

Technical Debt Forecasting Based on Deep Learning Techniques

Maria Mathioudaki, Dimitrios Tsoukalas(✉), Miltiadis Siavvas, and Dionysios Kehagias

Centre for Research and Technology Hellas, Thessaloniki, Greece
{mariamathi,tsoukj,siavvasm,diok}@iti.gr

Abstract. Technical debt (TD) is a metaphor commonly used to reflect the consequences of quality compromises that can derive short-term benefits but may result in quality decay of software products in the long run. While a broad variety of methods and tools have been proposed over the years for the identification and quantification of TD during the software development cycle, it is not until recently that researchers have turned their interest towards methods aiming to forecast the future TD evolution of a software project. Predicting the future value of TD could facilitate decision-making tasks regarding software maintenance and assist developers and project managers in taking proactive actions regarding TD repayment. In previous relevant studies, time series analysis and Machine Learning techniques have been employed in order to generate meaningful TD forecasts. While these approaches have been proven capable of producing reliable TD predictions, their predictive performance has been observed to decrease significantly for long-term predictions. To this end, in the present paper we investigate whether the adoption of Deep Learning may lead to more accurate long-term TD prediction. For this purpose, Deep Learning models are constructed, evaluated, and compared based on a dataset of five popular real-world software applications. The results of our analysis indicate that the adoption of Deep Learning results in TD forecasting models with sufficient predictive performance up to 150 steps ahead into the future.

Keywords: Software quality · Technical debt · Forecasting · Deep learning

1 Introduction

The Technical Debt (TD) term was first introduced by Ward Cunningham [1] to outline the issue of additional software maintenance effort caused by technical shortcuts taken usually in favor of shorter time-to-market. Since then, it is widely used in practice to facilitate the discussion among technical and non-technical stakeholders, due to its ability to convey the consequences of deliberate or inadvertent quality compromises in monetary terms. While initially associated with the software implementation phase, the TD metaphor was later broadened to

© Springer Nature Switzerland AG 2021
O. Gervasi et al. (Eds.): ICCSA 2021, LNCS 12955, pp. 306–322, 2021.
https://doi.org/10.1007/978-3-030-87007-2_22

other phases of the software development lifecycle (SDLC) [2], such as design and architecture. Similarly to financial debt, TD that is accumulated due to problematic design and implementation choices needs to be repaid early enough in the SDLC, mainly by performing refactoring operations. If not done so, then it can generate interest payments in the form of increased future costs that would be difficult to be paid off and in extreme situations may lead to an unmaintainable software product and thus, to technical bankruptcy [3].

Managing TD across the entire SDLC is a challenging task and therefore, a multitude of methods and tools have been proposed by researchers and adopted by practitioners [4] towards this direction. Most of these approaches however focus on the TD value (in terms of money or effort) of a software project in its current state and often tend to overlook the potential utility that a future approximation of this value would introduce in the TD Management process [5]. More specifically, being able to predict the future TD evolution of a software project could allow project managers and developers to timely react to the accumulation of TD and, thus, employ suitable repayment activities promptly [5] to preserve a software application to a satisfactory quality level. Although to date numerous research endeavours have dealt with predicting the evolution of various aspects directly or indirectly related to TD, such as software maintainability [6,7], fault-proneness [8–11], change-proneness [12,13] and code smells [14], the contributions that exist in the literature regarding the forecasting of TD itself are limited.

In our previous work towards this direction [15–17], we have contributed to the TD forecasting challenge by investigating the performance of a multitude of different forecasting methods ranging from simple statistical time series models to more sophisticated approaches, such as causal and Machine Learning (ML) models. The outcome of our work indicated that these methods are indeed capable of providing accurate and useful results. However, despite the fact that these approaches were proven able to produce reliable TD predictions for relatively short forecasting horizons, their predictive performance dropped significantly for longer forecasting horizons (i.e., long-term predictions). More specifically, the performance of statistical time series models was found to be satisfactory for forecasts up to 8 commits ahead, while the causal and ML models were able to provide reliable results up to 40 commits ahead. This obviously affects the practicality of the produced models, as they cannot be applied to cases where a software company requires a long-term TD management plan.

To this end, in the present paper we investigate whether the adoption of Deep Learning models may lead to accurate and practical TD forecasts, giving particular emphasis on the long-term predictions that have been the most important limitation of our previous relevant studies [15,16]. For this purpose, we utilize TD measurements throughout the different commits of five open-source Java projects obtained from the Technical Debt Dataset [18]. Subsequently, based on this dataset, a Multi-layer Perceptron (MLP) model is constructed and experiments are performed for each of the five projects for 5 to 150 commits ahead.

In each one of these cases, the produced models are evaluated with respect to their accuracy and practicality based on a set of performance metrics.

The rest of the paper is organised as follows: Sect. 2 explores the related work in the field of TD forecasting. Section 3 discusses details regarding the methodology that we followed throughout this work, while Sect. 4 presents the results of the analysis and a discussion on the findings. Finally, Sect. 5 concludes the paper and proposes directions for future work.

