Received <day> <Month>, <year>; | Revised <day> <Month>, <year>; | Accepted <day> <Month>, <year>

DOI: xxx/xxxx

EMPIRICAL

A Practical Approach for Technical Debt Prioritization based on Class-Level Forecasting

Dimitrios Tsoukalas[1,2] | Miltiadis Siavvas[1] | Dionysios Kehagias[1] | Apostolos Ampatzoglou[2] | Alexander Chatzigeorgiou[2]

[1]Information Technology Institute, Centre for Research and Technology Hellas, Thessaloniki, Greece

[2]Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece

Correspondence

*Dimitrios Tsoukalas, Information Technology Institute, Centre for Research and Technology Hellas, Thessaloniki, Greece and Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece. Email: tsoukj@iti.gr

Abstract

Monitoring Technical Debt (TD) is considered highly important for software companies, as it provides valuable information on the effort required to repay TD and in turn maintain the system. When it comes to TD repayment however, developers are often overwhelmed with a large volume of TD liabilities that they need to fix, rendering the procedure effort demanding. Hence, prioritizing TD liabilities is of utmost importance for effective TD repayment. Existing approaches rely on the current TD state of the system; however, prioritization would be more efficient by also considering its future evolution. To this end, the present work proposes a practical approach for prioritization of TD liabilities by incorporating information retrieved from TD forecasting techniques, emphasizing on the class-level granularity to provide highly actionable results. Specifically, the proposed approach considers the change proneness and forecasted TD evolution of software artefacts and combines it with proper visualization techniques, to enable the early identification of classes that are more likely to become unmaintainable. To demonstrate and evaluate the approach, an empirical study is conducted on six real-world applications. The proposed approach is expected to facilitate developers better plan refactoring activities, in order to manage TD promptly and avoid unforeseen situations long-term.

KEYWORDS:

technical debt, technical debt forecasting, technical debt prioritization, technical debt repayment

1 | INTRODUCTION

Technical Debt (TD)[1] is commonly used to indicate quality compromises that can yield short-term benefits in the software development process, but may negatively affect the long-term quality of software. In a software affected by TD, refactoring is the only effective way to reduce it on existing source code. However, companies usually cannot afford to repay all the TD that is generated continuously[2]. In fact, TD issues need to be translated into economic consequences before choosing the TD items that should be removed. In addition to this, strict production deadlines often force companies to focus on the delivery of new functionality, reducing the time that they can invest on TD repayment activities, leading to the accumulation of TD. Therefore, before applying refactoring activities, it is necessary to identify which items should be resolved first, by prioritizing them based on their TD values, but also based on stakeholders' objectives and preferences[3].

[0]**Abbreviations:** TD, Technical Debt; ML, Machine Learning;

Prioritizing TD liabilities is of utmost importance for effective TD repayment. This fact stresses the need for methods and tools that would provide companies with insights regarding where and when to apply refactoring activities. Therefore, what the stakeholders require is a decision-support system (DSS) to help them make such choices and support long-term effective TD repayment. Existing prioritization approaches rely on the current state of the system (TD principal/interest). However, a decision regarding whether to repay or not a TD item has different consequences depending on when it is made. For instance, the cost of refactoring a component in the current release is different than the cost of refactoring in a future release[4]; a class that might have non-alerting value of TD principal in the current version, might accumulate a large amount of TD in subsequent versions. Therefore, a DSS that would also take into account the future TD evolution by incorporating TD forecasting techniques could prioritize TD items not only based on their current TD, but also based on their potential future TD accumulation. To explore such an opportunity, the current state-of-the-art lacks TD forecasting approaches at the class level, since most studies focus on the project level[5,6].

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

In order to demonstrate our approach, we conducted an empirical study on six real-world open-source Java applications retrieved from GitHub[1]. During the first step of the approach, we retrieved the artifacts (i.e., classes) of each software application with their history (past commits). Next, for each application, we analyzed the associated artifacts using two static analysis platforms in order to calculate several TD measures for the construction of the initial class-level datasets. Afterwards, we applied a change proneness analysis to detect the most change-prone classes of each application in terms of size and TD. We filtered out the classes that were not suitable for the construction of forecasting models and ranked the remaining classes based on their calculated change proneness. Then, we used the remaining data to build class-level TD forecasting models and assess the TD evolution of the selected classes. Finally, we used visualization techniques to facilitate the understandability of the results and, in turn, the decision-making tasks of the developers and project managers regarding the repayment of the identified TD liabilities. To provide confidence that proposed prioritization approach captures the actual criticality of the project's classes from a TD viewpoint, we performed a qualitative evaluation using the Jira issue tracker.

The rest of the paper is structured as follows: Section 2 presents the related work in the field of TD forecasting and its potential effect on repayment activities. Section 3 thoroughly describes the proposed methodology, while Section 4 presents the empirical validation of the methodology, using six real-world applications as case studies. Section 5 reports the limitations and validity threats of this empirical study. Finally, Section 6 concludes the paper and discusses ideas for future work.

2 | RELATED WORK

According to Lehman's laws of software evolution[7], software systems must evolve over time or they will become irrelevant. The multitude of models that are available in the literature for predicting the evolution of specific quality attributes and properties that are directly or indirectly related to the TD of a software system reveal the importance of quality prediction and forecasting in the software engineering community. For instance, in their work Wagner[8] and Van Koten et al.[9] implement Bayesian Belief Networks for predicting the Maintainability of a software application, while Zhou et al.[10] approach the same problem by using multivariate adaptive regression splines. Similarly, to the high-level quality attributes, a large number of methods have been proposed to estimate the future evolution of software quality properties, such as code smells[11], fault-proneness[12,13,14], future changes[15,16], and software defects[17,18].

Since TD is an indicator of software quality (with an emphasis on maintainability), predicting its future value is considered equally important. While the previously mentioned studies indicate that there has been extensive research with respect to predicting the evolution of quality attributes and properties, directly or indirectly related to TD, only a few contributions exist so far regarding TD forecasting[6,5,19], indicating that it is a scarcely investigated field. The need for forecasting the evolution of TD

[1] https://github.com/

has been highlighted by a relatively recent study[20], in which the authors raise the awareness of the gap in the field of TD Management. 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 by facilitating TD Management activities.

In an initial study towards this challenge[6], the authors investigate statistical time series models for system-level TD forecasting and conclude that their approach can yield reliable short-term predictions for each of the five projects studied. They do observe, however, that the models' accuracy drops significantly for long-term forecasting horizons (more than 8 steps ahead). In an attempt to extend their previous work and introduce a more holistic TD forecasting methodology, the same authors conduct a follow-up study[5], where they examine the ability of more sophisticated ML methods to predict the system-level TD evolution of 15 open-source projects. The results reveal that the proposed ML approaches are suitable and able to provide accurate predictions. More specifically, their findings showcase that a nonlinear Random Forest regression can achieve sufficient accuracy for even longer forecasting horizons (up to 40 steps ahead).

Among the various TD Management activities, TD Prioritization is considered one of the most important. The TD prioritization process is used for scheduling of planned refactoring initiatives, by ranking identified TD items according to certain predefined rules to support decisions regarding which TD items should be repaid first and which TD items can be tolerated until later releases. Several different prioritization approaches and methods have been proposed by researchers on how to prioritize TD[21]. Regarding code TD, prioritization is mostly based on code smells[22,23]. In addition, other metrics such as time[24], cost to fix a violation[25], and quality rules[26] have also been considered. However, while all previous research works investigate the feasibility of prioritization and repayment of TD liabilities based on historical TD data, to the best of our knowledge, there are no research endeavors considering the future evolution of TD to address this issue.

Under those circumstances, being able to forecast the TD evolution of a software system is of great significance for TD prioritization and repayment activities. Such a work would enable the early identification of classes that are more likely to become unmaintainable in the future, and therefore allow project managers and developers plan precise payback strategies. Hence, an interesting topic is to investigate the possibility of applying TD forecasting techniques to lower levels of granularity of a software project (e.g., package, or class level). This would enable the prioritization and ranking of a software project's specific level components based on predictions of their future TD values to facilitate efficient identification of the aspects that might cause potential TD accumulation. In this way, TD items with the highest impact on the overall TD can be identified quickly and easily.

3 | APPROACH AND METHODOLOGY

As already mentioned, companies will probably not be able to repay all the accumulated TD of a software at a certain point in time. Furthermore, additional factors that have immediate economic impact like client demands and market adaptation might force managers to allocate resources on them, leaving a small portion of developer workforce effort to be utilized for maintenance activities. Therefore, the significance of prioritizing which software components to refactor is highlighted even further.

The TD prioritization approach proposed in the present paper is partially inspired by a related study introduced by Guo et al.[27]. According to their methodology, TD artifacts are initially identified and their respective TD measures are calculated. Subsequently, based on these measures, a decision on which TD items should be repaid or ignored is reached. However, the practical approach proposed in the present work differs from the aforementioned methodology, since the incorporation of TD forecasting techniques enables the prioritization of TD liabilities of the analyzed software not only based on their current TD, but also based on their potential future TD accumulation. In addition, it gives emphasis on class-level granularity, to provide highly actionable results. The proposed approach for TD liabilities prioritization is summarized in Figure 1 .

As can be seen in Figure 1 , our approach comprises five steps:

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

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

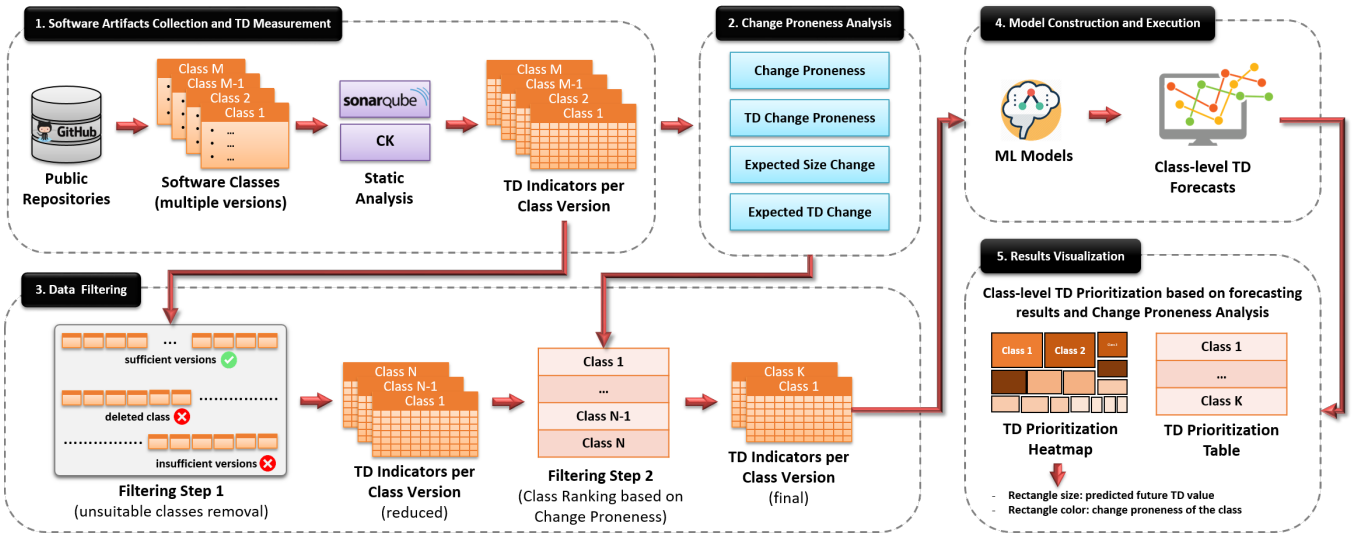


FIGURE 1 The proposed approach for prioritizing TD liabilities at class-level granularity leveraging TD forecasting techniques

3. **Data Filtering**. During the third step, a two-step filtering process is applied on the dataset. First, it filters out the classes that are not suitable for the next steps of the analysis (e.g., due to insufficient version history for the construction of forecasting models). Second, it ranks the classes based on their calculated change proneness, allowing in that way the stakeholders to focus only on a desired number of classes that are interesting from a TD viewpoint (i.e., they are modified frequently).

