eISSN 2783-6851

VILNIUS TECH
Vilnius Gediminas Technical University

NEW TRENDS in COMPUTER SCIENCES

2024
Volume 2
Issue 2
Pages 91–100
https://doi.org/10.3846/ntcs.2024.22274

JUST-IN-TIME SOFTWARE DEFECT PREDICTION USING A DEEP LEARNING-BASED MODEL

Rodrigo Alexandre DOS SANTOS

Department of Software Development, CPQD Foundation, Campinas, SP, Brazil

Article History:
- received 22 September 2024
- accepted 6 December 2024

Abstract. The increase in software complexity, driven by technological developments and user demands, has created major challenges for companies in Software Quality Assurance. Companies seek efficient ways to identify and mitigate defects, recognizing that they cause high financial costs and other problems with negative impacts on business. Among defect prediction approaches, Just-In-Time Software Defect Prediction has received increased attention from software industry professionals in recent years. This technique aims to identify and treat defects early, to improve the quality of the software development cycle. This study proposes a Deep Learning-based approach for Just-In-Time Software Defect Prediction using a large dataset of historical data from several popular software projects. The Deep Learning model was trained to identify defects by analyzing the software metrics provided by the dataset. The model achieved an accuracy of 82.08% in its predictions, and it was possible to determine the most relevant metrics for its conclusions through interpretability techniques. The results obtained demonstrate the potential of Just-In-Time Software Defect Prediction as a tool for improving software quality and encouraging the development of new studies and improvements in this area of research.

Keywords: software defect prediction, machine learning, deep learning.

Corresponding author. E-mail: *rodrigoasantos1981@gmail.com*

1. Introduction

Software Quality Assurance (SQA) is a topic that has received increasing attention from companies in recent years. Software has become increasingly complex to keep up with technological developments and meet user demands, and this increase in complexity has created several challenges for companies to ensure that their products reach high levels of quality. Although there have been many advances in tools, methodologies, and processes for monitoring and managing quality during the software development lifecycle, companies still face a high occurrence of defects in their systems. The existence of many defects in software can generate user dissatisfaction, increased costs related to corrective maintenance, commitment of resources that could be directed to improvements and developments in the product, and a negative impact on the company's reputation.

According to Krasner (2022), concern about the high financial costs generated by software quality problems has increased in recent years and this fact highlights the need for companies to increasingly invest in standards, practices, and tools aimed at mitigating the factors generators of this problem. One of the recommendations highlighted by Krasner (2022) for improving software quality is the investment in tools based on Artificial Intelligence (AI) for automatic detection of defects to improve the productivity of software engineers and mitigate the occurrence of defects. The author highlights the concern with the productivity of software engineers because his research has shown that a significant percentage of the current costs of software projects are spent on corrective activities rather than creation activities, and it is estimated that software developers have on average 30% of their working time committed to software quality problems. AI-based analysis and monitoring tools have the potential to assist in the rapid detection and diagnosis of defects, contributing to the reduction of efforts and costs spent on SQA activities.

This study proposes an approach based on Deep Learning for predicting defects in software. The objective is to evaluate the potential of using Deep Learning models as tools to help improve software quality. The study is focused on product metrics, that is, the metrics associated with the software artifacts produced. The scope of the study covers advanced defect prediction techniques and large software projects that are widely popular in the market. The use of popular projects aims to reproduce realistic scenarios and contribute to the dissemination of the studied practices in the software industry.

2. Literature review

SQA is the set of practices and processes carried out to ensure that the software meets established quality standards. According to Rathore and Kumar (2019), SQA activities are performed during all phases of the software development process and aim to monitor and control resources, costs, and deadlines related to the process. The activities commonly carried out in SQA are quality audits, requirements management, formal technical reviews, code inspections, configuration management, performance and usability assessments, software testing, and software defect prediction.

Software Defect Prediction (SDP) has been one of the most covered research topics in Software Engineering in the last two decades and, according to Mahbub et al. (2023), one of the reasons for this is the fact that companies recognize the importance of detecting and correcting software defects before releasing them to end users. According to Li et al. (2018), the increase in attention focused on the topic of SDP in recent years can be seen in the number of research articles published. SDP is used during the initial phases of the software development process to identify the software artifacts most likely to present defects and to enable the prioritization of tests and validations of these artifacts and the correction of possible defects as soon as possible.