2 Related Work

A groundbreaking study on the laws of software evolution by Lehman [19] states that while any software system needs to be continually evolving and adapting to the environment, doing so will increase its complexity and decrease its quality unless maintained properly. Thereby, analyzing, understanding, and predicting the evolution of a software system can provide estimates and insight on the evolution of its quality as well. In a relevant study, Mens [20] raises the awareness on the need for accurate models able to track software evolution and thus estimate the quality and predict the future maintenance cost of a software system. Towards this direction, numerous research endeavors have addressed the topic of predicting the evolution of various quality aspects directly or indirectly related to the TD of a software project, such as software maintainability [6,7], fault-proneness [8,9,11], change-proneness [12,13] and code smells [14].

As regards the software maintainability prediction, in a study by Chug and Malhotra [6], the authors employ 17 ML techniques using object-oriented metrics as predictors and conclude that the best performance is achieved by genetically adaptive learning models. In a similar study, Elish and Elish [7] compare various ML techniques and conclude that the TreeNet model is the best-performing model for predicting software maintainability. Regarding fault-proneness prediction, a study by Arisholm et al. [8] proposes a multivariate model that uses historical fault and change data of various open-source systems as predictors. The authors report that a satisfactory level of accuracy can be achieved using their model. In another study, Nagappan et al. [9] accurately predict future fault-proneness using regression models trained on code metrics, while in a similar study, Gondra et al. [11] employ and train an Artificial Neural Network having software metrics as input and conclude that their model outperforms those of similar studies on fault-proneness prediction. Regarding change-proneness prediction, in a study by Pritam et al. [12], the authors investigate various ML techniques using code smells as predictors and conclude that code smells are indeed a good predictor of class change proneness. In another study by Abbas et al. [13], the authors apply 10 ML techniques to investigate the effectiveness of object-oriented metrics in predicting software change-prone modules. They report that Random Forest is the best-performing model. Finally, regarding code smells prediction, Fontana et al. [14] employ 16 ML techniques on 74 software projects and conclude that J48 and Random Forest demonstrate the highest performance among the investigated models.

The large number of relevant studies trying to contribute towards predicting the evolution of various software quality aspects unveils the importance of this challenge in the software engineering community [21]. However, since TD is considered as an indicator of software quality, predicting its future value can be deemed equally significant. A careful search in the relevant literature reveals that while several existing research works deal with the evolution of TD and its implications for the software development process [22], it is not until recently that researchers have turned their interest towards investigating efficient methods aiming to forecast the future TD evolution of a software project.

The potential impact of developing forecasting models for accurate prediction of TD evolution has been discussed in a related study [5], in which the authors point out that it would be interesting to investigate if the adoption of such models in practice would result in higher-quality software products. In a first attempt towards modelling and forecasting the TD evolution of long-lived, open-source software projects, a study by Tsoukalas et al. [15] proposes the use of time series analysis techniques and more specifically, an autoregressive integrated moving average (ARIMA) model. While concluding that a single ARIMA model can generate accurate short-term TD forecasts for each one of the five investigated open-source projects, the authors notice that the performance of this model decreases significantly for forecasting horizons longer than eight commits ahead. In an attempt to extend their previous work and present a more holistic view on the TD forecasting concept, the same authors [16] conduct a follow-up study, where they investigate the ability of a large set of ML methods to predict the TD evolution of 15 open-source projects. By examining various forecasting horizons, they conclude that the non-linear Random Forest regression is able to achieve a sufficient level of accuracy for forecasts up to 40 commits ahead.

Existing TD forecasting methods have not demonstrated satisfactory results in long-term TD forecasts. Under those circumstances, we aim to examine whether taking advantage of deep learning approaches could be the key for overcoming this challenge. Such a contribution would enable software companies to plan accurate TD management strategies on time and thus, avoid unforeseen situations long-term.

3 Methodology and Experiment Setup

In this section, detailed information is provided on the overall methodology that was adopted in the present paper. More specifically, this section describes the dataset that was used for the purposes of this study, the data pre-processing that took place before the experiments, the model construction process and model performance metrics that were used, and last but not least, the strategy that was followed for the experimental execution.

3.1 Dataset

The first step of the proposed approach comprises the construction of the dataset. For the purposes of investigating whether the adoption of Deep Learning models

may lead to accurate short-, but most importantly, long-term TD forecasts, our dataset needs to consist of multiple software projects along with their past TD evolution, i.e., TD measurements spanning across their commit history. These TD measurements will serve as input to train and evaluate our models.

For the purposes of the present study, we decided to exploit the "Technical Debt dataset"[1], that is, a dataset that was constructed and made publicly available in a study by Lenarduzzi et al. [18] to facilitate TD research. This dataset is based on 33 real-world open-source Java projects obtained from the Apache Software Foundation and contains TD measurements for all available commits of the aforementioned projects. TD measurements were obtained using SonarQube[2], a widely-used open-source static code analysis tool that has been used as proof of concept in multiple research studies [4]. Each one of the 33 projects is provided as a csv file, where the columns contain the software-related metrics (including TD measurements), as extracted from SonarQube, and the rows correspond to the different commits through time. In the context of the present work, we decided to select five projects for constructing and evaluating our TD forecasting models. The selection criteria were based on the commit frequency (at least once per week) and project lifespan (at least 3 years of commit activity). Therefore, our final set consists of the following projects: Ambari, Commons-codec, Commons-io, Httpcomponents-client and Mina-sshd. These projects are presented in Table 1 accompanied by additional information on the total number of analyzed commits and the analysis lifetime.