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

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

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

3.1 | Software Artifacts Collection and TD Measurement

As mentioned previously, this step is responsible for the collection of all the artifacts of a selected software application along with their history, as well as for the calculation of important artifact-level TD indicators. Since the proposed approach operates at class-level granularity, the type of artifacts that we emphasize on are software classes. The reasoning behind the selection of this type of artifacts is that class-level granularity usually leads to predictors with more practical results[28]. In fact, the higher the level of granularity, the lower the practicality of the produced results, since the focus of the developers is not pointed to an actionable subset of TD liabilities. Software classes are the main components that developers work on while designing and implementing a system, whereas the number of the liabilities that they contain is usually manageable. At this point, it should be also noted that the history (or at least an adequate number of past instances of each class) is also required for the construction of the class-level TD forecasting models that are produced in Step 4 of the approach (see Figure 1), as well as for the final prioritization of TD liabilities.

After retrieving all the classes of the selected software application (including their history), the *TD Principal* of each class is calculated. TD Principal is quantified, in most of the approaches that exist in the literature, by summing up the estimated effort to fix each individual inefficiency that is identified through automated analysis tools[29]. In this study, SonarQube[2] (v7.9, 2019), i.e., a popular open source platform for static code analysis and continuous inspection of code quality, is used as a proof of concept for the calculation of the *TD Principal*, since according to various studies[30,31], it is the most frequently used tool for

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

estimating and monitoring TD. Due to the adoption of SonarQube, the *TD Principal* of each class is actually the summation of the *Reliability Remediation Effort* (i.e., sum of remediation effort of bugs), the *Security Remediation Effort* (i.e., sum of remediation effort of vulnerabilities), and the *TD Remediation Effort* (i.e., sum of remediation effort of code smells). However, it should be noted that the proposed approach is platform-agnostic, and therefore the *TD Principal* can be calculated based on a different TD platform (or tool) of choice that meets the requirements of our approach (e.g., the quantification of TD Principal as the estimated remediation effort expressed in a measurable unit, such as minutes, hours, cost, etc.).

In the approach presented in this paper, in addition to *TD Principal*, we also considered that various TD indicators (capable of acting as TD predictors) should be included as input to the TD forecasting models, in order to enhance their predictive performance. In the literature, OO metrics, code smells, issues extracted from ASA tools, and software quality metrics extracted from quality assessment tools have been widely used as indicators able to monitor and quantify TD and the quality of software maintainability in general[5,30,32]. Most of the proposed TD indicators are related to software metrics[30,32], that allow the assessment of attributes, features, or characteristics of software artefacts. More specifically, in the context of object-oriented (OO) programming, various sets of metrics, such as the metric suit proposed by Chidamber and Kemerer (C&K)[28], make it possible to characterize the size, complexity, coupling and cohesion of the code among others. To this end, C&K metrics have been intensively studied and used for their ability to predict maintainability and maintenance effort[33], 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[32,34]. 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[35]. These problems may impede the software maintenance process and impose the need for code refactoring.

Since SonarQube calculates *TD Principal* based on different issue categories, similarly to our recent relevant study[5], we opted for the TD-related metrics that are provided by this tool as our primary *TD Principal* predictors. Subsequently, and again in line with our previous work[5], to complement the TD predictor set we decided to account also for the popular C&K metrics[28]. For this purpose, we chose the popular CK tool[36] (v0.6.3, 2020). CK allows the calculation of class-level metrics in Java projects by means of static analysis, and can be used to compute various OO metrics, such as CBO, DIT, and LCOM.

The combination of SonarQube and CK tools results in a set of 16 metrics in total. For reasons of brevity, the full list of selected TD indicators is located at the online supporting material[37]. However, providing a model with such an input may result in what is called the "curse of dimensionality", which is the unlikely event of a significant drop in the model's predictive performance when a large dimension of variables is provided as input. Consequently, features that are not (or are slightly) associated with the target variable, i.e., *TD Principal*, should be filtered out before any model construction attempt. To identify the most important TD indicators that could act as strong predictors for TD forecasting, our previous study[5] introduced an extensive feature selection analysis, including correlation, univariate and multivariate regression analysis. Among the initial metrics (TD indicators) under investigation, four of them were found to have statistically significant effects on TD, namely: (i) *bugs*, (ii) *code smells*, (iii) *lines of duplicate code*, and (iv) *afferent coupling (Ca)*. Therefore, the same TD indicators will be used as independent variables also in this study, during the next steps of the methodology.

In the context of the proposed approach, these metrics are calculated for each version of the classes of a selected software application. By "version" we refer to a past instance of a class, i.e., a previous snapshot of a class as part of a past commit of the respective software application. The interval in which the past versions of a class are collected (e.g., daily, weekly, or monthly time distance) depends on the commit frequency of an application, as well as on user preference, since it affects the time distance between the generated TD forecasts. Therefore, it can be adapted to the point of analysis that fits the needs of a particular company. Within the context of our empirical study presented in Section 4, we chose to collect versions at weekly intervals - more specifically we opted for the last commit in every analyzed week - as we believe it is a more viable solution. The rationale behind this choice is also presented in Section 4.1. Ultimately, this step results in a long dataset of classes with their associated TD-related measurements, which render valuable sources for the construction of class-level TD forecasting models.

3.2 | Change Proneness Analysis

This step is responsible for assessing the change proneness of the classes of a software application. The change proneness of a given class is of high interest from a TD viewpoint, since the frequent changes that are applied to a class increase the probability of TD accumulation, mainly due to the potential introduction of quick fixes, bugs, vulnerabilities, and code smells[38]. Several studies have shown that change-prone artifacts are more likely to contain important bugs and vulnerabilities[39,40].

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

Hence, we decided to consider the change proneness of the classes in the filtering process of the proposed approach, and, in turn, in the final prioritization of the TD liabilities. More specifically, as will be discussed in Section 3.3, we included the results of the change proneness analysis to apply a second filtering process with the purpose to remove classes that are not so change prone either in terms of Lines of Code (LoC), or in terms of their TD values.

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

$$CP = \frac{n_{altered}}{n_{total}} \tag{1}$$

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

$$CP_{TD} = \frac{n_{TDaltered}}{n_{total}} \tag{2}$$

Apart from the probability of change, we are also interested in knowing how much the LoC and the TD of a given class change on average between versions. These values are useful for TD repayment planning since they provide an estimate of the magnitude of change that is expected to be observed in the next version of a given class. To this end, we define the *Expected Size Change* ($E[D_{LOC}]$) of a class as the average change in its size (expressed in LoC) that is observed between two sequential versions. This metric is given by the following formula:

$$E[D_{LOC}] = \frac{\sum D_{LOCi}}{n_{total} - 1} \tag{3}$$

where:

- D_{LOCi}: The total LoC of the class that changed between version i and version $i - 1$
- n_{total}: The total number of versions of the selected class

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

$$E[D_{TD}] = \frac{\sum D_{TDi}}{n_{total} - 1} \tag{4}$$

where:

- D_{TDi}: The total TD of the class that changed between version i and version $i - 1$
- n_{total}: The total number of versions of the selected class

At this point it should be noted that although the latter two metrics are not used directly by the proposed approach for the filtering of the selected classes (see Section 3.3), reporting these values can be proved very useful for facilitating decision making regarding the TD repayment planning. In fact, these values can actually supplement the results of the proposed approach, in order to help the developers and project managers make more informed decisions.

3.3 | Data Filtering

This step is responsible for preparing the final dataset that will be used for the construction of the class-level TD forecasting models in the next step of the proposed approach (see Section 3.4). More specifically, the *Data Filtering* step is responsible

for (i) removing classes that are not suitable for constructing TD forecasting models, and (ii) keeping the classes that are more interesting from a TD viewpoint. Hence, a two-step approach is adopted for filtering the classes of a selected software application.

The first step of the *Data Filtering* approach is responsible for removing classes that are not suitable for the construction of TD forecasting models. First of all, classes that do not have sufficient version history (i.e., a sufficient number of past commits) are removed from the analysis, since training forecasting models for predicting the TD evolution of individual classes requires a substantial number of past instances. More specifically, in our approach, we exclude classes that the number of their past versions is below a specific threshold. This threshold is defined in a heuristic manner by taking into account the specific characteristics of the corresponding application under analysis. More details on that are provided in the empirical study in Section 4. Apart from the version history, the proposed approach also eliminates classes that are not present in the latest version of the software application. This is reasonable since classes that no longer exist in the code base of the application are of no interest for the developers.

The second step of the *Data Filtering* process is responsible for identifying and keeping classes that are more interesting from a TD viewpoint. As already mentioned, classes that are modified frequently are more likely to affect the future value of the TD of the corresponding software application, as these source code modifications normally lead to an alteration (either positive or negative) of the class's TD. Hence, the results of the *Change Proneness Analysis* (described in Section 3.2) are exploited, for the final selection of the classes that will be used for the production of class-level TD forecasting models.