SDP is used as a tool to help improve software quality. According to Sirshar et al. (2019), one of the expected benefits of using SDP is that, by identifying which software artifacts are most likely to present defects, the development team can focus its efforts on testing and validating these artifacts to direct resources available for validation of software modules considered most critical. It is also expected that the prediction and fixing of defects in the early phases of the software development process will provide cost reduction as practical experience shows that these costs become greater when defects are identified in later phases of the process or when the software is already in production.

Software quality is normally assessed through metrics that represent aspects of its intrinsic complexity and the complexity of its development process. SDP is carried out through the analysis of metrics generated from historical data on defects. According to Thota et al. (2020), the data for generating metrics is obtained from software versioning control systems and issue-tracking systems. According to Rathore and Kumar (2019), several works have explored the use of different types of metrics to identify which type provides better performance in predicting defects, but the results have varied between the different works due to the differences between the different contexts in which the studies were carried out.

Several studies have proposed the use of Machine Learning (ML) techniques for SDP in recent years. These works used ML models to predict defects through learning patterns in datasets composed of metrics collected from historical data from software projects. According to Rathore and Kumar (2019), these studies have explored and evaluated several aspects that can influence the performance of defect prediction and the results observed allow us to conclude that there are still several opportunities for improvements and challenges to be overcome in this area of research.

Table 1 presents some studies carried out in recent years. The authors evaluated several ML models for SDP on several open datasets. The results obtained show that there is no single model with the best performance in all scenarios and that the results may vary depending on the characteristics of each dataset.

Table 1. Studies about software defect prediction

Authors	Trained models	Datasets	Best model
Shah e Pujara (2020)	Logistic Regression, Random Forest, Naïve Bayes, Gradient Boosting, Support Vector Machine, and Artificial Neural Network	7 datasets from the NASA Promise project (binary classification)	Artificial Neural Network (93% accuracy)
Goyal and Sinha (2023)	Logistic Regression, Random Forest, Support Vector Machine, Naive Bayes, and Artificial Neural Network	5 datasets from the NASA Promise project (binary classification)	Artificial Neural Network (93.8% accuracy)
Kaur et al. (2023)	Decision Tree Regression and K-nearest Neighbor	A dataset from the Promise Software Engineering project (binary classification)	Decision Tree Regression (99.37% accuracy)
Santos et al. (2020)	Logistic Regression, Naive Bayes, K-Nearest Neighbor, Artificial Neural Network, Decision Tree, Support Vector Machine, Random Forest, and XGBoost	9 datasets from the NASA Promise project (binary classification)	XGBoost (91.7% Area Under the Curve)
Khalid et al. (2023)	K-means, Support Vector Machine, Naive Bayes, and Random Forest	A dataset from the NASA Promise project (binary classification)	Support Vector Machine (99.8% accuracy)
Nevendra and Singh (2021)	Support Vector Machine, K-Nearest Neighbor, AdaBost, and Convolutional Neural Network	19 datasets from the tera-Promise project (binary classification)	Convolutional Neural Network (79% accuracy)
Lamba et al. (2019)	Linear Regression, Random Forest, Artificial Neural Network, Support Vector Machine, Decision Tree, and Decision Stump	9 datasets from Apache Software Foundation projects (clustering)	Support Vector Machine (88% accuracy)

Cross-Project Defect Prediction (CPDP) is an issue that has sparked the interest of software industry professionals. According to Thota et al. (2020), CPDP consists of using metrics from different software projects to build datasets and prediction models capable of generalizing pattern learning so that it is possible to predict defects in new projects distinct from those used in model training. An advantage of using CPDP cited by the author is that companies with insufficient historical project data to create prediction models can benefit from pre-trained models on their previous projects or projects from other companies.