Table 1. Selected projects of the TD dataset

Project name	Analyzed commits	Analysis lifetime
Ambari	13397	09/11–06/15
Commons-codec	1726	04/03–05/18
Commons-io	2118	01/02–06/18
Httpcomponents-client	2867	12/05–06/18
Mina-sshd	1370	12/08–06/18

For the purposes of the current study, we did not exploit all of the software-related metrics that are available in the Technical Debt dataset. In fact, to construct the TD forecasting models we followed a *univariate* modelling approach, meaning that no indicators other than the TD value itself were taken under consideration during the model training process. The reason behind this choice is that we are interested in investigating whether the MLP models are capable of discovering patterns within the TD evolution itself, setting aside causes and relationships with other software-related metrics that could potentially act as TD indicators (i.e., predictors).

[1] https://github.com/clowee/The-Technical-Debt-Dataset.
[2] https://www.sonarqube.org/.

That being said, we set *total technical debt* as our target variable, i.e., the metric that we try to forecast. Similarly to our previous work [16], we define *total technical debt* as the total remediation effort (in minutes) that is required in order to fix all reliability, maintainability and security issues that are detected in the source code of a software project. This metric is therefore calculated as the sum of three metrics taken from the "Technical Debt dataset", namely *reliability remediation effort*, *security remediation effort*, and *maintainability remediation effort*. As explained in Sect. 3.2, values of *total technical debt* in previous time steps will act as independent variables.

3.2 Data Pre-processing

In this section, we briefly describe the data pre-processing techniques that were applied on the final dataset (described in Sect. 3.1), in order to transform it in a form that is suitable for further analysis. In particular, we first describe the sliding window method, i.e., a commonly used method able to transform time series data into a form that is suitable for ML tasks. Secondly, we describe the data scaling step, which is deemed necessary in order to improve the accuracy and performance of Deep Learning models.

Sliding Window. As mentioned earlier, the dataset that was used in this study consists only of TD values computed throughout the whole evolution of five different software projects. Therefore, we eventually end up with five tables, where each table contains a single column representing the *total technical debt* of a specific commit, throughout different commits in time (rows) of each project, in a form of time series data. As ML (and therefore Deep Learning) models do not at first hand support the idea of evolution over time, time series data need to be reconstructed in a form appropriate for supervised learning problems.

Sliding window is a commonly used method that can be applied to address such problems. In short, this method transforms each initial sample (i.e., row) of a time series dataset by introducing, besides the current information, also past information (e.g., the TD values of multiple prior commits) and future information (e.g., the TD value of a future commit) simultaneously into a single row. By that, all rows of the dataset will be extended to include new columns containing not only information regarding the present, but also information regarding the past and the future. In our case, after applying the sliding window method, TD values of prior commits will be treated as independent variables, while the TD value of the future commit will become the target variable. Obviously, the selection of the future commit depends on the forecasting horizon that we want to make predictions for. The number of new columns (i.e., past time steps) that we want to include into each new row is called the "window size" and in fact denotes the number of independent variables that will be considered as input to our models. Generally speaking, there is no "golden rule" that can point to an ideal window size for a particular dataset or forecasting horizon. Most of the time, repeated tests with a wide range of window sizes need to take place, so

that one can empirically decide on the most suitable number based on error minimization criteria. In our case, after examining the performance of the models while testing for different window sizes, we observed that for forecasts generated for 5–40 steps ahead, the window size that provides us with the lowest errors is 5. For predictions concerning larger steps ahead in time (i.e., 60–150) we ended up using a window size of 10. These observations apply to each one of the five different project datasets that are investigated in this study.

Data Scaling. One of the most critical steps before employing any ML or Deep Learning algorithm is data scaling, due to the fact that ML techniques (especially neural networks) produce better results when data are normalized. Therefore, we used *StandardScaler*, which scales the data so that the final distribution centers around 0 with a standard deviation of 1 (Gaussian Distribution). Centering and scaling are two processes that happen independently by calculating the corresponding statistics on the values of the dataset samples. StandardScaler transformation was integrated into the Walk-forward Train-Test validation process (described in Sect. 3.4) where, during each iteration, it was fitted only to the training set (to create realistic conditions) and then, used to transform both the training and test set. For data pre-processing tasks we used Python programming language and more specifically the scikit-learn[3] ML library.

