sciendo INTERNATIONAL JOURNAL ON SMART SENSING AND INTELLIGENT SYSTEMS

Research Article | DOI: 10.2478/ijssis-2025-0012 Issue 1 | Vol. 18 (2025)

Automatic detection of technical debt in large-scale java codebases: a multi-model deep learning methodology for enhanced software quality

Dr. Pooja Bagane[1,*], Chahak Sengar[1], Sumedh Dongre[1], Siddharth Prabhakar[1], Obsa Amenu Jebessa[2]

[1]Department of Computer Science and Engineering, Symbiosis Institute of Technology-Pune Campus, Symbiosis International (Deemed University), Pune, India

[2]Department of Information Technology, Jimma Institute of Technology, Jimma, Oromia, Ethiopia

*E-mail: pooja.bagane@gmail.com

Received for publication January 10, 2025.

Abstract

Management of technical debt (TD) is crucial in long-term software projects for sustaining code quality. We proposed an effective deep learning-based approach to automatically detect and analyze self-admitted TD from large-scale Java codebases. Using a dataset consisting of over 55 million Java source files, we have designed several insightful machine learning models, including random forest, gradient boosting, long short-term memory, and gated recurrent unit, for making predictions about the presence and severity regarding TD. This proposed approach automates the risky component identification; therefore, one can manage TD proactively, thus reducing its costs and augmenting the overall project outcomes. Our results also confirm that these models have much increased detection accuracies of TD, thus giving a lot back to the software engineering domain.

Keywords

technical debt, self-admitted technical debt, machine learning, random forest, gradient boosting, long short-term memory, gated recurrent unit, Java codebases, software engineering, automated detection

I. Introduction

Technical debt (TD) is a common phenomenon within the software engineering domain, leading to shortcuts that meet present needs. Quite often, these are found in solutions that more or less threaten the maintainability and evolvability of software over the long term. In general, TD can be accrued at various activities from design and analysis to implementation and testing throughout the whole software development life cycle [1]. Several factors contribute to the accumulation of TD in legacy code, such as lines of code, fan-in (incoming dependencies), total number of methods and variables, and the frequency of specific occurrences. These factors directly influence the likelihood of TD and are, therefore, critical when assessing a codebase for potential debt. TD comprehension and control are the preconditions to assure that software projects remain maintainable and of high quality during a long period. Unmanaged TD highly increases the costs of maintenance, decreases satisfaction by a customer because of an insufficient final product, blocks innovation, and limits the capability of a company to respond to market opportunities.

In these challenges, researchers and practitioners respond by formulating a variety of strategies. These approaches range from analyzing source code comments to exploring issue tracking systems for effective TD identification. This paper focuses on identifying TD within open-source Java projects using a comprehensive dataset of millions of Java code files. We estimated the risk and likelihood of TD to gain insights

into effective detection strategies [2]. Additionally, we employed several machine learning and deep learning models for TD detection and developed a custom frontend to enable real-time visualization of relevant variables. This architecture forms the core of our research, leading to practical approaches for TD management. We also proposed and discussed several strategies for effective TD handling.

II. Literature Review

In software engineering, TD has emerged as a notion of basic concepts realized in trade-off during software development that put constraints on future maintainability and evolution enforcements. Several Machine Learning (ML) and Deep Learning (DL) techniques have been designed and developed over the last few years to tackle the challenges associated with the identification and management of TD. Results, methodologies, and limitations of important studies in this field are critically reviewed here to ultimately identify the research gaps that are going to be filled by the present study.

Machine learning approach for identification of tech debt, an approach discussed by Tsoukalas [1], in the year 2022, compared random forest (RF) and other machine learning classifiers to find the most accurate model to detect presence of TD. RF outperformed other models and obtained an accuracy of 92% in test data. This approach has shown oversampling bias and is subjected to limited external validity.

Detection and remediation of TD in issue trackers, a study conducted by Li et al. [2], in the year 2020, compared different types of TD detected in issues and source code comments and highlighted the importance of customizing identification approaches based on specific characteristics of each case. The study found on average 71.7% of debt is repaid, and the rest is paid by identifiers and creators (i.e., 47.7% and 44.0%). However, this approach relies on manual identification and categorization of TD in issue trackers, which may introduce subjectivity and potential bias.

