

Forecasting Optimal Timing for Software Refactorings based on Technical Debt Issues

Vasilica Moldovan
vasilica.moldovan@ubbcluj.ro
Babes-Bolyai University
Cluj-Napoca, Romania

Abstract

As the domain of software engineering grows, refactoring plays a key role in improving code quality. However, deciding when and how to refactor remains difficult and often relies on developer judgment. Existing tools struggle with adaptability and data relevance. This study applies machine learning to predict refactoring needs based on technical debt data from static analysis. Two data formats were tested: aggregated and detailed instances. Models evaluated include RandomForest, ExtraTrees, and RNNs. Using data from three Java projects and 28 output classes, the best model reached 0.96 accuracy, highlighting the potential of classical classifiers in refactoring prediction.

CCS Concepts

• **Software and its engineering** → **Maintaining software.**

Keywords

Software Refactoring, Machine Learning, Artificial Intelligence, Recurrent Neural Networks

ACM Reference Format:

Vasilica Moldovan. 2025. Forecasting Optimal Timing for Software Refactorings based on Technical Debt Issues. In . ACM, New York, NY, USA, 2 pages.

1 Introduction

The relationship between refactoring and software maintainability has been widely studied, with findings showing that refactorings are often applied to complex, low-quality code to improve sustainability. While many approaches rely on static metrics and machine learning, most focus on general code properties rather than technical debt. This highlights the potential for more targeted refactoring predictions using technical debt data [2].

Given the importance of timing in refactoring decisions and the limited adoption of existing tools—often due to poor adaptability and outdated datasets—this work explores a data-driven approach using modern classification techniques.

This study investigates several machine learning-based methods for predicting the type of software refactoring needed for individual

components, based on technical debt data collected via static analysis. The model classifies components into 28 classes: 27 refactoring types as defined by Fowler [1], and one class for cases requiring no refactoring. The dataset was built by combining outputs from static analysis and actual refactorings identified across software versions.

Two modeling strategies were considered: one compressing all technical debt information into a single instance per component (for use with LazyPredict classifiers), and another preserving full detail for each issue (for RNNs). Algorithms such as RandomForest, ExtraTrees, and custom RNNs were evaluated.

2 Investigated Approach

The goal of this project is to identify software refactoring opportunities by analyzing technical debt issues, with a high accuracy.

Two main approaches were explored. The first compressed all technical debt issues into a single entry per software component using aggregation methods like mean, sum, and median. Various classifiers from the lazypredict.Supervised.LazyClassifier library—such as ExtraTrees, DecisionTrees, RandomForest, SVC, and LGBM—were tested with default parameters.

The second approach used a custom recurrent neural network (RNN) on uncompressed data, where each issue remained a separate entry. The model included an Embedding layer, two LSTM layers each followed by Dropout layers, a Dense layer, and a final Softmax-activated Dense layer for classification.

The prediction was done based on the following features, for each detected technical debt issue: the severity, the debt value and the issue type. Model performance was evaluated using metrics such as accuracy, recall, precision, and F1 score—the latter being especially important due to class imbalance.

Three open-source Java projects were used: jEdit, FreeMind, and TuxGuitar, with technical debt data sourced from SonarQube static analysis. The data for the static analysis part was the dataset presented in [3]. A total of 54,617 issues were found and grouped into code smells (45,453), bugs (7,421), and vulnerabilities (1,743), while severity levels ranged from Info to Blocker.

Two preprocessing strategies were applied. One compressed issues per component and normalized the features, while the other retained multiple entries and excluded those with missing debt values. The first strategy supported traditional machine learning models, and the second supported sequence-based models like RNNs.

To label the dataset, the RefactoringMiner tool [4] was used to detect actual refactorings between project versions. Each component was linked to the most frequent refactoring type applied to it, with ties resolved in favor of the less common type to improve balance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, Washington, DC, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM

Due to the imbalanced nature of the dataset, SMOTE and Random Under-sampling techniques were applied to improve model performance by balancing class distribution.

3 Results and Discussion

This section presents the results of a comprehensive evaluation of two distinct data reduction strategies applied to time-series measurement data:

- (1) **Aggregation-based reduction**, where each component is represented by a single statistical summary using functions such as *mean*, *sum*, or *median*.
- (2) **Sequence-based modeling**, where all individual time-series records are retained and processed using a Recurrent Neural Network (RNN) to capture temporal dependencies.

The objective was to assess how these two data representation approaches affect the predictive performance of various machine learning classifiers.