More specifically, the final classes that passed the first step of the *Data Filtering* process are ranked based on their *Change Proneness* (CP) metric (see Section 3.2) in a descending order. Subsequently, the top N classes are selected to be part of the final dataset that will be used for the construction of the class-level TD forecasting models. The value of N is defined by the user (e.g., developer) based on the number of classes that he/she would like to have TD forecasts for. By determining the number of considered classes, a user can adjust the volume of information that they will be presented with. Actually, this feature acts as a filter (e.g., "show me the top-10 most change-prone classes", "top-100 most change-prone classes", etc.), which the developers could use to focus only on the number of classes that their company could afford to inspect, and potentially target for refactoring activities. It should be noted that instead of the CP metric, the *TD Change Proneness* (CP_{TD}) metric can be used as a measure of the class change proneness. Several empirical evaluations that we performed revealed that in the vast majority of the cases a statistically significant strong correlation exists between the two metrics, and therefore they can be used interchangeably for measuring change proneness. An example of this empirical evaluation is provided in the empirical study described in Section 4.

3.4 | Model Construction and Execution

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

The procedure that is adopted for the construction of the class-level TD forecasting models is as follows. For each one of the selected classes of a software application, the version history is retrieved along with the TD metrics that were computed in step 1 (see Section 3.1) of the overall process. Subsequently, due to ML models not directly supporting the notion of observations over time, the resulting class-specific dataset is restructured based on the "sliding window" approach[41], in order to be transformed in a format that is suitable for supervised ML tasks. In short, this method extends each initial sample (i.e., row) of the dataset by including, besides the current information, also past information (i.e., the TD indicator values of multiple prior commits) as inputs (X)) and future information (i.e., the TD value of a future commit as output (Y)) simultaneously into a single row. This approach is described in more detail below.

The number of past time steps that we want to include as input into each sample is called the "window width". As a first step, the width of the sliding window needs to be chosen. Window width, illustrated as a red box in Figure 2 , 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 Figure 2 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 Figure 2 , 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 at the bottom of Figure 2 . The process is repeated by shifting the two boxes simultaneously over the samples, one step at a time, creating new rows until

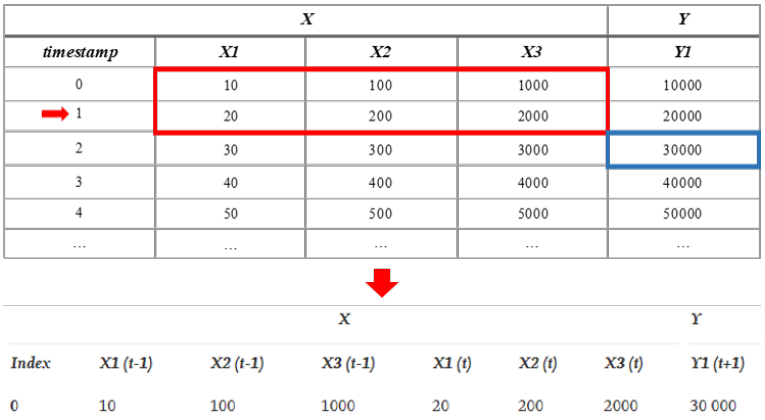


FIGURE 2 The sliding window method

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 "sliding window" approach has been effectively used for similar data restructuring tasks in our previous related studies[5,19].

Subsequently, several linear, non-linear, and ensemble ML models (e.g., Linear Regression, Support Vector Regression, Random Forest, etc.) are built and tested on the dataset for various forecasting steps ahead[3]. To better assess prediction accuracy of the produced models and similarly to our previous related studies[5,19,42], the Walk-forward Train-Test validation[43] is adopted. 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. The produced models are compared based on three different performance metrics, particularly the *Root Mean Square Error (RMSE)*, the *Mean Absolute Error (MAE)*, and the *Mean Absolute Percentage Error (MAPE)*. The model that demonstrates better values in these three metrics is selected as the TD forecasting model of the given class.

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

3.5 | Results Visualization

To better prioritize the TD repayment activities, the results of the proposed approach need to be properly visualized, so that the underlying information is effectively conveyed to the developers and project managers of the software application, assisting them in making more informed decisions regarding TD repayment. Several approaches for visualizing the produced results can be adopted. In the proposed approach, emphasis is given on heat maps. An example of such a heat map is depicted in Figure 3 .

As can be seen in Figure 3 , the heat map consists of a number of rectangles. Each rectangle corresponds to a specific class of the software application. The size (i.e., area) of the rectangle is proportional to the future value of the *TD Principal* of the class as reported by the associated class-level TD forecasting model. On the other hand, the color of the rectangle denotes the change proneness of the corresponding class. The greener the rectangle, the higher the probability to change in the next versions. Hence, the heat map allows the developers to take into account two different criteria for prioritizing their TD repayment activities, namely the future value of the TD Principal and the change proneness of the selected classes. For example, a class that is expected to have relatively higher *TD Principal* in the upcoming versions and that it is highly likely to change (e.g., Class 2 in Figure 3), may probably require immediate remediation actions compared to a class that changes less frequently (e.g., Class 1 in Figure 3), in order to avoid further TD accumulation.

It should be also noted that, apart from the heat map, a table comprising the detailed results of the analysis is considered necessary. This table should contain supplementary information including the additional metrics of the *Change Proneness*

[3] Since the commits in this study were collected in weekly intervals, steps actually refer to weeks - see Section 4.1

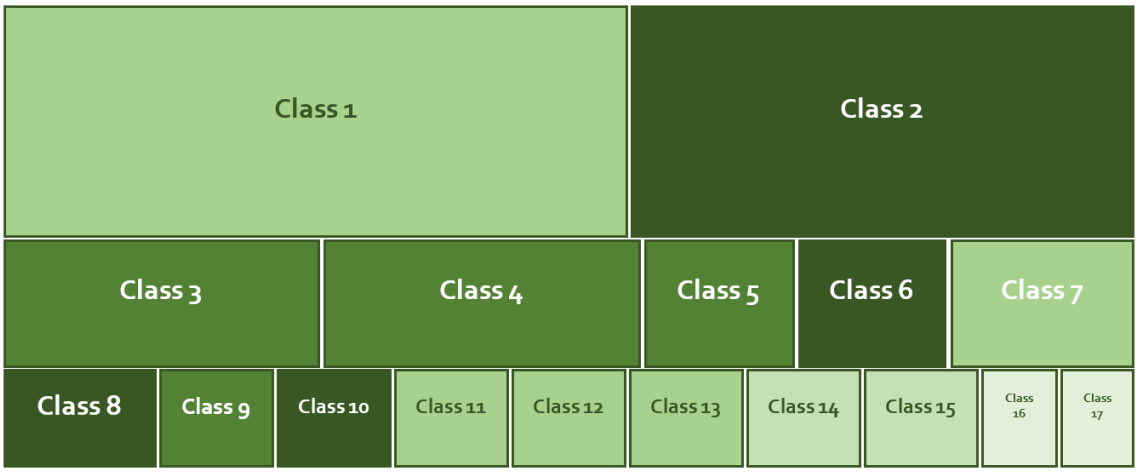


FIGURE 3 Heat map visualizing the future value of the TD Principal and the change proneness of the selected classes

Analysis that were defined in Section 3.2. This additional information is expected to help the developers take even more informed decisions regarding the prioritization of their TD repayment activities.

4 | EMPIRICAL STUDY

In this section, the proposed approach is demonstrated through an empirical study on six real-world open-source software applications. This study also acts as a test bed for evaluating the correctness of the proposed approach, and for assessing the feasibility of generating class-level TD forecasting models of sufficient predictive performance. For reasons of brevity, the steps of the approach presented below will focus mainly on the Apache Kafka software system. However, results and model comparisons will include also the rest of the applications.

4.1 | Data Collection, Analysis and Filtering

4.1.1 | Software Artifact Collection and TD Measurement

The data used in this study were obtained from six popular open source Java projects available on GitHub[4]. The selected six applications have different sizes and belong to different application domains, which range from Networking Software (e.g., Kafka, OKHttp) to Scientific Software (e.g., SystemML) and Utilities Software (e.g., Guava, Jenkins). The selection criteria were based on the software popularity, activity level, data availability, and the Java programming language. We selected only applications whose commit activity was frequent (at least once per week) and long-lived (at least 3 years). Table 1 presents the applications that were selected for constructing the codebase, along with additional information.

As a first step towards building our dataset, for each application in Table 1 , we collected 150 snapshots (commits) in weekly intervals, spanning up to almost 3 years of each system's evolution. To do so, we opted for the last commit in every analyzed week as the time point of analysis. Similarly to our previous related study[5], the rationale behind the choice to ensure fixed and weekly time intervals between the studied 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 or monthly intervals is a more viable solution as i) rarely do projects keep daily commits, and ii) monthly intervals would result in significantly fewer data and thus significantly lower forecasting performance.

Subsequently, as described in Section 3.1, we used both SonarQube and CK tools in order to statically analyse each snapshot of the six applications across their 150 timely-ordered commits. In total, $6 \times 150 = 900$ commits were analyzed. Since in this work we focus on software classes as our unit of analysis, our selected TD indicators that act as predictors, namely number of *bugs*, *code smells*, *lines of duplicate code*, and *Ca*, as well as the *TD Principal* were extracted at class-level, for each class across

[4]https://github.com

TABLE 1 Selected Java Applications

Application	Analysis Timeframe	Last Commit LoC	Total Analysed Class Instances	Unique Analysed Classes	Description
Apache Kafka	30/10/2015 - 07/09/2018	116.000	85.773	1.155	Kafka is a platform used for building realtime data pipelines and streaming apps.
Apache SystemML	02/10/2015 - 10/08/2018	200.000	220.617	4.445	SystemML provides an optimal workplace for machine learning using big data.
Apache Groovy	25/12/2015 - 02/11/2018	210.000	193.131	2.221	Groovy is a powerful, dynamic language, with static capabilities for the Java platform.
Google Guava	25/12/2015 - 02/11/2018	114.000	68.051	518	Guava is a set of libraries that includes graphs, utilities for I/O, hashing, string processing, etc.
JenkinsCI Jenkins	25/03/2016 - 01/02/2019	147.000	208.564	1.320	Jenkins is the leading open-source development workflow automation server.
Square OkHttp	18/12/2015 - 26/10/2018	24.000	23.125	241	OKHttp is an HTTP & HTTP/2 client for Android and Java applications.

the 150 commits of its associated project. Table 1 presents information regarding the number of classes that were analysed per application. Taking the Apache Kafka as an example, by statically analyzing each of the 150 commits we derived from the process 150 CSV files containing in total 85.773 analysed class instances, while the number of unique classes among these instances is 1.155. The above process resulted in a long dataset of classes and their TD-related measurements across their commit history, which render valuable sources of time series data for the construction of class-level TD forecasting models.

4.1.2 | Change Proneness Analysis

After acquiring the dataset of multiple class instances with their TD-related measurements, the next step is to apply a *Change Proneness Analysis* in order to detect the most change-prone classes of the given software applications. This process is of high interest not only from a TD viewpoint, but also for selecting the classes that are more suitable for the construction of class-level forecasting models. By following the *Change Proneness Analysis* process described in Section 3.2, we computed the 4 metrics of interest, namely *Change Proneness* (CP), *TD Change Proneness* (CP_{TD}), *Expected Size Change* ($E[D_{LOC}]$), and *Expected TD Change* ($E[D_{TD}]$) for each unique class of the six selected software applications.