3.3 Model Construction and Performance Metrics

Trying to forecast the future TD evolution of a software project can be considered as a regression problem, as the target variable (i.e., *total technical debt*) is a continuous variable and thus, it should be treated accordingly. Contrary to the previous studies, where statistical time series and ML models were examined, in the present work we chose to employ the Multilayer Perceptron (MLP), a supervised Deep Learning model suitable for regression tasks. In this section, we will describe in detail the model that was employed for this purpose, as well as the performance metrics that were used in order to evaluate its performance.

Model Construction. An MLP is a class of feedforward Artificial Neural Networks (ANNs). It receives as input a set of features and a target variable and it uses a non-linear function for either classification or regression problems in order to "learn" how to predict the values of the target variable. Most of the times, it consists of an input layer that corresponds to a set of neurons representing the input features, one or more hidden layers that transform the values of the previous layer using a weighted linear sum followed by a non-linear activation function, and finally, an output layer that takes the values from the last hidden layer and transforms them into output values.

An MLP model is extremely sensitive to input features and therefore the configuration of its parameters strongly depends on the dataset that will be

[3] https://scikit-learn.org/stable/index.html.

used to train it. These parameters include the number of hidden layers, neurons, epochs but also, the different activation and loss functions, the batch size and the optimizer that can be put in practice. Thus, there is a process that needs to take place in order to determine the optimal parameters, called hyperparamater tuning. In the context of this work, we performed hyperparameter tuning using the *GridSearch* method, which is commonly used to find the optimal hyperparameters, by performing an exhaustive search over specified parameter values for an estimator. We chose the Mean Absolute Percentage Error (MAPE) (more details in Sect. 3.3) as the objective function of the estimator.

In Table 2 we present the optimal MLP parameters as calculated during GridSearch. It should be noted that these parameters are common for each one of the five datasets (i.e., five projects) that are examined in this study.

Table 2. The optimal hyperparameters of the Multi-layer Perceptron (MLP).

Hyperparameter name	Value
Number of layers	4
Number of hidden layers	3
Number of hidden units (per hidden layer)	200/100/50
Gradient descent optimizer	Adam
Batch size	5
Epochs	50
Activation function	Relu
Loss function	Mean squared error

For the implementation of the MLP model we used Python and more specifically the Tensorflow Keras[4] library.

Performance Metrics. As in any ML task, in this study we aim to generate predictions for the target variable (i.e., *total technical debt*) which will be as close as possible to the ground truth values and at the same time avoid overfitting. In a few words, we aim to maximise the accuracy of our models towards forecasting the future TD evolution of a software project. For evaluating the models predictive power, we decided to use the *Mean Absolute Percentage Error* (MAPE). MAPE is a commonly used measure for the evaluation of forecasting models that uses absolute values to calculate the error in percentage terms. The equation describing MAPE is offered below:

$$MAPE = \frac{100}{n} \sum_{i=1}^{n} \frac{|y_i - \widehat{y_i}|}{y_i} \tag{1}$$

where n is the number of observations, y_i the real and $\widehat{y_i}$ the predicted values.

[4] https://www.tensorflow.org/guide/keras/sequential_model.

To complement the evaluation of our results and at the same time remain inline with our previous studies [15,16], we also calculated the *Mean Absolute Error* (MAE) and the *Root Mean Squared Error* (RMSE), whose presence is usual when it comes to forecasting tasks. MAE calculates the magnitude of the errors coming from a group of forecasts by not taking under consideration their trend. RMSE represents the quantification of the quadratic mean of the differences between the observed and the predicted values. The equations of MAE and RMSE are the following:

$$MAE = \frac{\sum_{i=1}^{n} |y_i - \widehat{y_i}|}{n} \tag{2}$$

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y_i})^2} \tag{3}$$

again, n is the number of observations, y_i the real and $\widehat{y_i}$ the predicted values.

3.4 Model Training and Execution

Once the data pre-processing and model construction phases are over, we may start the experimental process aiming towards the evaluation of the constructed model. In this section, we provide a brief description of the experiments which were performed towards two different directions so that we can be able to evaluate whether the constructed MLP model can provide meaningful short-term and most importantly long-term forecasts. First, Walk-Forward Train-Test experiments are performed so as to compute and evaluate the performance metrics of the model for different horizons in the future by means of time series validation. Secondly, we present indicative examples where we execute the constructed model in order to visualise the results and hence, gain an extensive perception on the model's ability to capture the potential patterns hidden in the datasets.

Walk-Forward Validation. In ML problems, k-fold cross-validation is a widely used method in order to efficiently evaluate a model's performance. However, the concept of k-fold cross-validation is to randomly split the dataset into k folds, from which k−1 participate in training and the remaining one participates in testing. As a result, it is not the right tool as far as time series data are concerned, since in the case of time series data, samples cannot be randomly split into folds without respecting the temporal order. To overcome this issue, we adopted *Walk-forward Train-Test validation*, a process similar to cross-validation that is suitable for evaluating time series data based on the notion that models are updated when new observations are made available. Shortly, in Walk-forward Train-Test validation, the dataset is split into n equally distanced points, hence, we end up with *n+1* subsets. Initially, the first subset is used to train a model whose accuracy is tested against the consecutive subsets. Then, the first subset is merged with the second one and a new model is trained and tested against

the consecutive remaining subsets. The process is repeated until there is only one subset left to serve as a test set. The results of the n constructed models are then averaged to offer an estimation of the model's accuracy.