Self-admitted technical debt (SATD) detector, a text mining tool was developed by Liu et al. [3], in the year 2018, in order to detect the presence of SATD. The tool's performance was compared using different baseline approaches including Naïve Bayes Model (NBM), Support Vector Machine (SVM), BestSub, and Natural Language Processing (NLP). It outperformed every project in terms of F1 score but lacked the capability for real-time detection.

Lifecycle-based approach identified by Tan et al. [4], in the year 2023, presented the identification of 312 items consisting of TD using manual analysis of issue trackers. A total of 87.4% of items that exhibited active TD were identified and resolved in issue trackers. However, this approach has shown limited scope and interpretation complexity.

Beyond the Code: Mining Self-Admitted Technical Debt in Issue Tracker Systems by Xavier et al. [5] in the year 2020. The study examined SATD in issue-tracking systems by conducting a survey of developers, which provided valuable insights into their practices. However, this approach has demonstrated a restricted ability to apply knowledge to new situations and is susceptible to reliance on specific tools.

Detecting outdated code element references in software by Tan et al. [6] in the year 2022. The study detected outdated citations by employing regular expressions to compare previous and current iterations of the project. The methodology effectively identified outdated code references in widely used GitHub projects, uncovering persistent longevity. However, this approach has a restricted range of documentation and relies on a specific platform.

Automatic identification of SATD from four different sources by Li et al. [7] in the year 2023. The study utilized a Multitask Text Convolutional Neural Network (MT-Text-CNN) to automatically detect SATD. The model attained the maximum F1-score of 0.611. However, this method encountered difficulties, such as an uneven distribution of data and the lack of negative outcomes.

Automatically learning patterns for SATD removal by Zampetti et al. [8] in the year 2020. The research paper discusses SAtd Removal using DEep Learning (SARDELE), a classifier that combines recurrent and convolutional neural networks at multiple levels. The AUC of SARDELE is 0.73, while its average precision and recall are 55% and 57%, respectively. However, SARDELE has limitations due to its lack of transparency, which highlights the need to complement it with additional methods and resources.

a. Gaps in existing research and novelty of the current study

The existing body of research on TD detection demonstrates a significant focus on machine learning and deep learning techniques. However, several gaps persist, particularly concerning the scalability, real-time applicability, and generalization of these

methods across diverse software environments. Recent studies emphasize high accuracy in controlled environments, but their methods often struggle with external validity and real-world application due to oversampling bias and data imbalance [9]. Moreover, tools, such as the SATD detector, face challenges in real-time detection, which is critical for large-scale, dynamic software projects.

Among such research work, the following will try to fill this gap by providing a deep learning-based framework operating on an entire data set of over 55 million Java source files and using several machine learning models, such as RF, gradient boosting (GB), long short-term memory (LSTM), and gated recurrent unit (GRU), in achieving higher detection while ensuring scalability and real-time applicability. Thus, the study would dwell on proactive TD management [10] with a view to reducing maintenance costs for improved project outcomes and making a meaningful contribution to existing methodologies [11].

III. Data and Methodology

This paper aims to estimate the probability of TD in existing codebases using machine and deep learning models. To support a swift and effective environment to execute code analysis, main components, such as RF regressor, GB regressor, GRU, and LSTM networks, will be combined using an integrated workflow that can process, train, and validate a large dataset comprised of >55 million Java code files [12, 13]. The detailed TD prediction workflow is described in below steps:

a. Data collection and preprocessing

It involves collecting a large number of Java codebases from various projects in software development. The collected dataset needs to be preprocessed to be consistent and of high quality, which it does by the following:

a.i Data cleaning

This is done to ensure that the null values in the dataset do not contribute toward maligning the data.

a.ii Outlier handling

In most cases of regression and classification, finding outliers and handling them is very crucial because these cause skewed performances in the models.

a.iii Normalization

Min-Max scaling is applied for the normalization of data, scaling the features to a comparable range.