A wide range of models was evaluated, including Random Forest, Extra Trees, K-Nearest Neighbors (KNN), Extra Tree, Decision Tree, Support Vector Classifiers (SVC and LinearSVC), Logistic Regression, Perceptron, and RNN. For each model and data reduction method, we conducted hyperparameter tuning using randomized search cross-validation and applied mutual information for feature selection. Evaluation was performed using standard metrics, with a primary focus on the F1-score, as summarized in Table 1. We chose to present the F1-scores, as they provide a more accurate representation of performance, particularly for imbalanced datasets.

The results clearly indicate that tree-based models, particularly Random Forest, consistently achieve the highest F1-scores across all datasets and reduction strategies. Among the aggregation methods, the sum-based approach often yields the best results, especially when paired with ensemble methods. For example, Random Forest achieved an F1-score of 0.93 on the jEdit dataset using the sum aggregation method.

Interestingly, despite the RNN's ability to model temporal sequences, its performance lags behind most traditional models using aggregated features. This suggests that for the studied software engineering datasets, temporal granularity may not provide a significant predictive advantage.

These findings highlight the practical value of statistical aggregation in time-series reduction, offering not only simpler and more interpretable models but also superior or comparable predictive performance. Moreover, this approach is computationally efficient and easier to deploy in production settings.

4 Conclusions and Future Work

The evolution of software systems has highlighted the need to improve maintainability, with increasing attention given to analyzing source code through artificial intelligence techniques. Numerous studies suggest strong correlations between software maintainability, technical debt, and refactorings.

This paper investigates different approaches for detecting software refactoring opportunities based on technical debt issues. Three open-source Java projects—jEdit, FreeMind, and TuxGuitar—were analyzed. Technical debt data were obtained via static analysis, while refactoring data were collected using RefactoringMiner by

Model	jEdit			FreeMind			TuxGuitar		
	Avg	Sum	Median	Avg	Sum	Median	Avg	Sum	Median
RandomForest	0.92	0.93	0.91	0.91	0.92	0.90	0.90	0.91	0.89
ExtraTrees	0.90	0.91	0.90	0.90	0.90	0.88	0.88	0.90	0.87
KNN	0.88	0.89	0.87	0.86	0.87	0.86	0.85	0.86	0.85
ExtraTree	0.88	0.89	0.89	0.87	0.87	0.87	0.85	0.86	0.86
DecisionTree	0.83	0.84	0.84	0.81	0.82	0.83	0.80	0.81	0.82
SVC	0.86	0.87	0.87	0.85	0.86	0.86	0.83	0.85	0.85
LogisticRegression	0.79	0.80	0.80	0.77	0.78	0.78	0.76	0.77	0.77
LinearSVC	0.75	0.76	0.75	0.74	0.75	0.74	0.72	0.74	0.73
Perceptron	0.68	0.69	0.67	0.66	0.67	0.65	0.65	0.67	0.64
RNN	0.65	0.65	0.65	0.64	0.64	0.64	0.63	0.63	0.63

Table 1: F1-Score values for each model, dataset, and reduction method

comparing two consecutive versions of each project. The combined dataset was processed using two distinct approaches.

The first approach compressed technical debt issues to a single entry per software component and included the count of detected issues as an additional feature. Classical machine learning algorithms were applied, with the ExtraTrees algorithm achieving the best results—0.95 accuracy and 0.93 F1-score.

The second approach retained multiple entries per component, using a Recurrent Neural Network (RNN). However, this method underperformed, achieving only a 0.66 accuracy and a 0.65 F1-Score.

A key limitation of the study is the exclusive use of Java-based projects, which may reduce the generalizability of the results to other programming languages. Additionally, the study does not consider certain details such as the content of technical debt messages or specific characteristics of refactorings, which may improve predictive performance. Future work could also include additional software metrics to enhance the model, though these may be better suited for the first approach due to data structure constraints. Another direction that is worth investigating, is representing by using other data augmentation techniques in order to balance the dataset. In this study, we chose to directly augment the features, but another possibility would be to augment the source code that will be further analyzed in the static analysis and by the refactoring detection tools.

Further research is warranted into better methods for reducing technical debt data and improving the performance of the second approach. Overall, the findings offer valuable insights into the relationship between technical debt and refactoring needs, contributing to better assessment of software maintainability.

References

- [1] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.
- [2] Peter Hegedus, Istvan Kadar, Rudolf Ferenc, and Tibor Gyimothy. 2018. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology* 95 (2018), 313–327. doi:10.1016/j.infsof.2017.11.012
- [3] Arthur-Jozsef Molnar. 2020. *Collection of technical debt issues in FreeMind, jEdit and TuxGuitar open source software*. doi:10.6084/m9.figshare.12363581.v1
- [4] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930–950. doi:10.1109/TSE.2020.3007722