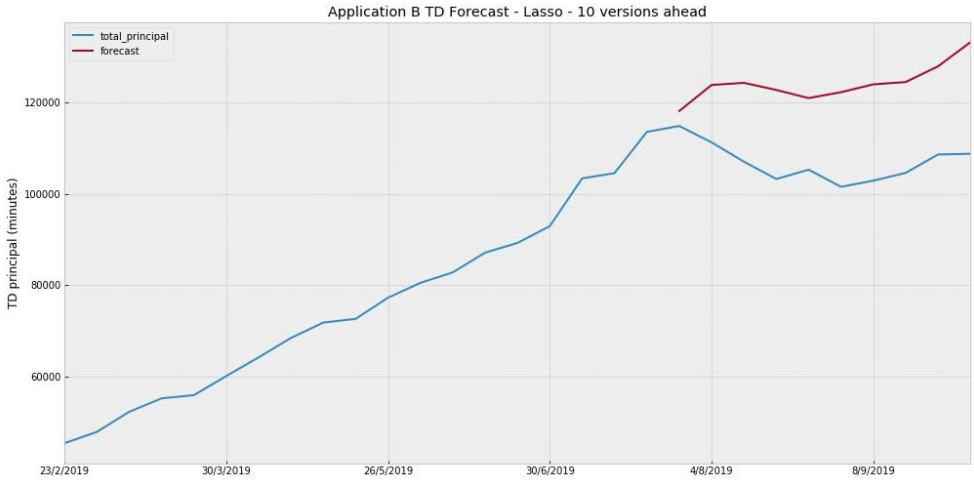


Figure 20: Project B TD Principal forecasting for 10 steps ahead using Ridge (the ground truth from the starting point of the forecasts and onwards was hidden from the participants)

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.

7 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 6, 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 paper 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 paper 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 6. 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[36]. Moreover, the source code repositories of the 15 selected projects are available on GitHub to obtain the same data.

8 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.

8.1 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

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

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 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. [63] 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.

8.2 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 [125]. 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.

9 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 regarding whether to repay or not 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 paper is to examine whether and to what extend 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.

10 Acknowledgment

This work is funded by the European Union's Horizon 2020 Research and Innovation Programme through SDK4ED project under Grant Agreement No. 780572.

11 Appendix

Table 14: Apache Ofbiz TD predictions using Walk-forward Train-Test validation

Model	*MAE (minutes)*	*RMSE (minutes)*	*MAPE (%)*
MLR	11800.06	14939.92	5.70
Lasso regressor	2586.26	2881.59	1.25
Ridge regressor	3580.03	4043.14	1.73
SGD regressor	3998.45	4749.17	1.93
SVR regressor (linear)	4041.08	4945.64	1.96
SVR regressor (rbf)	5371.22	6492.32	2.60
Random Forest Regressor	1842.31	2100.87	0.89

Table 15: Apache SystemML TD predictions using Walk-forward Train-Test validation

Model	*MAE (minutes)*	*RMSE (minutes)*	*MAPE (%)*
MLR	4489.38	5151.21	3.78
Lasso regressor	3949.70	4864.14	3.32
Ridge regressor	4125.15	5070.44	3.50
SGD regressor	4182.12	4712.35	3.53
SVR regressor (linear)	8771.71	10464.37	7.49
SVR regressor (rbf)	4201.63	4850.27	3.52
Random Forest Regressor	4096.68	4661.39	3.46

Table 16: Apache Groovy TD predictions using Walk-forward Train-Test validation

Model	*MAE (minutes)*	*RMSE (minutes)*	*MAPE (%)*
MLR	44141.98	57592.53	20.41
Lasso regressor	24910.16	32278.33	11.56
Ridge regressor	26379.48	33607.69	12.16
SGD regressor	29316.00	38734.54	13.51
SVR regressor (linear)	22512.98	29346.43	10.56
SVR regressor (rbf)	15716.78	20060.46	7.38
Random Forest Regressor	10384.53	13494.23	4.92

Table 17: Apache Nifi TD predictions using Walk-forward Train-Test validation

Model	*MAE (minutes)*	*RMSE (minutes)*	*MAPE (%)*
MLR	5293.48	6251.46	5.65
Lasso regressor	1920.77	2197.68	1.95
Ridge regressor	2041.70	2320.67	2.09
SGD regressor	2386.59	2678.10	2.47
SVR regressor (linear)	6359.22	7132.60	6.66
SVR regressor (rbf)	3852.61	4252.67	4.03
Random Forest Regressor	3233.18	3461.34	3.34