The aforementioned steps help in pre-processing the data to prepare for any machine learning or deep learning tasks that need to be performed, since it helps the model learn from the cleaned and standardized data effectively.

b. Data splitting and model training

Then, the preprocessed data are split into training and testing sets, with 80% of the data for training and 20% for testing. To capture various facets of TD prediction, the following models are trained in parallel:

b.i RF regressor

This model especially performs well in those datasets that involve huge numbers of variables. It creates multiple decision trees on random subsets of the data, averaging the predictions to reduce overfitting and improve generalization. Since software development projects have so many nuances and complexities, the RF model captures them best and delivers robust predictions.

b.ii GB regressor

Unlike in RF, the trees are grown sequentially in GB. This means each tree tries to solve the errors of its forerunners. This is an iterative method of improving the accuracy of the model to deal with the eventual complex interaction between codebases [14].

b.iii GRU

GRUs process sequential data, and they are mostly applied in analysis where long-term dependencies are crucial. The GRU model takes as input the sequences of code changes and embeds contextual information that is indicative of TD. Due to its gating, relevant past information is kept, while irrelevant data get filtered out.

b.iv LSTM

LSTMs will be used to provide insight into predicting the probability of TD in Java projects by investigating the temporal sequences of the data. This model is set

up to include two LSTM layers that have 64 and 128 units, respectively, with an increase in feature extraction and performance of the prediction [15]. Dropout layers are implemented to avoid overfitting, while this model is optimized using the Adam optimizer.

c. Model validation and evaluation

The performance of the models is validated by the following sets of metrics:

c.i Mean squared error

It is one of the measures that tell about the average squared difference between forecasted and actual values. It gives a hint about the accuracy of the model.

$$\text{MSE} = (1/n)\,\Sigma\left(y_j - \hat{y}_j\right)^2 \quad (1)$$

In Eq. (1) n is the number of samples, y_j is the true value, and y_j is the predicted value.

c.ii Root mean squared error (RMSE)

This is the measurable size of the difference between predictions versus observed values, which gives insight into the performance of a model in real-world applications.

$$\text{RMSE} = \sqrt{1/n\ \Sigma\left(y_j - \hat{y}_j\right)^2} \quad (2)$$

In Eq. (2), n represents the total number of data points or observations in the dataset, y_i represents the actual or observed value for the *ith* data point, and *ith* represents the predicted value for the *i*th data point.

c.iii R^2 score

It gives the percentage of variance in the dependent variable explained by the independent variables. The larger the R-squared score, the stronger the model fit.

$$R^2 = 1 - \frac{\text{SSres}}{\text{SStotal}} \quad (3)$$

In Eq. (3), SSres is the sum of squares of residuals (or errors) from the regression line. SStotal is the total sum of squares, which represents the total variation in the dependent variable γ around its mean.

All the above-listed models will be compared in the validation process, the result of which will determine its actual deployment. Among others, metrics, such as Mean Squared Error (MSE), RMSE, and R^2, are primary representatives of the predictive capabilities for these models.

d. Deployment and integration

The final step is putting these top-performing models into a real-world application environment. A frontend, created using Flask, was provided for a TD calculator where the user could provide the codebases with the intent of getting insights on potential TD. This system will enable continuous integration to keep updating the system and building active feedback loops. This will further improve the predictions and user experience.

Figure 1 shows the architectural diagram for illustrating the comprehensive workflow, starting from

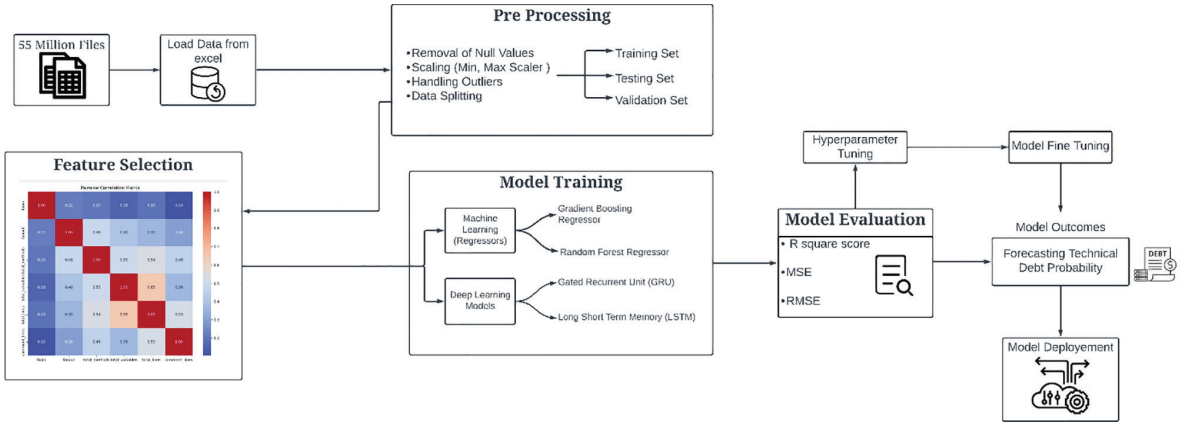

Figure 1: TD prediction workflow. TD, technical debt.

data loading to model deployment. The workflow shows the data collection, preprocessing, feature engineering, model training, model evaluation, fine tuning, and then prediction followed by deployment. The stages are so interconnected that the transition to a successive phase from any previous one is essentially smooth. It is such a structured methodology that calls for best practices, facilitating effective data-driven decision making and amplifying predictive analytics to enhance TD management.