Among the various approaches already proposed for SDP, Just-In-Time Software Defect Prediction (JIT-SDP) has received a lot of attention from researchers in recent years. According to Zhao et al. (2023), JIT-SDP is a technique to determine whether a change is potentially defect-inducing when the change is registered in versioning control systems. According to Cabral et al. (2023), what differs JIT-SDP from other traditional SDP approaches is the level of granularity of the artifacts considered in the predictions, since while traditional SDP approaches consider artifacts at the file or module level, JIT-SDP considers artifacts at the source code level (fine granularity). According to Mahbub et al. (2023), JIT-SDP has offered the most accurate predictions among the various approaches to SDP proposed in recent years.

According to Zhao et al. (2023), the JIT-SDP approach has the following advantages over traditional SDP approaches: reducing efforts for code review and correction activities because it always considers smaller artifacts (fine granularity); the possibility for predictions to be made incrementally and defects to be identified earlier, which is more in line with continuous deployment practices used in modern software development; facilitate the attribution of Quality Assurance activities (corrections and tests) to the responsible specialists since fine granularity changes are more specific and punctual.

3. Methods

3.1. Dataset description

The dataset used in this study is called ApacheJIT. This dataset was derived from the work developed by Keshavarz and Nagappan (2022) and the main motivation for its creation is the scarcity of large public datasets focused on CPDP. The authors made the dataset publicly available for use in research related to CPDP that requires large amounts of data for training Machine Learning and Deep Learning models.

ApacheJIT was built from information obtained from 14 Java projects maintained by the Apache Foundation. The projects used were ActiveMQ, Camel, Cassandra, Flink, Groovy, HDFS, HBase, Hive, Ignite, MapReduce, Kafka, Spark, Zeppelin, and Zookeeper. Initially, the authors identified issues that were reported as bugs in the issue-tracking systems of these projects. Using techniques recommended in the literature, the commits associated with these issues were retrieved from the project repositories and classified into defective commits or defect-free commits. Additionally, change metrics associated with each commit were collected and included in the dataset. The dataset has 106,674 samples that represent the verified commits, of which 28,239 are defective commits and 78,435 are defect-free commits. Table 2 presents the description of the features included in the dataset.

Table 2. ApacheJIT features

Feature	Description	Type
COMMIT_ID	Commit hash	String
PROJECT	Project name	String
BUGGY	Commit classification (True = defective, False = defect-free)	String
FIX	Is fixing any previous bug? (True = yes, False = no)	String
YEAR	Year of registration	String
AUTHOR_DATE	Change date	String
LA	Number of lines added	int
LD	Number of deleted lines	int
NF	Number of files changed	int
ND	Number of directories changed	int
NS	Number of modules changed	int
ENT	Distribution of changed code across each file (Entropy)	float
NDEV	Number of developers who participated in the change	float
AGE	Average time interval between last commit and current commit on affected files	float
NUC	Number of unique commits to modify each file	float
AEXP	Developer experience level measured by the number of changes the developer made previously	int
AREXP	Developer experience level measured by the number of recent changes the developer made	float
ASEXP	Developer experience level is measured by the number of changes the developer made in the module affected by the commit	float

3.2. Preprocessing

The features LA, LD, NF, ND, NS, ENT, NDEV, AGE, NUC, AEXP, AREXP, and ASEXP were selected for use in this study because they represent metrics about software changes. The COMMIT_ID, PROJECT, FIX, YEAR, and AUTHOR_DATE features were disregarded because they do not represent metrics.

Normalization techniques were used on the features because their values were on different scales and their distribution was asymmetric. Another reason for using normalization techniques was to minimize negative impacts on the model's performance that could be caused by the presence of outliers that were verified in all features.

The dataset samples are divided into 2 classes: defect-free (class 0) and defective (class 1). The dataset was a little unbalanced as 74% of the samples belonged to class 0 and 23% of the samples belonged to class 1. The Synthetic Minority Over-sampling (SMOTE) technique was used to balance the dataset and the total number of samples became 156,870 with 50% of the samples belonging to each class.