Table 18: Google Guava TD predictions using Walk-forward Train-Test validation

Model	*MAE (minutes)*	*RMSE (minutes)*	*MAPE (%)*
MLR	11800.06	14939.92	5.70

Lasso regressor	2586.26	2881.59	1.25
Ridge regressor	3580.03	4043.14	1.73
SGD regressor	3998.45	4749.17	1.93
SVR regressor (linear)	4041.08	4945.64	1.96
SVR regressor (rbf)	5371.22	6492.32	2.60
Random Forest Regressor	1842.31	2100.87	0.89

Table 19: Square Okhttp TD predictions using Walk-forward Train-Test validation

Model	*MAE (minutes)*	*RMSE (minutes)*	*MAPE (%)*
MLR	990.90	1116.48	10.24
Lasso regressor	620.66	725.74	6.24
Ridge regressor	434.47	524.53	4.38
SGD regressor	902.72	1018.00	9.36
SVR regressor (linear)	3096.27	3414.23	32.97
SVR regressor (rbf)	749.76	862.88	7.66
Random Forest Regressor	316.21	383.99	3.20

Table 20: Square Retrofit TD predictions using Walk-forward Train-Test validation

Model	*MAE (minutes)*	*RMSE (minutes)*	*MAPE*
MLR	603.55	678.93	13.25
Lasso regressor	247.41	283.32	5.65
Ridge regressor	239.37	274.13	5.48
SGD regressor	470.46	517.63	10.42
SVR regressor (linear)	902.16	1061.68	19.68
SVR regressor (rbf)	423.43	469.03	9.54
Random Forest Regressor	198.28	233.01	4.67

Table 21: Spring Boot TD predictions using Walk-forward Train-Test validation

Model	*MAE (minutes)*	*RMSE (minutes)*	*MAPE (%)*
MLR	128.56	151.70	5.50
Lasso regressor	122.08	141.07	5.23
Ridge regressor	124.33	142.50	5.30
SGD regressor	119.47	138.43	5.11
SVR regressor (linear)	642.80	786.51	28.05
SVR regressor (rbf)	141.77	163.29	6.11
Random Forest Regressor	101.38	116.44	4.30

Table 22: Apache CommonsIO TD predictions using Walk-forward Train-Test validation

Model	*MAE (minutes)*	*RMSE (minutes)*	*MAPE (%)*
MLR	240.97	311.95	6.80
Lasso regressor	160.64	208.43	4.35
Ridge regressor	160.98	210.00	4.36
SGD regressor	233.35	294.95	6.52
SVR regressor (linear)	230.17	311.15	6.38
SVR regressor (rbf)	220.05	289.50	6.10
Random Forest Regressor	314.67	369.86	8.44

Table 23: Apache Incubator TD predictions using Walk-forward Train-Test validation

Model	*MAE (minutes)*	*RMSE (minutes)*	*MAPE (%)*
MLR	8666.60	10957.26	22.13
Lasso regressor	3221.57	3770.47	8.33
Ridge regressor	3085.79	3576.34	7.97
SGD regressor	6060.73	7022.08	15.40
SVR regressor (linear)	2932.74	3404.16	7.56
SVR regressor (rbf)	2913.77	3432.83	7.51
Random Forest Regressor	2731.86	3110.55	7.06

Table 24: Java WebSocket TD predictions using Walk-forward Train-Test validation

Model	*MAE (minutes)*	*RMSE (minutes)*	*MAPE (%)*
MLR	652.78	713.54	43.29
Lasso regressor	368.33	412.58	25.98
Ridge regressor	409.87	449.94	29.94
SGD regressor	645.63	703.79	43.82
SVR regressor (linear)	586.90	654.24	39.97
SVR regressor (rbf)	682.55	742.21	50.80
Random Forest Regressor	444.89	494.17	32.52

Table 25: Zxing TD predictions using Walk-forward Train-Test validation

Model	*MAE (minutes)*	*RMSE (minutes)*	*MAPE (%)*
MLR	205.99	238.87	1.57
Lasso regressor	94.77	105.33	0.72
Ridge regressor	77.02	85.17	0.58
SGD regressor	105.05	119.87	0.79
SVR regressor (linear)	114.15	130.16	0.86
SVR regressor (rbf)	112.55	127.93	0.85
Random Forest Regressor	79.54	89.46	0.60