As already mentioned, these metrics are very important for the *Data Filtering* step of the overall approach. More specifically, the CP and CP_{TD} metrics are actually used as the basis for selecting the main classes of interest, which are subsequently used for the construction of class-level TD forecasting models. Apart from the *Data Filtering* step, these four metrics provide additional useful information to the developers and project managers of the software application, allowing them to reach more informed decisions regarding the TD repayment activities. In addition, the aforementioned metrics helped us decide whether to analyze a particular class or not, thus deterring from wasting computational power and man-hours. This will be further explained below.

4.1.3 | Data Filtering

The *Data Filtering* step is responsible for preparing the final dataset that will be used for the construction of the class-level TD forecasting models. As mentioned in Section 3.3, the first part of this step involves the removal of classes that are not suitable for the construction of TD forecasting models, i.e., they do not have sufficient version history. In our approach, since 150 snapshots (i.e., commits) of each application were available, we decided to set the threshold to 100 versions. This number was computed in a heuristic manner, after applying dedicated experiments (within the present study and our previous related studies[5,19,42]) in order to assess what would be the minimum number of samples that would result in an acceptable forecasting error, when given as input into various ML forecasting algorithms. Taking Apache Kafka as an example, 764 out of 1.155 classes were filtered out as a result of applying this filtering process, leaving us with 391 unique classes. Apart from classes with insufficient number of

version history, we also eliminated classes that were not present in the latest version of the software application (i.e., 7/9/2018), as they no longer exist in the code base of the selected software application. This extra filtering step removed another 20 classes, leaving us with 371 classes for further analysis.

The second part of the *Data Filtering* step involves identifying and keeping classes that are modified frequently and therefore are more interesting from a TD viewpoint. Hence, we exploited the results of *Change Proneness Analysis* for the remaining 371 classes that passed the first filtering step, by ranking them based on their *Change Proneness* (CP) metric in a descending order so as to focus on the classes that are more likely to affect the future *TD Principal* value. As regards the Apache Kafka application, an indicative number of 10 classes along with their computed *Change Proneness Analysis* metrics ranked by *Change Proneness* (CP) in a descending order are presented in Table 2 . The complete ranked set of the 371 classes can be found at the online supporting material[37]. Supporting material also presents the *Data Filtering* step results for the rest of the software applications under examination.

TABLE 2 Change Proneness Analysis metrics for first 10 classes of the Apache Kafka ranked by CP

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

It should be noted that alternatively, the CP_{TD} could have been used as the measure of the class's change proneness, and, in turn, as the basis for the ranking. However, by inspecting Table 2 , we can observe that a correlation may exists between *Change Proneness* (CP) and *TD Change Proneness* (CP_{TD}) metrics. In order to reach safer conclusions, formal statistical testing was applied. More specifically, we ranked the selected 371 classes of Apache Kafka based on the CP and CP_{TD} metrics, leading to the generation of two individual rankings. Subsequently we compared the two resulting rankings in order to determine whether a statistically significant and strong positive correlation exists. For this purpose, we defined the following Null Hypothesis (H_0), along with its corresponding alternative hypothesis (H_1), and tested it in the 95% confidence interval:

- $\mathbf{H_0}$: No statistically significant correlation exists between the two rankings
- $\mathbf{H_1}$: A statistically significant correlation exists between the two rankings

In order to test the Null Hypothesis the *Spearman rank correlation coefficient* (ρ) was used, which is a non-parametric test, not affected by outliers. The calculated ρ was found to be 0.85, which is a positive and strong (according to Cohen et al.[44]) correlation. The *p-value* was found to be 0.0048, which is lower than the threshold of 0.05, which led us to the rejection of the Null Hypothesis, and, thus to the acceptance of the alternative hypothesis. As a result, we can conclude that a statistically significant positive and strong correlation exists between the rankings of CP and CP_{TD}, at least for the selected dataset.

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

Qualitative Evaluation of TD Prioritization Approach

This paper essentially proposes a practical approach that aims to facilitate refactoring activities planning by providing a fine-grained prioritization of TD liabilities. Therefore, a qualitative evaluation of its usefulness in practice (i.e., to investigate whether

the selected classes are actually critical from a TD viewpoint) would normally require seeking feedback from practitioners, that is, developers of the six investigated applications, in order to ask their opinion on how relevant is the proposed ranking of critical classes to the actual effort and, in turn, the cost that is required for maintaining and extending these classes. However, developers of open-source projects are difficult to reach and are usually unresponsive.

To overcome this obstacle, we decided to evaluate our approach through a proxy that would give us an insight into the actual development workflow. More specifically, we exploited the Jira[5] issue tracking system of the Apache Kafka with the purpose of investigating whether the specific ranking of classes as suggested by our approach is in line with the actual ranking of classes that were indeed critical for the developers, based on the reported fault information. In brief, we measured the criticality of a given class based on how many times it has been reported in the project's Jira issue tracker. To do so, we performed queries to the Jira API and we fetched the total number of issues related to each of the 371 classes through the investigated 3 years of the system's evolution (i.e., from 30/10/2015 to 7/9/2018). This information can be found online[37]. Subsequently, we ranked the selected 371 Apache Kafka classes based on the total number of Jira issues and compared the resulting ranking with the ranking based on the CP metric in order to determine whether a statistically significant and strong positive correlation exists. We defined the Null (H_0) and alternative hypothesis (H_1), and tested it in the 95% confidence interval.

Again, to test the Null Hypothesis we used the *Spearman rank correlation coefficient* (ρ). The calculated ρ was found to be 0.741, which is a positive and strong (according to Cohen et al.[44]) correlation. The *p-value* was found to be 4.32e-104, a value significantly lower than the threshold of 0.05, which led us to the rejection of the null hypothesis, and thus, to assume that a statistically significant positive and strong correlation exists between the rankings of Jira issues and CP between the 371 classes.

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

4.2 | Model Construction and Execution

The *Model Construction and Execution* step is responsible for the construction and execution of a class-level TD forecasting model for each one of the classes of the examined datasets (i.e., software applications) in order to retrieve class-level *TD Principal* forecasts. For reasons of brevity, we decided to focus on the top 10 classes of the six selected software applications in terms of CP metric as part of the datasets used for the construction of the class-level TD forecasting models. A snapshot of these datasets, including TD-relevant metrics for the selected 10 classes of each application can be found online[37]. We believe that the selected number of classes (10 per application - 60 in total) is sufficient for evaluating the correctness of the proposed approach (i.e., the feasibility of constructing class-level TD forecasting models), as well as for demonstrating the overall usefulness of the proposed TD prioritization methodology. However, the reader can easily replicate the present analysis using a much larger number of software classes, depending on their preferences. In addition, in case that the proposed approach is incorporated by a dedicated tool, the user may be equipped with the option to manually define the number of classes for which they would like to have TD forecasts (e.g., as shown later in Figure 6).

4.2.1 | Model Construction

As an initial step towards constructing the class-level TD forecasting models, we retrieved the version history (i.e., commits collected in weekly intervals) for each of the selected classes of each application, along with their *TD Principal* values and TD metrics that act as predictors, extracted during *Software Artifact Collection and TD Measurement* step of the overall process (see Section 3.1). Subsequently, due to ML models not directly supporting the notion of observations over time, we restructured each class-specific time series dataset using the "sliding window" method (see Section 3.4). Similarly to our previous related study[5], we found out that choosing a sliding window of size = 2 across different models resulted in the minimum *MAPE* when trying to forecast for 5 steps ahead. This means that two past lag values plus the current lag will be used to forecast future values. Respectively, for longer forecasting horizons (e.g., 10 steps ahead), a larger window appeared to be more suitable and resulted in better model performance.

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

After performing the above data restructuring process, we split each class-specific dataset into training and test sets. In particular, to assess model prediction accuracy and at the same time respect the temporal order of our class-level time series data, we adopted the Walk-forward Train-Test validation (see Section 3.4). As an example, most of the selected Apache Kafka classes consist of 150 observations (i.e., commits). For Walk-forward Train-Test validation we chose the number of splits = 5, meaning that training set will start from 25 samples and will expand up to 125 samples during the last iteration. Test set will constantly contain 25 observations.

Subsequently, a set of Causal and ML models, namely Multiple Linear regression (MLR), Ridge and Lasso regression, Support Vector regression (SVR) with both linear and Gaussian kernel, and Random Forest regression were selected for a class-level evaluation. Most of these models have been extensively compared and evaluated in the literature for their ability to predict important software attributes[5,45,33,46,47,48]. In order to tune selected models in the best possible way, we used the Grid-search method[49]. Grid-search is commonly used to find the optimal hyper-parameters of a model that result in the most accurate predictions, by performing an exhaustive search over specified parameter values. These models were built on each class-level training set, and then tested on the respective test set. To test their predictive performance for different future horizons, we repeated the Walk-forward Train-Test validation process three times, where predictions were made for the next n+1, n+5, and n+10 future steps (weeks) respectively. The best parameters per model, as selected during the hyper-parameter tuning process described above, can be found online[37].

As mentioned in Section 3.4, the produced models were compared based on three different performance metrics, particularly the *MAPE*, the *RMSE*, and the *MAE*. The *MAPE* is a popular measure for forecast accuracy that uses absolute values to measure the size of the error in percentage terms. *RMSE* and *MAE* are also widely used in forecasting to express average model prediction error in units of the variable of interest.

As an example, we will illustrate the detailed results of training TD forecasting models on the StreamThread.java class, which was found to be the most change-prone class of the Apache Kafka project. The StreamThread.java class dataset is comprised of 150 versions (i.e., commits collected in weekly intervals). In Table 3 , we report a comparison of prediction errors of the regression models trained on the StreamThread.java dataset for multiple (1, 5 and 10) time steps (weeks) into the future. Prediction errors in each cell of the table are averaged values of the testing errors for all train-test splits that were performed during Walk-forward Train-Test validation. Prediction errors indicated in bold are averaged values of the specific models that were created for each week-ahead prediction category.

As far as shorter forecasting horizons are concerned (i.e., 1 week ahead), it is clearly depicted in Table 3 that linear models, such as Multivariate, Lasso, Ridge Regression, and SVR(linear), have generally lower *MAPE* values and outperform non-linear models, such as SVR(rbf) and Random Forest Regression. In fact, we observe that forecasting the TD of the StreamThread.java class for 1 step ahead (1 week) using Lasso regression gives a *MAPE* of 4.168%, while for the same horizon SVR with a Gaussian kernel gives 19.821% and Random Forest Regression gives 8.270%.

