Forecasting the Principal of Code Technical Debt in JavaScript Applications

Ioannis Zozas, Stamatia Bibi, Apostolos Ampatzoglou

Abstract— JavaScript (JS) is one of the most popular programming languages for developing client-side applications mainly due to allowing the adoption of different programming styles, not having strict syntax rules, and supporting a plethora of frameworks. The flexibility that the language provides may accelerate the development of application, but also pose threats to the quality of the final software product, e.g., introducing Technical Debt (TD). TD reflects the additional cost of software maintenance activities to implement new features, occurring due to poorly developed solutions. Being able to forecast the levels of TD in the future can be extremely valuable in managing TD, since it can contribute to informed decision making when designating future repayments and refactoring budget among a company's projects. Despite the popularity of JS and the undoubtful benefits of accurate TD forecasting, in the literature, there is available only a limited number of tools and methodologies that are able to: (a) forecast TD during software evolution, (b) provide a ground-truth TD quantifications to train forecasting, since TD tools that are available are based on different rulesets and none is recognized as a state-of-the-art solution, (c) take into consideration the language-specific characteristics of JS. As a main contribution for this study, we propose a methodology (along with a supporting tool) that supports the aforementioned goals based on the Backward Stepwise Regression and Auto-Regressive Integrated Moving Average (ARIMA). We evaluate the proposed approach through a case study on 19,636 releases of 105 open-source applications. The results point out that: (a) the proposed model can lead to an accurate prediction of TD, and (b) the Number of appearances of the "new" and "eval" keyword along with the number of "anonymous" and "arrow" functions are among the features of JavaScript language that are related to high levels of TD.

Index Terms—Software Quality, JavaScript, Code Technical Debt, Source Code Quality

1 Introduction

JavaScript (JS) is constantly gaining ground in the software industry as it is considered to be ideal for implementing web and mobile applications [20][44]. The usage of JS has been boosted by the fact that it is: (a) a weakly-typed language—no strict programming rules; (b) a multi-paradigm language—allowing object-oriented, functional, and imperative programming; and (c) supported by a variety of open-source programming frameworks and libraries [18]. Currently, more than 400K JS repositories are hosted and maintained in GitHub, the majority of which is presenting a lifespan of more than five years, proving the need for performing maintenance. Nevertheless, the previously mentioned "*selling points*" of JS despite accelerating the implementation pace, also pose threats to the quality of the final product that is often very unstructured and hard to understand. For example, a developer may choose to benefit from the "*arrow*" function that has a shorter syntax and thus speed up development, but may cause readability issues during maintenance. To resolve this problem, the developer may need to convert the "*arrow*" function into a typical function to change the scope of the function and improve its maintainability, also known as repaying "*code technical debt*".

- *Ioannis Zozas is with the Department of Electrical and Computer Enigneering, University of Western Macedonia, Kozani, Greece. E-mail: izozas@uowm.gr.*
- *Stamatia Bibi is with the Department of Electrical and Computer Enigneering, University of Western Macedonia, Kozani, Greece. E-mail: sbibi@uowm.gr.*
- *Apostolos Ampatzoglou is with the Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece. E-mail: a.ampatzoglou@uom.edu.gr.*

Technical Debt (TD) is a metaphor [11] used to reflect the additional maintenance effort, caused by quality compromises. The amount of money "*deposited*" (saved) from compromising maintainability (using less development time) refers to the term TD Principal, whereas the additional effort that needs to be paid due to the lowered maintainability is termed TD Interest. We note that for the rest of this paper when referring to the value of TD, we: (a) refer to the value of TD Principal; and (b) to TD Principal at the code level, since other types of TD (such as architecture, requirements, etc.) are not considered in this study. By studying the literature (see Section 2.3), we have identified the following limitations:

(L1) Lack of methods and tools to forecast TD Principal throughout the evolution of a software system. Predicting the value of TD Principal along evolution is a challenging task, since both the system and the associated TD Principal emerge in parallel [40]. Although, in the literature there are studies that focus on the quality assessment of JavaScript applications [16], [30], their maintainability [45], and their evolution [10], there is limited work on TD management [39]. Being able to forecast TD Principal along software evolution can help towards the effective prioritization of the project maintenance effort allocation: a quality manager can allocate more budget for TD management in projects that are more heavily affected by the negative consequences of TD. For instance, if a company maintains 3 different systems, out of which 2 have accumulated high and similar amounts of TD Principal; being aware of the system, whose TD will grow more in the near future can guide resources allocation for quality improvement. We note that in corner cases that TD Principal and TD Interest are not correlated, as expected based on the metaphor and the literature [46], prioritization is expected to follow the ranking of TD Interest.

(L2) Lack of a ground-truth TD Principal quantification in JS. Currently, there are several tools that are available for quantifying TD Principal (e.g., SonarQube, CAST, SQUoRE). However, none is recognized as a state-of-the-art solution for quantifying TD. This shortcoming arises mostly from the different methodologies that are used to capture TD Principal: various metrics, rulesets, and score mechanisms to quantify it. The majority of studies focused on forecasting, use the TD quantification of a single tool rendering the credibility of the forecast into question [28], [39], [40]. In this direction, providing a TD benchmark that showcases TD Principal scores in which several tools agree, can increase the credibility of TD forecasts.

(L3) Current methods do not take into consideration the specific characteristics of JS. Most of the TD tools found in the literature use a set of metrics, usually object-oriented ones, that can be uniformly applied to different languages, without taking into consideration their specifics. This might pose a major threat to the completeness of TD quantification, concerning JS applications, since it is recognized that TD quantification should depend on language-specific models [38]. The consideration of a plethora of metrics (Process, Activity, and Product metrics), including JS-specific ones in a TD model can increase the accuracy of the TD prediction models.

Based on the above, the contribution of the paper is the provision of a novel 3-step methodology that aims at alleviating the aforementioned limitations, as explained below. The scope and motivation of the proposed methodology is illustrated in Figure 1.

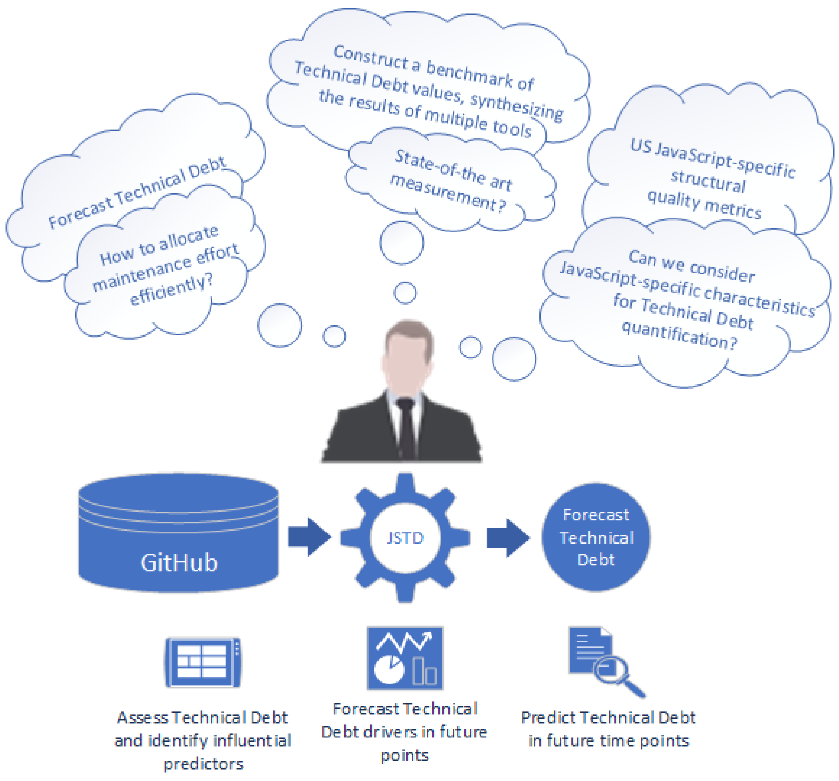

Construct a benchmark of Technical Debt values, synthesizing the results of multiple tools
State-of-the art measurement?
Forecast Technical Debt
How to allocate maintenance effort efficiently?
US JavaScript-specific structural quality metrics
Can we consider JavaScript-specific characteristics for Technical Debt quantification?
GitHub
JSTD
Forecast Technical Debt
Assess Technical Debt and identify influential predictors
Forecast Technical Debt drivers in future points
Predict Technical Debt in future time points

Fig. 1. Scope and Motivation of this Study

In particular, we combine two well-known ML techniques (namely: *supervised Backwards Stepwise Regression*, and *Autoregressive Integrated Moving Average (ARIMA)*) for forecasting TD Principal in three steps:

[Step 1] *Predict the value of TD Principal relying on several parameters and identify the most significant ones*. The goal of this step is to identify the most important predictors that contribute to accumulating TD. For this purpose, data from different already implemented JS applications (retrieved by mining the GitHub repository) are analyzed to extract product and process metrics that are able to predict TD Principal, through the application of Backwards Stepwise Regression. Based on regression theory [13] a predictor is considered to be an independent variable (in our case product and process metrics) that provides information on an associated dependent variable (in our case TD Principal). To increase the credibility of TD Principal quantification (*L2*), we rely on a benchmark developed by Amanatidis et al. [2], targetting the inter-agreement of three well-established tools. The metrics that are recorded include, apart from typical metrics, JS-specific ones [16][18][26][29][30] (*L3*).

[Step 2] *Forecast TD Principal predictors*. The goal of this step is to build a model that forecasts the values of the TD predictors identified in step 1. For this purpose, the complete history of the JS applications are analyzed. The formalization employed in this step is the ARIMA Box-Jenkins model for time-series analysis. In this step, we focus on forecasting the values of TD predictors instead of TD Principal itself (step 3). ARIMA is a univariate forecasting model, relying on historical values of a single variable—the one that is being forecasted; therefore, it is ideal for predicting the future values of TD predictors (*L1*).