IV. Results and Discussion

During the course of our extensive investigation, we focused on TD management. Our goal was to determine the most effective model or hybrid method to reach the maximum level of efficiency achievable. To achieve this, a variety of machine learning and deep learning models, including GB regressor, RF regressor, GRU, and LSTM, were put through rigorous training and comparative analysis.

Table 1 indicates that LSTM model outperformed other models in terms of performance and obtained a fairly low MSE of 0.003357 and a high R^2 score of 0.93. This suggests that the LSTM model produced predictions that were more accurate by best capturing the underlying patterns and relationships in the dataset. The outcomes highlight how well LSTM handles sequential data and how well it can perform in regression tasks.

After analyzing the data, we were able to build our frontend using Flask, as illustrated in Figure 2 and integrate the best model. This was completed without any issues. This integration allows us to accurately assess the risk of discovering self-reported TD in software projects.

Table 1: Comparison of validation metrics

Model	MSE	RMSE
LSTM	0.003357	0.057947
GRU	0.005403	0.073505
RF regressor	0.006093	0.078061
GB regressor	0.007900	0.08888

GB, gradient boosting; GRU, gated recurrent unit; LSTM, long short-term memory; RF, random forest; RMSE, root mean squared error.

V. Conclusion

Our research work is an important contribution to the continuous efforts in software development industry to improve software quality, maintenance, and decision making. We have expanded our knowledge of TD management and its implications for Java codebases by incorporating cutting-edge technology and approaches. By this research, we have prepared the way for additional successful approaches to reduce TD and improve software development procedures, which will promote advancement and innovation in the sector.

Our goal for the future is to create an open-source platform that incorporates our verified models in

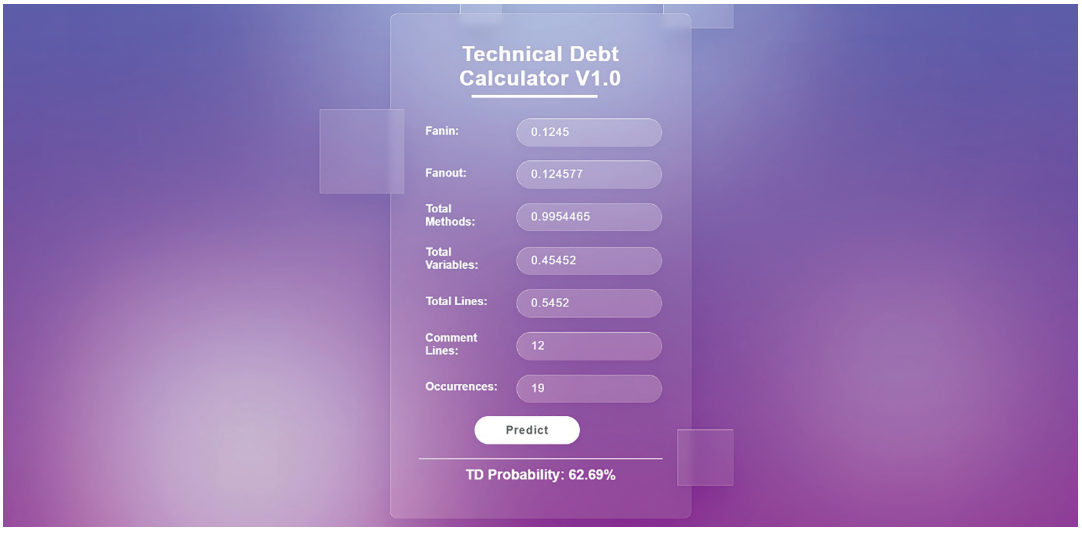

Figure 2: Frontend interface of the project. TD, technical debt.

order to further expand the impact of our study. By giving developers a convenient and effective way to recognize, rank, and resolve TD concerns in Java codebases, this tool will give workable ways for dealing with TD. By providing such a tool to software development teams, we hope to ease the development process, lessen the load on maintenance staff, and eventually improve the general caliber and long-term viability of software products.