3.3. Model training

A Multilayer Perceptron model was developed to predict software defects using ApacheJIT. Figure 1 shows the architecture of the proposed model. This architecture was the one that

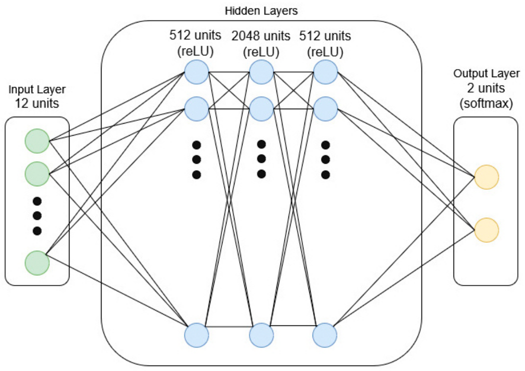

Figure 1. Multilayer perceptron architecture

achieved the best accuracy among several tested configurations. The architecture is composed of 3 hidden layers with 512, 2048, and 512 neurons. The model was implemented with the TensorFlow framework and the Keras library. The dataset was stratified into 80% of training samples and 20% of testing samples. The training was carried out in 500 epochs with a learning rate of 0.001 (default value in Keras library), batch size of 32 (default value in Keras library), and "he_normal" kernel initializer. The "he_normal" kernel initializer was chosen because it provides better results in layers that use the Rectified Linear Activation Unit (ReLU) activation function.

The model has an input layer with 12 neurons that receive the values of the 12 features used. The hidden layers (with 512, 2048, and 512 neurons, respectively) use the ReLU activation function. ReLU was chosen because it is a function that presents simplicity of execution and computational efficiency when compared to other popular functions. Furthermore, the use of ReLU aims to reduce the possibility of the vanishing gradient problem, which can cause problems of slowness or stagnation in model training.

4. Results and discussion

The Multilayer Perceptron Model achieved an accuracy of 82.08% on the test set. The accuracy achieved was considered satisfactory due to the complexity of dealing with a large dataset that has 156,870 samples and encompasses a variety of defects from different projects. Table 3 and Figure 2 show the detailed metrics and the training confusion matrix, where it can be seen that the model achieved similar performance in identifying the 2 classes.

Table 3. Detailed training metrics

Class	Precision	Recall	F1-Score
0	0.84	0.79	0.82
1	0.80	0.85	0.83

The SHapley Additive exPlanation (SHAP) technique was applied to the model to determine the contribution of each feature to the predictions. SHAP is a technique proposed by Lundberg and Lee (2017) for the interpretability of Machine Learning model predictions. This technique determines the contribution of each feature to the model prediction based on game theory concepts. The values calculated by SHAP determine the contribution of each feature to an individual prediction (local interpretation) and the average of the values makes it possible to determine the feature's contribution to the entire data set (global interpretation). In this study, the SHAP technique was applied to the test set using all 31,374 samples. Figure 3 shows the importance of each feature according to the values calculated by SHAP. The features that contributed most to the model predictions were LA, AEXP, and NUC.