[Step 3] Predict TD Principal. The goal of this step is to predict TD Principal in future time points. For this purpose, the forecasted values of the TD predictors (as calculated by the ARIMA model in Step 2) serve as an input to the TD Principal estimation model (as calculated in Step 1). The final output of the proposed methodology is the value of TD Principal in various future time points (*L1*), considering JS-specific characteristics (*L3*), and relying on the quantification of different TD tools for reliability (*L2*).

To validate our approach, we performed an empirical study on 19,636 releases of 105 JS applications, hosted in GitHub. The rest of the paper is organized as follows: Section 2 presents related work and the necessary background information; whereas Section 3 presents in detail the proposed methodology. In Section 4, we present the validation case study design. In Section 5, we present the results, whereas, in Section 6, we discuss the results and provide implications to researchers and practitioners. Finally, in Section 7 we present threats to validity; whereas in Section 8, we conclude the paper.

2 Related Work & Background Information

In Section 2.1, we present works that are directly comparable to ours, i.e., studies that aim at the forecasting of TD—related to *L1*. In Section 2.2, we provide comprehensive literature, based on [5], on the tools that can be used for quantifying TD. The information in this section explains in detail *L2*, i.e., the need for a TD measurement benchmark. Finally, Section 2.3 is related to works that focus on JavaScript; thus, providing information regarding *L3*.

2.1 Technical Debt Forecasting

Forecasting [8] refers to the process that exploits data from previous events, along with recent trends, to estimate future events. On this basis, Mathioudaki et. al. [28] proposed deep learning techniques for providing a more accurate long-term TD prediction. The authors quantified TD based on the SonarQube platform and subsequently applied Random Forest, Multi-layer Perceptron, and ARIMA models for producing predictions. The methods were evaluated on five Java open-source projects. According to the results, Multi-layer Perceptron outperformed the rest of the methods presenting higher long-term accuracy, even for 150 steps ahead in the future, presenting a mean RMSE 2,361.09, mean MAPE 4.99% and mean MAE 1,960.52). Tsoukalas et. al. [39], employed the ARIMA time series modeling for forecasting TD based on code violations (code smells, bugs and vulnerabilities). The ARIMA method was evaluated on a dataset of five open-source java projects, while TD was estimated based on the SQALE index of the SonarQube platform. The authors concluded that the ARIMA (0,1,1) model presented the best fit, outperforming the random walk model. The accuracy model fit statistic metrics (RMSE 0.02, MAPE 3.62%, and MAE 0.02) indicated that ARIMA overall presents a satisfactory prediction power for up to 2 steps ahead in the future.

Tsoukalas et. al. [40] conducted an empirical study, to examine the applicability of machine learning algorithms such as Multivariate Linear Regression, Lasso Regression, Ridge Regression, Stochastic Gradient Descent, Support Vector Machines and Random Forests on TD forecasting. For this purpose, the authors analyzed weekly observations, for a 3-year period, of 225 open-source Java applications. The TD Principal was quantified based on SonarQube while the metrics that participated in the study were the number of bugs, vulnerabilities, code smells, size, code coverage, complexity, coupling, and cohesion. The study concluded that machine learning algorithms perform better (RMSE 4,767.65, MAPE 8.66%, and MAE 4,262.78) in longer forecasting horizons compared to linear models such as ARIMA. Kumar et. al. [23] applied time series for forecasting the TD of SaaS-based applications. The authors quantified TD Principal through an equation that is based on metrics that measure the service utilities, the recompositing decisions, and the service level agreement violations. The proposed methodology forecasts future TD by applying ARFIMA models. The latter was evaluated in one real-world case scenario, on the Sales CRM application and its' services, presenting MAE and RMSE accuracy within 15% of the actual values.

Limitations: Currently the majority of tools and methodologies that exist focus on quantifying TD in a current version of an application, without producing long-term forecasts [6], [12], [17]. Also, there are some studies that predict the existence of high-TD software artifacts [2], [41] but these studies do not produce a TD estimate related to the effort required to fix quality issues. ***Contribution***: In this study, we focus on producing long-term forecasts of TD principal, itself.

TABLE 1: Metric evidence per study

Study	Metrics variability	TD Quantification	Language	Tool
[23]	Maintainability	Custom formula	Generic	
[28]	Activity, TD	SonarQube	Java	
[39]	TD	SonarQube	Java	
[40]	Activity, Source code size & Modularity, Source code Complexity, Maintainability, Other	SonarQube	Java	
This study	External quality indicators, Activity Source code Size & Modularity, Source code Complexity, Maintainability, JS metrics, TD	TD benchmark [2]	JS	X

2.2 Technical Debt Quantification Tools

In the literature and the market, there are several tools for quantifying TD Principal [5]. These tools consider different types of TD, such as architectural, design, documentation, testing, and code TD. In this section we focus on tools capturing TD introduced at the code level, driven by the scope of the proposed methodology. Therefore, tools like VisMiner, Anacondebt, CodeMRI, etc. are omitted. In Table 2 we present an overview of all tools identified by Avgeriou et al. [5], quantifying TD from source code. For each tool, we present the metric used to quantify TD Princial, the index of code TD Principal, and whether the tools are able to analyse JS applications, along with file-level calculations (which are more suitable for JS).

TABLE 2: TOOL OVERVIEW

Tool	JS	Principal Definition	Code TD index	File Level
CAST[1]	✓	Time to remove issues	Violations * Rule criticality * Effort	✓
NDepend[2]	-	Man-time to fix issues	Violations * Fix effort	-
SonarQube [3]	✓	Time to remove issues	Cost to develop 1 LOCe * Number of lines of code.	✓
SQuORE[4]	✓	Time to remove issues	Issues * Fix time	✓
Code Inspector[5]	-	Effort to avoid TD	Function of violations, duplications, readability/maintainability issues.	-
Symphony Insight[6]	-	Time to remove issues	Issues * Fix time	-

Focusing on the tools that can be used to quantify the TD Principal of JavaScript applications, SonarQube [5] is the most widely used. SonarQube's TD estimation algorithm is based on the detected maintainability issues, calculated with the help of the following metrics: code duplications, comment density, bad distribution of complexity, bugs, vulnerabilities, coding rules violations, bad code design, and unit test bugs. TD is calculated either at class- or file-level and is defined as the remediation effort required to fix all maintainability issues, measured in minutes. At project-level TD Principal is computed as the sum of each remediation effort function for every class or file. Furthermore, the tool provides a SQALE rating index [24] that is the ratio between the TD of the application to the estimation of the cost to rewrite the application from the beginning. Similarly, to SonarQube, the SQuORE platform [6], adopts the ISO rule standards to quantify TD Principal. The platform quantifies the values of quality attributes such as Maintainability, Reliability, Efficiency, Portability, Security, Testability, and Changeability, based on a variety of quantifiable metrics. The definition of TD Principal is similar to SonarQube, in the sense that it uses remediation functions without providing an index. From a different perspective, the estimations provided by CAST [12] are based on performance, security, robustness, transferability, and changeability violations. TD Principal is defined as the cost to fix structural quality problems, i.e., violations, that may cause future major disruptions. In particular, each violation is assigned a weight that reflects its criticality. The tool provides a TD Principal index that is the product of the violation with the criticality, and the effort required to fix the violation.

The aforementioned tools are commercial products that provide a free license for academic or research purposes. One of the main shortcomings while performing qualitative empirical studies in the TD domain is the selection of the tool that will be used for quantifying TD Principal since there is no state-of-the-art solution. This problem becomes even worse, by considering that TD quantification tools adopt different metrics and quality models to capture TD, a fact that in many cases leads to contradicting TD estimations among tools [2], [21]. To capture the diversity of TD quantification of different tools, Amanatidis et al. [2], developed a benchmark that compares the different TD Principal quantification of three tools (SonarQube, CAST and SQuORE) and calculates their level of agreement. A main output of this work is the construction of benchmark, i.e., a list of code artifacts (i.e., classes or files) which all tools identify as TD items. In the same direction, Tsoukalas et. al. [41] argue that the reliability of the findings derived by a single TD tool is limited. Therefore, the authors used the benchmark developed by Amanatidis et al. [2] to classify software code artifacts as being High-TD or not (using source code, repository activity, issue tracking, refactoring, duplication and commenting rate as predictors).

Limitations: Different tools produce different types of estimates based on the different ruleset they adopt, a fact that can cause large deviations within the estimates produced by different tools for the same application. The majority of studies focused on TD Principal forecasting use the TD quantification of a single tool rendering the credibility of the forecast into question [23], [28], [39], [40].
Contribution: In this study we address this issue by adopting the TD benchmark [2] that identifies projects, for which there is an agreement for their value of TD Principal, based on the three tools (SonarQube, CAST and SQuORE).

2.3 JavaScript Quality Assessment

There are several studies found in literature for assessing the quality of Javascript applications. The majority of such studies adopt metrics that are language-agnostic; highlighting the need for language-specific metrics [42]. Gizas et. al. [18] used size metrics (LOC, Numb. Of statements, comments, comments intensity), complexity met-

[1] CAST (1998): https://www.castsoftware.com
[2] NDepend (2007): https://www.ndepend.com/
[3] SonarQube (2007): https://www.sonarqube.org
[4] SQuORE (2010): https://www.vector.com
[5] Code Inspector (2019): https://www.codiga.io
[6] Symphony Insight (2019): https://insight.symfony.com/

rics (McCabe complexity, branches& depth), and maintainability metrics (Halstead metrics) to compare the quality of six popular JavaScript frameworks (ExtJS, Dojo, jQuery, MooTools, Prototype and YUI). Misra and Cafer [29] introduced a cognitive complexity index for assessing the quality of JS applications. The index utilizes metrics such as the lines of code, the number of arbitrarily name distinct variables, the number of meaningfully named variables, the number of operators, and the cognitive weights of basic control structures. The authors evaluated the proposed index on 30 JS scripts and concluded that it performs better than the typical complexity metrics when used separately, stressing out the need for more specialized JS metrics. Lin et. al. [26] presented a set of metric units and quantization rules for the React.js framework, based on the existing metrics proposed for assessing the code quality of traditional software applications. The developed domain-specific quality model includes three types of metrics: (a) JS metrics (code indentation, code annotation, correct variable naming, code replication); (b) React.js component metrics (lines of code, dependent graph leaf nodes, dependent graph depth, functions cyclomatic complexity, coupling); and (c) React.js state metrics (self-state, parent-transition state, state utilization, state transmissibility, state transition weight).

