The Journal of Systems & Software 194 (2022) 111481

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Using deep temporal convolutional networks to just-in-time forecast technical debt principal☆

Pasquale Ardimento [a], Lerina Aversano [b], Mario Luca Bernardi [b], Marta Cimitile [c], Martina Iammarino [b,*]

[a] University of Bari Aldo Moro, Department of Informatics, Bari, Italy
[b] University of Sannio, Department of Engineering, Benevento, Italy
[c] UnitelmaSapienza University, Rome, Italy

ARTICLE INFO

Article history:
Received 19 November 2021
Received in revised form 18 June 2022
Accepted 4 August 2022
Available online 12 August 2022

Keywords:
Technical debt
Technical debt forecasting
Process metrics
Software quality metrics
Deep learning
Temporal convolutional network

ABSTRACT

Technical debt is a widely used metaphor to summarize all the consequences of poorly written code. Managing technical debt is important for software developers to allow adequate planning for software maintenance and improvement activities, such as refactoring and preventing system degradation. Several studies in the literature investigate the identification of technical debt and its consequences. This work aims to explore a deep learning approach to just-in-time predict the impact on technical debt when changes are performed on the source code. In this way the developer can work better, trying to improve the quality of the code that is being modified. Knowing what the TD trend will be in just-in-time source code with the change made is the key to avoiding a project taking a long time to remediate or improve. The model exploits the knowledge of quality and ad-hoc process metrics evolution over time. To validate the approach, a large dataset, including metrics evaluated from commits of ten Java software projects, was built. The results obtained show the effectiveness of the proposed approach in predicting the Technical Debt accumulation within the source code.

1. Introduction

In recent years, numerous studies have focused on the Technical Debt (TD) of software systems. Technical debt was introduced by Cunningham (1992a) as a metaphor to describe the impact of shortcuts taken during development and possible complications that arise in a project if adequate actions are not taken to maintain the best quality overall solution. It indicates the extent of the cost of reworking a solution caused by choosing an easy but limited solution. Similar to what happens in the financial world, where even interest must be paid to pay off a debt, the effort to recover a project developed without a correct methodology can also increase significantly over time if action is not taken promptly. The concept of TD, therefore, represents the additional development work that occurs when a mediocre code is implemented in the short term and indicates the difference we see between what has actually been achieved and what is ideally necessary to consider a task completed.

There are different types of TDs, related to design, architecture, documentation, or testing. In this study, we have focused on code debt, which manifests itself in the form of poorly written code and requires developer intervention with refactoring (Stopford et al., 2017; Aversano et al., 2020a). In particular, it refers to faulty pieces of code, such as duplicates, errors that reduce readability, or that show poorly organized logic (Tom et al., 2013). When writing code, it happens very often that developers take shortcuts to reach the goal in a shorter time. Therefore, one of the causes that induce the introduction of the TD in the source code is represented by the rapidity of development that implies a code characterized by some deficiencies that a programmer will have to rework or clean up later. These deficiencies affect the overall quality of the code by compromising its functionality because although the software may still work, it cannot reach its full potential until the deficiencies within are addressed (Digkas et al., 2021). The main problem with TD in the code is that, like many debts, if left behind it can give rise to interest which, if accumulated, could fatally compromise the quality of the project (Cunningham, 1992b; Zazworka et al., 2011).

In the literature, several studies were conducted on methodologies and techniques that can help developers and project managers to estimate and measure the TD (Alves et al., 2016b; Letouzey, 2012). Li et al. (2015a) carried out systematic mapping of the TD and its management. From these studies, it emerged

☆ Editor: Matthias Galster.

* Corresponding author.
E-mail addresses: pasquale.ardimento@uniba.it (P. Ardimento), aversano@unisannio.it (L. Aversano), bernardi@unisannio.it (M.L. Bernardi), marta.cimitile@unitelmasapienza.it (M. Cimitile), iammarinomartina@gmail.com (M. Iammarino).

https://doi.org/10.1016/j.jss.2022.111481
0164-1212/

that there is a need to conduct more empirical analysis with high-level evidence on the entire management process. Recently, Yli-Huumo et al. (2016) investigated how software development teams manage technical debt in a real-world environment. To identify the procedures and techniques for debt management, they interviewed 25 people, bringing to light that management is carried out at different levels. This study allowed them to develop a framework to support developers, capable of explaining the activities, processes, stakeholders, and responsibilities of TD management.

However, TD measurement may not be sufficient to avoid total compromise of the quality of a software system. Instead, forecasting the future value would be of great support because it would facilitate development and maintenance.

According to Isaac Griffith et al. (2014), the management of the TD should be about identifying, monitoring, and how to pay the debt. The tools currently in use can identify TD by measuring its value but are not able to predict how it changes. Therefore, this could be a problem, because a developer may find that they have introduced the TD when its value is already too high.

Previous studies have been conducted on predicting the evolution of quality properties or other characteristics closely related to TD (Aversano et al., 2020b). In particular, in this study, the authors analyzed the trend of software metrics (CK) when removals of self-admitted technical debt occur. The results obtained show that when there is a worsening of the measure of technical debt there is also a worsening of software quality metrics. This then indicates the effect of TD on the quality of the software and highlights how the quality metrics chosen in our feature model can be considered as an indicator of debt in the source code. But as far as we know, to date very few studies have focused on predicting the impact of change on TD, therefore further studies are needed in this direction. More specifically, Skourletopoulos et al. (2014) has created two models that allow us to identify which aspects could lead to the accumulation of interest. Tsoukalas et al. (2019) tested a model based on time series and managed to predict the TD 8 weeks into the future, and Tsoukalas et al. (2020) applied several Machine Learning models for predicting the TD value, considering a limited set of commits of the software projects considered.

The goal of this work, therefore, is to develop a deep learning-based model for predicting the impact of software changes on TD just at the time when the software maintainers are modifying the source code (just-in-time). It involves the adoption of fine granularity, a collected commit by commit, of the whole history of software systems. By leveraging the metrics collected for previous commits, we aim to predict the trend of the TD before the commit is even changed. This means that considering the current commit, the model is able to predict whether the TD value will increase, decrease or remain stable in the commit following the modified one, based on the trend of the other metrics of the code. In this way the developer can work better on his modification, because aware of the impacts that this will produce on the TD, he can try to improve the quality of the code he is modifying. Just-in-time knowledge of what the TD trend will be in the source code is the key to avoiding it taking a long time to heal a developed project.

To train and validate our model a large dataset has been created. In particular, the data were extracted from the source code repositories of the ten software systems considered for the study and from the measurement of several indicators such as source code quality metrics, process metrics, and TD. The dataset includes the history of the analyzed systems with a fine-grained assessment. In particular, it contains commits by commits (that is change after change) metrics evaluations of the software systems analyzed. Therefore, the proposed approach allows us to predict the change of TD relating it to a change in the software. The training and validation of our model take place on this large dataset that was created specifically for it. More specifically, we investigate (i) whether our model is efficient in predicting the impact on the TD commit by a commit with the optimization of some deep learning network hyperparameters; (ii) which of the metrics considered and connected to the impact on the TD allow a more accurate forecast, and finally (iii) if our model is also effective to perform a cross-project prediction.

This article extends a preliminary study we performed (Aversano et al., 2021) in the following directions:

- replication of the experiments on previously analyzed systems plus the addition of 6 new systems
- adding process metrics to the features model and performing feature selection for identifying the most relevant functions for TD just-in-time prediction
- performing cross-project experiments that would allow the application of pre-existing forecasting models on new projects that do not have a broad enough history.

The document is divided into 10 main sections. More specifically, the next section describes similar works already in the literature, comparing them with this study. Section 2 offers an overview of the topics on which the work is based while Section 3 discusses the main related work. Section 4 describes the proposed approach whereas Section 5 shows the details of the experiment conducted. In Section 6 we report the results obtained, and in Section 7 a discussion of them. Finally Section 8 reports implications, Section 9 discusses threats to validity, and Section 10 reports the conclusions.

2. Background

2.1. Deep learning algorithms

Deep Learning (DL) represents a segment of Machine Learning within which it is possible to enclose the Artificial Intelligence algorithms designed both at a structural and functional level to replicate the characteristics of the human brain and beyond. Compared to Machine Learning where the algorithms are no longer scalable when reached a certain level of performance, Deep Learning systems improve their performance as data increases.

The goal of an approach based on Deep Learning is to create neural networks through which machines can learn data, even peripherally preprocessed through edge computing in a cloud-based context, exploiting the action of algorithms designed to offer a representation of data themselves.

In the case of artificial neural networks, learning is not only automatic but also deep, hierarchical, and adaptive, this is possible thanks to a multilevel architecture where the passage of information between the layers allows to simulate the abstraction capacity of biological neural networks, a process by which machines can elaborate formulas and identify symbols that allow them to solve complex problems (Deng et al., 2014).

A Deep Learning algorithm provides a representation of reality with which to define patterns that allow the activation of different behaviors based on the inputs coming from the reference context and its characteristics. To reach the solution of the problem, the algorithms provide for the selection and classification of the data with greater importance.

The source data can be labeled by providing a reference to their nature (supervised learning) or a model (unsupervised) can be adopted in which the algorithms are fed via inputs, leaving the burden of classification to the system. In both cases, the goal is to maximize the performance of the training sessions focused on the principle of trial-and-error. In a neural network, the passage of data between the layers can in fact involve even the less accurate

information, for this reason, it becomes essential the intervention of the algorithms for the backpropagation which, in the presence of an error, remove the origin by going up all the levels which led to his generation.

In particular, two phases are foreseen to train the DL network, forward and backward. During the first, the activation signals of the nodes are propagated from the input level to the output level; during the second, weights and biases are changed to improve overall network performance.

2.1.1. Temporal convolutional networks (TCN)

The temporal convolutional network (Albawi et al., 2017; Ardimento et al., 2020), briefly called TCN, represents a variation of convolutional neural networks that employ convolutions and random dilations so that it is adaptable for sequential data with its temporality and large receptive fields.

TCN is a combination of simplicity, autoregressive prediction, and very long memory compared to recurring architectures with the same capacity. Its main advantages are parallelism, flexible size, stable gradients, and low memory requirement. While in the residual neural networks future predictions must always wait for the previous ones, this type of network allows the execution of convolutions in parallel because at each level it uses the same filter. Secondly, we speak of flexible dimensions of the receptive field because it is able to modify them, thus making it more suitable for different domains, and able to better control the memory of the model. Again, it is based on stable gradients because it has a back-propagation path different from the temporal direction of the sequence. Finally, training is characterized by a low memory requirement.