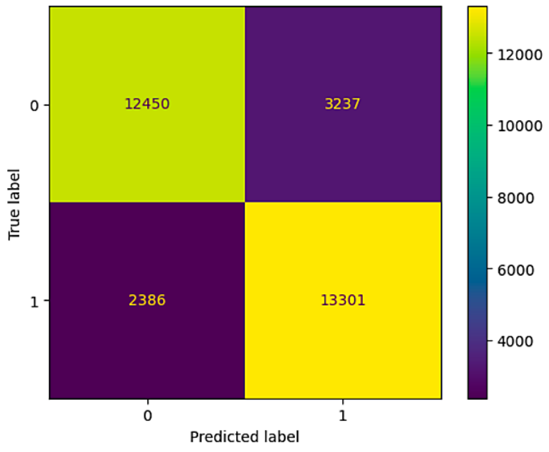

Figure 2. Confusion matrix of training

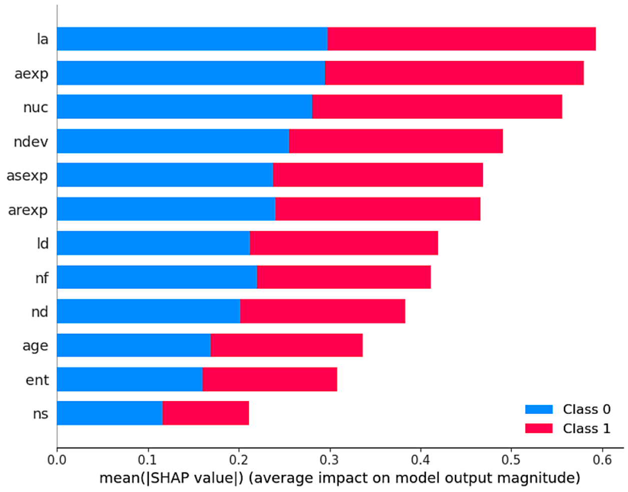

Figure 3. SHAP explanation for multilayer perceptron predictions

The LA, AEXP, and NUC metrics are widely covered in Software Engineering and, although it cannot be said that these are the most relevant metrics in all contexts, the model's conclusions can be considered coherent with what is observed in many real cases. The LA metric represents the number of lines added in a change and, in certain contexts, adding many lines can increase code complexity and introduce new defects. The AEXP metric represents the experience level of the software engineer, and although experience is not a guarantee of the absence of defects, more experienced engineers are typically expected to introduce fewer defects. The NUC metric represents the number of different areas of the software that are being changed, and a high number for this metric may be an indication that the changes are dispersed throughout the code and there is a greater risk of introducing defects. Therefore, these 3 metrics have recognized importance in real scenarios.

The use of interpretability techniques such as SHAP adds the characteristic of transparency to the proposed model, as it allows the understanding of how the model makes its predictions and this increases software engineers' confidence in its results. An explanation of the contribution level of each feature makes it possible to identify metrics that contribute little or are irrelevant to the predictions, and these metrics could be discarded to simplify the model and improve its performance. SHAP results can also be useful for the continuous improvement of the software development process because software engineers can focus their efforts on improving the metrics identified by SHAP as most relevant.

Identifying which metrics contribute most to discovering which factors most influence the introduction of defects in software allows software engineers to focus on critical issues and improve their development techniques, and development teams can adjust their processes to mitigate these factors in future projects. On the other hand, identifying that a certain metric does little to discover these factors allows development teams to stop collecting or monitoring it and save efforts and resources. Identifying relevant and irrelevant metrics helps refine the defect prediction process, making it more accurate and aligned with the factors that impact software quality.

5. Limitations and future direction

The model uses 2,107,394 trainable parameters (weights and biases of artificial neural network) and can be considered a model of moderate complexity. This characteristic can be considered a trade-off of the proposed solution because it demands greater computational cost and more time for training. A suggestion for future work is the evaluation of other types of artificial neural networks for training with ApacheJIT to find a model that achieves a better balance between performance and computational costs.

Another approach for executing JIT-SDP is to consider changes in the source code itself to determine how syntactic and semantic aspects of the source code can influence the occurrence of defects. ApacheJIT does not provide features containing the source code of the commits made, but there are other public datasets with this feature. A suggestion for future work is to train a Deep Learning model with a dataset containing source code to evaluate whether this approach allows for better performance than the metrics-based approach used in this study.

6. Conclusions

The use of defect prediction techniques is a very desirable alternative for improving software quality. Techniques based on Machine Learning and Deep Learning are considered promising in this scenario due to the possibility of automating analyses and reducing efforts and costs spent in the process. Reusing historical data from other software projects through CPDP is an approach that can mitigate difficulties that could arise and prevent a certain company from adopting defect prediction in its projects. The early discovery and treatment of defects generate the opportunity to improve the execution of the phases of the software development cycle, as it allows for greater predictability and control over the allocation of resources involved in the process.

The use of interpretability techniques to generate transparency and a better understanding of how Machine Learning models reach their conclusions should become an increasing trend given the need to increase users' confidence in model decisions. This characteristic is considered a requirement mainly in domains in which decisions made are considered critical due to the consequences they can cause or in domains in which decisions must comply with strict regulations. The interpretability of models can also be useful for discovering insights into the data or for mitigating possible biases present in the data.