Language-agnostic metrics have also been used to assess the maintainability [45] and the evolution of JS applications [10]. Zozas et. al. [45] investigated the maintenance activities performed in 60 JS projects. Out of this research, two indices were proposed, the Maintenance Effort index and the Maintenance Changes index. The indices were evaluated based on correlation, consistency, predictability, discriminative power, and reliability evaluation criteria. The metrics that participate in these indices are: Number of Bugs, Duplicate Lines of Code, Lines of Code, Number of attributes, Number of corrective activities, Complexity, Number of commits, and Number of files. Chatzimparmpas et. al. [10] focused on JS application quality and evolution trends over time, by examining the relevant Laws of Lehman. Each law was investigated with respect to maintenance data coming from 20 popular open-source JS applications. The authors concluded that complexity remains stable over time.

Despite the above efforts which consider traditional quality metrics for accessing quality, there are a few studies that consider the unique nature of JS language for assessing the quality of the associated applications. Gallaba et. al. [16] pointed out that callbacks are a key feature of JS applications that uses them to handle and respond to events. The authors suggest that the increased number of callbacks decreases the understandability and the maintainability of the source code as they introduce a non-linear control flow that is declared anonymously and executed asynchronously. By performing an empirical study on 138 JS projects, the study concluded that 10% of functions include callbacks, while 43% out of these are anonymous, and over 50% of these are nested and asynchronous. On the same track, Mirghasemi et. al. [30] explored anonymous function declaration in 10 large JS projects and concluded that only 7% of functions are named by developers, thus having an impact on maintenance and quality overall. To address this problem, the authors proposed an automated approach, called Static Function Object Consumption, that provides names to anonymous JavaScript functions based on local source code analysis. Similarly, Richards et. al. [32] focused on the eval keyword, which is among the dynamic features of the language, arguing that the extended use of this keyword may cause unpredictable behavior to JS applications. The authors presented an approach to transform common uses of eval into other language constructs.

Limitation: The majority of tools are focused on TD introduced in OO applications (i.e., Java) [28][39][40], considering metrics that are focused on the OO nature. This issue poses a major threat when adopting these models to applications developed in JS since the unique aspects of the language are not taken into consideration. ***Contribution***: In this study we introduce a TD forecast model with four types of metrics: (a) External Quality and Activity metrics [45]; (b) Source code size and complexity metrics [18][29]; (c) Maintainability metrics [10][18]; and (d) Language-specific metrics as introduced by [16][30][32].

2.4 Background Information

In this section, we present the necessary background on the data analysis methods employed in this study

Regression Analysis (RA) aims at predicting the value of a dependent variable, based on the value of one or multiple independent variables [13]. In this study we applied Backward elimination for two reasons: (a) this method does not present the suppressor effect, i.e., a predictor is significant when another predictor is held constant [7]; and (b) the method can handle many independent variables being able to eliminate predictor variables to just the most important ones. The Backward Stepwise Regression (BSR) is a stepwise regression approach, in which the model at the start, is saturated with the independent variables [43]. The end outcome of Backward Stepwise Regression is a function, in which independent variables measured in metric contribute towards the prediction, with a specific weight Bi, as the summary of each metric.

The ***Autoregressive Integrated Moving Average (ARIMA)*** time series forecasting model aims at forecasting the value of a particular variable based on past observations [36]. We chose to employ ARIMA as it does not require computational power nor extensive parameter tuning, relying solely on historical data [36]. An alternative to this would be the use of SARIMA, which considers seasonality in the dataset, which is not the case for software development data [39]. The produced models [8] are denoted as ARIMA (p, d, q). The adjusting parameters are p (the autoregressive AR part of the model as the maximum number of lags in it), d (the integrated I part of the model as the number of differencing observations), and q (the moving average MA assuming current error dependency on the error of lags allowing linear combination between successive lags). The main assumptions are normality, stationarity, and invertibility [8]. The modeling strategy involves the steps: (a) *Stationarity Identification* [4]; (b) *Estimation*; (c) *Testing;* and (d) *Application*.

3 JSTD Methodology

In this section, we present the proposed methodology for forecasting TD Principal in JS applications (namely JSTD), as outlined in Figure 2. In **Phase 1**, we predict the value of TD principal based on several parameters; in **Phase 2**, we forecast the value of each TD predictor in the future; and in **Phase 3**, we predict TD Principal in future time points, based on the forecasted values of TD predictors. To automate the application of the proposed methodology, we developed a tool that implements the aforementioned steps. The input of the tool is a GitHub repository, whereas the output is guided by the methodology steps.

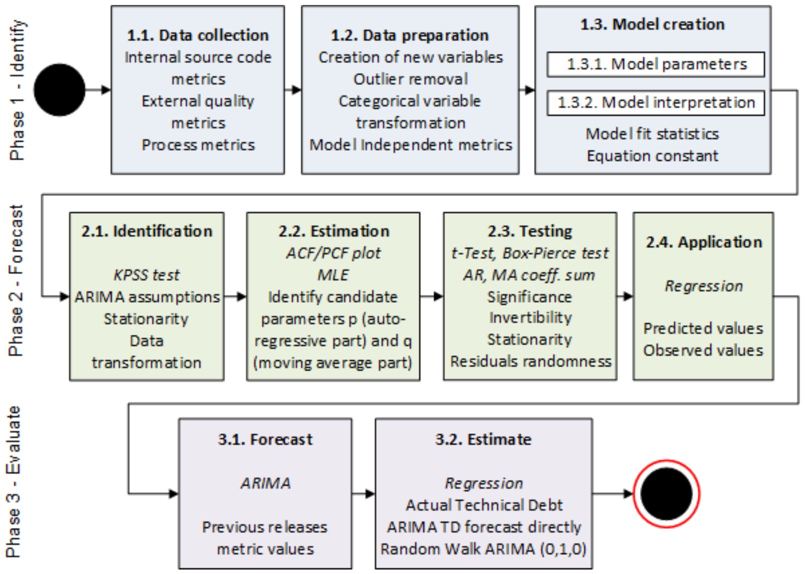


Fig. 2. Proposed Approach

3.1 Predict TD Principal

In this step we identify the most important predictors that can be used to predict TD Principal. First, we focus on ***data collection (step 1.1)***: in this activity, we record a wide range of available metrics, so as to isolate the most significant predictors that affect TD. The list of metrics (candidate TD predictors) belongs to the four main categories defined in section 2.3*: External quality and activity metrics* (Operational metrics, End-User Activity metrics), *Code size & complexity metrics* (Source code size metrics, Source code complexity metrics), *Maintainability Metrics* (Code Smells, Vulnerabilities) and *JS metrics*. JSTD calculates source code and maintainability metrics by reusing third-party components (i.e., Snyk, Lizard, and ES-check). The external quality metrics are derived by JSTD by using the related GitHub APIs. TD Principal quantification is derived by using the benchmark provided by Amanatidis [2]. To identify a unified TD Principal value, we have synthesized the values of the benchmark tools, as follows:

- we have executed CAST, SonarQube, and Squore for each project
- we retained files whose TD Principal values are similar (present less than 5% deviation between the max and min TD Principal values from the 3 tools)
- we summed the TD Principal of these files to aggregate to the project level, which is the unit of analysis

Table 3 presents the metrics collected by JSTD that are used in the scope of this study. Due to size limitation, Table 3 presents only a brief description of the metrics, whereas a full-fledged metric definition, along with representative examples are given in Appendix B.

TABLE 3: Predictors

Group	Metric Name	Calculation Method
External Indicators	POPL	Popularity – Number of stars
	AGE	Reverse days to the latest release.
	OPEN_ISSUES	Open issues (bugs)
	CLOSED_ISSUES	Closed/resolved issues (bugs)
	DEVLP	Developers/contributors
	PART	Commits for every release.
	DOC	Comments per commit.
	UPD	Frequency of updates
Source Code Size & Complexity Metrics	SLOC	Physical source code lines
	LCOM	Lines of comments.
	LOC	Total lines of code = SLOC + LCOM.
	NOA	The number of attributes.
	NOC	The number of classes.
	NOM	The number of methods.
	FILES	The number of files.
	DIRS	The number of directories.
	SIZE	Release size in bytes.
	PARM	Number of function parameters.
	DIT	Depth of inheritance tree.
	MEM	Memory heap.
	CC	Cyclomatic complexity.
	CCDEN	Cyclomatic complexity density.
	HEFF	Halstead effort.
	HPV	Halstead program volume.
	HPL	Halstead program level difficulty.
	CLONE	Duplicate (cloned) lines.
	COVRG	Source code coverage percent.
Maintainability	OBFS	Number of obfuscation incidents.
	CSMELL	Total count of code smell issues.
	VULN	Total count of vulnerability issues.

Group	Metric Name	Calculation Method
JS metrics	WITH	WITH keyword statements.
	EVAL	EVAL keyword statements.
	VECMA	Version of ECMAScript applied.
	NEW	NEW keyword statements.
	ANONYM	Number of Anonymous functions.
	ARROW	Number of Arrow functions.
TD	TD	TD Principal calculated based on the benchmark of Amanatidis et al. [2]

Upon the data collection, we proceed ***with Data preparation (step 1.2)*** This activity includes the calculation of derived variables (e.g., average size of functions in an application). Within the scope of this study, we also calculated the project-level TD introduced in the source code by summing the TD Principal of each file. The ***development of the TD Prediction model (step 1.3)*** is the last activity of this step. During the development of the model, we need to ***specify the parameters of the model (step 1.3.1)***. Initially, we select the type of statistics that will be used to fit the model (i.e., f-measure, t-measure, existence or not of a constant value). To apply Backward Stepwise Regression, we selected T-measure as a statistical measure, along with the inclusion of a constant value in the equation. The JSTD tool applies BSR to the most recent releases of the 104 open-source JS projects. The next activity is to ***interpret the model (1.3.2)*** and confirm that the results are meaningful, i.e., that can be intuitively confirmed. For this reason, we use the values of regression coefficients to identify the impact of each independent variable on TD Principal. The outcome of this step is an equation that uses the values of several parameters to predict the value of TD Principal.