To make a prediction they can look very far into the past, so they are built to have very long historical dimensions. To do this, a combination of very deep augmented networks with residual layers and dilated convolutions is employed. It has been designed taking into consideration some fundamental characteristics:

- Sequence Modeling: Includes autoregressive prediction in which you try to predict a signal based on its past, then set the target output to be simply the shifted input by a time step.
- Causal Convolutions: Assumes that there are no losses from the future in the past and that the output produced is the same length as the input provided. Therefore, the output at time t is only convoluted with elements from time t and earlier in the previous level. Instead, to reach the second point, a fully convolutive 1D network architecture is used, in which the input layer and the hidden layers have the same sizes, and to make the subsequent layers also have the same length a fill is added. The convolutional operations used by the TCN architecture have been addressed in Bai et al. (2018).
- Residual blocks: Each block of this type has two layers of dilated causal convolution, and the results of the final convolution are added to the inputs to obtain the outputs of the block. If the second causal convolution has a different number of input channels, which correspond to length, and filters, which correspond to width, a 1D convolution must be applied to the inputs before adding the convolution outputs to match the widths .

2.2. Technical debt

Technical debt is a metaphor invented by Cunningham (1992a) to describe the delayed maintenance costs in software development and to represent all choices, conscious or unintentional, that reduce the maintainability and evolvability of the code, therefore the speed and quality. Since entropy increases in the absence of control, keeping debt at an acceptable level requires expertise and commitment. Like any other type of debt, TD implies convenience now and higher interest payments in the future.

Several studies have investigated the effects and consequences of TD. Rios et al. (2020) recently published a very interesting study, in which they examined all possible causes and effects, identifying 78 causes and 66 effects, and organizing the identified set into probabilistic cause and effect diagrams. In particular, the results show that there are six macro-categories of the most common effects that occur as a result of debts related to development, external and internal quality, people, planning and management, and organizational issues. For each of these macro-categories, sub-categories have been identified such as the difficulty in carrying out tests for the first macro-category, or delays in the release relating to problems related to management and planning.

When we talk about TD we are therefore referring to a set of design choices which, accumulating over time, make the system difficult to maintain and evolve. Hence, it is something that negatively affects the internal and non-functional characteristics of a system, in particular its maintainability and evolvability (Li et al., 2015b). This happens because software development tools and standards are constantly evolving, regardless of the reference resources, and this leads to invalidating the original design and construction choices (Boehm and Beck, 2010).

Overall, it highlights that in a software project, from its design to source code, there is no adequate solution, and is used to describe maintenance problems caused by (i) deadline limits, when hasty delivery of the project results in software design problems; (ii) budget limits; (iii) poor development choices or poor development skills, because the development team should have conventions so that there are no differences, or at most not significant in design and implementation (Rios et al., 2018).

To avoid TD, quality assurance activities are required, therefore the process of checking the code for problems (before release, after recurring planning, or through refactoring). This demands a technique to modify the internal structure of portions of code without modifying their external behavior (Fowler, 1999).

There are currently several tools available for TD detection (e.g. Ndepend,[1], Teamscale,[2] Cast[3]), but in our study we have used Sonarqube, the most used tool (Avgeriou et al., 2021), which provides the value in time/person, necessary to calculate the TD. To estimate this value it uses the method of Letouzey (2012), the Software Quality Assessment Based on Lifecycle Expectations (SQALE), that refers to TD principal without interest. This approach allows identifying the elements of a software system that introduce debt into the code, evaluating the consequences on quality characteristics, quantifying the costs of correcting the factors that make up the debt. It is based on a quality model associated with a model of analysis, whose operation consists in assigning a repair cost for each rule and calculating the cost at the level of the various elements that make up the software system defining the aggregation rules.

3. Related work

The TD is perceived as a debt precisely because it lacks what is expected. It can be considered as the gap between what has been achieved and what would be needed. Therefore, if left behind it can give rise to interests that, if accumulated, could compromise the quality of the project.

[1] https://www.ndepend.com.
[2] https://www.cqse.eu/en/teamscale/overview/.
[3] https://www.castsoftware.com.

For this reason, its identification has been the subject of numerous studies (Li et al., 2015a; Yli-Huumo et al., 2016), that have carried out systematic mapping of the TD management.

On the other hand, however, none of the tools currently in use can predict TD and its behavior in code. This would be of fundamental importance as TD can be considered as an index of the quality of the source code and its knowledge could facilitate the implementation and maintenance of the software.

In this regard, numerous studies have been conducted on the relationships between TD and quality metrics (Wehaibi et al., 2016; Li et al., 2014; de Freitas Farias et al., 2015).

Other studies, on the other hand, have analyzed the impact that the evolution of the TD has on the qualitative aspects of software development. In particular, Chatzigeorgiou et al. (2015) coined the term "breaking point", which indicates the moment in which the benefit is exceeded by the cost, with a consequent increase in accumulated interest. For the need to be aware of the evolution of the TD, Ampatzoglou et al. (2018) proposed a framework that evaluates the breaking point of the source code modules. It is an in-depth decision-making tool to support managers to facilitate the decision-investment process to improve software quality.

Therefore, estimating the point at which the software product could become unreachable is critical to predicting the evolution of TD capital and interest. Therefore, to improve the TD reimbursement strategy, it would be appropriate to develop an effective forecasting model that allows monitoring and analyzing the constant evolution of the TD.

To date, while a lot of research has been conducted that focuses on predicting the maintainability of a software system through quality metrics, Chug and Malhotra (2016), Pandey et al. (2021) and Nagappan et al. (2006), on the other hand, many questions are still open in the field of TD prediction, a few authors have addressed this issue.

Skourletopoulos et al. (2014) created two models that support technical debt forecasting in the cloud service tier. The first approach uses COCOMO as the software cost model, while the second calculates the TD in the case of a SaaS cloud lease. These allow you to identify which aspects could lead to the accumulation of interest.

Tsoukalas et al. (2019) conducted an empirical analysis of long-term open source software projects to predict TD across time series. The authors created a repository containing 150 commits at weekly intervals from five open-source software systems, for a total of 750 instances. The results show that the ARIMA model (autoregressive integrated moving average model) cannot accurately predict for long periods, but only for 8 weeks into the future. The same authors have expanded the study of the previous work (Tsoukalas et al., 2020) by shifting the focus from univariate time series to multivariate methods of Machine Learning, thus taking into account not the evolution of the target variable, but also other additional characteristics related to the 'last. In particular, they expanded the previous dataset, building a dataset containing information on 15 software systems belonging to different application domains, for each of which about 150 commits were collected on a weekly basis, covering up to 3 years of evolution of each project and several software quality metrics. In this article, they applied several machine learning models (regression, regularization, vector regression support, and regression trees) to create TD master prediction approaches. From the validation of these models through the mean absolute percentage error, it emerged that for short periods the linear regularization models work better, while for long periods the Random Forest regression as predictor works better.

As an extension of the work (Tsoukalas et al., 2020), Tsoukalas et al. (2021) have carried out a TD prediction between projects using a data clustering approach, with the aim of evaluating whether clustering can be considered as a solution to improve the accuracy of the TD prediction. They collected data from different software systems and divided them by similarity into clusters. The approach involves the use of five regression algorithms to predict periods that oscillate in intervals from 1 to 10 steps forward. The results are promising and allow predicting TDs in unknown software systems, assigning them to known clusters.

Furthermore, the study of Tsoukalas et al. (2018) has brought to light that the right maturity has not been reached for the methods and tools capable of estimating the TD and that there are still no interesting contributions regarding the field of debt forecasting.

Overall, to the best of our knowledge, there is still a lack of adequate approaches to predicting the behavior of TD in a software system. This study represents an extension of a preliminary (Aversano et al., 2021) study in which we presented a deep learning-based approach capable of predicting the trend of the TD in the source code. This work aimed to investigate to what extent the use of deep learning models can be considered effective for predicting TD. In particular, the model was trained and tested on data from four open-source Java systems, for which only product metrics were collected, hence the quality of the source code, and TD-related indicators. The results are really interesting, showing an F1 score of 99% for two software systems and greater than 91% for the other two systems being analyzed.

Compared to the contributions already present, the aforementioned study builds a model of functionality based not only on metrics relating to TD and software quality but also on two influential process metrics. For each software system, regular intervals are not considered, but the entire history is collected, collecting all the information commit by commit, without neglecting anything and a convolutional temporal neural network model is proposed, capable of predicting the trend of the TD in the source code and indicate if, given a specific class, in a given commit, the TD value will increase, decrease or remain stable in the next commit for that same class. Compared to the previous work, the analyzes relating to 6 software systems were added, and above all cross-project experiments were carried out that allowed us to validate the effectiveness of the model by testing it on completely unknown data compared to those on which it had been trained.

Therefore,this work aims to give life to a model capable of evaluating the future evolution of the debt in the code to allow developers to take proactive action regarding the repayment of TD. Therefore, the main contribution of this work concerns the granularity with which the analyzes were carried out.

4. Approach

This section presents the proposed approach to perform a just-in-time prediction on TD of changes performed on a software system. The main contributions of the study focus on four main aspects: (i) the prediction is just-in-time (ii) the fine granularity with which the forecast is made, (iii) the metrics chosen to build the forecast model, and (iv) the deep learning approach used.

More in detail, we have chosen to carry out analyzes at a very deep granularity, collecting a large amount of data for the training of our deep learning model. We predict the evolution of the TD in each class commit by commit, following each modification performed by the developers.

As for the characteristics chosen to build our forecasting model, these can be divided into three subgroups:

- Product metrics indicating the presence of the TD;
- Product metrics related to software quality;

- Process metrics inherent to the modality and quality of developers' work.

The last innovative aspect of this study concerns the type of approach used to make a just-in-time prediction. To the best of our knowledge, there are no studies in the literature that exploit deep neural network models for this purpose. In our case, this prediction is possible due to the huge amount of data collected at a fine level of granularity. In particular, we propose a temporal convolutional network model with a hierarchy of attention levels, to investigate more precisely the very complex relationships that coexist between process and product metrics. We chose this type of network because, employing dilated convolutions that enable an exponentially large receptive field, it is capable of detecting the causality between the presence of the TD in the source code and the evolution of the metrics.

4.1. Feature set

The model is trained using a large set of metrics. These metrics are evaluated on the source code and VCS logs of a software project. Specifically, our features include three sets of metrics: those related to the software product, divided into TD indicators and quality indicators, and others strictly related to the evolution process.

We have chose these metrics because they are the ones that in the literature have been defined as most correlated to the presence of TD, and the most used (Baldassarre et al., 2020). These allow identification of the TD through the analysis of artifacts created during the software life cycle. In particular, the TD-related metrics are the result of the analysis of the source code and report the presence of any bugs, vulnerabilities or violations of good programming practices that lead to failures or a deterioration in the quality of the software. In particular, among these, we also find code smells, which according to Alves et al. (2016a) and Palomba et al. (2018) are symptoms of poor design and implementation choices that can hinder the comprehensibility and maintainability of the code. These are useful indicators because they lead to the need to proceed with refactoring.

To account for internal quality, our functionality model also includes object-oriented quality metrics. These metrics allow to evaluate the different characteristics of the systems from the point of view of long-term maintainability (Li et al., 2015a; Alves et al., 2016a). In particular, the metrics previously defined by Chidamber and Kemerer (1994) evaluate the code in terms of cohesion, complexity, size and coupling. These aspects, if not properly managed, lead to an increase in the TD.