References

[1] D. Tsoukalas, "Machine learning for technical debt identification," IEEE Transactions on Software Engineering, p. 1, Jan. 2021, doi: 10.1109/tse.2021.3129355.

[2] Y. Li, M. Soliman, and P. Avgeriou, "Identification and Remediation of Self-Admitted Technical Debt in Issue Trackers," 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 495–503, Aug. 2020, doi: 10.1109/seaa51224.2020.00083.

[3] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "SATD detector," 2018 IEEE/ACM 40th International Conference on Software Engineering, May 2018, doi: 10.1145/3183440.3183478.

[4] J. Tan, D. Feitosa and P. Avgeriou, "The life-cycle of Technical Debt that manifests in both source code and issue trackers", Information and Software Technology, Volume 159, 2023, 107216, ISSN 0950-5849, doi:10.1016/j.infsof.2023.107216.

[5] L. Xavier, F. Ferreira, R. Brito and M. Valente, "Beyond the Code: Mining Self-Admitted Technical Debt in Issue Tracker Systems," in 2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR), Seoul, Korea, Republic of, 2020 pp. 137-146. doi: 10.1145/3379597.3387459

[6] W. S. Tan, M. Wagner, and C. Treude, "Detecting outdated code element references in software repository documentation," arXiv (Cornell University), Jan. 2022, doi: 10.48550/arxiv.2212.01479.

[7] Y. Li, M. Soliman, and P. Avgeriou, "Automatic identification of self-admitted technical debt from four different sources," Empirical Software Engineering, vol. 28, no. 3, Apr. 2023, doi: 10.1007/s10664-023-10297-9.

[8] F. Zampetti, A. Serebrenik and M. Di Penta, "Automatically Learning Patterns for Self-Admitted Technical Debt Removal," in 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 2020 pp. 355-366. doi: 10.1109/SANER48275.2020.9054868

[9] P. Bagane, C. Sengar, S. Dongre, S. Prabhakar, S. Baldua, and S. Gurav, 'Total Electron Content Forecasting in Low Latitude Regions of India: Machine and Deep Learning Synergy', Communications in Computer and Information Science, vol. 2054 CCIS, pp.104–119, 2024.doi: 10.1007/978-3-031-56703-2_9

[10] P. Bagane, M. Thawani, P. Singh, R. Ahmad, R. Mital, and O. A. Jebessa, 'Breaking the Silence: An innovative ASL to Text Conversion System Leveraging Computer Vision & Machine Learning for Enhanced Communication', International Journal of Intelligent Systems and Applications in Engineering, vol. 12, no. 14s, pp. 246–255, 2024.

[11] E. Gama, S. Freire, M. Mendonça, R. O. Spínola, M. Paixao, and M. I. Cortés, 'Using Stack Overflow to Assess Technical Debt Identification on Software Projects'. In Proceedings of the XXXIV Brazilian Symposium on Software Engineering (SBES '20). Association for Computing Machinery, New York, NY, USA, 2020, pp. 730–739. doi:10.1145/3422392.3422429

[12] F. Bi, B. Vogel-Heuser, Z. Huang, F. Ocker 'Characteristics, causes, and consequences of technical debt in the automation domain', Journal of Systems and Software, vol.204, 2023.doi: 10.1016/j.jss.2023.111725

[13] C. Jaspan and C. Green, "Defining, Measuring, and Managing Technical Debt," in IEEE Software, vol. 40, no. 3, pp. 15-19, May-June 2023, doi: 10.1109/MS.2023.3242137.

[14] D. Pina, A. Goldman and G. Tonin, "Technical Debt Prioritization: Taxonomy, Methods Results, and Practical Characteristics," 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Palermo, Italy, 2021, pp. 206-213, doi: 10.1109/SEAA53835.2021.00034.

[15] J. S. De Jesus and A. C. V. De Melo, "Technical Debt and the Software Project Characteristics. A Repository-Based Exploratory Analysis," 2017 IEEE 19th Conference on Business Informatics (CBI), Thessaloniki, Greece, pp. 444–453, Jul. 2017, doi: 10.1109/cbi.2017.62.