3.2 Forecast Techical Debt Predictors

In this step we forecast the values of the TD predictors identified in the regression model (Step 1). The selected forecasting modeling procedure is Box-Jenkins ARIMA. **The Identification (step 2.1)** activity aims to ensure that the ARIMA model assumptions are fully satisfied. The collected data are sampled to create time series and will be tested for stationarity. It is unknown which sampling period would work best, and as such, three different sampling periods are tested: for one day, one week, and one month. In our case, the time series were found to be non-stationary based on the KPSS test, and as such, differencing data transformation was applied, as first-order differencing and one-month sampling was found to remove non-stationarity, the same finding is supported by Tsoukalas et al. [40]. Before the development of the forecasting models the dataset is split into a training and a testing set: The training set for each participating project includes 90% of the oldest observations; whereas the testing set for each project includes the rest 10% of the observations (the most recent ones).

The **Estimation (step 2.2)** activity, estimates candidate p (auto-regressive part) and q (moving average part) parameters of the models. This activity derives a group of ARIMA (p, d, q) models – in our case (p, 1, d) – to be further tested for best fit for each TD predictor identified in step 1.3.2. The third activity is **Testing (step 2.3)** and involves testing the diagnostics of the ARIMA models, to identify whether they are satisfied. All models are tested for significance, stationarity and invertibility conditions, and residuals randomness, as described in Section 3.2. Finally, in the **Application (2.4)** activity of this step, a regression analysis is performed between the predicted values of the model and the observed values of the most recent observations that belong to the testing data set. Thus, the predictive power of the derived model has been tested by evaluating RMSE, MAPE, and MAE metrics.

3.3 Predict Code Techical Debt Principal

During this step, we **forecast the value of TD Principal** using the models derived in the previous steps. The forecasted values of TD predictors serve as an input to the estimation model developed in step 1.3 to **predict future TD Principal (step 3.2).** During this activity, we derive an early estimation of debt, based on the forecasted values of TD predictors.

4 Case Study Design

In this section, we present the design of the case study performed to evaluate the proposed methodology, as presented in Section 3. The case study was designed, based on the guidelines of Runeson et al. [33].

4.1 Research Questions

The main goal of this study is to validate the JSTD methodology. For each one of the steps of JSTD, a relevant research question has been set:

[RQ1] *Which are the most significant predictors that influence the most the technical debt of JavaScript applications*?

This question aims to identify the most significant predictors that influence the value of TD Principal for JS applications. Our target is to create a model that can be used for accurately estimating TD Principal by isolating the most significant TD predictors. This research question is related to the 1st step of the methodology.

[RQ2] *Is it possible to accurately forecast the TD predictors through time-series forecasting methods?*

This question aims to validate the accurate forecasting of the future values of TD predictors, by analyzing the evolution of the project. This research question is related to the 2nd step of the methodology.

[RQ3] *Is it possible to accurately estimate TD Principal based on the forecasted values of the TD predictors?*

This question aims to evaluate the accuracy of estimating TD Principal in a particular future time point, based on the forecasted values of TD predictors. This research question is related to the 3rd step of the methodology.

4.2 Case Selection

Since the proposed methodology aims at predicting the future values of TD Principal, the evaluation focuses on evaluating the predictive power of the derived models. The proposed approach was evaluated on a dataset containing 105 popular JavaScript applications hosted in GitHub. The dataset has been included in the Github repository of the JSTD tool[7]. For these applications, in total 19,636 releases have been analyzed. Several descriptive statistics on the dataset are presented in Appendix A. The applications are selected, based on the following criteria:

- The applications are among the most popular written in JS in GitHub (the 105 projects with the largest number of stars), so as to ensure that the applications are not trivial or toy-examples.
- JS is the primary scripting language, more than 50%, so that the results are JS-specific–We note that to this percentage the GitHub platform considers HTML, CSS and variants as separate programing languages which are markup scripts. For example, Bootstrap is reported as 50% JS while the rest 50% is HTML and CSS scripts.
- The evolution contains more than 10 releases, to ensure that of its fitness for time series analysis [8], [27].

4.3 Analysis Process

TD Prediction Based on Parameters (RQ$_1$): To answer RQ$_1$, we use a dataset that consists of the latest release of each of the 105 projects that participate in the analysis. Since the employed method does not rely on timeseries analysis, we need to select a single version. Among the first and the latest version, we opt for the latest, since it is timelier relevant in terms of used technologies and the current status of the projects. The data from these 105 releases is randomly split into 90% of the projects, (94 projects) and the validation set that consists of the rest 10% of the projects, (11 projects). This process is repeated 10 times with 10 random training and test sets. The regression model that presents the highest accuracy is selected as the most appropriate one. To calculate the accuracy of the achieved regression models, we use R^2 and adjusted R^2 to evaluate the derived estimations.

Dependent Variable: Value of TD Principal
Independent Variables: TD Predictors of Table III
Method: Backward Regression Anlysis
Evaluation Process: 90%-10% Splitting for Training and Tetsing – repeated 10 times

Forecast TD Parameters (RQ$_2$): In the case of RQ$_2$ the evaluation process is performed for each of the 105 projects participating in the analysis, separately. In particular the available releases of each project are split into the training set, that consists of the 90% of the oldest releases and the testing set that consists of the 10% of the most recent releases. Then, the R^2 metric was employed to calculate the accuracy of the ARIMA model, produced by the training set, when applied to the observations of the test set, as well as the *Root Mean Square Error* (RMSE), *Mean Absolute Percentage Error* (MAPE), *Mean Absolute Error* (MAE), and *Normalized Bayesian Information Criterion* (Normal BIC) fit statistic measures.

Dependent Variable: TD predictors in some time steps
Independent Variables: TD predictors in previous steps
Method: ARIMA
Evaluation Process: 90%-10% splitting of versions. 10% of most recent versions are used for Tetsing

[7] https://github.com/zozas/jstd/blob/main/Manuscript%20dataset.zip

Prediction of Future TD Principal (RQ$_3$): With respect to RQ$_3$ the estimate is performed for each project separately by performing walk-forward validation [37]. The data set is split from a temporal viewpoint—see Figure 3. We adopted walk-forward validation [37] to evaluate the prediction accuracy as each model is updated when new observations are made available. The initial model is trained on an initial set of consecutive observations, and the accuracy is tested against future steps (the model prediction is evaluated against known value). The processes are repeated by moving the time window one-step forward (one month) to include the known value into the new training set. We applied 12 walk-forward validation processes to predict the next 12 iterations respectively. Furthermore, as a common practice in time series analysis [37], [39], we included a random walk model ARIMA (0,1,0) for the purpose of comparison with the trained model, as it. Furthermore, we included an ARIMA prediction model based on the known TD Principal values to compare the predicting accuracy of the proposed model to compare whether the suggested multivariate analysis (BSR model) is more accurate compared to univariate analysis [39]. The metrics used to evaluate forecasts for all models are the RMSE, MAE, and MAPE.

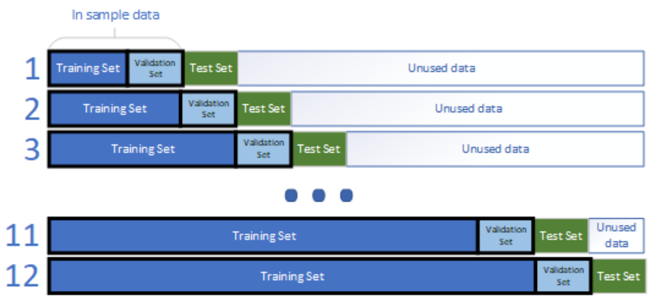


Fig. 3. Data Splitting Approach

Dependent Variable: Value of TD Principal in future
Independent Variables: Values of TD predictos in future
Method: Backward Regression Anlysis
Evaluation Process: Walk-forward validation

5 Results

5.1 TD Predictors Identification (RQ$_1$)

To identify the significant TD predictors for JS applications we have applied Backward Stepwise Regression. In the 8th step of the process, the value of adjusted R^2 is overall satisfactory, explaining 74% of the variance of the dependent variable. Through this process we have identified 8 predictors as the most prevalent ones, as presented in the folowing equation:

$$\begin{aligned}\boldsymbol{TD\ Principal\ Index} &\\ &= 1.060 - 0.063 \times NOC + 0.143 \times CLONE \\ &- 0.023 \times EVAL + 0.0738 \times NEW \\ &- 0.002 \times NOM + 0.123 \times PARM \\ &+ 0.13 \times ANONYM - \ 0.96 \times ARROW\end{aligned}$$

From the formula, we observe that the significant predictors of TD are solely source code metrics, while external quality and activity metrics do not participate in the equation. Out of the most significant TD predictors, four are general-purpose structural metrics (*NOC, CLONE, NOM, PARM*), while the rest are JS-related (*EVAL, NEW, NOM, ARROW*). The predictors that have a positive impact on TD (minimizing it) are *NOC, NOM, EVAL*, and *ARROW* while the predictors that increase TD are *CLONE, NEW, PARM*, and *ANONYM*. Table 4 presents the accuracy statistics for the TD Principal index for both the training and the validation set. Concerning the training set, the simple correlation R is 0.861, whereas the R^2 is 0.741 and the adjusted R^2 value is 0.710. As for the test set, the simple correlation R is 0.855 and the R^2, as well as the adjusted R^2 values, are 0.722 and 0.711 respectively. The regression model is statistically significant presenting a p-value < 0.05, performing with a high accuracy [14].