Overall, for each of these sets of metrics, we report a table in which each row contains the name and a brief description of the metric. More specifically, in the Table 1 we report the metrics indicating the presence of the TD and therefore strictly linked to it, while in Table 2, we report the metrics relating to the quality of the source code, including which metrics CK.

Finally, we included as features some process metrics, which explain the quality of the development activities carried out by the developers. These metrics include the cost and effort employed considering both the type of change made on the code and the characteristics of the developers related to the experience. They are defined as follows in Table 3, where we report the name of metric in the first column, the formula for calculating each of them in the second, and a brief description in the last column. Once the necessary information reported in the table had been extracted, these metrics were calculated by developing specific scripts.

4.2. Data collection

The dataset[4] used for this study is the evolution of the history of 10 open source Java projects. The choice fell on these systems because they share Java as a programming language, are characterized by different domains for application and size, are available on Github, and have several commits greater than 500.

Table 4, in the first column, shows the names, in the second the label we assigned to speed up their identification and to follow the number of classes containing TD, the number of commits under observation, and the time-lapse.

4.3. Data extraction

Fig. 1-(a) describes the main phases that made up the process used to extract and collect the characteristics and generate the datasets necessary to conduct the analysis object of this research work. Collecting useful information begins with extracting the change history of software systems. The source code of each class of each software system has been inspected commit by commit to evaluate its evolution over time. More specifically, as you can see in the figure, all the information was extracted from Github, a hosting service for software systems. In detail, the code was subjected to an analysis aimed at detecting the metrics indicating the presence of the TD and its value, shown in Table 1. A revision data extraction allowed the collection of source code quality indicator metrics, described in the Table 2, and the log extraction was used to collect information useful for the calculation of the process metrics described above, such as the author of the commit or the number of changes made.

Respectively, in the first case, SonarQube was used. It is an open-source tool to support developers, which guarantees continuous inspection of the code and provides thousands of automated rules aimed at the static analysis of the code. This is capable of detecting bugs, code smells, and security vulnerabilities on over 20 programming languages. For quality metrics, on the other hand, the CK tool was used, capable of calculating class and method level metrics in Java projects without the need for compiled code, but therefore using static analysis. For the process metrics, on the other hand, python scripts have been created that use the information collected from the logs.

At the end of the analyzes conducted with the two tools used, a merging phase is executed in which the JSON files produced by SonarQube and the CSV files produced by CK have been combined into a single CSV file. Notice that CK unlike SonarQube generates an entry for each class and subclass present in the file. Sonarqube, on the other hand, brings everything together in one voice. This causes a different number of entries which prevents the integrity check. To work around this problem, the CK output is treated by merging all the entries generated from a single file using different strategies depending on the metric in question. In particular, for CBO, LCOM, WMC the average is considered, for DIT and for the Max metric nested blocks the maximum value, and for all the rest the sum.

Finally, the three datasets have been integrated giving life to the final dataset, where there is the history in terms of changes committed for each java class considered. More specifically, for each software system considered, for each java class we have collected the complete history in terms of commits, collecting the metrics described above for each single commit. Subsequently, after an appropriate normalization and windowing of the data, these are fed to the Deep Learning model used for the various stages of training, validation, and testing. Furthermore, as can be seen in the figure, a set of traces labeled $T = (M, l)$ have been created, where each line M represents an instance of the system to which a multi-label is associated indicating the impact on the TD.

[4] https://doi.org/10.6084/m9.figshare.19404419.

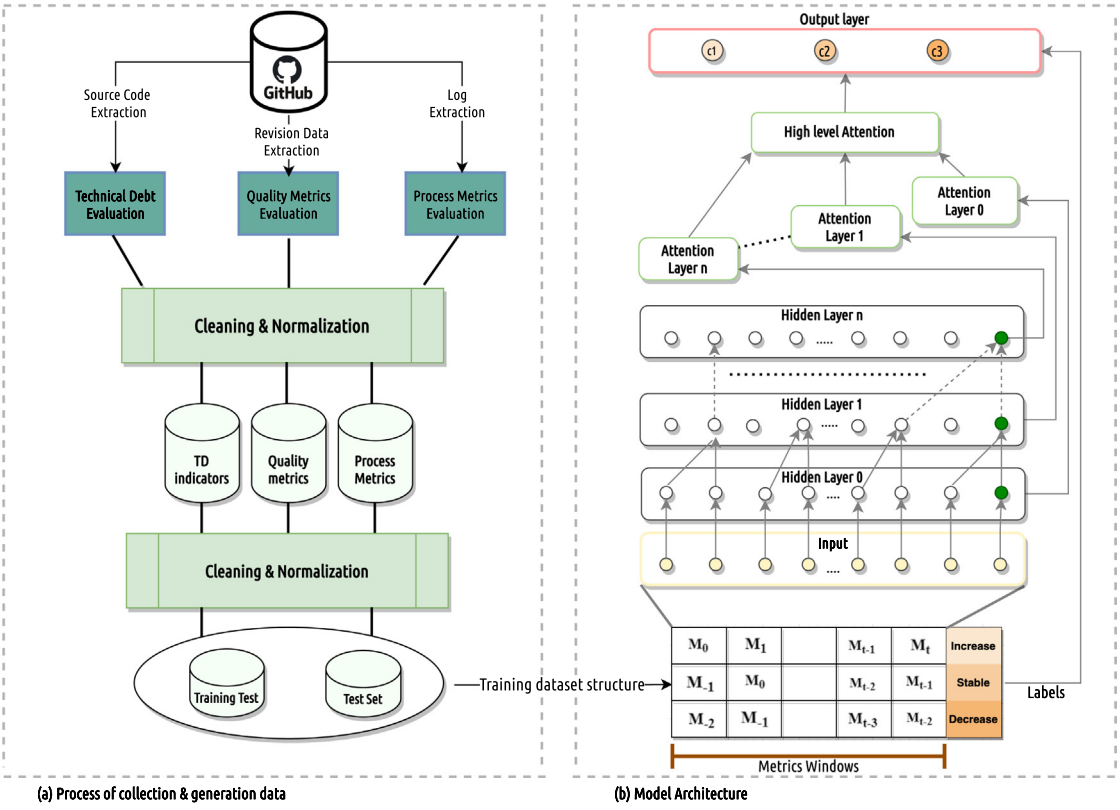

Fig. 1. Data extraction process and model architecture.

Table 1
Product metrics and TD related metrics.

Name	Description
Code smells	Total of code smells in the source code.
Bugs	Total of bugs.
Classes	Total of classes (all types: nested classes, interfaces, enumerations, and annotations).
Functions	In the java language, it indicates the number of methods present in the classes, ignoring those belonging to the anonymous classes.
Vulnerabilities	Amount of vulnerabilities.
Complexity	Indicates the cyclomatic complexity of a section of code and is the number of linearly independent paths through the source code. If the source code does not contain decision points such as IF or FOR loops, then the complexity will be 1. If the code has a single IF containing a single condition, then there will be two possible paths.
Cognitive complexity	Measurement of the difficulty of understanding control flow.
Sqale rating	Represents the score assigned to the technical debt ratio. It is divided into several intervals: A = 0–0.05, B = 0.06–0.1, C = 0.11–0.20, D = 0.21–0.5, E = 0.51–1.
Sqale debt ratio	Remediation cost/Development cost, which can be expressed as: Remediation cost/(Cost for developing 1 line of code ∗ Total of lines of code). Ratio of the cost of developing the software to the cost of repairing it.
Ncloc	Number of lines of code not commented.
Comment lines	Total sum of lines containing comments and commented code. It does not consider blank lines or lines containing only special characters.
Comment lines density	Comment line density = Comment lines/(Code lines + Comment lines) ∗ 100 With such a formula, values equal to: (a) 50% mean that the number of lines of code equals the number of comment lines (b) 100% indicates that the file contains only comment lines

Table 2
Product metrics — Quality indicators.

Name	Description
Lack of cohesion of methods	The cohesion of a method expresses the property of a method to exclusively access attributes of the class. The lack of cohesion derives from the presence of multiple methods that access common attributes.
Weight method count per class	Weighted sum of the methods of a class, where the weight of a method is given by a complexity factor of your choice.
Coupling between objects	Number of collaborations of a class, that is, the number of other classes to which it is coupled
Depth of inheritance tree	Maximum distance of a node (a class) from the root of the tree representing the hereditary structure. The greater the depth of the class in the hierarchy, the greater the number of methods it can inherit.
Response for a class	Indicator of the 'volume' of interaction between classes. At high values the design complexity of the class increases and the effort for testing increases
Number of methods	Total amount of methods: static, public, abstract, private, protected, predefined, final, and synchronized.
Number of static invocation	Total number of invocations through static methods
Non-Commented, non-empty Lines of Code	Number of lines of code, except of blank lines.
Number of unique words	Count of unique words.
Number of fields	Number of set of fields: static, public, private, protected, default, final, and synchronized.
Parenthesized expressions	Count of expressions in parentheses.
Comparisons	Total of comparisons (e.g. == or !=).
Returns	Count of return statements.
Try/catches	Total of try and catches.
Loops	Amount of loops (for, while, do while, generalized for).
Number	Quantity of numbers (i.e. int, long, double, float).
Variables	Numerical index of variables declared.
Math operations	Count of mathematical operations (divisions, remainder, plus, minus, right and left shift).
Anonymous classes, subclasses and lambda expressions	Number of anonymous declarations of classes or subclasses.
String literals	Amount of string literals (e.g. "John Doe"). Strings that are repeated are counted as many times as they appear.
Usage of each field	Calculate the usage of each field in each method.
Usage of each variable	Calculate the usage of each variable in each method.
Modifiers	Number of public/abstract/private/protected/native modifiers of classes/methods.
Max nested blocks	The highest number of blocks nested together.

Table 3
Process metrics.

Metrics	Formula	Description
Developer's seniority	$SE(c_i, d_j) = Cd(c_i) - Fc(d_j)$	Measure developer seniority for a given commit based on days. It is calculated by making the difference between the commit date and the dates of its first commits in the repository.
Commit's ownership	$OC(d_i, f_j, c_k) = d_i \in O(f_j, c_k)$	The set of developers who collectively performed at least half of the most significant total changes on a given file.
Number of file owners	$NFOWN(f_j, c_k) = \lvert O(f_j, c_k)\rvert$	The cardinality of the collection for a specific file and a specific commit.
Owned file ratio	$\frac{R(d_i, f_j, c_k)}{\text{Changes}([c_s, \ldots, c_k])}$	Calculates the ratio of changes R made by the developer d_j on file f_j to the total changes Changes$(\cdot)$ made by others to a specific file in the commit interval $[c_s, \ldots, c_k]$, since the start of the observation period on the same file.

Table 4
Systems details.