By observing Table 3 , we also notice that linear models that apply Regularization in order to prevent overfitting, i.e., Ridge and Lasso Regression, are good candidates even for the mid-term forecasting horizon category of 5 steps (weeks) ahead. In fact, Ridge Regression demonstrates the best performance with a *MAPE* of 11.212%. However, while most of the linear models' predictive power drops significantly as we try to forecast longer into the future, the non-linear Random Forest model seems to have an almost stable performance over the holdout sample for all steps ahead.

For longer horizons (i.e., 10 weeks ahead), linear models are in general performing equally average. However, the non-linear Random Forest model seems to perform significantly better than the other models (with Ridge Regression being the only exception). In fact, forecasts for 10 steps ahead using Random Forest give the lowest *MAPE* error (15.524%), as well as the lowest *MAE* and *RMSE* errors. These results were also verified during the model execution phase described in Section 4.2.2, where we observed that Random Forest regression is able to provide accurate forecasts that lie very close to ground truth, even for 10 weeks ahead.

To further examine the ability of the investigated algorithms to forecast *TD Principal* and get an understanding of how the models perform across different datasets, we repeated the same experiments for each of the top 10 classes (in terms of CP) of the six software applications under examination (60 classes in total). We will not go through each class of each application one by one, but instead we will provide averaged scores. Detailed prediction scores for each class across the six examined applications can be found online[37]. Figure 4 illustrates the *MAPE* of the forecasting models for the three forecasting horizon cases, i.e., 1 (in orange), 5 (in yellow) and 10 (in blue) time steps (weeks) into the future, averaging the 10 most change-prone classes for each one of the six applications. The average value of the three forecasting horizon cases is also depicted (in green).

TABLE 3 StreamThread.java TD predictions using Walk-forward Train-Test validation

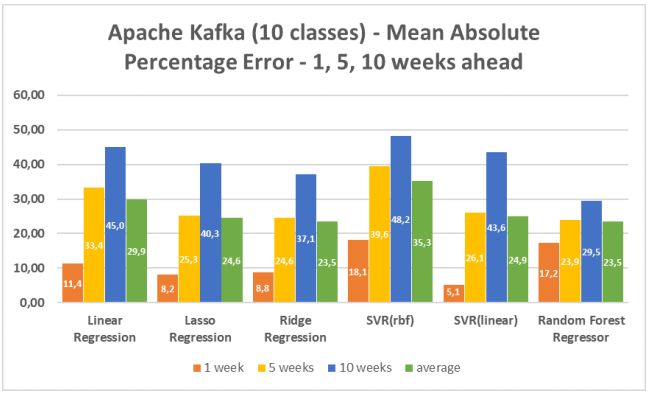
Model	Weeks ahead	MAE (mins)	RMSE (mins)	MAPE (%)
MLR	1	26.343	32.041	7.795
	5	61.473	66.236	17.786
	10	76.763	82.888	22.119
	Average	**54.860**	**60.388**	**15.900**
Lasso regressor	1	14.134	20.287	4.168
	5	45.438	51.163	13.203
	10	68.177	74.424	19.737
	Average	**42.583**	**48.625**	**12.369**
Ridge regressor	1	14.453	20.406	4.249
	5	38.152	44.244	11.212
	10	53.555	60.679	15.603
	Average	**35.387**	**40.518**	**10.355**
SVR regressor (linear)	1	20.023	27.879	6.018
	5	60.193	66.518	17.196
	10	90.241	99.637	25.843
	Average	**56.819**	**64.678**	**16.352**
SVR regressor (rbf)	1	67.630	79.692	19.821
	5	129.270	148.530	37.390
	10	108.331	134.356	30.286
	Average	**101.744**	**120.859**	**29.166**
Random Forest Regressor	1	27.759	34.180	8.270
	5	42.150	46.883	12.424
	10	52.388	59.118	15.524
	Average	**40.766**	**46.727**	**12.073**

By inspecting Figure 4 , we observe that the Random Forest Regressor provides generally lower average MAPE values compared to the rest of the models. By delving deeper into the models' prediction scores per forecasting horizon (averaged for the 10 classes across each of the six examined applications), we further observe that, similarly to the StreamThread.java class, for shorter forecasting lengths, linear models that apply Regularization, such as Lasso and Ridge regression, demonstrate generally higher performance compared to the non-linear candidates. We also observe that again, while the predictive power of linear models drops as we forecast longer into the future, the non-linear Random Forest seems to have an almost stable performance in comparison to the other models.

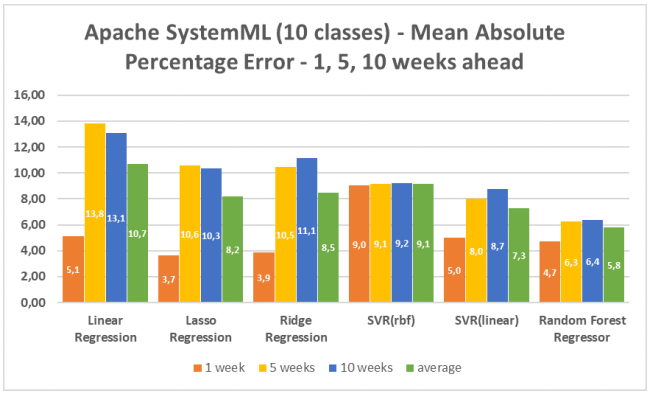
The general outcome of the *Model Construction* phase is that selection of a forecasting model really depends on the horizon that we want to forecast. By inspecting the results presented both in the paper and online, we conclude that for shorter horizons the best accuracy is usually presented on models that apply Regularization, i.e., Lasso and Ridge regression. However, when it comes to longer horizons the Random Forest regression model is a better-performing candidate. These results are in line with the findings of our previous empirical study[5], and will be visually presented in the rest of this section.

4.2.2 | Model Execution

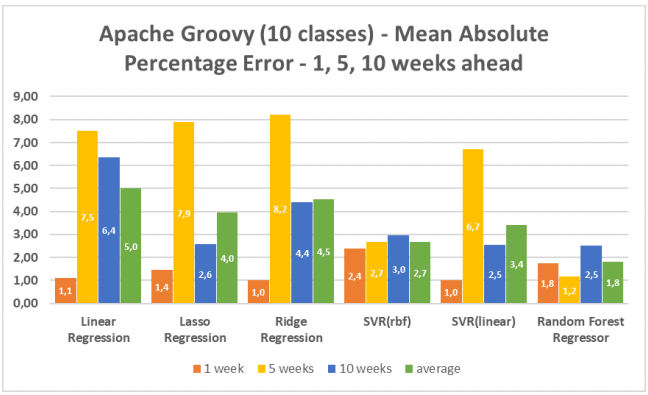
After constructing our models as described above, in this section we present the *Model Execution* phase. Towards executing our models for multi-step forecasts, we adopted the "Direct" approach, which means that a separate model is developed to forecast each forecast lead time. The main reason behind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



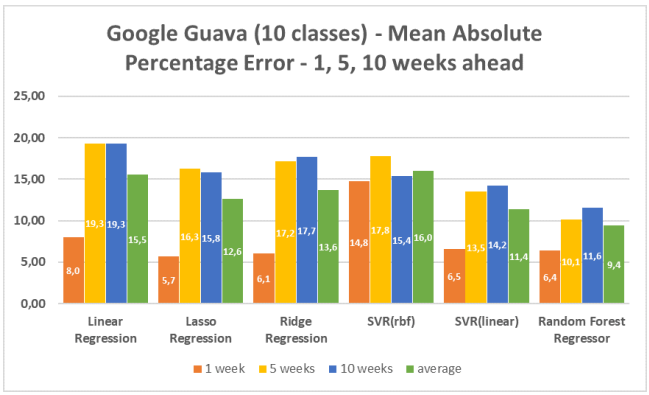
(a) Apache Kafka



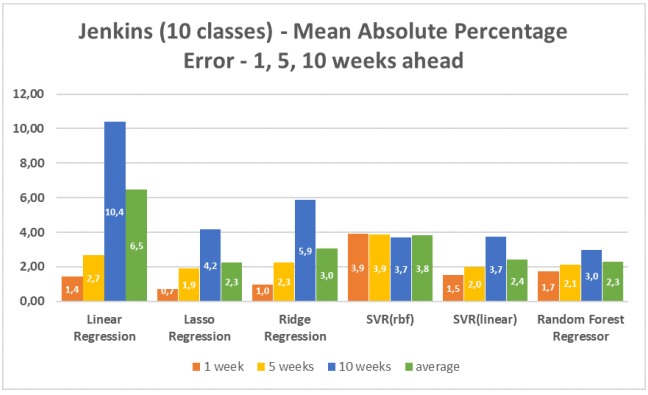
(b) Apache SystemML



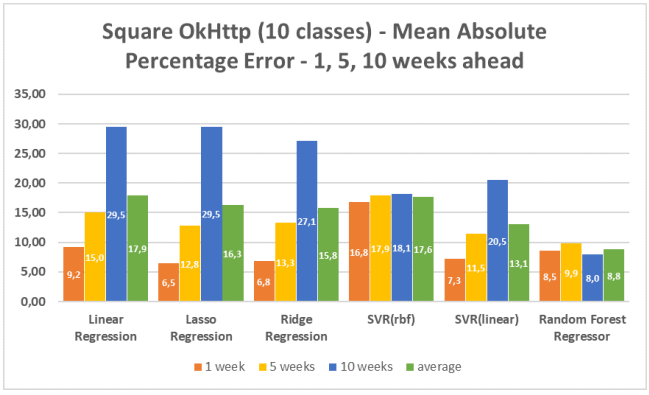
(c) Apache Groovy



(d) Google Guava



(e) Jenkins



(f) Square OkHttp

FIGURE 4 Six projects (top 10 classes) TD predictions – MAPE for 1 to 10 steps-ahead using Walk-forward Train-Test validation

In Figure 5 , we provide an example of forecasting the evolution of StreamThread.java class of the Apache Kafka application, for 10 versions ahead using Random Forest regression, which during the model construction phase was reported to perform better than the other examined models for longer forecasting horizons. The red line denotes the forecast, while the blue line denotes the ground truth. Behind the scenes, 10 models were executed, one for each specific horizon of interest (starting from 1 step to 10 steps), while their forecasted TD values where aggregated into a common vector, and then plotted as the projected TD evolution.