TABLE 4: MODEL ACCURACY

	R	R^2	Ad. R^2	Std. Error
Training set	.861	.741	.710	0.74
Test set	.855	.722	.711	0.82

5.2 Forecast of TD Predictors (RQ_2)

The next step is to forecast the values of TD predictors with the use of the ARIMA models. For each one of the projects participating in this study, we have applied ARIMA to forecast the values of the TD predictors (i.e., *NOC, CLONE, EVAL, NEW, NOM, PARM, ANONYM, ARROW*). We should mention that we have excluded the ARIMA (0,1,0) "Random walk" model to compare it with the selected model during the next research question. This approach is often followed by related literature [37]. For each predictor, based on our analysis, a number of competitive ARIMA models were identified. As a start, we have analyzed the goodness of fit and the residuals of these selected models, as well as the BIC metric to conclude to the models that present the best satisfactory fit. Table 5, displays the mean statistics over the forecasting models for all 105 projects.

TABLE 5: ARIMA MODELS ACCURACY

Metric	ARIMA	BIC	Ljung-Box		RMSE	MAPE	MAE
NOC	1,1,0	14.2	18	14	1211	71.773	91.07
CLONE	0,1,1	12.7	12	12	590	55.168	75.39
EVAL	1,1,1	16.4	284	9	3633	44.619	692.78
NEW	1,1,1	9.2	159	14	100	89.727	17.95
NOM	0,1,1	0.5	242	12	1	58.910	0.41
PARM	0,1,1	32.1	32	14	947244	20.259	20838
ANON	1,1,0	13.2	294	10	737	177.17	162.43
ARROW	1,1,0	13.7	186	9	988	104.50	171.80

The Ljung-Box Q test indicates that the residuals are independent and all models are suitable and well-adjusted to the time series, though a high significance value over 0.05 for all models. The Q number of the test indicates the randomness of the residual errors, while the DF number (Degrees of Freedom) is the number of model parameters that are free to vary when estimating debt. The statistic test follows a chi-square distribution with DF degrees of freedom. On these results, the data values are independent. The BIC number of the best fit models presents a lower score than the candidates. All models have been evaluated apart from the BIC number, by comparing the Root Mean Squared Error (RMSE), the Mean Absolute Percentage Error (MAPE), and the Mean Absolute Error (MAE), resulting in that the proposed models are good for all steps ahead. Furthermore, the coef-ficients present in all AR and MA parameters a p-value less than 0.05 indicating the significance of each weight as well as the predictive performance of the model. The dominating ARIMA models are of type:

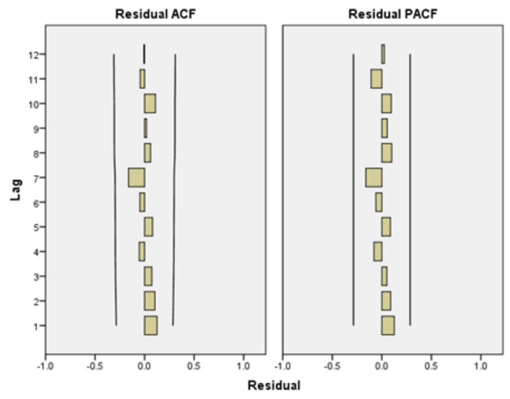
ARIMA (0,1,1) a simple exponential smoothing with growth for three predictors (*CLONE, NOM, PARM*).
ARIMA (1,1,1) additionally takes into consideration as an autoregressive term the value of 1 previous observation. Two predictors follow this model (*EVAL, NEW*).
ARIMA (1,1,0) a first-order autoregressive model concerning three predictors (*NOC, ANONYM, ARROW*).
All models include a maximum of two moving average terms. The next step is to present the mathematical specification of the above models, as presented in Table 6, where Y_t is the forecasting value, and εt is the white noise (forecasting errors), with zero means.

TABLE 6: MATHEMATICAL ARIMA MODELS

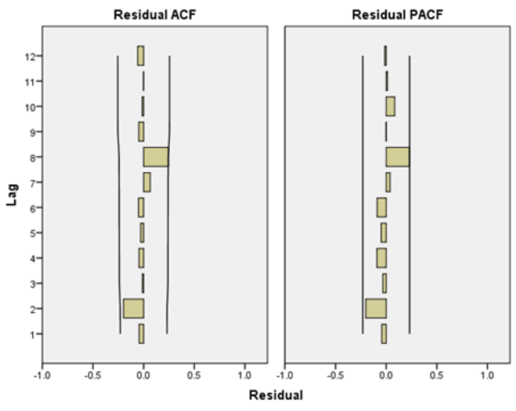
Metric	Model	Mathematical specification
NOC	1,1,0	$Y_t = 0.041\times Y_{t-1} - 0.057\times Y_{t-1} + 0.021\times e_t$
CLONE	0,1,1	$Y_t = - Y_{t-1} + 0.051\times e_t - 0.052\times e_{t-1}$
EVAL	1,1,1	$Y_t = 0.421\times Y_{t-1} - 0.126\times Y_{t-1} + 0.078\times e_t - 0.045\times e_{t-1}$
PARM	0,1,1	$Y_t = 2.726 - Y_{t-1} + 0.105\times e_t - 0.860\times e_{t-1}$
NEW	1,1,1	$Y_t = 0.282\times Y_{t-1} - 0.157\times Y_{t-1} + 0.030\times e_t - 0.057\times e_{t-1}$
NOM	0,1,1	$Y_t = - Y_{t-1} + 0.242\times e_t + 0.174\times e_{t-1}$
ANONYM	1,1,0	$Y_t = 1.206\times Y_{t-1} - 0.976\times Y_{t-1} - 0.711\times e_t$
ARROW	1,1,0	$Y_t = 0.476\times Y_{t-1} - 0.125\times Y_{t-1} - 0.051\times e_t$

Following the above, a residual analysis was performed to evaluate the model goodness of fit, as presented in Table 7. Concerning the residuals over time, as mentioned in Table 5, they do not display any obvious seasonality. The Q-Q plots in Table 7 follow the linear trend of the samples taken from a standard normal distribution which is an indication that the residuals are normally distributed. Furthermore, the ACF and PACF plots show a low correlation with lagged versions of the residuals. Based on the above, our model residuals are normally distributed and uncorrelated.