Software name	ID	# classes	# commit	Commit time-lapse
Jackson-core	JC	440	2244	22 December 2011–17 August 2020
Jackson-dataformat-xml	JDF	336	1406	30 December 2010–1 October 2020
Commons BeanUtils	CB	819	1404	27 March 2001–29 August 2021
Commons-imaging	CI	1568	1334	12 October 2007–31 August 2020
Commons-io	CIO	702	2957	26 January 2002–28 February 2021
Guice	GU	1312	1967	25 August 2006–29 January 2021
Javassist	JA	545	944	22 April 2003–21 September 2021
Jfreechart	JF	2936	3951	29 June 2007–2 July 2020
Xerces2-j	XE	1800	4900	9 November 1999–17 December 2008
Zookeeper	ZO	1490	1867	3 November 2007–18 February 2021

4.4. Data analysis

Having a large set of features included in the model, we also conducted an analysis using feature selection techniques to identify the best subset. Therefore, a hierarchical cluster analysis has been performed on the variables with the purpose of grouping the most similar ones (Köhn and Hubert, 2014). The result is a set of clusters, where each cluster differs from the others and the features within each cluster are very similar to each other. In this approach, we evaluated redundancy and collinearity measures among all the features. This allowed grouping the features with a high value of redundancy and collinearity in a single cluster, thus

leading to a significant and effective reduction of the features to be considered in the model.

More specifically, we have used the R language *Hmisc* package, and in particular, the *varclus* function which automatically combines the infrequent cells of these variables using an auxiliary function *combine.levels*.

As a method of similarity between variables, we have used Spearman which allows us to identify monotonous but not linear relationships. This correlation coefficient is a non-parametric index, hardly conditioned by the outliers, which allows us to evaluate the strength of the relationship between two variables.

Therefore the purpose of the cluster analysis is to create groups of statistical units present in the collective and on which some characters have been detected so that the units belonging to the same group are as similar as possible and between the groups, there is the maximum dissimilarity.

The final product of the hierarchical methods consists of a series of partitions that can be represented graphically through a dendrogram in which the distance level is reported on the ordinate axis, while the individual units are reported on the abscissa axis. Each branch of the diagram (vertical line) corresponds to a cluster. The (horizontal) junction line of two or more branches identifies the distance level at which the clusters merge. In particular, we have applied clustering to the three feature groups defined in the Tables 1, 2, 3 for each system. Therefore, having obtained the three dendrograms, one for each group of features, we have chosen 0.5 as the pruning threshold and we have eliminated from our feature model all those above this threshold.

After obtaining the dendrograms for all systems considered in the study, for the clusters that exceeded the chosen threshold, we kept only one metric for each cluster in our set of metrics. In particular, we have tried to eliminate the over-threshold characteristics common to most systems.

4.5. Predictive model

The Keras,[5] and Tensorflow[6] API were used to build our classifier. The first is the library of the neural network, the second is used for high-performance numerical computations.

Below are the three main phases that make up the work of the predictive model:

- Pre-processing phase: phase with multiple objectives, (i) cleaning of data, with annexed removal of incomplete or incorrect ones (ii) normalization of features with min–max normalization, (iii) selection of the most correlations, and (iv) labeling of the target variable in three classes.
- Training phase: the classifier is trained with a supervised learning algorithm, receiving the labeled traces as input. Furthermore, the hyperparameters are optimized.
- Test phase: The model is tested through nested cross-validation (Varma and Simon, 2006). A sequence of test sets is generated for each observation. This is a resampling procedure used to evaluate machine learning models and access model performance for an independent test dataset. It consists of two cycles, internal and external. Respectively, in the first one the parameters are chosen, in the second one the subdivision of the data into several training sets is performed (the error is calculated for each set) and finally, the average of all the errors is calculated.

More in detail, Fig. 1-(b) shows the architecture of the proposed model, which is based on a neural network of the TCN type. The layers of which it is composed are:

- Input layer: it receives data from the outside, and is composed of several nodes equal to the number of attributes considered;
- Hidden Layers: it is an intermediate level between the input and the output composed of artificial neurons also called "perceptrons". This is where the actual processing of the data takes place, with an accompanying interpretation of the complex relationships existing between them. Its output is represented by the weighted sum of the inputs after being processed by the activation function;
- Attention Layer: it introduces the hierarchical mechanism of attention (Yang et al., 2016) aimed at modeling relationships, both in input and output, independently of distance;
- Batch normalization: it mitigates the effect generated by unstable gradients, allowing greater precision in both the test and validation phases (Ioffe and Szegedy, 2015). The outputs are scaled to have an average of 0 and a variance of 1. This normalization allows you to train the network faster using a faster learning rate and simplifies parameter initialization.
- Output Layer: it generates the final result and contains as many neurons as the number of classes to predict.

5. Experiment description

In this section, we specifically define the objectives of the research through the description of the three research questions on which the study is based, and describe the details of the experiment conducted.

5.1. Research questions

The objectives of this research study can be summarized in the three research questions.

RQ1: *Is the TCN model effective for forecasting the technical debt during the open-source software systems evolution?*

This research question aims to evaluate the performance of the TCN model, proposed with the variant of the attention levels, to predict the trend of the TD in the investigated software systems, taking into consideration the metric model presented in Section 4.1.

RQ2: *Can the performance of the TCN model be improved applying a feature selection?*

The choice of the inputs of the neural network, which constitute the independent variables, are fundamental for the performance of the model because each adds a dimension. Therefore, having defined the feature model, this research question aims to reduce the number of inputs, through a feature selection operation. In this way we can understand the weight of each chosen metric, keeping only those that are most relevant for the just-in-time prediction of the trend of the TD.

RQ3: *Can the model trained on a set of projects successfully be applied for predictions on a different project (transfer-learning)?*

The first two research questions individually investigate the history of each software system considered for the study, thus considering a vast amount of data for each project. This question instead aims to evaluate a cross-project approach. In this sense, all the data on the systems have been merged, to verify whether the proposed predictive model can also be used on systems with a short history based on what the network learns on other systems. This approach would be critical for developers in the early stages of development when there are few commits left. In this way, thanks to the information of other systems, the model is already able to know the TD trend.

[5] https://keras.io.

[6] https://www.tensorflow.org.

5.2. TCN experiment settings

The experiments were conducted to evaluate the predictive capacity of the proposed model of the trend of the TD in the source code of the analyzed software systems. The network was built using Python as a programming language. More in detail, Tensorflow[7] and Keras, respectively an open-source software library for high-performance numerical computations, and a Neural Networks API.

To achieve the aforementioned goal, a product and process metrics model was built, described in the Section 4.1 section, and we used the sliding window methodology, setting 10 as the window size, to prepare the data and better follow the evolution over time of the chosen metric. Furthermore, to find the best possible results, we have chosen to do an optimization of the hyperparameters (Bengio, 2000). We have optimized the following architectural parameters: the learning rate, the number of levels and the size of the network, the batch size, the abandonment rate, and the timestamp. For the choice of the best parameters, we have relied on an approach SBMO (Sequential Bayesian Model-based Optimization) implemented through the Tree Parzen Estimator (TPE) algorithm as defined in the work of Bergstra et al. (2011).

In more detail, we report below the architectural parameters on which the proposed model is based:

- **Network size**: two types of size, small and medium. A small network is a network made up of a maximum of 1.5 million learning parameters; while the average network has several parameters that can fluctuate between 1.5 million and 7 million;
- **Activation function**: indicates the transformation of the input with the help of a weighted sum to the output. It uses a single node or multiple nodes for the network to generate the prediction. In our case we used RELU.
- **Learning rate**: between 3 and 6;
- **Number of layers**: varies between 6 and 8;
- **batch size**: 128 or 256;
- **Dropout rate**: to be chosen between 0.15 and 0.20;
- **Timestamp**: 5 or 10.

All possible combinations of the parameters listed above have been formulated, but only the best is taken into consideration. The results of the work refer precisely to the best possible combination.

Furthermore, as an optimization algorithm, we used stochastic gradient descent (SGD) (Schaul et al., 2013), an iterative method for the optimization of differentiable functions, the softmax function as the last activation, and for the loss function for training, we chose categorical cross-entropy (Mannor et al., 2005).

In our experimentation, a maximum of 50 epochs is defined. We also have introduced an early stop for the target metric to avoid wasting resources and time and to avoid excessive adaptation. In this way, if there is no improvement, the training is interrupted and the model is set aside for testing.

Finally, to evaluate the accuracy of the model we calculated the harmonic mean of the precision and recall, F_1. The precision is the ratio between the number of correct predictions of an event over the total number of times that the model predicts it, and recall measures the sensitivity of the model because it indicates the ratio between the correct predictions for a class on the total of cases in which occurs.

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

6. Results

In this section, we report the results obtained for each research question.

6.1. RQ1: is the TCN model effective for forecasting the technical debt during the open-source software systems evolution?

This question investigates the performance of the deep neural network model based on the proposed metrics for predicting the impact on TD in the source code of software systems. First of all, we report in Fig. 2 the accuracy and loss curves obtained during the training phase (note that, since early stop is used, they have different lengths). For each system, the accuracy trend is shown on the top and the loss obtained during the best permutation performed on the bottom. It is possible to note that the systems JDF, CB, CIO, JF, and XE show 6 epochs for the given permutation, while the remaining JC, CI, and ZO, respectively 8, 15, and 20. This means that in the former the early stop was activated to the sixth epoch because no further improvements were found. Finally, the results show excellent performance of the model in the training phase, as already at the first epoch we obtain an accuracy higher than 94% for JC, 95% for JDF, and AL 99% for all the other systems. For the loss, less than 26% for JC, 22% for JDF, and 7% for the remainder.

For each software system analyzed, Table 5 reports the best performances obtained with the predictive approach based on a TCN neural network. More in detail, the table shows the permutation during which the model obtained the best performance, in particular the best F1 value. The first two columns show the system considered and the activation function, while the six following columns show the parameters for which the optimization has been carried out: the learning rate, the size of the network, the number of levels, the timestamp, the batch size, and finally the dropout rate. Instead, the last columns report two of the validation metrics used for the model: accuracy obtained during the validation phase, and an F1 score obtained during the testing phase. Each row of the table reports the best permutation for each system with the parameters used by the model and the results obtained.

Taking the F1 score as a reference, the model shows excellent performance. In fact, except in the case of JC, F1 is always greater than 0.98, even reaching approximately 1 in the case of JF, XE, and ZO.

Furthermore, observing the table, it can be seen that (in 6 cases out of 10 total) the best permutation is obtained when the learning rate, the number of layers, the size of the network, and the timestamp are respectively 6, Small, 7 and 5. For the batch size and the dropout rate, the best results are obtained for the values 128 and 0.15.

Accuracy during the model validation phase shows the excellent performance of the approach. Even if for JC this is equal to 81%, and therefore indicative of the fact that the model behaves fairly with the data of this system, and that for JDF it is equal to 96%, for all the other systems it always reaches values higher than 99% thus demonstrating a very high efficiency of the proposed approach.

Furthermore, except for JC, where the model achieves a good F1 of 0.96, all the values of interest of the other systems show excellent performance. In fact, we get 0.98 for JDF, 0.99 for CB and CI, and the maximum achievable value in the case of the remaining systems. Finally, we can conclude that the proposed approach behaves in a good way for JC, is excellent for JDF, CB, and CI, and is perfect for CIO, GU, JA, JF, XE, and ZO where the model seems to be wrong with a very small number of instances to predict (see Table 5).