The task of predicting defects in modern software represents a great challenge for the following reasons: there are different types of defects such as functional bugs, performance problems, and security flaws, which generates a greater variety of situations to be predicted; software has millions of lines of code spread across different modules and integrations, which increases the difficulty of predicting the location of defects; there are several technologies and tools used together to develop software, which increases the difficulty of detecting patterns in the characteristics of artifacts. The use of advanced pattern prediction techniques such as Deep Learning and the assertive choice of software metrics are essential to achieve good performance and overcome these challenges.

References

Cabral G. G., Minku, L. L., Oliveira, A. L. I., Pessoa, D. A., & Tabassum, S. (2023). An investigation of online and offline learning models for online just-in-time software defect prediction. *Empirical Software Engineering*, *28*(1), Article 121. https://doi.org/10.1007/s10664-023-10335-6

Goyal, J., & Sinha, R. R. (2023). Machine learning-based defect prediction for software efficiency. *International Journal of Intelligent Systems and Applications in Engineering*, *11*(6s), 257–266.

Kaur, G., Pruthi, J., & Gandhi, P. (2023). Machine learning based software fault prediction models. *Karbala International Journal of Modern Science*, *9*(2), Article 9. https://doi.org/10.33640/2405-609X.3297

Keshavarz, H., & Nagappan, M. (2022). ApacheJIT: A large dataset for just-in-time defect prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories* (pp. 191–195). Association for Computing Machinery. https://doi.org/10.1145/3524842.3527996

Khalid, A., Badshah, G., Ayub, N., Shiraz, M., & Ghouse, M. (2023). Software defect prediction analysis using machine learning techniques. *Sustainability*, *15*(6), Article 5517. https://doi.org/10.3390/su15065517

Krasner, H. (2022). *The cost of poor software quality in the US: A 2022 report*. CISQ. https://www.it-cisq.org

Lamba, T., Kavita, & Mishra, A. K. (2019). Optimal machine learning model for software defect prediction. *International Journal of Intelligent Systems and Applications*, *11*(2), 36–48. https://doi.org/10.5815/ijisa.2019.02.05

Li, Z., Jing, X.-Y., & Zhu, X. (2018). Progress on approaches to software defect prediction. *IET Software*, *12*(3), 161–175. https://doi.org/10.1049/iet-sen.2017.0148

Lundberg S. M., & Lee S. (2017). *A unified approach to interpreting model predictions*. arXiv. https://doi.org/10.48550/arXiv.1705.07874

Mahbub, P., Shuvo, O., & Masudur Rahman, M. (2023). Defectors: A large, diverse Python dataset for defect prediction. In 2023 *IEEE/ACM 20th International Conference on Mining Software Repositories* (pp. 393–397). IEEE. https://doi.org/10.1109/MSR59073.2023.00085

Nevendra, M., & Singh, P. (2021). Software defect prediction using deep learning. *Journal of Applied Sciences*, *18*(10), 173–189. https://doi.org/10.12700/APH.18.10.2021.10.9

Rathore, S. S., & Kumar, S. (2019). A study on software fault prediction techniques. *Artificial Intelligence Review*, *51*(1), 255–327. https://doi.org/10.1007/s10462-017-9563-5

Santos, G., Figueiredo, E., Veloso, A., Viggiato, M., & Ziviani, N. (2020). Predicting software defects with explainable machine learning. In 19th *Brazilian Symposium on Software Quality* (Article 18). Association for Computing Machinery. https://doi.org/10.1145/3439961.3439979

Shah, M., & Pujara, N. (2020). *Software defects prediction using machine learning*. arXiv. https://doi.org/10.48550/arXiv.2011.00998

Sirshar, M., Mir, H., Amir, K., & Zainab, L. (2019). *Comparative analysis of software defect prediction techniques*. Preprints. https://www.preprints.org/manuscript/201912.0075/v1

Thota, M. K., Shajin, F. H., & Rajesh, P. (2020). Survey on software defect prediction techniques. *International Journal of Applied Science and Engineering*, *17*(4), 331–344. https://doi.org/10.6703/IJASE.202012_17(4).331

Zhao, Y., Damevski, K., & Chen, H. (2023). A Systematic survey of just-in-time software defect prediction. *ACM Computing Surveys*, *55*(10), Article 201. https://doi.org/10.1145/3567550