AI Assisted/Automated code refactoring

How the development of AI may impact the future of code refactoring

Loris Tomassetti
Linz, Austria
loris.tomassetti@outlook.com

Alexander Weißenböck
Linz, Austria
alewei934@gmail.com

***Abstract*—This Paper aims to shed light onto developments in AI Assisted/Automated Code refactoring, how it can help the indtustry and which models work most efficiently to tackle different challenges code refactoring brings. This will be done by going over various literature describing first the challenges at hand and afterwards discussing several possible solutions that have been tested to gain a greater understanding and to generate an informed outlook into further developments of this technology.**

***Index Terms*—machine learning algorithms, software code refactoring, deep neural network**

I. Introduction

Refactoring, as defined by Fowler [6], is "the process of changing a software system in such a way that does not alter the external behaviour of the code yet improves its internal structure". More and more empirical studies have since established a positive correlation between refactoring operations and code quality metrics. All this evidence hints at refactoring being a high-priority concern for software engineers. [1].
However, deciding when and how to refactor can prove to be a challenge for developers. Refactoring in an early stage may cost too much for what you're getting out of it, and refactoring too late may cause the refactor to be an even bigger time commitment. [9]
Tools have been in the hands of many developers to make this process more streamlined for years now. Analytics tools to sniff out bugs or give hints on how to improve code quality such as PMD, ESLint, and Sonarqube can be integrated into different stages of a developers' workflow, e.g. inside IDEs, during code review or as an overall quality report. [1]
Taking a closer look at these tools, however, reveals that they commonly have a lot of false positives, making developers lose their confidence in them. Often, the detection strategies are based on hard thresholds of just a handful of metrics, such as lines of code in a file (e.g. PMD's famous "problematic" classification occurring once a method reaches 100 lines per default). These simplistic ways of detection simply aren't able to capture the full complexity of modern software systems.
Manually analyzing hundreds of metrics and figuring out which ones are the cause of technical dept is very hard and almost impossible for tool developers, which is where machine learning-based solutions come into play. [8] [10]
We will take a closer look at how exactly different models go about this task in section III

II. Approaches for Automation

This section will cover the different approaches for automation based on machine learning algorithms and will take a closer look at large language models used by thousands of software developers today. Additionally we will look at supervised machine learning algorithms that have been experimented with and create a comparison between them. Lastly, we will be taking a look at dedicated models with the example of DNNFFz.

A. Large Language Models based on GPT Models

This section covers the experiments' results of the paper [2, AI-Driven Refactoring: A Pipeline for Identifying and Correcting Data Clumps in Git Repositories].
In the fields of ai assisted workflows, large language models (LLMs) have shifted from a niece speciality to an almost omnipotent tool which finds applications from image creation to code generation. [11] LLMs are huge deep-learning models pre-trained on enormous amounts of data. They are especially known for their ability to be trained on datatest of specific domains but can also be used on a broad spectrum of general knowledge, making them incredibly flexible. [2]
Being the best-known LLM, OpenAI's Generative Pre-trained Transformer series (GPT) with its versions GPT-3.5, GPT-3.5 Turbo and GPT-4. Temperature is a key parameter for GPT models, having a value ranging from zero to one, this parameter determines the predictability of the results. A higher temperature leads to more variety in the LLM and vice versa.

1) Detection: Taking a look at detection itself, the median sensitivity of GPT-3.5-Turbo is 0 which indicates many data clumps are undetected and many false positives. Submitting all files in bulk also made the model trade off its sensitivity with the specificity parameters, with the median sensitivity reaching 50 percent, but the specificity only being 14 percent. Apparently, the model is looking at all the information and is finding more data clumps. But also potentially leading to more false positives. The temperature also has a similar trade-off. Higher temperatures lead to a lower sensitivity and the other way around.

2) Refactoring: If you prompt a GPT model to refactor source code while also giving it the location of the data clumps, GPT-3.5 and GPT-4's median is identical, lying at 68 percent. These results show if you know where to look for data clumps and which places to refactor, both models can

refactor the source code just as well as the other. GPT-3.5-Turbos arithmetic mean is less which can be explained by the existence of more overall compiler errors. On the same note: the median of the three instruction variants is also identical. Higher temperature values also resulted in a median of 0 percent, indicating more non-compilable code.