Table 26: Jenkins TD predictions using Walk-forward Train-Test validation

Model	*MAE (minutes)*	*RMSE (minutes)*	*MAPE (%)*
MLR	2689.58	3244.52	3.35
Lasso regressor	2694.02	3216.50	3.35
Ridge regressor	3009.70	3519.39	3.73
SGD regressor	2286.04	2774.47	2.83
SVR regressor (linear)	4443.72	5393.16	5.54
SVR regressor (rbf)	2469.89	3029.77	3.05
Random Forest Regressor	2309.38	2737.34	2.87

Table 27: Openfire TD predictions using Walk-forward Train-Test validation

Model	*MAE (minutes)*	*RMSE (minutes)*	*MAPE (%)*
MLR	9764.29	11254.28	18.30
Lasso regressor	7943.13	9464.07	14.94

Ridge regressor	9844.82	11965.04	17.84
SGD regressor	6584.61	7622.49	11.53
SVR regressor (linear)	12931.75	14325.71	21.48
SVR regressor (rbf)	8136.43	12571.07	14.42
Random Forest Regressor	10991.14	12090.32	17.55

12 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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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," 2018, doi: http://dx.doi.org/10.1109/IS.2018.8710521.

[10] 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 press, doi: 10.13140/RG.2.2.33152.79367.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] A. K. Palit and D. Popovic, *Computational intelligence in time series forecasting: theory and engineering applications*. Springer Science & Business Media, 2006.

[19] J. Das, *Statistics for Business Decisions*. Academic Publishers, 2012.

[20] 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.

[21] 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.

[22] P. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," Harvard University, Cambridge, 1974.

[23] 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.

[24] A. Lapedes and R. Farber, "Nonlinear signal processing using neural networks: Prediction and system modelling," United States, 1987.

[25] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*, vol. 1. Springer series in statistics New York, 2001.

[26] E. Alpaydin, *Introduction to Machine Learning, 2nd edn.*, 2nd ed. The MIT Press (February 2010), 2010.

[27] 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.

[28] 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.
[29] 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.
[30] 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.
[31] L. Breiman, *Classification and Regression Trees*. Routledge, 2017.
[32] 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.
[33] L. Breiman, “Bagging predictors,” *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996, doi: http://dx.doi.org/10.1007/BF00058655.
[34] M. Fowler, “Technical Debt,” 2003. http://www.martinfowler.com/bliki/TechnicalDebt.html (accessed Jul. 30, 2018).
[35] M. Fowler, “Technical Debt Quadrant,” 2009. http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html (accessed Jul. 30, 2018).
[36] 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.
[37] 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).
[38] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.
[39] C. Seaman and Y. Guo, “Measuring and monitoring technical debt,” in *Advances in Computers*, vol. 82, Elsevier, 2011, pp. 25–46.
[40] 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.
[41] 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.
[42] 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.
[43] 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.
[44] 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.
[45] 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.
[46] 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.
[47] 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.
[48] 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.
[49] 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.
[50] 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.
[51] 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.
[52] 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.
[53] 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.
[54] 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.
[55] 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.
[56] 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.
[57] 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.
[58] 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.
[59] 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.
[60] 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.
[61] 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.

[62] A. Ampatzoglou, A. Ampatzoglou, P. Avgeriou, and A. Chatzigeorgiou, “Establishing a framework for managing interest in technical debt,” 2015, doi: 10.5220/0005885700750085.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.
[91] 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.
[92] 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.
[93] 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.
[94] 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.
[95] 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.
[96] 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.
[97] 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.
[98] 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.
[99] 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.
[100] 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.
[101] 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.
[102] 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.
[103] 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.
[104] 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.
[105] R. E. Bellman, *Dynamic Programming*. Dover Publications, 2003.
[106] 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.
[107] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic Press, 1977.
[108] 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.
[109] M. Efroymson, “Multiple regression analysis,” *Mathematical methods for digital computers*, pp. 191–203, 1960.
[110] 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.
[111] 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.
[112] 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.
[113] 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.
[114] 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.
[115] 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.
[116] 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.
[117] 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.
[118] 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.
[119] 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.
[120] 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.

[121] 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.

[122] 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.

[123] 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.

[124] 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.

[125] 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.