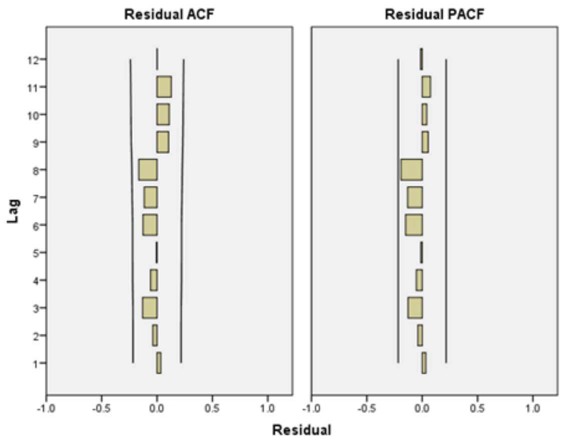
TABLE 7: RESIDUAL ANALYSIS



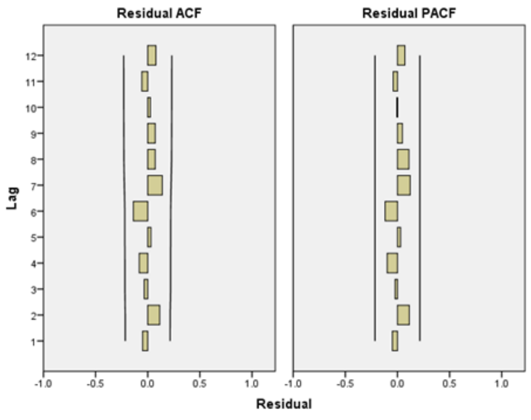
NOC 1,1,0



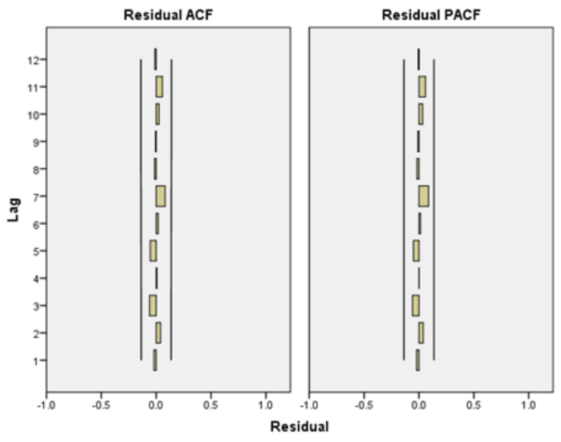
CLONE 0,1,1



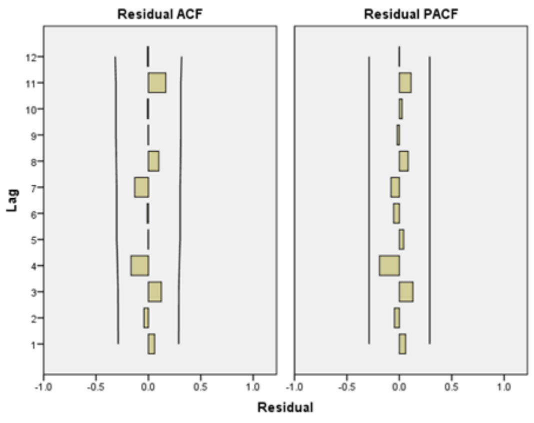
NEW 1,1,1



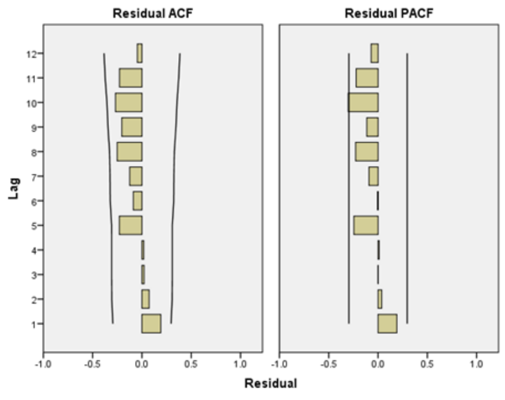
EVAL 1,1,1



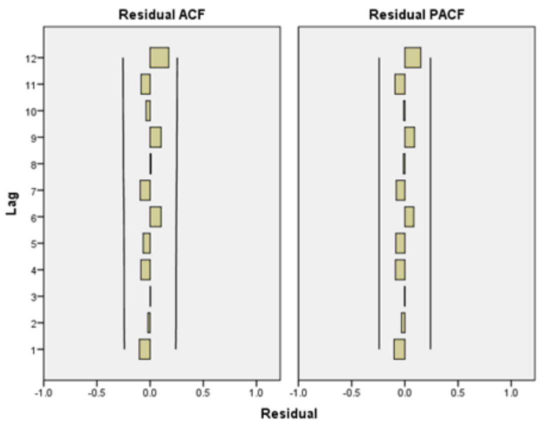
ANONYM 1,1,0



PARM 0,1,1



ARROW 1,1,0



NOM 0,1,1

5.3 Forecast of Technical Debt Principal (RQ_3)

Based on the forecasts of TD predictors performed in RQ_2 and the model presented in RQ_1, in this step we proceed to forecasting the TD Principal value of the latest observations for each project. By moving from (t) to (t+1) lags it is possible to predict multile iterations ahead. We have performed a walk-forward validation [37][39] with 12 iterations-ahead. The results will be tested on observation data that have not been used during training, to ensure the ability of the model to generalize and penalize errors.

Table 8 presents the estimation accuracy of the proposed methodology that is based on the combination of ARIMA and BSR and compares the accuracy of the method with the ARIMA model for forecasting directly TD Principal and with the Random walk prediction. All three models are evaluated against the observed technical debt values. The comparison is presented for multiple time steps into the future [39]. In the case of the Random Walk, the ARIMA (0,1,0) model have been applied as a common method followed by related literature [37]. In the case of the application of ARIMA for the direct forecasting of TD Principal, a regression analysis on the past TD values has been applied to forecast future TD Principal values [39]. For iterations 1 to 12 we evaluated the models by comparing the RMSE, MAPE and MAE. The results indicate that the proposed model is relative stable for all 12 steps ahead. In the case of RMSE, MAPE and MAE, our model presents lower errors than random walk.

TABLE 8
RANDOM WALK, REGRESSION AND ARIMA TD COMPARISON

Steps ahead	ARIMA random walk (0,1,0)				ARIMA TD Forecast				Proposed Methodology				Observed
	Forecast	RMSE	MAPE	MAE	**Forecast**	RMSE	MAPE	MAE	**Forecast**	RMSE	MAPE	MAE	**Observed**
1	5.96	5.97	8.51	5.93	5.02	5.60	16.2	5.53	**4.54**	**4.99**	**4.73**	**4.92**	4.18
2	5.12	5.97	6.95	5.93	4.65	5.64	16.0	5.58	**4.87**	**4.99**	**4.55**	**4.91**	4.55
3	5.18	6.01	6.69	5.97	4.72	5.68	16.7	5.61	**4.80**	**5.01**	**4.44**	**4.92**	4.65
4	5.25	6.05	6.45	6.01	4.76	5.72	17.5	5.65	**4.66**	**5.02**	**4.50**	**4.93**	4.67
5	5.33	6.09	6.14	6.05	4.32	5.77	18.3	5.70	**4.93**	**5.04**	**4.73**	**4.94**	4.92
6	5.49	6.13	6.02	6.09	4.32	5.81	18.7	5.75	**4.56**	**5.07**	**4.98**	**4.98**	4.95
7	5.64	6.16	5.73	6.12	4.36	5.88	19.0	5.82	**4.79**	**5.11**	**4.80**	**5.02**	4.98
8	5.43	6.20	5.27	6.15	4.55	5.94	19.4	5.88	**4.66**	**5.16**	**4.86**	**5.06**	5.15
9	5.77	6.24	5.26	6.20	5.13	6.02	19.9	5.96	**5.19**	**5.19**	**4.54**	**5.09**	5.35
10	5.91	6.28	5.07	6.23	5.03	6.07	21.1	6.02	**5.24**	**5.20**	**4.66**	**5.09**	5.42
11	6.06	6.30	4.78	6.26	4.62	6.14	22.1	6.09	**5.42**	**5.21**	**4.49**	**5.09**	5.61
12	6.02	6.32	4.50	6.27	4.33	6.22	22.5	6.17	**5.69**	**5.26**	**4.14**	**5.13**	5.68

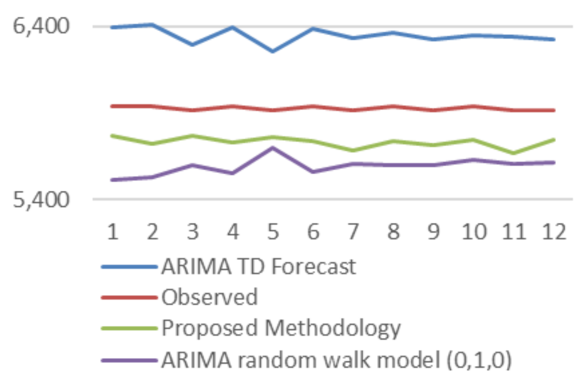


Fig. 4. Proposed approach (Node.js)

In Figure 4, we present the 12 forecasted steps based on the JavaScript project "*Node.JS*" (13th project as presented in Appendix). In all iterations, the proposed methodology outperforms the random walk and the regression model. The random walk model presents a fluctuation in step 5. Based on the results, the predictive power of all models decreases slowly in further future steps, while they tend in many cases in long-future prediction to present similar MAPE and APE error values. By average, for all projects, our model based on ARIMA outperforms both Random Walk as well as regression based on TD values.

6. DISCUSSION

In this study we proposed a methodology for identifying significant TD predictors; forecasting the values of these predictors in future observations and subsequently predicting TD Principal based on these values. The results of the study will be therefore discussed with respect to: (a) the ability of the methodology to forecast TD in future version of the system—see Section 6.1; (b) the predictors that affect TD Principal of JS applications—see Section 6.2; and (c) the implications to researchers and practitioners—see Section 6.3.

6.1 JSTD Predictive Power & Interpretation

Regarding the ability of JSTD to forecast the values of the TD predictors, our findings show that the ARIMA model is capable to forecast accurately future observations. This finding agrees with other studies that applied ARIMA for forecasting software quality attributes that are primitive (i.e., can be directly calculated from the source code) [19], [31]. It seems though that when using ARIMA to forecast TD that is a synthesized metric (i.e., its value depends on other metrics), the accuracy of the method deteriorates as we proceed in long-term predictions (increased future steps). This finding is in accordance with [28] and [41] who argue that machine learning techniques that are able to synthesize the values of independent variables to make predictions of TD Principal appear to be more sufficient and accurate in long-term forecasting compared to time-series models that take into consideration just the previously observed values of TD Principal.

6.2 TD Predictors' Analysis

Quantitative Findings & Theoretical Interpretation: The majority of traditional source code size metrics such as *NOC, CLONE, NOM, PARM* are found to be highly significant TD predictors. These results are aligned with related work that also appoints that software formulated in classes (*NOC*) [39] and methods (*NOM*) [40] presents less TD Principal while software with duplications and many parameters [40], [3] tend to be more complex and thus present more TD. Unlike other studies that associate TD with object-oriented metrics [22], in this study we were not able to confirm or solidly reject this finding, since for the majority of the participating projects we were not able to calculate all OO metrics as defined by Chimader and Kemerer [9]. This is indicative of the special nature of the JS applications, where the object-oriented paradigm is very loosely defined compared to stricter OO languages, such as Java; thus, typical OO metrics have not emerged as important TD predictors for the case of JS projects.
However, keeping in mind that OOP was first introduced with ECMA script in 2015, we can assume that strict OO

practices may have not yet spread to the community. Nevertheless, there are practices that are language-specific and involve directly or indirectly OO practices such as the creation of anonymous objects and functions. The *NEW* keyword creates new empty JavaScript objects in the run-time, including a constructor prototype object. The language compiler calls the constructor and returns the new object. *ANONYM* functions on the other hand can be invoked immediately, used as an argument to other functions, or assigned to a variable. Both anonymous functions and objects (*NEW* and *ANONYM*) metrics are found to increase TD Principal, probably due to the fact that they are associated with the creation of objects in the run-time, a fact that increases system complexity and deteriorates code readability. On the other hand, several controversial characteristics of the language such as the *ARROW* functions and the *EVAL* strings seem to decrease TD. The *ARROW* functions are syntactically compact functions that are created dynamically to evaluate an expression and return the result, without allowing binding to "*this*", "*super*" or "*new*" target keywords. In both cases, there is "hidden" source code that is incorporated indirectly and has a one-time-run and non-repeatable effect on the overall program. Despite the general opinion that these two aspects of the language should be carefully adopted and only in suitable cases, it seems that JS programmers make rational use of them.

Qualitative Interpretation based on Expert Opinion: One of the main limitations of the purely quantitative studies in software engineering is the usually narrow interpretation of the results, and the lack of the deep understanding of the findings. To proceed one step further than interpreting the quantitative findings from our point of view, we have seeked for the expert opinion of JS developers. To this end, we have organized a short survey with 15 medium experience JS developers (5-10 years of programming) and asked them to interpret our findings. As a first step, we asked them to confirm or oppose the identified relations between predictors and the level of TD in JS applications. The results are summarized in Figure 5. The results of the study suggest that the developers confirmed at a rate higher than 90% that *NOC*, *PARAM*, and *CLONE* metrics are related to TD. The most marginal evaluations (2 out of 3 developers are "seeing" the relation) were identified for *NOM* and *NEW*, closely followed by *EVAL* and *ARROW*.