In our case, each of the five datasets (i.e., projects) was split at five equally distanced points and Walk-forward Train-Test validation was followed as described above. Experiments were performed for producing forecasts for 5, 10, 20, 40, 60, 80, 100, 120 and 150 steps ahead for each iteration of the process. All of the forecasts that were generated were then compared to the corresponding real values to compute the averaged MAE, RMSE and MAPE errors. In this way, we extract information which step by step illustrate the model's performance.

Model Execution. After evaluating the performance of the produced models through the Walk-forward Train-Test validation, the next step is to execute the produced models and present indicative visualizations of the forecasting results. This process will allow us to visually judge whether the models are able to catch the underlying trends or patterns throughout the evolution of the TD.

Since the structure of the Deep Learning models examined in this study does not directly support more than one outputs, in this type of experiments we decided to use the *Direct* approach, which basically means that for a chosen forecasting horizon, an equal number of independent models will be used to forecast each of the intermediate steps. To make this more clear, let's assume that we want to forecast the TD evolution of a software project for k steps ahead. To achieve this, k models will be executed, where each one of them will be responsible for predicting a specific step (starting from 1 step to k steps), while their forecasted TD values will be aggregated into a common vector, and then plotted as the projected TD evolution.

4 Results and Discussion

In this section, we present the results of our experiments, following the strategy that has been described in Sect. 3.4. Firstly, we present and discuss the results of the Walk-forward Train-Test validation experiments using the constructed MLP model (described in Sect. 3.3) on the selected five projects under investigation. In Table 3, the reader may find the main performance metric used in this study (i.e., MAPE), along with the two complement performance metrics (i.e., MAE and RMSE) across the five different projects that were analysed, for 5 up to 150 versions ahead.

By having a first look at Table 3, we observe that the three performance metrics of the MLP model across the five investigated projects indicate a quite sufficient level of accuracy. This is also depicted in their total averaged values (in bold font) for all investigated forecasting horizons. More specifically, by inspecting the average MAPE values of each project, we observe that they range between 3.04% for the Ambari project and 8.28% for the Mina-sshd project. In addition, while observing the MAPE fluctuation between the various forecasting horizons, we can state that the MLP model has achieved a stable performance for all

Table 3. TD forecasting performance metrics of the MLP on the 5 projects using walk-forward train-test validation

Project	Commits ahead	MAE (min)	RMSE (min)	MAPE (%)
Ambari	5–50 (avg)	1526.332	1873.303	1.252
	60–100 (avg)	3815.803	4451.718	3.503
	110–140 (avg)	6745.792	7584.719	6.121
	150	6864.296	7702.084	6.677
	Total Average	**3341.409**	**3888.185**	**3.037**
Commons-codec	5–50 (avg)	362.476	461.219	3.475
	60–100 (avg)	738.392	900.343	6.630
	110–140 (avg)	858.872	1066.487	7.581
	150	946.965	1131.661	8.420
	Total Average	**580.437**	**722.427**	**5.268**
Commons-io	5–50 (avg)	519.536	621.641	2.684
	60–100 (avg)	980.595	1143.73	4.778
	110–140 (avg)	1099.282	1330.886	5.473
	150	896.301	1050.964	4.223
	Total Average	**737.863**	**878.834**	**3.644**
Httpcomponents-client	5–50 (avg)	1450.848	1974.844	3.468
	60–100 (avg)	2288.026	2836.171	5.377
	110–140 (avg)	3207.025	3868.439	7.614
	150	3294.445	4028.666	7.433
	Total Average	**2031.452**	**2589.210**	**4.763**
Mina-sshd	5–50 (avg)	1647.925	2037.843	4.492
	60–100 (avg)	3792.056	4484.627	10.371
	110–140 (avg)	4877.660	5742.890	12.264
	150	5719.797	6886.471	14.436
	Total Average	**3111.452**	**3726.793**	**8.283**

steps (i.e., commits) ahead. Of course, it is reasonable to expect that while the model's predictive power is higher for shorter forecasting horizons, it decreases as we try to forecast for longer horizons into the future. This is an expected outcome in the field of forecasting, verified also by our previous studies on TD. However, even though longer forecasting horizons have a noticeable impact on the model's performance, the quality of the forecasts remains quite satisfactory. More specifically, the MAPE values of the MLP model for 150 commits-ahead forecasts range between 4.22% for Commons-io and 14.44% for Mina-sshd.

In order to gain a deeper understanding on the results and on their potentially underlying patterns, Fig. 1 shows the evolution of the MAPE values for each project across the various forecasting horizons. By taking a closer look at Fig. 1, we can observe that the lowest MAPE values (ranging from 0.27% to 6.67%) were

obtained when the MLP model was trained and tested on the Ambari project (green line), except for the cases of 120 and 150 commits ahead, where the MLP on Commons-io (blue line) demonstrated slightly better results (i.e., MAPE values of 5.47% and 4.22% respectively). On the other hand, the highest MAPE values (ranging from 3.26% to 16.18%) were observed in the case of the Mina-sshd project (orange line). The MLP model exhibited quite similar performance with respect to Commons-codec (red line) and Httpcomponents-client (purple line) projects.