For reasons of brevity, forecasts for 10 versions ahead using Random Forest regression for the rest of the classes are available online[37]. As can be seen, similar observations can be made for the other nine classes of the Apache Kafka application. In

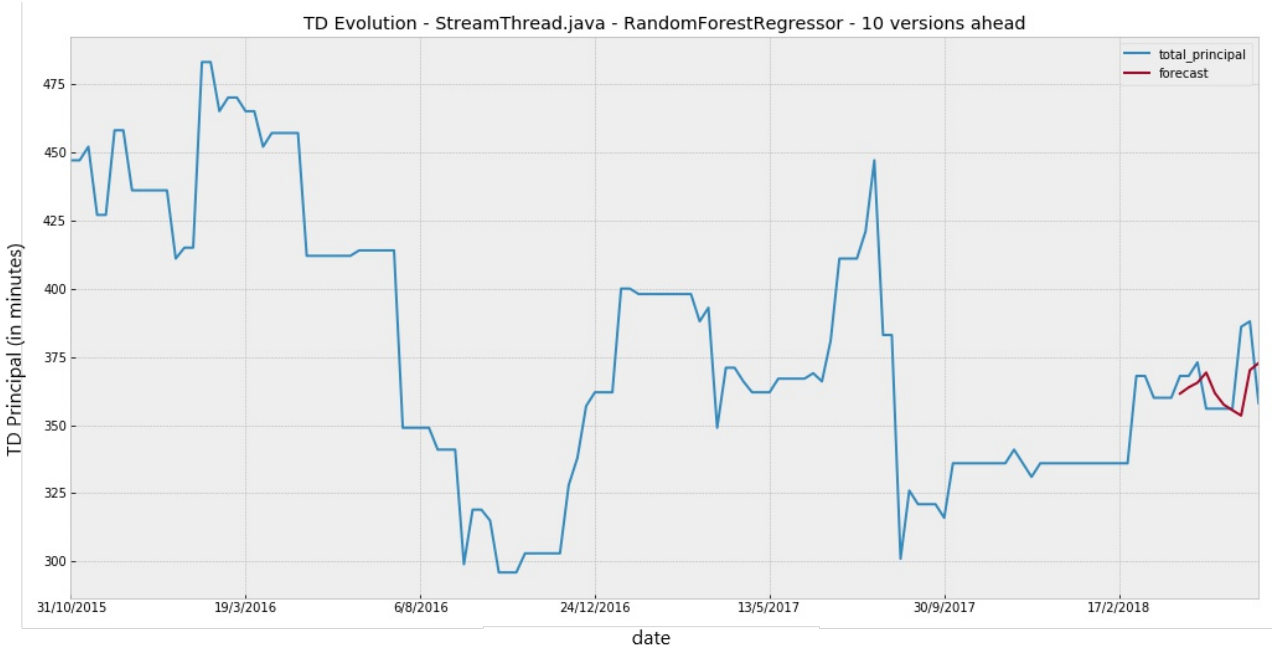


FIGURE 5 StreamThread.java TD forecasting for 10 steps (weeks) ahead using Random Forest

particular, the Random Forest regression seems to provide meaningful long-term forecasts for each one of the studied cases. 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 its future value is also captured with a sufficient level of accuracy.

4.3 | Results Visualization

As described in Section 3, the proposed approach introduces a more fine-grained prioritization of TD liabilities by incorporating information retrieved from TD forecasting techniques. To properly combine the two proposed criteria for prioritizing TD repayment activities, namely the future forecasted value of the *TD Principal* and the change proneness of the selected classes, a heat map is used as a means of visualization. Heat maps are easy to read and understand, since their underlying information is effectively conveyed to the developers and project managers of the software application, assisting them in making more informed decisions regarding TD repayment.

Figure 6 illustrates an indicative heat map for the Apache Kafka application, combining information retrieved from the analysis that we described above. In particular, the rectangles correspond to the specific 10 selected classes of the Apache Kafka application, as extracted and ranked (see Table 2) during the *Data Filtering* step of the methodology. The color of each rectangle denotes the change proneness of the corresponding class, while the size is proportional to the future value of the *TD Principal*, as reported by the class-level TD forecasting model. For the purpose of this indicative example, we have selected 4 steps-ahead (i.e., 1 month) as the horizon of the forecasts. However, users could easily change the forecasting horizon depending on the TD repayment strategy that they are interested in, by simply using a drop-down menu, as shown at the top-right of the indicative screen. In addition, users can also change the number of classes that they wish to inspect by using a drop-down menu, such as the one depicted below the heat map in Figure 6 .

An example illustrating the usefulness of the proposed method in enabling the early identification of classes that are more likely to become unmaintainable is the following: In the heat map depicted in Figure 6 , class StreamThread.java is expected to have relatively high *TD Principal* (big rectangle size) in the upcoming versions, whereas it is also highly likely to change (deep green color). On the other hand, while class KTableImpl.java is expected to have relatively higher *TD Principal* compared to StreamThread.java (bigger rectangle size), it is less likely to change (light green color). As a result, StreamThread.java needs to be prioritized higher than KTableImpl.java, as it probably requires immediate remediation actions in order to reduce the risk of its TD accumulation.

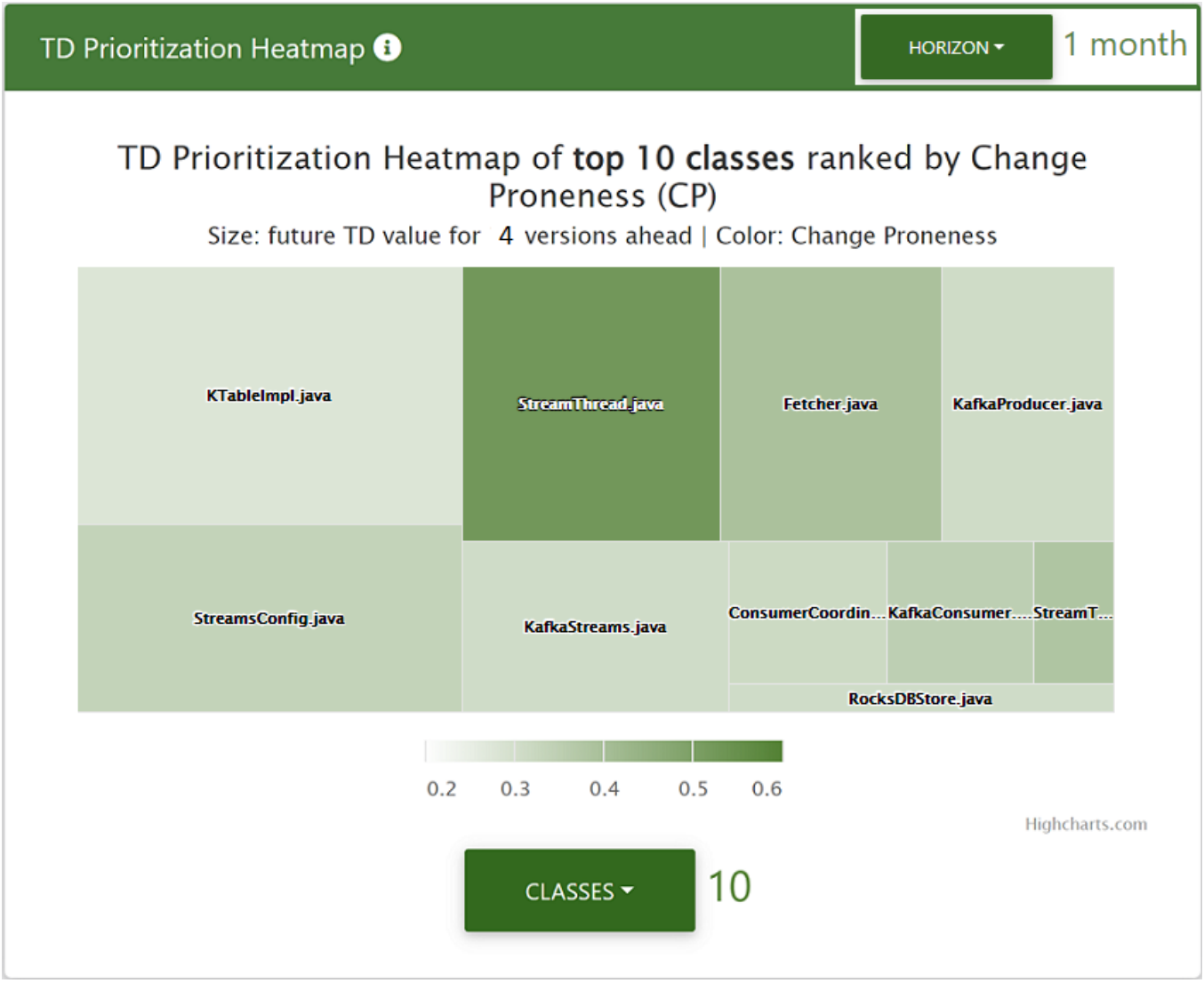


FIGURE 6 Heat map visualizing the future value of the TD Principal and the change proneness of the selected classes of Apache Kafka application

Apart from the heat map, an indicative complementary table comprising the detailed results of the analysis is presented in Figure 7 . This table contains supplementary information including the metrics that were computed during the Change Proneness analysis (e.g., CP, CP_{TD}, etc.), the forecasted class-level TD value, as well as the trend between the current TD value and the forecasted TD value, which can act as an indicator regarding whether TD of a specific class will increase or decrease. This additional information is expected to help the developers take even more informed decisions regarding the prioritization of their TD repayment activities.

To complement the analysis and further highlight the added value of the proposed TD Prioritization approach compared to approaches relying only on historical data, an additional practical scenario demonstrating how our approach enables the early identification of seemingly harmless but potentially "dangerous" classes is presented below. More specifically, we revisit the particular example presented above, but instead of focusing on 4 weeks ahead, we increase the forecasting horizon to 10 weeks ahead. The analysis results are depicted in the table of Figure 8 , containing TD forecasting and CP supplementary metrics of the selected ten classes of Apache Kafka application.