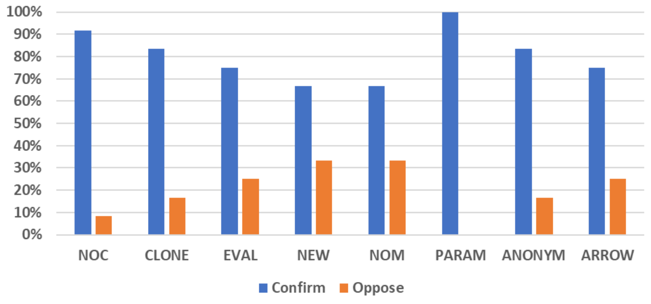


Fig. 5. Experts' Opinion on TD Predictors

Next, we asked them to present the reasons for which they believe a specific metric is an indicator of TD Principal. After analysing and synthesizing the opinions of the experts, the following interpretations have emerged as the most prominent ones:

- ***Size Metrics***: The developers highlighted that a larger system (in terms of **NOC**) is inevitable to accumulate more TD, due to the provided functionality (*needed complexity*). However, in terms of TD Principal density a system with more classes will probably yield less TD Principal. A similar argumentation has been provided for **NOM**: high levels of NOM and NOC is a sign of *good fragmentation* and *single responsibility* adoption. On the other hand, the number of **PARAMS** was well-accepted as an indicator of *method complexity* and thereof a sign of high TD Principal accumulation. It is worth mentioning that there was no developer who opposed this relation.
- ***CLONED*** lines of code: The develeopers suggested that code duplication is a *smell* itself, being responsible for potential *repetition of bugs and future code inconsestencies*. Additionally, they highlighted that apart from *making the codebase bigger*, in case of a change you should find and change all the cloned blocks, *hardening the maintainance process*.
- ***EVAL*** function: The developers showcased two views with the use of EVAL. First, it hides *complexity* into the code. However, more importantly the developers characterized EVAL as a *huge security threat* since malicious code can run inside the application without permission and third-party code can see the scope of the application, which can lead to possible attacks. In that sense, any use of EVAL will definitely need to be re-written in refactoring sessions. Nevertheless, at this point we need to note that security is a quality property whose relevance to TD Principal is controversial, since it is a run-time quality attribute.
- ***NEW*** keyword: The developers perceive NEW as possible source of TD, since it introduces additional *coupling* and hurts *modularity*. One developer mentioned that: "*…direct instantiations of objects make the client bound to a concrete class; thus, the system less modular. The use of factory patterns and dependency inversion are preferable…*". On the other side, some developers mentioned that using NEW is "*a more OO way of calling functions*", so it makes sense to using it. This is a probable reason for the marginal results of the qualitative evaluation.
- ***ANONYM*** functions: The developers that agreed that anonymous functions are harmful to system TD, supporting that each *function must have a name*, so as to be *easily spotted* in the code. If there are no names for functions one needs to rely on the willingness of the developer to add comments into the code ("*The mainetenance becomes very difficult without method names, especially if there are no comments*"). Additional concerns were raised in terms of *hindering reuse* for similar purposes. On the other end, some developers mentioned that the use of *anonymous functions reduce the size of the code* and therefore can reduce TD Principal. However, usually this decrease is low in absolute values and TD issues are not identified in method signatures.

- ***ARROW*** functions: Although the rationale of arrow functions and anonymous funtions is similar, the developers validated the positive impact of arrow functions in TD, highlighting mostly on *hiding complexity* and *reducing code duplication*. An important difference in the use of the two is the fact that arrow functions are usually *smaller in size* and *simpler*, compared to anonymous functions, which can be *lengthier* and usually *nested as arguments* in other functions, *hurting* the *readability* of the code.

6.3 Implications to Researchers and Practitioners

Under this scope, the overall findings of the current research indicate that the proposed methodology for predicting TD Principal of JavaScript applications based on the combination of ARIMA and BSR model can offer useful impli-cations to both researchers and practitioners.

We encourage ***researchers*** to adopt JSTD and the proposed methodology and enhance them by incorporating expert knowledge in the definition of the regression model used, in the parametrization of the ARIMA model, and in the inclusion / exclusion of the TD independent variables. Additionally, researchers are encouraged to evaluate the proposed methodology on closed source software. Additionally, given the importance of cross-project validation, we encourage a replication of the evaluation of RQ2-RQ3 on groups of projects that the training set (RQ1) is different from the testing set. Furthermore, based on other research efforts [41], we encourage researchers to enhance the current methodology on JS class-level basis considering that while JS is capable of object-oriented programming, few research efforts focus on this subject [18]. The latter would be interesting to observe differences concerning TD significant parameters. Another interesting implication for researchers would be to explore if the groups of parameters studied in this work present some kind of uniform and joint effect on TD Principal, exploiting fitting analysis methods (e.g., factor analysis). Finally, researchers are encouraged to experiment with different forecasting / prediction techniques in the first two steps of the approach and compare the plethora of the available statistical, ML, DL algorithms that exist in the litertaure.

Regarding ***practitioners***, we encourage them to adopt the proposed methodology and tool in order to be able to predict TD Principal in future observations. Also, it is important for the community to understand that TD is project-specific and language-specific. Therefore, the use of language specific-tools on software repositories that are project-specific or company-specific may boost the ability of a company to manage its own TD.

7 Threats to Validiy

In this section, we discuss the threats to validity that we have identified for this study. Regarding *Conclusion Validity* that refers to how reasonable are the findings of the analysis, we mention that regarding the statistical power of the results, we calculated a variety of regression and ARIMA models for all projects participating, in order to validate the proposed approach. The model comparisons based on statistical tests, as described in Sections 3.1 and 3.2, showed a high level of accuracy in the proposed methodology. Regarding the error rate probability problem, we selected common, pre-defined validation procedures (KPSS test, Ljung-Box test). Furthermore, regarding the heterogeneity of data, we used project-specific datasets to ensure the relativity of the data included in the analysis. The set of metrics that we have selected to use in our study for quantifying technical debt (see Table 3) belong to well-known metric suites. These include the most popular suites [15] Chidamber & Kemerer metrics [9] and the Li & Henry suite of metrics [25]. However, some metrics cannot be applied to JavaScript language programs due to the language nature (lack of class notation formalization or even class support to early versions [44]). In expansion to this, metrics-based specifically on the JavaScript programming language were introduced, such as the number of obfuscation incidents [42], the version of ECMAScript applied [18], the number of anonymous and arrow type functions, and the usage of language-specific keywords (like *"NEW"*, *"WITH"* and *"EVAL"*) [20]. Finally, the estimated ground-truth TD principal value highly depends on the threashold (i.e., 5%) that we have set to consider agreement among TD benchmarker tools. The use of a different threashold (stricter—e.g., 1% or more relaxed—e.g., 10%) might have yield to a different result. Nevertheless, to the best of our knowledge there is no other dataset to reuse having commonly agreed values of TD principal.

Regarding *Internal Validity* an attempt to formalize a methodology over TD based on quality parameters over time is presented. The causal relationships identified in this study, as documented in Section 6.1, can be considered as indicators of possible cause-effect relationships among the participating parameters without excluding other relationships between variables that may affect the maintainability of applications that did not participate in this study.

Concerning *Reliability*, we believe that the replication of our research is safe, and the overall reliability is ensured. The process that has been followed in this study has been thoroughly documented in Section 4, to be easily reproduced by any interested researcher. All metrics calculated, as well as the overall extraction of the defined data set was performed with the use of widely used research tools, as documented in Section 4.1, as well as the JSTD tool developed for our research and is based on the source code of these tools. In either case, future verifications of the accuracy of this tool would be valuable. Nevertheless, we cannot guarantee the replicability of results if different study design decisions were made (e.g., using a 6-month or 24-month sliding window for RQ3).

Concerning the *External validity* and in particular the generalizability supposition, changes in the findings might occur if the application data set for which the sample releases were analyzed is altered in both projects and size. Future replications of this study, on data from other projects, would be valuable to verify these findings.

8 Conclusion

Technical Debt has a large impact on software quality and maintenance cost over time. Forecasting this concept is considered to be critical for managing the software lifecycle, and both are influenced by a variety of parameters. In this study, we presented and validated a methodology to model the predictors related to the technical debt of JavaScript applications by applying Backwards Stepwise Regression and Autoregressive Integrated Moving Average methods. We exploited the influence of these predictors, the predictive power of the constructed model, and the ability to forecast both concepts over time.

To investigate the validity of the proposed approach we performed a case study on 105 JavaScript applications analyzing in total 19,636 releases of JavaScript applications, testing the estimation accuracy of the derived models on all releases. Furthermore, we developed a tool to automate the whole process of data retrieval, storage, evaluation, and analysis. The results from the case study suggested that the proposed approach is capable of providing stable accurate forecasts with high accuracy. Furthermore, JavaScript-explicit features as the "*new*" and "*eval*" keywords, as well as the "*anonymous*" and "*arrow*" functions, are among the features of JavaScript language that affect TD.

Overall, JSTD is a tool, that can be a useful for practitioners when monitoring the TD Principal of JS applications. JSTD can help towards the creation of TD Principal prediction models, trained on a large number of project releases, by just accessing a GitHub repository. On the other hand, the results of the analysis show that the forecasts actually require information from just 1 or 2 previous releases and therefore the need of thousands of releases is actually not a prerequisite. In this manuscript we experimented on a large number of releases in order to exemplify the proposed approach, but we believe, that the existence of a large number of historical releases is not necessary in order to produce accurate results. Based on these results, a methodology and tool have been provided for the benefit of both researchers and practitioners. Future replications of the current study, as well as the comparison of the provided methodology would be valuable.

Acknowledgment

This work has received funding from the European Union's H2020 research and innovation programme, under grant agreement: 871177 (SmartCLIDE).