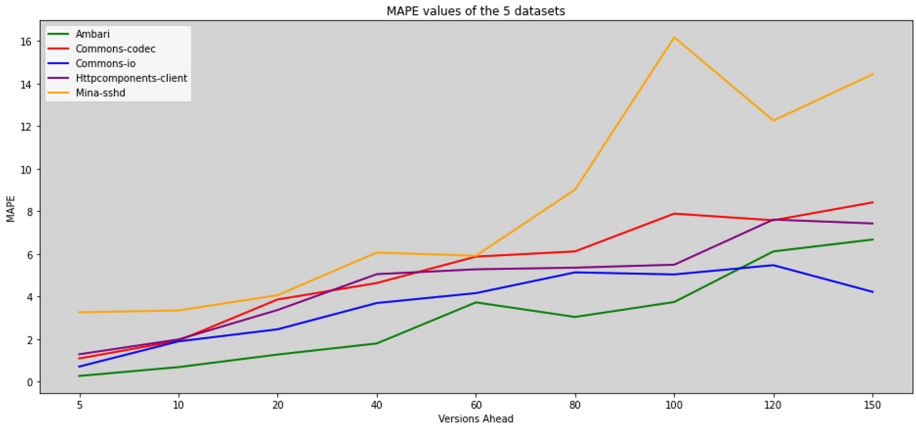

Fig. 1. MAPE values of the MLP on the 5 projects using Walk-forward Train-Test validation for 5 to 150 commits ahead. (Color figure online)

While there is a natural tendency for MAPE values to increase proportionally to the horizon of forecast, there are some exceptions identified throughout our experiments that need to be noted. More specifically, regarding the Mina-sshd (orange line) and Httpcomponents-client (red line) projects, we can observe a spike of the MAPE value at 100 commits-ahead, followed by a drop and then an increase. Similarly, Httpcomponents-client (purple line) and Commons-io (blue line) exhibit a slight decrease of the MAPE value at 150 commits ahead compared to the higher MAPE value observed at 120 commits ahead. As these "abnormalities" do not seem to appear consistently at the same forecasting horizons for every project, we can only assume that they are caused by strong variabilities in the structure of the data itself (i.e., a smooth TD evolution would be much easier to predict compared to a TD evolution with many fluctuations).

As previously stated, in our past studies we have employed a wide range of statistical and ML techniques, which have proven capable of producing reliable short-and mid-term TD predictions, but failed to do so for longer-term forecasts. Specifically, in our study on ML models [16], our best performing model (i.e., Random Forest) demonstrated good performance for forecasts up to 40 commits ahead (average MAPE value 5.94%) whereas, in our study on time series

forecasting [15], the ARIMA model's predictions were satisfactory for up to 12 steps ahead (average MAPE value of 1.77%). We believe that the MLP model employed within the context of this study managed to fill this gap by demonstrating promising results even for long-term forecasts. That said, we need to examine and compare the short- mid- and long-term performance of our MLP model with the models used in our previous studies. To do so, we decided to employ both the Random Forest and ARIMA as *baseline models* and directly compare their performance with the MLP so that we may validly assess the contribution that the latter possibly offers.

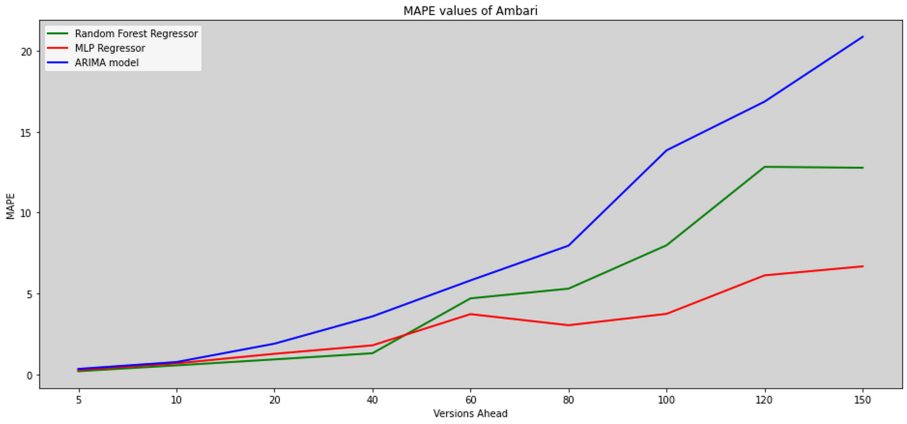

Fig. 2. MAPE values of MLP, Random Forest and the ARIMA model on Ambari project for 5 to 150 commits ahead. (Color figure online)