By inspecting the table, we point out again that class KTableImpl.java has a relatively higher current TD Principal value (358 minutes) compared to StreamThread.java (243 minutes). However, it can be seen that the future TD value of StreamThread.java is expected to not only increase in 10 weeks from now, but also surpass the TD value of KTableImpl.java. More specifically, we

TD Prioritization Table ⓘ HORIZON 1 month

Show entries 10 Search Search

Class Name	Change Proneness (CP)	TD Change Proneness (CPTD)	Expected Size Change (E[DLOC])	Expected TD Change (E[DTD])	Current TD Value	Forecasted TD Value	TD Trend %
StreamThread.java	0.53	0.39	3.14	-0.6	243	252.33	3.84
Fetcher.java	0.4	0.24	4.15	1.17	235	243.44	3.59
StreamTask.java	0.39	0.12	2.16	0.1	80	77.9	-2.62
KafkaConsumer.java	0.35	0.09	1.8	0.7	101	102.38	1.36
StreamsConfig.java	0.34	0.1	3.45	0.7	211	214.45	1.64
ConsumerCoordinator.java	0.32	0.13	1.73	0.53	140	116.32	-16.91
KafkaProducer.java	0.31	0.13	1.72	-0.84	206	203.72	-1.11
KafkaStreams.java	0.31	0.16	3.61	1.26	152	154.73	1.80
RocksDBStore.java	0.31	0.15	1.73	0.22	42	41.11	-2.12
KTableImpl.java	0.28	0.17	2.28	1.1	358	359.26	0.35

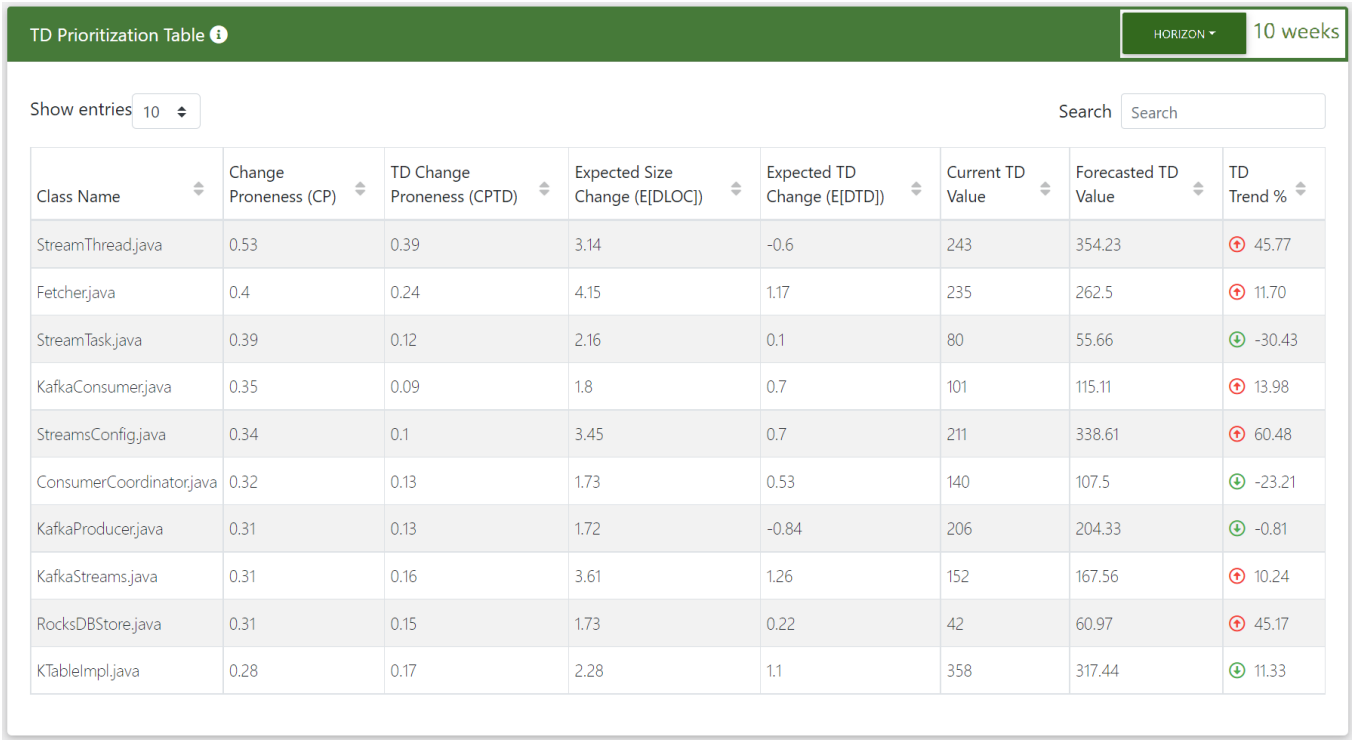
FIGURE 7 Table containing TD forecasting and CP supplementary metrics of the selected classes of Apache Kafka application - 4 steps-ahead forecasts

can observe that the forecasted TD of StreamThread.java is predicted to increase by ∼45% and reach the value of 354 minutes, while the forecasted TD of KTableImpl.java is predicted to decrease by ∼11% and reach the value of 317 minutes. This outcome is in line with the suggestions drawn from the previous example, where we had proposed that StreamThread.java needs to be prioritized higher than KTableImpl.java, as it probably requires immediate remediation actions (due to its high CP) in order to reduce the risk of its TD accumulation.

That being said, a "conventional" approach relying only on present and/or historical data for TD Prioritization would probably prioritize KTableImpl.java higher than StreamThread.java, judging only by their current TD values. On the other hand, our approach can foresee that StreamThread.java is more likely to become unmaintainable in the future and therefore should be prioritized higher. Similar suggestions can be also drawn by inspecting other classes. For instance, StreamsConfig.java has a lower current TD value (211 minutes) compared to Fetcher.java (235 minutes). However, it is expected that in 10 weeks from now StreamsConfig.java will have a much higher TD value (338 minutes) than Fetcher.java (262 minutes), since its estimated TD increment is ∼60%. As a result, StreamsConfig.java should be prioritized higher during a strategic refactoring planning, in order to avoid its further TD accumulation.

5 | LIMITATIONS AND THREATS TO VALIDITY

This section discusses the limitations and validity threats of this empirical study. Any forecasting model's accuracy is limited by definition, especially in the software domain, where the future evolution of software quality is significantly influenced by several business-related factors such as planned features, release deadlines, and changes in the size of the development team. As a result, predicting such planned or unplanned events would be a difficult task outside the scope of our research. We believe however that a project's history captures recurrent events over longer time horizons, and thus, that developing a forecasting model based on historical data can provide insight into future evolution. Nonetheless, we acknowledge that the proposed approach is unable to account for anticipated or unanticipated business-related events. Aside from the aforementioned limitations, the methodology proposed in this study suffers from the usual validity threats.



TD Prioritization Table

HORIZON 10 weeks

Show entries 10 Search Search

Class Name	Change Proneness (CP)	TD Change Proneness (CPTD)	Expected Size Change (E[DLOC])	Expected TD Change (E[DTD])	Current TD Value	Forecasted TD Value	TD Trend %
StreamThread.java	0.53	0.39	3.14	-0.6	243	354.23	45.77
Fetcher.java	0.4	0.24	4.15	1.17	235	262.5	11.70
StreamTask.java	0.39	0.12	2.16	0.1	80	55.66	-30.43
KafkaConsumer.java	0.35	0.09	1.8	0.7	101	115.11	13.98
StreamsConfig.java	0.34	0.1	3.45	0.7	211	338.61	60.48
ConsumerCoordinator.java	0.32	0.13	1.73	0.53	140	107.5	-23.21
KafkaProducer.java	0.31	0.13	1.72	-0.84	206	204.33	-0.81
KafkaStreams.java	0.31	0.16	3.61	1.26	152	167.56	10.24
RocksDBStore.java	0.31	0.15	1.73	0.22	42	60.97	45.17
KTableImpl.java	0.28	0.17	2.28	1.1	358	317.44	11.33

FIGURE 8 Table containing TD forecasting and CP supplementary metrics of the selected classes of Apache Kafka application - 10 steps-ahead forecasts

External validity refers to the ability to generalize results. Since the applicability of ML models to forecast TD is examined on six software applications, the study's findings are inevitably subject to external validity threats. It is always possible that a different set of applications might exhibit different phenomena. However, the fact that the selected applications are different in terms of application domains, size, and other factors helps to limit threats to generalization. Furthermore, the present study focuses on the class-level granularity. This practically means that the six examined applications are decomposed into a sample set of 9.900 unique software classes, a number that can be considered adequate for examining the generalizability of the produced results. In addition, a large part of the proposed methodology consists of constructing forecasting models that learn from past commits and may thus be easily adapted to any software application, provided that adequate and trustworthy class-level historic data are available. A similar threat arises from the fact that our dataset is made up entirely of open source Java applications, restricting the potential to generalize the findings to applications from other domains or programming languages. On the other hand, the process of developing TD forecasting models presented in this study is based mostly on the output of the tools used to calculate software-related metrics that can be used as indicators of TD. Therefore, the proposed models can be easily extended to forecast the TD of applications developed in a different programming language, as long as there are tools that facilitate the extraction of software-related metrics that can act as TD indicators for the respective language. This also helps to reduce threats to generalization. However, we cannot make any assumptions about closed-source applications. Further investigation into commercial systems and other object-oriented programming languages is a subject of future work.

Concerning the *internal validity*, i.e., the possibility of having undesirable or unexpected associations between the parameters that might affect the variable that we aim to forecast, it is reasonable to expect that a variety of other metrics that affect TD might have not been taken into consideration. However, the fact that we consider as TD predictors various software-related metrics that have been widely used in the literature as indicators of the presence of TD, such as OO software metrics and code smells, limits this threat. Moreover, the thorough feature selection analysis (including univariate and multivariate regression) performed in our previous related study[5], allowed us to confidently "exploit" the final TD predictors set within the context of this study as well.

Construct validity concerns the extent to which the measurements, i.e., the independent and dependent variables, are correctly represented and accurately measured. The main threats related to construct validity in this study are due to possible inaccuracies in identifying software-related metrics that act as TD indicators, as well as identifying and measuring TD itself. To mitigate this

risk, we decided to employ two popular and widely-used tools, namely SonarQube and CK. Both of these tools were used as a proof of concept of the proposed methodology. The approach described in this study is not dependent on the selected tools, and therefore it could be applied on top of the measurements produced by another similar set of tools, based on user desire. The findings of this study, however, are dependent on the measurements collected by these tools and, as a result, on the tools themselves. As a result, more experimentation is needed to assess the accuracy of results obtained through other tools. As for the threats concerning the reliability of the ML algorithms themselves, we relied on the implementation provided by the scikit-learn library, which is widely considered as a reliable tool. Finally, regarding our decision in Section 4.1 to filter out classes whose number of past versions is below the threshold of 100, we acknowledge that a greater (or smaller) threshold value would result in a smaller (or greater) number of software classes being considered for the next step of the approach. However, we relied on this value as a "rule of thumb", after performing dedicated experiments within the context of not only the present study, but also in our previous related empirical studies[5,19,42], in order to assess what would be the minimum number of samples that would result in an acceptable forecasting error. We point out that this threshold can also be adapted to specific needs, allowing the user to decide (based on their expertise) what an acceptable time frame is, having in mind however that choosing a very small amount of past history would result in an insufficient amount of data, thus affecting the accuracy of the produced forecasting models.

Reliability threats concern the possibility of replicating this study. To facilitate such replication studies, we provide an experimental package containing both the datasets and the scripts that were used for our analysis. This material can be found online[37]. Moreover, the source code of the six examined projects is available on GitHub.