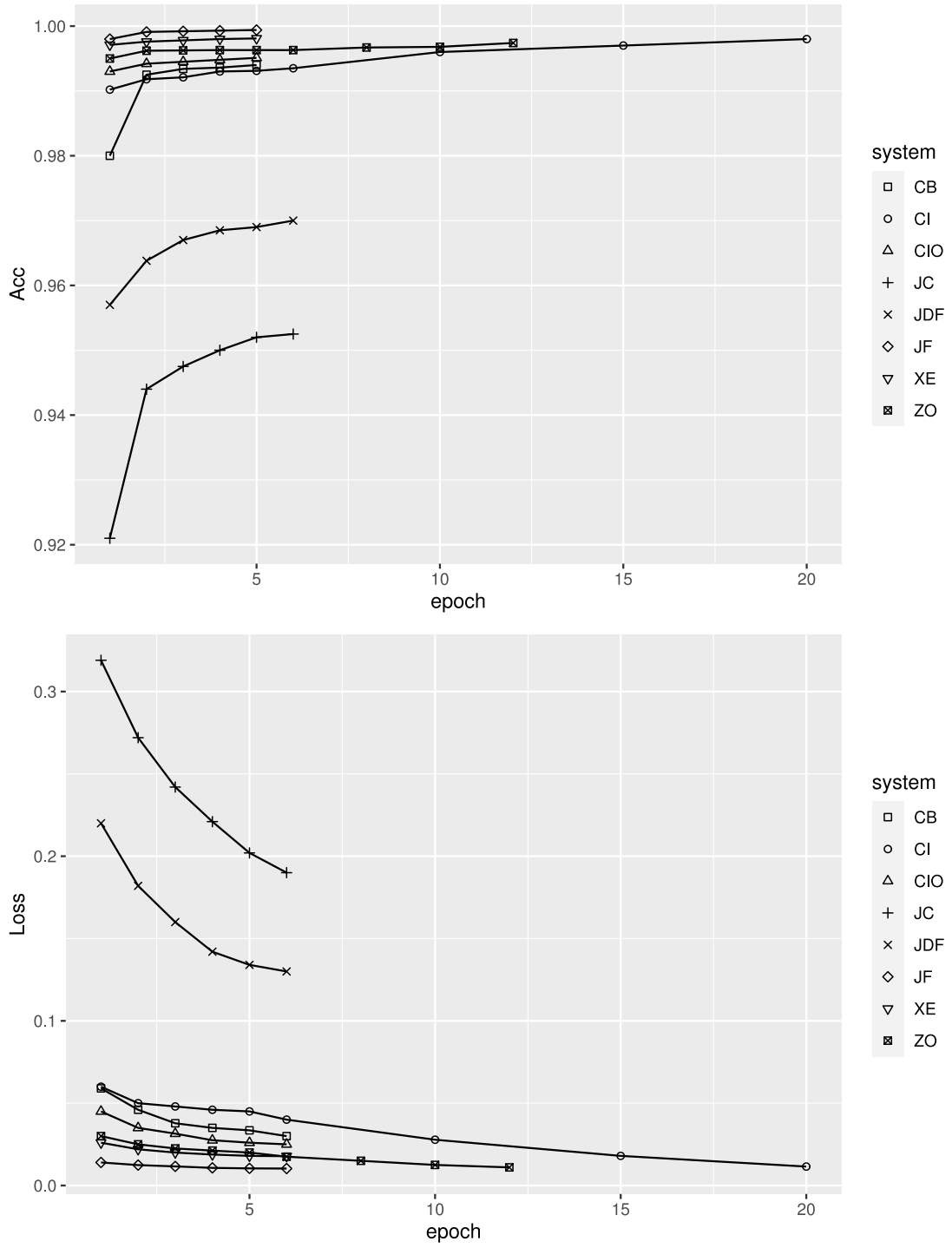

Fig. 2. Accuracy (top) and loss (bottom) plots of the best permutation for all the analyzed systems.

Finally, in Table 6 we report for each system the average time taken by the model per epoch in the best permutation obtained. The unit of measurement of times is seconds. The results show that the fastest system is JDF (43″), which as you can see in the Table 4 is the system with the fewest java classes analyzed, while the slowest system is JF (2910″), which has the largest number of java classes and a large number of TD-affected commits.

6.2. RQ2: can the performance of the TCN model be improved applying a feature selection?

In the first research question, we have trained the model for the prediction of the TD trend in the code of the systems under examination considering all the features reported in Section 4.1 for a total of 58 variables considered as input. This initial set of

Table 5
Results of the best permutations.

System	Activation function	Learning rate	Size of network	Number of layers	Timestamps	Batch size	Dropout rate	Validation accuracy	Test F1-Score
JC	RELU	3	Medium	7	10	128	0.15	0.8121	0.9598
JDF	RELU	6	Small	6	10	128	0.15	0.9604	0.9828
CB	RELU	6	Small	6	5	128	0.15	0.9930	0.9935
CI	RELU	6	Small	7	5	128	0.15	0.9957	0.9937
CIO	RELU	3	Medium	7	5	128	0.15	0.9932	0.9963
GU	RELU	6	Medium	7	10	128	0.15	0.9952	0.9966
JA	RELU	3	Small	8	5	128	0.15	0.9945	0.9954
JF	RELU	6	Small	7	5	128	0.20	0.9987	0.9986
XE	RELU	3	Small	8	10	128	0.15	0.9978	0.9971
ZO	RELU	6	Medium	7	5	128	0.15	0.9979	0.9975

Table 6
Best permutation average training time per epoch.

System	Training time
JC	744.56
JDF	43.01
CB	136.18
CI	691.99
CIO	753.77
GU	683.32
JA	325.86
JF	2910.06
XE	2569.39
ZO	993.50

features includes all features used in similar study in the literature. However, considering that such a large number of features could affect and condition the training times of the model, we have carried out a feature selection activity, to understand which of these had a greater influence on the performance of the TD, to identify the best set of features to obtain satisfying results and minimizing the time needed by the model for training.

As detailed in Section 4.4, we have used a hierarchical clustering analysis and in Fig. 3, we report as an example the dendrograms obtained for the JF system. In particular, in Fig. 3-(a) we report the dendrogram for the group of TD indicators, in Fig. 3-(b) for the process metrics, and in Fig. 3-(c) for the quality metrics.

So, after this selection process, we have removed 22 features from our model, moving from a set of 58 to a set of 36 features. In particular, for the TD indicators, we have discarded cognitive complexity, classes, nloc, vulnerabilities, complexity, comment line density and code smells. For the process metrics, owner of the commit and frequency, as these are strongly correlated respectively to ownership and seniority. For the quality indicators, we have discarded 13 features: comparisonQty, statistic Fields, WMC, UniquewordsQty, VariablesQty, Final Fields, MaxnestedBlocks, AssignmentsQty, LOC, Total Fields, Total Methods, and Public Methods.

Once the feature selection was made, we re-trained the model and still obtained excellent results, indicators of good performance.

As in the first RQ, also in this case we report for each software system analyzed, the Table 7 with the results of the best permutations of the model.

As you can see from the table, the only parameter that assumes the same value for all the best permutations of the systems considered is the batch size, always equal to 128. For the other parameters, there is no unanimity in the choice of the ideal value, because these vary according to the system data that the neural network is using as input. Nevertheless, it is possible to verify that almost all systems, except one, achieved the best performance with a permutation in which the dropout rate was equal to 0.15. In eight out of ten systems the size of the network, the number of levels, and the timestamp, are respectively equal to Medium, 6, and 10, while in six out of ten systems the learning rate for the best permutation is equal to 6.

The accuracy obtained during the model validation phase is always greater than 99%, reaching the maximum peak of 99.87% in the case of JF. The only systems that show lower accuracy are JDF, which still shows excellent model performance with 97% of accuracy, and JC, for which the model behaves fairly well with almost 92% of accuracy.

Therefore, the results of our approach show the excellent performance of the proposed model. For JC we obtain an F-score equal to 95.82%, for JDF equal to 98.82%, and for all other systems considered higher than 99%, almost 100% in the case of JF with 99.87%. All the validation metrics used assume very high values, highlighting excellent performances, which means that our model can predict the technical debt by failing only very few instances.

In Table 8 we report the average times per epoch expressed in seconds and calculated for the best permutations of each system. As can be seen from the table, the training times of the model have been considerably reduced after the features selection used on the dataset. The training times of the model vary from 32 s to 2445 s, because the time taken is strongly influenced by the number of input data. The system with which the network we have used takes the least time for training to predict the impact on TD is JDF (32″), followed by CB (106″), JC (159″) and JA (198″). These are the systems that have the least number of classes and commits. Following are the larger systems, CI (246″), CIO (289″), GU (350″), ZO (681″), JF (1749″) and XE (2445″).

6.3. RQ3: can the model trained on a set of projects successfully be applied for predictions on a different project (transfer-learning)?

The first two research questions concern the prediction of the impact on TD in the source code in the individual software systems considered. Therefore the proposed model is trained on the data of a specific software system, and the validation is carried out on other data always connected to that specific system. With this research question, however, we intend to predict cross-project TD. This means that we aim to evaluate the forecasting capacity of the proposed model in software projects that are not part of the training set, and therefore consequently previously unknown.

To conduct this type of experiment we created four different combinations of the systems considered for this study. In the first combination (C1) we considered five systems of the total ten. In particular, we built our training set by combining all the data belonging to the systems, CIO, XE, CI and JF, and the test set with the data relating to the ZO system. In this way we trained the proposed model on a varied set of instances, and then we tested it on data completely unknown to the model, that is, those belonging to a system not included in the training set. In the second combination we have (C2) we have selected the remaining five systems, not considered in the previous combination. In particular, we trained the proposed model on a training set