References

[1] Amanatidis, T., Chatzigeorgiou, A., Ampatzoglou. A. 2017. The relation between technical debt and corrective maintenance in PHP web applications. Inf. and Soft Technology.

[2] Amanatidis, T., et al., (2020). Evaluating the agreement among technical debt measurement tools: building an empirical benchmark of technical debt liabilities. Empir Software Eng 25, 4161–4204.

[3] Alégroth, E., Steiner, M., Martini, A. 2016. Exploring the Presence of Technical Debt in Industrial GUI-Based Testware: A Case Study, 2016 IEEE 9th Int. Conf. on Soft. Testing, Verification and Validation Workshops, pp. 257-262.

[4] Amin, A., Grunske, L., Colman, A. 2013. An approach to software reliability prediction based on time series modeling, Journal of Systems and Software, 2013, 1923-1932.

[5] Avgeriou, P., et al. An Overview and Comparison of Technical Debt Measurement Tools, IEEE Software, vol. 38, no. 3, pp. 61-71, May-June 2021.

[6] Baldassari, B. 2012. SQuORE: a new approach to Software Project Quality Measurement, Intern. Conference on Software & Systems Engineering and their Applications, 25 Oct. 2012

[7] Borboudakis, G., Tsamardinos, I. 2019. Forward-backward selection with early dropping. J. Mach. Learn. Res. 2019.

[8] Box, G., Jenkins, G., 1976. Time Series Analysis: Forecasting and Control. Holden Day, San Francisco.

[9] Chidamber, S., Kemerer, C. 1994. A Metrics Suite for Object-Oriented Design", IEEE Trans. Software Engineering, vol. 20.

[10] Chatzimparmpas, A., Bibi, S., Zozas, I., Kerren, A. 2019. Analyzing the Evolution of Javascript Applications. ENASE

[11] Cunningham, W. 1992. The WyCash Portfolio Management System. In Addendum to the proceedings on Object oriented programming systems, languages, and applications. 29-30.

[12] Curtis, B., Sappidi, J., Szynkarski, A. 2012. Estimating the Principal of an Application's Technical Debt. IEEE Software.

[13] Draper, N., Smith, H. 1981. Applied Regression Analysis, 2d Edition, New York: John Wiley & Sons, Inc.

[14] Fahrmeir, L., Kneib, T., Lang, S., & Marx, B. 2013. Regression: models, methods and applications. Springer Science.

[15] Fioravanti, F., Nesi, P. 2001. Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems. IEEE Transactions on software engineering 2001.

[16] Gallaba, K., Mesbah, A., Beschastnikh, I. 2015. Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript, 2015 ACM/IEEE Int. Symp. on Empirical Software Engineering and Measurement (ESEM), pp. 1-10.

[17] Gaudin, O. 2009. Evaluate your technical debt with sonar, Sonar, Jun 11, 2009.

[18] Gizas, A., Christodoulou, S., Papatheodorou, T. 2012. Comparative evaluation of javascript frameworks." 21st International Conference on World Wide Web, pp. 513-514. ACM.

[19] Goulão, M., Fonte, N., Wermelinger, M., Brito e Abreu, F. 2012. Software Evolution Prediction Using Seasonal Time Analysis: A Comparative Study. European Conference on Software Maintenance and Reengineering, CSMR.

[20] Hafiz, M., Hasan, S., King, Z., Wirfs-Brock, A. Growing a language: An empirical study on how (and why) developers use some recently-introduced and/or recently-evolving JavaScript features, Journal of Systems and Software, 2016.

[21] Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyev, S., Fedak, V., Shapochka, A. 2015. A Case Study in Locating the Architectural Roots of Technical Debt, 37th IEEE Int. Conf. on Software Engineering, 2015, pp. 179-188.

[22] Kosti, M., Ampatzoglou, A., Chatzigeorgiou, A., Pallas, G., Stamelos I., Angelis, L. 2017. Technical Debt Principal Assessment Through Structural Metrics, 43rd Software Engineering and Advanced Applications (SEAA), pp. 329-333.

[23] Kumar, S., Bahsoon, R., Chen, T., Buyya, R. 2019. Identifying and Estimating Technical Debt for Service Composition in SaaS Cloud, 2019 IEEE International Conference on Web Services (ICWS), pp. 121-125.

[24] Letouzey, J.L. 2012. The SQALE method for evaluating technical debt. 3rd International Workshop on Managing Technical Debt, 2012, Zurich, Switzerland, June 5, 2012. 31–36.

[25] Li, W., Henry, S. 1993. Maintenance Metrics for the Object-Oriented Paradigm, 1st Int. Soft. Metrics Symp., 52-60.

[26] Lin Y., Li M., Yang C., Yin C. 2017 A Code Quality Metrics Model for React-Based Web Applications. Intelligent Computing Methodologies. ICIC 2017., vol 10363. Springer, Cham.

[27] Ljung, G., Box. G., 1978. On a measure of lack of fit in time series models, Biometrika, 65,2, August, pp. 297–303.
[28] Mathioudaki M., Tsoukalas D., Siavvas M., Kehagias D. 2021 Technical Debt Forecasting Based on Deep Learning Techniques. Comp. Science & Applications, v.12955, Springer.
[29] Misra, S., Cafer, F. 2012. Estimating Quality of JavaScript. International Arab Journal of Information Technology.
[30] Mirghasemi, S., Barton, J., Petitpierre, C. 2011. Naming anonymous javascript functions. Object oriented programming systems languages and applications companion, 277–288.
[31] Raja, U., Hale, D.P., Hale, J.E. 2009. Modeling software evolution defects: a time series approach. J. Softw. Maint. Evol.: Res. Pract., 21: 49-71.
[32] Richards G., Hammer C., Burg B., Vitek J. 2011 The Eval That Men Do. In: Mezini M. (eds) ECOOP 2011 – Object-Oriented Programming, vol 6813. Springer, Heidelberg.
[33] Runeson, P., Host, M., Rainer, A., Regnell, B. 2012. Case study research in software engineering: Guidelines and examples. John Wiley & Sons.
[34] Salamea, M., Farré, C. 2019. Influence of Developer Factors on Code Quality: A Data Study, 2019 19th International Conference on Software Quality, Reliability and Security, 120-125
[35] Yogesh S., Kaur, A., Ruchika, M. 2009. Comparative analysis of regression and machine learning methods for predicting fault proneness models. Int. J. Comput. Appl. Technol. 2009.
[36] Schwarz, G. 1978, Estimating the dimension of a model, Annals of Statistics, 6 (2): 461–464.
[37] Stone, M. 1974. Cross-validatory choice and assessment of statistical predictions, Journal of the Royal Statistical Society: Series B (Methodological), vol. 36, no. 2, pp. 111–133.
[38] Tsoukalas, D., et al., 2018. Methods and Tools for TD Estimation and Forecasting: A State-of-the-art Survey. IEEE Conf. on Intelligent Systems 2018: 698-705
[39] Tsoukalas, D., et al., 2019. On the Applicability of Time Series Models for Technical Debt Forecasting, 15th China-Europe Int. Symp. on Soft. Engin. Education.
[40] Tsoukalas, D., Kehagias, D., Siavvas, M., Chatzigeorgiou, A. 2020. Technical debt forecasting: An empirical study on open-source repositories. J. Syst. Softw. 170: 110777
[41] Tsoukalas, D., Mittas, N., Chatzigeorgiou, A., Kehagias, D., Ampatzoglou, A. 2021. Machine Learning for Technical Debt Identification, IEEE Transactions on Software Engineering.
[42] Wang, Y., Cai, W., and Wei, P. 2016. A deep learning approach for detecting malicious JavaScript code. Security Comm. Networks, 9: 1520– 1534.
[43] Wang, M., Wright, J., Buswell, R., Brownlee, A. 2013. A comparison of approaches to stepwise regression for global sensitivity analysis used with evolutionary optimization. 13th Conf. of the Int. Build. Perf. Sim. Assoc. 2551-2558.
[44] Wei, X. 2013. Verification and Validation of JavaScript. Doctoral thesis, Durham University.
[45] Zozas, I., Bibi, S., Ampatzoglou, A., Sarigiannidis, P.G. 2019. Estimating the Maintenance Effort of JavaScript Applications. SEAA 2019: 212-219
[46] A. Ampatzoglou, N. Mittas, A. A. Tsintzira, A. Ampatzoglou, E. M. Arvanitou, A. Chatzigeorgiou, P. Avgeriou, L. Angelis, "Exploring the Relation between Technical Debt Principal and Interest: An Empirical Approach", Information and Software Technology, Elsevier, 128, 2020.

Ioannis Zozas is currently a Ph.D. student from 2017 to the present, at the department of Electrical and Computer Engineering, University of Western Macedonia. Received a bachelor's and master's degree from the department of Applied Informatics, University of Macedonia. Currently working in the banking sector. His research interests are software source code quality, JavaScript, and Software Engineering.

Stamatia Bibi received the B.Sc. degree in informatics and the Ph.D. degree in software engineering from the Aristotle University of Thessaloniki, Greece, in 2002 and 2008, respectively. She is currently an Assistant Professor in software engineering with the Department of Electrical and Computer Engineering, University of Western Macedonia, Kozani, Greece. Her research interests include process models, software effort / cost estimation, quality assessment, technical debt management, and cloud computing.

Apostolos Ampatzoglou received the B.Sc. degree in information systems, in 2003, the M.Sc. degree in computer systems, in 2005, and the Ph.D. degree in software engineering from the Aristotle University of Thessaloniki, in 2012. He is currently an Assistant Professor with the Department of Applied Informatics, University of Macedonia, Greece, where he carries out research and teaching in the area of software engineering. Before joining the University of Macedonia, he was an Assistant Professor with the University of Groningen, The Netherlands. He has published more than 100 articles in international journals and conferences, and is/was involved in more than 15 research and development ICT projects, with funding from national and international organizations. His current research interests include technical debt management, software maintainability, reverse engineering software quality management, open-source software, and software design.