Figure 2 presents the MAPE values that were obtained when running the Random Forest (green line), the ARIMA (blue line), and the MLP Regressor (red line) on the Ambari project for 5 up to 150 commits-ahead forecasts. As can be seen, for 5 commits ahead the performance of the three models is quite similar. Starting from 10 commits ahead however, the lines are starting to be distinguishable with Random Forest providing slightly lower errors than the MLP, while the ARIMA model's performance starts to drop significantly. This pattern continues for up to 60 commits ahead, where we can clearly notice a shift between MLP and Random Forest. Overall, we conclude that the results are in line with the observations made not only within this paper, but also in our previous works. More specifically, for short-term forecasts (up to 10 commits ahead) the Random Forest, MLP and ARIMA models display equivalent performance. For 20 to 40 commits-ahead predictions, we obtain the lowest MAPE values when Random Forest is employed and the highest when the ARIMA model is used. The scene takes an interesting swift for horizons longer than 60 and up to 150 commits ahead. For these cases, the MLP model clearly outperforms the other two baseline models providing us with a maximum MAPE value of 6.67%.

Following the performance evaluation and comparison between the MLP and the baseline models through the Walk-forward Train-Test validation, the next step is to actually execute the three models on the projects under investigation and visualize the results. As stated in Sect. 3.4, this process will allow us to visually judge whether the MLP model is able to catch the underlying patterns throughout the evolution of the TD (especially for long-term forecasts), in a more accurate way than the other two baseline models do. In Fig. 3, we provide an example of forecasting the TD evolution of Ambari project for 150 steps (commits), using the three models under investigation. The blue line depicts the ground truth, while the red, green and purple lines depict the forecasted values using the MLP, ARIMA, and Random Forest models respectively. It should be noted that the samples covered by the lines that represent forecasts were excluded from the training set.

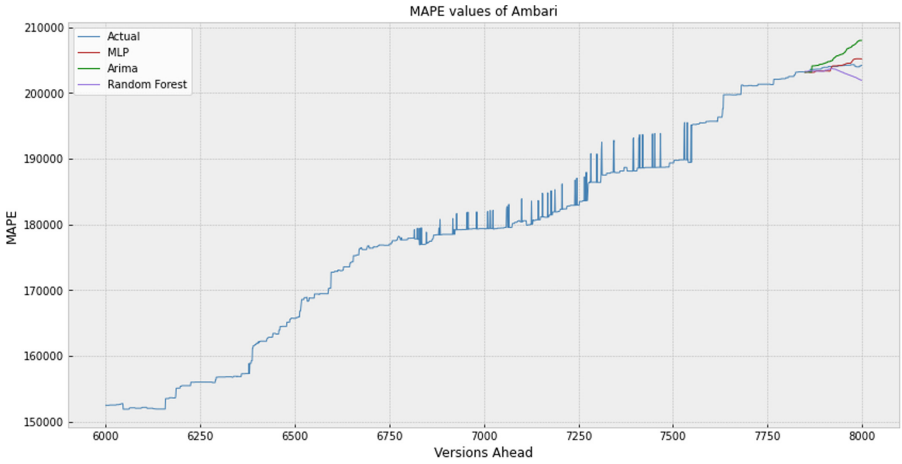

Fig. 3. Ambari TD forecasting for 150 commits ahead using MLP, Random Forest and the ARIMA model.

As can be seen by inspecting Fig. 3, the MLP model outperforms the rest of the models and seems to provide meaningful long-term forecasts for the studied case of Ambari project for up to 150 commits ahead. In fact, the selected model is able to capture the trend of the future TD evolution with a sufficient level of accuracy. For reasons of brevity, illustrations of TD forecasts for the rest of the studied projects are available online[5]. As can be seen, similarly to the case of Ambari, valuable forecasts were also obtained for the rest of the four projects.

To sum up, the novel contribution provided throughout this study is the ability of Deep Learning to accurately predict the future TD evolution of a software project in the long term, an area that was mainly unexplored until now. Furthermore, short and mid-term experiments also demonstrate meaningful

[5] https://drive.google.com/drive/folders/14QP6_gw3GGe6_mHSow9sJrfPKRrgd3q8.

results. In order to be able to value the new contribution of the MLP model, we employed two of the models that we widely used throughout our previous studies and thus, we were able to recognize that Deep Learning Neural Networks comprise a new promising path towards the TD Forecasting concept.

5 Conclusion and Future Work

In previous studies we have employed statistical and ML techniques, able to generate accurate short-and mid-term TD forecasts. However, the fact that TD Management activities usually tend to focus on the long term raised the need for more sophisticated forecasting approaches. The purpose of this work was to investigate whether Deep Learning may lead to more accurate long-term TD predictions. We utilized a dataset comprising five open-source Java applications and employed a MLP model, which was evaluated for various forecasting horizons. Our experiments showed that Deep Learning is indeed a promising approach for TD forecasting. In fact, its adoption resulted in models with sufficient predictive performance for both short-and long-term TD forecasts, demonstrating meaningful results even for 150 steps ahead into the future.