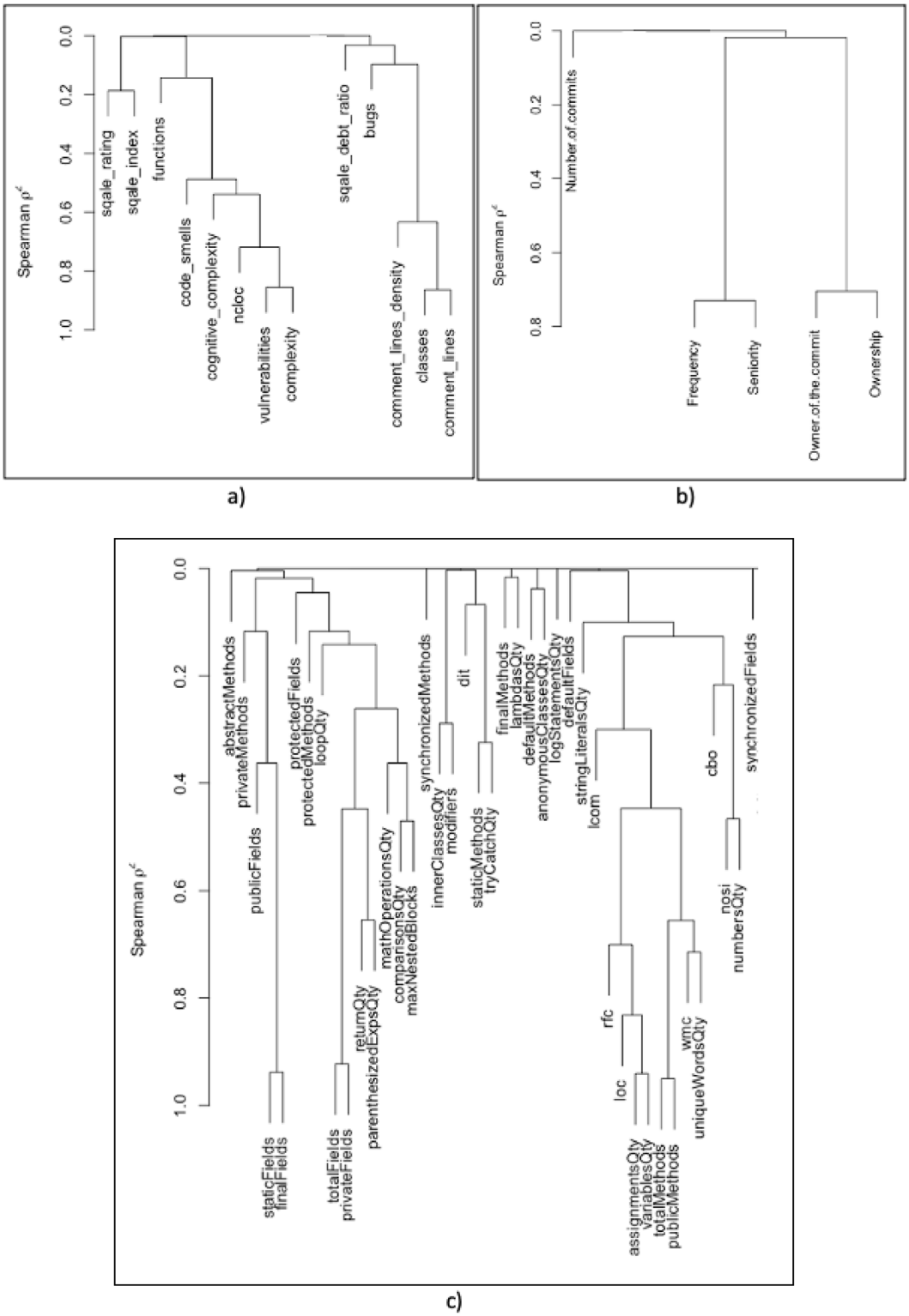

Fig. 3. Dendograms of the three groups of features.

consisting of the instances of the CB, JA, JC, JFD systems, and we tested the model on the instances of the GU system. For the third combination (C3) we have considered four systems, three for the training e one for the testing set. More specifically, the training set is composed by instances of CIO, CI, and GU, while the set of CB. Finally, the fourth combination (C4), like the previous one, involves four systems. For the training set we have used ZO, JDF, and JC instances, while we have tested the model on JA instances.

The systems chosen for the experiment were selected in a particularly random way. This allows us, even more, to underline the

Table 7
Results of the best permutations with the feature selection.

System	Activation function	Learning rate	Size of network	Number of layers	Timestamps	Batch size	Dropout rate	Validation accuracy	Testing F1-Score
JC	RELU	3	Medium	6	10	128	0.15	0.9179	0.9584
JDF	RELU	6	Medium	6	5	128	0.15	0.9777	0.9881
CB	RELU	6	Small	6	10	128	0.15	0.9926	0.9940
CI	RELU	6	Medium	6	5	128	0.20	0.9936	0.9944
CIO	RELU	3	Medium	6	10	128	0.15	0.9968	0.9956
GU	RELU	6	Medium	6	10	128	0.15	0.9956	0.9962
JA	RELU	3	Medium	6	10	128	0.15	0.9936	0.9945
JF	RELU	3	Medium	6	10	128	0.15	0.9987	0.9987
XE	RELU	6	Medium	7	10	128	0.15	0.9976	0.9977
ZO	RELU	6	Small	8	10	128	0.15	0.9982	0.9980

Table 8
Best permutation average training time per Epoch after the feature selection.

Systems	Training time
JC	159.00
JDF	32.55
CB	106.67
CI	246.47
CIO	289.64
GU	350.49
JA	198.72
JF	1749.55
XE	2445.66
ZO	681.58

effectiveness of the proposed model, because although these are different in size and characteristics, our approach shows excellent performance.

In the Table 9 we report the results obtained for the three best permutations of the model, for each considered combination. Respectively, we report three rows for each combination previously described, C1 in yellow, C2 in green, C3 in red, and C4 in blue. In the first column there is the number of the combination, and the number of the permutation, from the second to the seventh column there are the parameters with which we have set the neural network, and in the last two columns respectively there is the accuracy in the validation phase and the F1-Score in the test phase of the model.

In particular, it is possible to observe first of all that on the six variable parameters of the network, for three of these the best results are always obtained with the same value. This would allow a future experiment to eliminate certain values from the choice, thus saving time. More specifically, the optimal parameters for Learning Rate, Batch Size, and Dropout Rate are 3, 128, and 0.15, respectively.

The results show excellent performance of the model, able to predict the impact on the TD even on instances unknown to it and not belonging to the training set. The accuracy during the validation phase is always greater than 99%, except in the case of C4 where it assumes values equal to 97%. Likewise, the F1 score is almost 100%, except in the case of the C4 where it is 97%. These results are very satisfying, and show that the model can perfectly predict the impact on TD in the code, failing only with very few instances.

7. Discussion of results

Comparing the results of the first RQ, with the results of the second RQ, we can say that the results are satisfying and encouraging. If we take a look at the validation accuracy and the F1-Score, we can see that after the Feature Selection, these validation metrics continue to assume excellent values, demonstrating almost perfection of the model in forecasting the trend of the TD. It should also be emphasized that the validation accuracy for JC which in the first RQ was 82% here reaches 92%.

To highlight the results obtained in the two experiments conducted, respectively before and after the feature selection, we report in Fig. 4 the comparison between the values of the F1-Score obtained. We report the systems on the abscissa axis and the values of the F1-Score on the ordinates. With the blue bars, we find the values of the first RQ, in green those of the second.

As you can see, for five systems the metric improves after the feature selection, for JF it remains almost the same and in the case of JC,CIO, GU, and JA the value is higher for the first experiment conducted. The difference in values is minimal because already in the first RQ we have obtained very high values of the metric, which in some cases even touches 100%. But even if minimal, these increases are important because they denote a further improvement of the proposed approach.

The success of the model after feature selection is also shown by the average model training times per epoch. In Fig. 5 we show the comparison of the training times, plotting the systems on the abscissa axis and the seconds on the ordinate axis. In the graph, the blue bars indicate the times obtained in the first RQ, and the green bars those obtained after the features selection. From the graph, we note that in all cases there is a significant decrease in training times, almost always equal to more than 20%. Indeed, in the cases of JC, CI, CIO and JA the decrease is truly remarkable. In fact, in the case of JC, we go from 744 to 159 s, obtaining a reduction of almost 80%. For CI, 691 to 246 s, 64% less, and for CIO 753 to 289 s, 62% less. Finally, in the case of JA the training time is 325 s in the first experiment, while 198 in the second, so it means 61% less.

8. Implications

This section, outlines the main implications for practitioners and researchers working on technical debt, pointing out interesting topics to be investigated in more depth.

Currently, existing tools for managing TD, such as Sonarqube or CAST, allow practitioners to perform the detection of the TD value, without indications about its previous or future values. While, practitioners can benefit from approaches, like the one proposed in this study, to anticipate what kind of impact might occur on the TD avoiding an excessive rise in its value. This will drive practitioners to discard faster and poorly designed solutions. Moreover, they should also focus their attention on technical features that may lead to an increase in TD principal components. In this direction, our analysis shows that it is not necessary to have a large set of software metrics to predict the trend of the TD because the model obtains excellent performances also by reducing the number of features considered, with the advantage of spending much less time in the phase of training. Therefore, practitioners should identify and monitor the subset of features more related to TD principal, for their projects. Moreover, we observed that it is possible to predict the trend of

Table 9
Results of the best permutations.

Combination/ Permutation	Activation function	Learning rate	Size of network	Number of layers	Timestamps	Batch size	Dropout rate	Validation accuracy	Testing F1-Score
C1/1	RELU	3	Small	6	5	128	0.15	0.9979	0.9972
C1/2	RELU	3	Small	8	10	128	0.15	0.9979	0.9972
C1/3	RELU	3	Small	8	5	128	0.15	0.9978	0.9970
C2/1	RELU	3	Small	6	5	128	0.15	0.9954	0.9953
C2/2	RELU	3	Medium	6	5	128	0.15	0.9953	0.9950
C2/3	RELU	3	Medium	7	5	128	0.15	0.9953	0.9952
C3/1	RELU	3	Small	6	5	128	0.15	0.9933	0.9939
C3/2	RELU	3	Medium	7	10	128	0.15	0.9932	0.9938
C3/3	RELU	3	Small	6	10	128	0.15	0.9932	0.9939
C4/1	RELU	3	Medium	7	5	128	0.15	0.9719	0.9700
C4/2	RELU	3	Small	8	10	128	0.15	0.9755	0.9693
C4/3	RELU	3	Medium	6	5	128	0.15	0.9759	0.9669

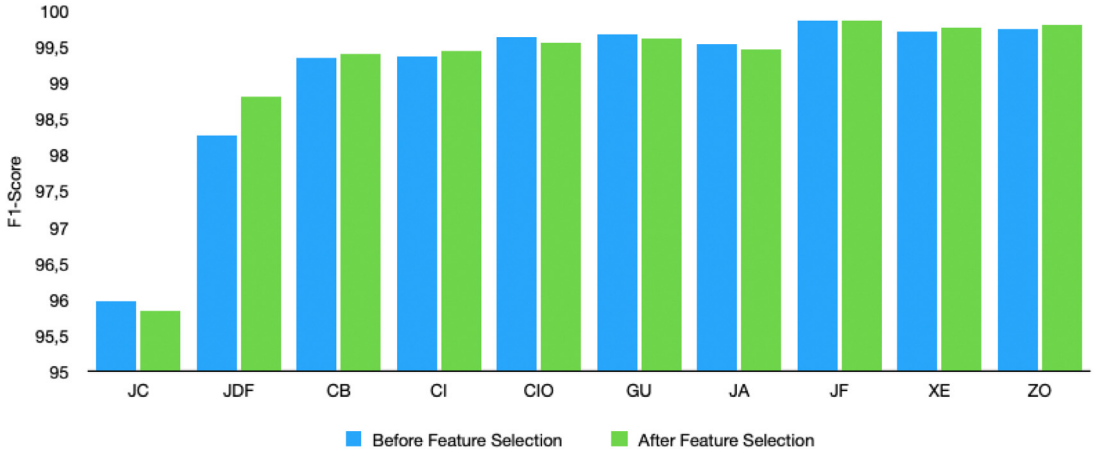

Fig. 4. Test F1-Score in comparison, before and after the features selection. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