Finally, a threat of different nature stems from the fact that the static analysis tools employed within our study during the Data Collection step (i.e., SonarQube and CK) do not explicitly track classes that were renamed during the studied evolution period. This may potentially affect the results of the approach, since the renamed files will be treated as new, and therefore might be excluded by the analysis if there is no sufficient commit history for the renamed class (i.e., the renaming was relatively recent). To examine how this phenomenon could affect our results, we performed a dedicated analysis aiming at quantifying the frequency of "Rename Class" refactorings applied across the studied evolution period for each of the six examined applications. The results indicate that the number of classes missed by our approach due to renaming is relatively negligible, ranging from 5% to 0% of the total classes of each application. The script and the analysis results are both available online[37]. It should be also noted that the renamed classes being wrongly filtered out due to insufficient historical data will eventually be reconsidered by the approach, either when a sufficient number of past commits (i.e., 100) is reached, or even sooner by decreasing the past-commits threshold (with a potential impact on the accuracy of the forecasting models). Therefore, we believe that the inability to handle classes being renamed is a "limitation" of the prototype tool and does not affect the validity of our methodology. The main goal of this work is to introduce an approach of using TD forecasting for prioritizing TD liabilities, which is something that has not been studied before and seems to be promising. Finally, since the proposed TD Prioritization approach is not inherently dependent on the selected tools, as both SonarQube and CK tools were used as a proof of concept, a potential future extension could be to replace them with other alternatives able to tackle this issue.

6 | CONCLUSIONS AND FUTURE WORK

Monitoring the evolution of TD is highly important for software development companies, as this provides valuable information regarding the effort and, in turn, the cost that is required for maintaining and extending the system. When it comes to TD repayment, the developers are often overwhelmed with a large volume of TD liabilities (e.g., code smells, bugs, vulnerabilities, etc.) that they need to fix, which renders the TD repayment procedure tedious, time consuming and effort demanding. In addition, strict production deadlines often force them to focus on the delivery of new functionality, reducing the time that they can invest on TD repayment activities, leading to the accumulation of TD. Hence, prioritizing TD liabilities is of utmost importance for effective TD repayment, since it allows developers to start their refactoring activities from constructs that are of high interest from a TD viewpoint.

To this end, in this study we proposed a practical approach for a more fine-grained prioritization of TD liabilities by incorporating information retrieved from static analysis, change-proneness analysis and forecasting techniques, while focusing on the class-level granularity. Through our empirical study across the 10 most change-prone investigated classes of six real-world open-source Java applications, we have shown that TD Principal patterns can be modeled adequately by ML techniques. An interesting observation that was made through our analysis is that linear regularization models are better in short-term TD forecasting, while the non-linear Random Forest model performs better in long-term prediction. We strongly believe that taking into

account the future evolution of TD during the TD management activities and combining it with proper change-proneness analysis and visualization techniques can enable the early identification of classes that are more likely to become unmaintainable in the future, and therefore allow project managers and developers plan more effective TD repayment strategies.

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

7 | ACKNOWLEDGEMENTS

An Acknowledgements section is started with `\ack` or `\acks` for *Acknowledgement* or *Acknowledgements*, respectively. It must be placed just before the References.

References

1. Cunningham W. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* 1993; 4(2): 29–30.
2. Martini A, Bosch J, Chaudron M. Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study. *Information and Software Technology* 2015; 67: 237–253.
3. Li Z, Liang P, Avgeriou P. Architectural debt management in value-oriented architecting. In: Elsevier. 2014 (pp. 183–204).
4. Falessi D, Shaw MA, Shull F, Mullen K, Keymind MS. Practical considerations, challenges, and requirements of tool-support for managing technical debt. In: IEEE; 2013: 16–19.
5. Tsoukalas D, Kehagias D, Siavvas M, Chatzigeorgiou A. Technical debt forecasting: An empirical study on open-source repositories. *Journal of Systems and Software* 2020; 170: 110777. doi: 10.1016/j.jss.2020.110777
6. Tsoukalas D, Jankovic M, Siavvas M, Kehagias D, Chatzigeorgiou A, Tzovaras D. On the Applicability of Time Series Models for Technical Debt Forecasting. In: ; 2019: 1–10
7. Lehman MM. Laws of software evolution revisited. In: Springer; 1996: 108–124.
8. Wagner S. A Bayesian network approach to assess and predict software quality using activity-based quality models. *5th International Conference on Predictor Models in Software Engineering - PROMISE '09* 2009: 1. doi: 10.1145/1540438.1540447
9. Van Koten C, Gray A. An application of Bayesian network for predicting object-oriented software maintainability. *Information and Software Technology* 2006; 48(1): 59–67. doi: 10.1016/j.infsof.2005.03.002
10. Zhou Y, Leung H. Predicting object-oriented software maintainability using multivariate adaptive regression splines. *Journal of Systems and Software* 2007; 80(8): 1349–1361. doi: 10.1016/j.jss.2006.10.049
11. Fontana FA, Mäntylä MV, Zanoni M, Marino A. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 2016; 21(3): 1143–1191.
12. Gondra I. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software* 2008; 81(2): 186–195. doi: 10.1016/j.jss.2007.05.035
13. Goulão M, Fonte N, Wermelinger M, Abreu eFB. Software evolution prediction using seasonal time analysis: a comparative study. In: IEEE; 2012: 213–222
14. Arisholm E, Briand LC. Predicting fault-prone components in a java legacy system. In: ACM; 2006: 8–17
15. Yazdi HS, Mirbolouki M, Pietsch P, Kehrer T, Kelter U. Analysis and prediction of design model evolution using time series. In: Springer; 2014: 1–15.

16. Kenmei B, Antoniol G, Di Penta M. Trend analysis and issue prediction in large-scale open source systems. In: IEEE; 2008: 73–82

17. Raja U, Hale DP, Hale JE. Modeling software evolution defects: a time series approach. *Journal of Software Maintenance and Evolution: Research and Practice* 2009; 21(1): 49–71. doi: 10.1002/smr.398

18. Nagappan N, Ball T, Zeller A. Mining metrics to predict component failures. In: ACM; 2006: 452–461

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

20. Tsoukalas D, Siavvas M, Jankovic M, Kehagias D, Chatzigeorgiou A, Tzovaras D. Methods and Tools for TD Estimation and Forecasting: A State-of-the-art Survey. In: IEEE; 2018

21. Lenarduzzi V, Besker T, Taibi D, Martini A, Arcelli Fontana F. A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools. *Journal of Systems and Software* 2021; 171: 110827. doi: 10.1016/j.jss.2020.110827

22. Fontana FA, Ferme V, Spinelli S. Investigating the impact of code smells debt on quality code evaluation. In: IEEE Press; 2012: 15–22

23. Choudhary A, Singh P. Minimizing Refactoring Effort through Prioritization of Classes based on Historical, Architectural and Code Smell Information. In: ; 2016: 76–79.

24. Akbarinasaji S, Bener A, Neal A. A heuristic for estimating the impact of lingering defects: can debt analogy be used as a metric?. In: IEEE Press; 2017: 36–42.

25. Nugroho A, Visser J, Kuipers T. An empirical model of technical debt and interest. In: ACM; 2011: 1–8.

26. Falessi D, Voegele A. Validating and prioritizing quality rules for managing technical debt: An industrial case study. In: IEEE; 2015: 41–48.

27. Guo Y, Spínola RO, Seaman C. Exploring the costs of technical debt management–a case study. *Empirical Software Engineering* 2016; 21(1): 159–182.

28. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 1994; 20(6): 476–493. doi: 10.1109/32.295895

29. OMG . Automated Technical Debt Measure V1.0. https://www.omg.org/spec/ATDM/1.0/PDF; 2018. Accessed: May 30, 2022.

30. Li Z, Avgeriou P, Liang P. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 2015: 193–220. doi: 10.1016/j.jss.2014.12.027

31. Ampatzoglou A, Ampatzoglou A, Chatzigeorgiou A, Avgeriou P. The financial aspect of managing technical debt: A systematic literature review. *Information and Software Technology* 2015; 64: 52–73.

32. Alves NSR, Mendes TS, Mendonça MGd, Spínola RO, Shull F, Seaman C. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology* 2016; 70: 100 – 121. doi: 10.1016/j.infsof.2015.10.008

33. Riaz M, Mendes E, Tempero E. A systematic review of software maintainability prediction and metrics. In: IEEE Computer Society; 2009: 367–377.

34. Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 2018; 23(3): 1188–1221. doi: 10.1007/s10664-017-9535-z

35. Fowler M. *Refactoring: improving the design of existing code*. Addison-Wesley Professional . 1999.

36. Aniche M. *Java code metrics calculator (CK)*. ; https://github.com/mauricioaniche/ck/: 2015.

37. Supporting Material. https://sites.google.com/view/granular-td-forecasting/home; 2022. Accessed: May 30, 2022.

38. Khomh F, Di Penta M, Gueheneuc YG. An exploratory study of the impact of code smells on software change-proneness. In: IEEE; 2009: 75–84.

39. Bosu A, Carver JC, Hafiz M, Hilley P, Janni D. Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study. *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* 2014: 257–268. doi: 10.1145/2635868.2635880

40. Shin Y, Meneely A, Williams L, Osborne JA. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering* 2011; 37(6): 772–787.

41. Dietterich TG. Machine learning for sequential data: A review. In: Springer; 2002: 15–30

42. Mathioudaki M, Tsoukalas D, Siavvas M, Kehagias D. Technical Debt Forecasting Based on Deep Learning Techniques. In: Springer International Publishing; 2021: 306–322

43. Stone M. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)* 1974; 36(2): 111–133. doi: 10.1111/j.2517-6161.1974.tb00994.x

44. Cohen J. *Statistical power analysis for the behavioral sciences*. Routledge . 2013.

45. Tsoukalas D, Mittas N, Chatzigeorgiou A, et al. Machine Learning for Technical Debt Identification. *IEEE Transactions on Software Engineering* 2021: 1–1. doi: 10.1109/TSE.2021.3129355

46. Chug A, Malhotra R. Benchmarking framework for maintainability prediction of open source software using object oriented metrics. *International Journal of Innovative Computing, Information and Control* 2016; 12(2): 615–634.

47. Elish MO, Elish KO. Application of TreeNet in Predicting Object-Oriented Software Maintainability: A Comparative Study. In: ; 2009: 69–78

48. Jin C, Liu J. Applications of Support Vector Machine and Unsupervised Learning for Predicting Maintainability Using Object-Oriented Metrics. In: . 1. ; 2010: 24–27

49. Feurer M, Klein A, Eggensperger K, Springenberg J, Blum M, Hutter F. Efficient and robust automated machine learning. In: ; 2015: 2962–2970.

How to cite this article: Tsoukalas D., Siavvas M., Kehagias D., Ampatzoglou A., and Chatzigeorgiou A. (2022), A Practical Approach for Technical Debt Prioritization based on Class-Level Forecasting, *J Softw Evol Proc.*, *2022;00:1–6*.