Future work includes the evaluation of the proposed approach on software products that belong to different domains and that are written in different programming languages, in order to verify the generalizability of our results. Based on previous work [17], we also plan to examine the adoption of Deep Learning for cross-project TD forecasting, that is, building a forecasting model based on data retrieved from one project and using it to get reliable forecasts for a new, previously unknown software project. Finally, recent studies have highlighted that TD may be closely related to software security [23]. Hence, another interesting direction would be to examine whether TD forecasting can be used as a means to project the future evolution of important security-related aspects of a software under development, such as the manifestation of vulnerabilities.

Acknowledgements. This work is partially funded by the European Union's Horizon 2020 Research and Innovation Programme through EXA2PRO project under Grant Agreement No. 801015.

References

1. Cunningham, W.: The WyCash portfolio management system. ACM SIGPLAN OOPS Mess. **4**(2), 29–30 (1993)
2. Brown, N., et al.: Managing technical debt in software-reliant systems. In: Proceedings of the Workshop on Future of Software Engineering Research (FSE/SDP), pp. 47–52. ACM (2010)
3. Suryanarayana, G., Samarthyam, G., Sharma, T.: Refactoring for Software Design Smells: Managing Technical Debt. Morgan Kaufmann (2014)
4. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. J. Syst. Softw. **101**, 193–220 (2015)

5. Tsoukalas, D., Siavvas, M., Jankovic, M., Kehagias, D., Chatzigeorgiou, A., Tzovaras, D.: Methods and tools for TD estimation and forecasting: a state-of-the-art Survey. In: International Conference on Intelligent Systems (IS 2018) (2018)
6. Chug, A., Malhotra, R.: Benchmarking framework for maintainability prediction of open source software using object oriented metrics. Int. J. Innovat. Comput. Inf. Control **12**(2), 615–634 (2016)
7. Elish, M.O., Elish, K.O.: Application of TreeNet in predicting object-oriented software maintainability: a comparative study. In: 2009 13th European Conference on Software Maintenance and Reengineering (CSMR), pp. 69–78, March 2009
8. Arisholm, E., Briand, L.C.: Predicting fault-prone components in a java legacy system. In: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 8–17. ACM (2006)
9. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proceedings of the 28th International Conference on Software Engineering (ICSE), pp. 452–461. ACM (2006)
10. Misra, S.: A step by step guide for choosing project topics and writing research papers in ict related disciplines. In: Information and Communication Technology and Applications: Third International Conference (2021)
11. Gondra, I.: Applying machine learning to software fault-proneness prediction. J. Syst. Softw. **81**(2), 186–195 (2008)
12. Pritam, N., et al.: Assessment of code smell for predicting class change proneness using machine learning. IEEE Access **7**, 37414–37425 (2019)
13. Abbas, R., Albalooshi, F.A., Hammad, M.: Software change proneness prediction using machine learning. In: 2020 International Conference on Innovation and Intelligence for Informatics, Computing and Technologies (3ICT), pp. 1–7 (2020)
14. Arcelli Fontana, F., Mäntylä, M.V., Zanoni, M., Marino, A.: Comparing and experimenting machine learning techniques for code smell detection. Empirical Softw. Eng. **21**(3), 1143–1191 (2015). https://doi.org/10.1007/s10664-015-9378-4
15. Tsoukalas, D., Jankovic, M., Siavvas, M., Kehagias, D., Chatzigeorgiou, A., Tzovaras, D.: On the applicability of time series models for technical debt forecasting. In: 15th China-Europe International Symposium on Software Engineering Education (CEISEE 2019). (2019, in press)
16. Tsoukalas, D., Kehagias, D., Siavvas, M., Chatzigeorgiou, A.: Technical debt forecasting: an empirical study on open-source repositories. J. Syst. Softw. **170**, 110777 (2020)
17. Tsoukalas, D., Mathioudaki, M., Siavvas, M., Kehagias, D., Chatzigeorgiou, A.: A clustering approach towards cross-project technical debt forecasting. SN Comput. Sci. **2**(1), 1–30 (2021)
18. Lenarduzzi, V., Saarimäki, N., Taibi, D.: The technical debt dataset. In: Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 2–11 (2019)
19. Lehman, M.M.: Programs, life cycles, and laws of software evolution. Proc. IEEE **68**(9), 1060–1076 (1980)
20. Mens, T.: Introduction and roadmap: history and challenges of software evolution. In: Software Evolution. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-76440-3_1
21. Siavvas, M.G., Chatzidimitriou, K.C., Symeonidis, A.L.: QATCH-an adaptive framework for software product quality assessment. Expert Syst. Appl. **86**, 350–366 (2017)

22. Digkas, G., Lungu, M., Chatzigeorgiou, A., Avgeriou, P.: The evolution of technical debt in the apache ecosystem. In: Lopes, A., de Lemos, R. (eds.) ECSA 2017. LNCS, vol. 10475, pp. 51–66. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65831-5_4
23. Siavvas, M., Tsoukalas, D., Jankovic, M., Kehagias, D., Tzovaras, D.: Technical debt as an indicator of software security risk: a machine learning approach for software development enterprises. Enterp. Inf. Syst., 1–43 (2020)