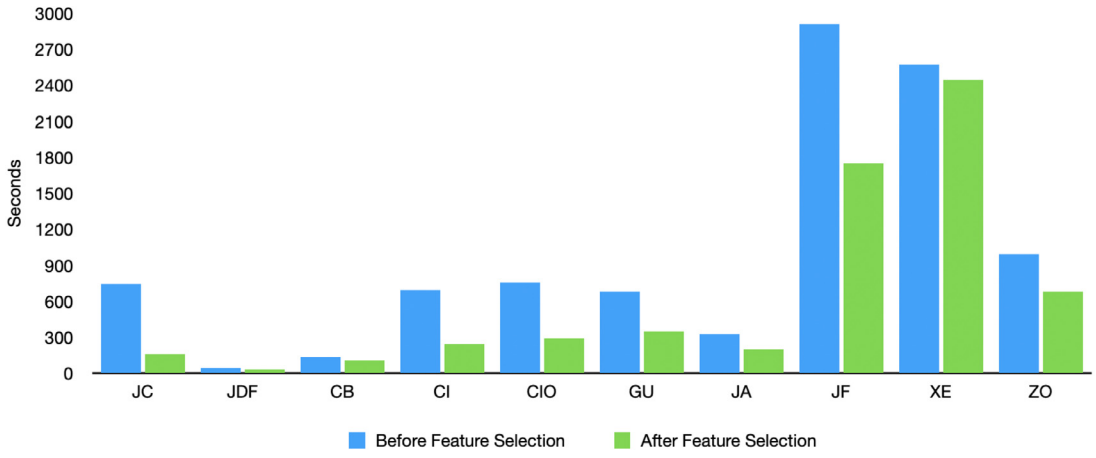

Fig. 5. Model training times in comparison, before and after the features selection. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the TD also cross-project and not only intra-project, thus testing the model on data completely unknown to it. This can be useful for practitioners to move a model from one project to another one.

Our research study aims to be one of the first attempts toward forecasting TD, indeed, there are still many aspects that remain unresolved and need to be addressed by researchers. First of all, it would be necessary to extend the domain of the software systems considered, having considered, in this study, only open-source systems with the Java programming language.

Researchers should also focus their attention on a standard measure to evaluate the interest of TD. Currently, methods and tools regarding the estimation of the TD are missing, while interest may lead to dramatic effects in terms of costs and decreasing in software quality.

Finally, practitioners and researchers can achieve important benefits from our toolkit. It includes a trained model that can be used as is to forecast TD principal, as well as a python script that can use to train a focused model using the company's data. Both the python script and the model can be easily integrated

into a company dashboard. Of course, it is not possible to ensure optimal results for any software project. However, the provided toolkit makes a significant contribution in the direction of TD Principal forecasting. An industrial company could leverage the predictive power from the models provided to deliver good predictions and adequately forecast future TD Principal trends of software applications. An interesting future direction to undertake would be the evolution of the proposed toolkit to develop a recommender system for the automatic prediction of the TD trend, able to give real-time suggestions to the developer for the management of the code quality of a software system. This would facilitate maintenance, and the software development community would also be able to forecast maintenance times and costs.

9. Threats to validity

This section discusses the threats to the validity of the proposed research that we have identified. In particular, we have identified different types of threats: construct validity, internal validity, external validity.

Respectively, for the first type of threat, it must be highlighted that a significant threat could concern the accuracy of the tools used to perform the detection of the TD in the source code and the detection of the metrics of the proposed model. This could skew the study results. Therefore, to mitigate this threat, we used the SonarQube and CKTool, tools used by most of the studies respectively for detecting some code indicators, including TD and software metrics.

On the other hand, the threat to internal validity concerns the consistency of the results based on the considered data. To this end, we performed a very precise data collection process for the construction of the dataset on which to train and test our model.

In conclusion, for threats to external validity, the number of software systems considered and the generalization of the results obtained are certainly a problem. The set considered is composed of ten systems, it is, therefore, smaller than the population of all OSS systems, and, again, our study includes only systems written in Java because the CK tool only works on this programming language. Therefore, we cannot speak of the generalization of systems in this sense. To mitigate this threat, we have chosen for our study all systems that are constantly evolving, with a very long history, and characterized by different dimensions, domains, sizes, time intervals, and the number of commits. Furthermore, the amount of data considered is really enormous, because for each system we have collected the entire evolutionary history, adopting a very fine granularity, commit by commit. However, several limits remain to the generalizability of the conclusions.

10. Conclusions

Technical debt is the term used to designate the consequences of intentional or unintentional negligence, errors, and deficiencies in the code, as corrections and maintenance slow down productivity and involve costly additional work.

The TD not only leads to greater effort and reduces productivity due to subsequent maintenance, but also generates higher costs. If the debt is not repaid regularly and on time, interest accrues, manifesting itself in a slowdown in development, a decline in productivity, and an increase in costs.

Precisely for this reason, TD has attracted both academic and corporate interest. To date, numerous studies have focused on various aspects concerning it, such as identification, management, or resolution, but to the best of our knowledge, there are still few contributions regarding its forecast.

What is missing is a tool to support decision-making mechanisms capable of understanding when the debt is becoming too large to be managed, to the point of leading to the decay of the software system.

In this regard, this study provides an approach based on a deep learning model to predict the trend of the TD in the source code of a software system. The major contribution of this study is due to the granularity of the forecast. We have used a TCN network, which, taking as input information relating to the TD, the quality of the software, and the process metrics can predict the trend of the TD in each class of the software system, commit by commit. We have considered the evolution of ten software systems, different from each other in characteristics and with histories belonging to very long periods.

The experimentation was first conducted on a metric model containing 58 different features and then following a feature selection process, these were reduced to 37, equally obtaining excellent efficiency of the forecasting model and significantly lowering the lead times. In fact, both in the first and in the second case we obtained an F-Score equal to 99% for seven systems, and 95% for the last system.

These results are very encouraging and demonstrate the high efficiency of the proposed model.

A further contribution is represented by the experimentation carried out on cross-project. Our model is in fact capable of predicting the trend of the TD, with an accuracy of 99% even for the evolution of software systems unknown to it, therefore not used during the training phase.

Therefore the results obtained demonstrate an excellent accuracy of the proposed model, which, as far as we know, is the first in the literature capable of verifying the trend of the TD during the evolution of a software system with a commit by commit granularity. These results are very encouraging and lead to further experimentation.

In the future it would be interesting to study the impact of the history of software systems on the accuracy of the model, and also to focus on the levels of increase and decrease of the TD in the source code, in order to understand the type of change.

CRediT authorship contribution statement

Pasquale Ardimento: Conceptualization, Methodology. **Lerina Aversano:** Conceptualization, Methodology, Writing – review & editing. **Mario Luca Bernardi:** Conceptualization, Methodology. **Marta Cimitile:** Conceptualization, Methodology, Writing – review & editing. **Martina Iammarino:** Investigation, Data curation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Albawi, S., Mohammed, T.A., Al-Zawi, S., 2017. Understanding of a convolutional neural network. In: 2017 International Conference on Engineering and Technology (ICET). pp. 1–6. http://dx.doi.org/10.1109/ICEngTechnol.2017.8308186.

Alves, N.S., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., Seaman, C., 2016a. Identification and management of technical debt: A systematic mapping study. Inf. Softw. Technol. 70, 100–121. http://dx.doi.org/10.1016/j.infsof.2015.10.008, URL: https://www.sciencedirect.com/science/article/pii/S0950584915001743.

Alves, N.S., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., Seaman, C., 2016b. Identification and management of technical debt: A systematic mapping study. Inf. Softw. Technol. 70, 100–121.

Ampatzoglou, A., Michailidis, A., Sarikyriakidis, C., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., 2018. A framework for managing interest in technical debt: An industrial validation. In: Proceedings of the 2018 International Conference on Technical Debt. In: TechDebt '18, Association for Computing Machinery, New York, NY, USA, pp. 115–124. http://dx.doi.org/10.1145/3194164.3194175.

Ardimento, P., Aversano, L., Bernardi, M.L., Cimitile, M., 2020. Temporal convolutional networks for just-in-time software defect prediction. In: van Sinderen, M., Fill, H., Maciaszek, L.A. (Eds.), Proceedings of the 15th International Conference on Software Technologies, ICSOFT 2020, Lieusaint, Paris, France, July 7-9, 2020. ScitePress, pp. 384–393. http://dx.doi.org/10.5220/0009890003840393.

Aversano, L., Bernardi, M.L., Cimitile, M., Iammarino, M., 2021. Technical debt predictive model through temporal convolutional network. In: 2021 International Joint Conference on Neural Networks (IJCNN). pp. 1–8. http://dx.doi.org/10.1109/IJCNN52387.2021.9534423.

Aversano, L., Bernardi, M.L., Cimitile, M., Iammarino, M., Romanyuk, K., 2020a. Investigating on the relationships between design smells removals and refactorings. In: Proceedings of ICSOFT 2020, Lieusaint, Paris, France, July 7-9, 2020. ScitePress, pp. 212–219. http://dx.doi.org/10.5220/0009887102120219.

Aversano, L., Iammarino, M., Carapella, M., Vecchio, A.D., Nardi, L., 2020b. On the relationship between self-admitted technical debt removals and technical debt measures. Algorithms 13 (7), URL: https://www.mdpi.com/1999-4893/13/7/168.

Avgeriou, P.C., Taibi, D., Ampatzoglou, A., Arcelli Fontana, F., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, A., Pigazzini, I., Saarimaki, N., Sas, D.D., de Toledo, S.S., Tsintzira, A.A., 2021. An overview and comparison of technical debt measurement tools. IEEE Softw. 38 (3), 61–71. http://dx.doi.org/10.1109/MS.2020.3024958.

Bai, S., Kolter, J.Z., Koltun, V., 2018. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. CoRR, abs/1803.01271. arXiv:1803.01271, URL: http://arxiv.org/abs/1803.01271.

Baldassarre, M.T., Lenarduzzi, V., Romano, S., Saarimäki, N., 2020. On the diffuseness of technical debt items and accuracy of remediation time when using sonarqube. Inf. Softw. Technol. 128, 106377.

Bengio, Y., 2000. Gradient-based optimization of hyperparameters. Neural Comput. 12 (8), 1889–1900. http://dx.doi.org/10.1162/089976600300015187, arXiv:https://doi.org/10.1162/089976600300015187.

Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B., 2011. Algorithms for hyper-parameter optimization. In: Proceedings of the 24th International Conference on Neural Information Processing Systems. In: NIPS'11, Curran Associates Inc., Red Hook, NY, USA, pp. 2546–2554.

Boehm, B., Beck, K., 2010. Perspectives [the changing nature of software evolution; the inevitability of evolution]. IEEE Softw. 27 (4), 26–29. http://dx.doi.org/10.1109/MS.2010.103.

Chatzigeorgiou, A., Ampatzoglou, A., Ampatzoglou, A., Amanatidis, T., 2015. Estimating the breaking point for technical debt. In: 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD). pp. 53–56. http://dx.doi.org/10.1109/MTD.2015.7332625.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20 (6), 476–493. http://dx.doi.org/10.1109/32.295895.

Chug, A., Malhotra, R., 2016. Benchmarking framework for maintainability prediction of open source software using object oriented metrics.

Cunningham, W., 1992a. The WyCash portfolio management system. In: Addendum To the Proceedings on Object-Oriented Programming Systems, Languages, and Applications. ACM.

Cunningham, W., 1992b. The WyCash portfolio management system. In: SIGPLAN OOPS Mess., Vol. 4. Association for Computing Machinery, New York, NY, USA, pp. 29–30. http://dx.doi.org/10.1145/157710.157715.

de Freitas Farias, M.A., de Mendonça Neto, M.G., d. Silva, A.B., Spínola, R.O., 2015. A contextualized vocabulary model for identifying technical debt on code comments. In: 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD). pp. 25–32. http://dx.doi.org/10.1109/MTD.2015.7332621.

Deng, L., Yu, D., et al., 2014. Deep learning: methods and applications. Found. Trends® Signal Process. 7 (3–4), 197–387.

Digkas, G., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., Matei, O., Heb, R., 2021. The risk of generating technical debt interest: A case study. SN Comput. Sci. 2, 12. http://dx.doi.org/10.1007/s42979-020-00406-6.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison Wesley.

Ioffe, S., Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37. In: ICML'15, JMLR.org, pp. 448–456.

Isaac Griffith, Taffahi, H., Izurieta, C., Claudio, D., 2014. A simulation study of practical methods for technical debt management in agile software development. In: Proceedings of the 2014 Winter Simulation Conference, Savanah, GA. pp. 119–128. http://dx.doi.org/10.1109/WSC.2014.7019961.

Köhn, H.-F., Hubert, L.J., 2014. Hierarchical cluster analysis. In: Wiley StatsRef: Statistics Reference Online. Wiley Online Library, pp. 1–13.

Letouzey, J., 2012. The SQALE method for evaluating technical debt. In: 2012 Third International Workshop on Managing Technical Debt (MTD). pp. 31–36. http://dx.doi.org/10.1109/MTD.2012.6225997.

Li, Z., Avgeriou, P., Liang, P., 2015a. A systematic mapping study on technical debt and its management. J. Syst. Softw. 101, 193–220. http://dx.doi.org/10.1016/j.jss.2014.12.027, URL: https://www.sciencedirect.com/science/article/pii/S0164121214002854.

Li, Z., Avgeriou, P., Liang, P., 2015b. A systematic mapping study on technical debt and its management. J. Syst. Softw. 101, 193–220.

Li, Z., Liang, P., Avgeriou, P., Guelfi, N., Ampatzoglou, A., 2014. An empirical investigation of modularity metrics for indicating architectural technical debt. In: Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures. Association for Computing Machinery, New York, NY, USA, pp. 119–128. http://dx.doi.org/10.1145/2602576.2602581.

Mannor, S., Peleg, D., Rubinstein, R., 2005. The cross entropy method for classification. In: Proceedings of the 22nd International Conference on Machine Learning. In: ICML '05, ACM, New York, NY, USA, pp. 561–568.

Nagappan, N., Ball, T., Zeller, A., 2006. Mining Metrics to Predict Component Failures. Association for Computing Machinery, New York, NY, USA, URL: https://doi.org/10.1145/1134285.1134349.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A., 2018. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. In: Proceedings of the 40th International Conference on Software Engineering. In: ICSE '18, Association for Computing Machinery, New York, NY, USA, p. 482. http://dx.doi.org/10.1145/3180155.3182532.

Pandey, S.K., Mishra, R.B., Tripathi, A.K., 2021. Machine learning based methods for software fault prediction: A survey. Expert Syst. Appl. 172, 114595. http://dx.doi.org/10.1016/j.eswa.2021.114595, URL: https://www.sciencedirect.com/science/article/pii/S0957417421000361.

Rios, N., Spínola, R.O., Mendonça, M., Seaman, C., 2018. The most common causes and effects of technical debt: first results from a global family of industrial surveys. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 1–10.

Rios, N., Spínola, R.O., Mendonça, M.G., Seaman, C.B., 2020. The practitioners' point of view on the concept of technical debt and its causes and consequences: a design for a global family of industrial surveys and its first results from Brazil. Empir. Softw. Eng. 1–72.

Schaul, T., Antonoglou, I., Silver, D., 2013. Unit tests for stochastic optimization. arXiv:1312.6055.

Skourletopoulos, G., Mavromoustakis, C.X., Bahsoon, R., Mastorakis, G., Pallis, E., 2014. Predicting and quantifying the technical debt in cloud software engineering. In: 2014 IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD). pp. 36–40. http://dx.doi.org/10.1109/CAMAD.2014.7033201.

Stopford, B., Wallace, K., Allspaw, J., 2017. Technical debt: challenges and perspectives. IEEE Softw. 34 (04), 79–81.

Tom, E., Aurum, A., Vidgen, R., 2013. An exploration of technical debt. J. Syst. Softw. 86 (6), 1498–1516.

Tsoukalas, D., Jankovic, M., Siavvas, M., Kehagias, D., Chatzigeorgiou, A., Tzovaras, D., 2019. On the applicability of time series models for technical debt forecasting. In: 15th China-Europe International Symposium on Software Engineering Education.

Tsoukalas, D., Kehagias, D., Siavvas, M., Chatzigeorgiou, A., 2020. Technical debt forecasting: An empirical study on open-source repositories. J. Syst. Softw. 170, 110777. http://dx.doi.org/10.1016/j.jss.2020.110777, URL: https://www.sciencedirect.com/science/article/pii/S0164121220301904.

Tsoukalas, D., Mathioudaki, M., Siavvas, M.G., Kehagias, D.D., Chatzigeorgiou, A., 2021. A clustering approach towards cross-project technical debt forecasting. SN Comput. Sci. 2 (1), 22. http://dx.doi.org/10.1007/s42979-020-00408-4.

Tsoukalas, D., Siavvas, M., Jankovic, M., Kehagias, D., Chatzigeorgiou, A., Tzovaras, D., 2018. Methods and tools for TD estimation and forecasting: A state-of-the-art survey. In: 2018 International Conference on Intelligent Systems (IS). pp. 698–705. http://dx.doi.org/10.1109/IS.2018.8710521.

Varma, S., Simon, R., 2006. Bias in error estimation when using cross-validation for model selection." BMC bioinformatics, 7(1), 91. BMC Bioinformatics 7, 91. http://dx.doi.org/10.1186/1471-2105-7-91.

Wehaibi, S., Shihab, E., Guerrouj, L., 2016. Examining the impact of self-admitted technical debt on software quality. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). 1, pp. 179–188. http://dx.doi.org/10.1109/SANER.2016.72.

Yang, Z., Yang, D., Dyer, C., He, X., Smola, A., Hovy, E., 2016. Hierarchical attention networks for document classification. In: Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Association for Computational Linguistics, San Diego, California, pp. 1480–1489. http://dx.doi.org/10.18653/v1/N16-1174, URL: https://www.aclweb.org/anthology/N16-1174.

Yli-Huumo, J., Maglyas, A., Smolander, K., 2016. How do software development teams manage technical debt? – An empirical study. J. Syst. Softw. 120, 195–218. http://dx.doi.org/10.1016/j.jss.2016.05.018, URL: https://www.sciencedirect.com/science/article/pii/S016412121630053X.

Zazworka, N., Shaw, M.A., Shull, F., Seaman, C., 2011. Investigating the impact of design debt on software quality. In: Proceedings of the 2nd Workshop on Managing Technical Debt. In: MTD '11, Association for Computing Machinery, New York, NY, USA, pp. 17–23. http://dx.doi.org/10.1145/1985362.1985366.

Pasquale Ardimento
pasquale.ardimento@uniba.it

Pasquale Ardimento received a Ph.D. degree in computer science from the University of Bari. In 2004, he collaborated as a Research Student with the Department of Engineering, University of Durham, U.K. From 2005 to 2007, he was a Contract Researcher with the University of Bari. Since 2008, he has been a Researcher with the Computer Science Department, University of Bari Aldo Moro. He has authored more than 60 articles published in journals and conference proceedings. His main research interests include software engineering (maintenance, testing, data mining on software systems, software quality assurance, and computational intelligence). He serves both as a member of the program and organizing committees of conferences and as a reviewer for articles submitted to some of the main journals and magazines in the field of software engineering and software maintenance. He was involved in several national research projects and European projects and served as a Reviewer for many international and national conferences.

Lerina Aversano
aversano@unisannio.it

Lerina Aversano is an associate professor at the Department of Engineering of the University of Sannio Benevento (Italy). She received the Ph.D. in Computer Engineering in July 2003 at the same University and was assistant professor from 2005. She also was a research leader at RCOST – Research Centre On Software Technology – of the University of Sannio from 2005. Her research interests include software maintenance, program comprehension, reverse engineering, reengineering, migration, business process modeling, business process evolution, software system evolution, software quality.

Mario Luca Bernardi
bernardi@unisannio.it

Mario Luca Bernardi received the Laurea degree in Computer Science Engineering from the University of Naples "Federico II", Italy, in 2003 and the Ph.D. degree in Information Engineering from the University of Sannio in 2007. He is currently an assistant professor of Computer Science at the Giustino Fortunato University. Since 2003 he has been a researcher in the field of software engineering and his list of publications contains more than 60 papers published in journals and conference proceedings. His main research interests include software engineering (maintenance, testing, business process management, reverse engineering and data mining on software systems, software quality assurance with particular interest on internal quality metrics and on new paradigms for software modularity, including aspectoriented software, component-based software and model-driven development). He serves both as a member of the program and organizing committees of conferences, and as associate editor and reviewer of papers submitted to some of the main journals and magazines in the field of software engineering, software maintenance and program comprehension.

Marta Cimitile
marta.cimitile@unitelmasapienza.it

Marta Cimitile is Assistant Professor and Aggregated Professor at Unitelma Sapienza University of Rome (Italy). She received a Ph.D. in Computer Science in 06/05/2008 at the Department of Computer Science at the University of Bari and she received the Italian Scientific Qualification for the Associate Professor position in Computer Science Engineering in April 2017. She published more than fifty papers at international conferences and journals. Her main research topics are: Business Process Management and modeling, Knowledge modeling and Discovering, Process and Data Mining in Software Engineering Environment. In the last year, she was involved in several industrial and research projects and she is a founding member of the SpinOff of University of Bari named Software Engineering Research and Practices s.r.l. (www.serandp.com). She was in the program and organizing committees of several international conferences, she is reviewer to some of the main journals and magazines in the field of Knowledge Management and Software Engineering, knowledge representation and transfer and data mining and she is in the Editorial Board of the Journal of Information and Knowledge Management, PeerJ Computer Science. She is IEEE Member and IEEE Access reviewer.

Martina Iammarino
iammarino@unisannio.it

Martina Iammarino is a Ph.D. student in Computer Engineering. She received a master's degree in Computer Engineering at the University of Sannio in Benevento, Italy. Her research interests include software quality, software metrics, refactoring, modeling, evaluation and evolution, technical debt, and software measurement.