3) Combining Detection and Refactoring: The final step of the experiment is combining detection and refactoring into one step. At this point, the limitations of GPT-3.5-Turbo become clear. The model scores a median score of 7 percent compared to 82 percent of GPT-4. Surprisingly, however, providing no definitions about data clumps leads to the best results, reaching a median of 46 percent. All other instruction types are 0 percent each. Another experiment, held in the paper mentioned in section I, [1, The effectiveness of supervised machine learning algorithms in predicting software refactoring], compared different machine learning models with each other, with the "Random Forest" Model performing best out of all the tested models.
It appears as if machine learning models commonly perform best in a close-to-random environment.

B. Comparing Linear Regression, linear SVM, Naive Bayes, Decision Trees, Random Forest, and Neural Network

This section covers the experiments' results of the article [1, The effectiveness of supervised machine learning algorithms in predicting software refactoring].

1) Overview of the used algorithms: First we will briefly go over the differences between these algorithms, starting with Logistic Regression.

- Logistic Regression (LR) [3] is centered on combining input values using coefficient values to predict an outcome value.
- (Gaussian) Naive Bayes algorithms [15] use training data to compute the probability of each outcome based on the information extracted from the feature values.
- Support Vector Machines (SVMs) [5] search for the best hyper-plane to separate the training instances into their respective classes in high-dimensional space.
- Decision Trees [12] yield hierarchical models composed of decision nodes and leaves, presenting a partition of the feature space.
- Random Forest [4] is an algorithm using a number of decision trees with random subsets of the training data.
- Neural Networks [7] create an architecture that is similar to neurons and are made up of one or more layers of these neurons. They essentially act as a function, mapping inputs to their respective classes.

In the conducted experiment, a pipeline for each of the previously mentioned algorithms validates the outcomes and returns the precision, recall, and accuracy of all the models. Once a classification model is trained for a given refactoring, the model would predict true in case an element (i.e. a class, method, or a variable) should undergo a refactoring or false if it should not.

2) Research Questions: The experiment aims to answer three research questions (RQs):

RQ1 How Accurate are Supervised ML Algorithms in Predicting Software Refactoring? This question explores how accurately the different models predict refactoring opportunities, while using Logistic Regression as a base line.

RQ2 What are the Important Features in the Refactoring Prediction Models? This question regards features that are most relevant to the models and have the biggest impact on the outcome.

RQ3 Can the Predictive Models be Carried Over to Different Contexts? Whether or not refactoring prediction models *need* to be trained for this specific context or a more generalized model is sufficient potentially reduces the cost of applying and re-training the models in the real-world. The question is tackled by comparing the accuracy of predictive models against independent datasets.

3) Data Sources: The targeted projects for the experiment are collected from three different sources: The Apache Software Foundation (ASF), F-Droid (a software repository of Android mobile apps), and GitHub. They also used a highly sophisticated method to extract the labelled instances using RefactoringMiner [14], a tool for refactoring detection having an average precision of 99.6 percent [13].

4) Answering the Research questions: Evaluation gave answers to the three research questions formulated above.

RQ1 Observation 1: *Random Forest models are the most accurate in predicting software refactoring.* It's average accuracy for class, method, and variable-level refactorings are 0.93, 0.90, and 0.94 respectively. The second best model is Decision Trees which only falls shortly behind with a score .04 less on almost every dataset compared to Random Forest.
Observation 2: *Random Forest was outperformed only a few times by Neural Networks.* Specifically it outperformed Random Forest 4 times in terms of accuracy in the F-Droid dataset, and in two opportunitiesin both the Apache and GitHub databases, with the difference always lying at around 1 percent.
Observation 3: *Naive Bayes models present high recall, but low precision.* They presented recalls of 0.94, 0.93, 0.94, and 0.84 in the combined dataset. Unfortunately, these models had by far the worst precision values: 0.62, 0.66, 0.62, and 0.67 in the same datasets.
Observation 4: *Logistic Regression shows good accuracy.* The most straightforward model in the study presented a fairly high average accuracy, consistently outperforming Naive Bayes models with an average of 0.83 across all datasets.

RQ2 Observation 5: *Process metrics are highly important in class-level refactorings.* The top ranking metrics are quantity of commits, lines added in a commit, and number of previous refactorings. Top-5 to Top-10 rankings mostly consist of process metrics with the occational ownership

metrics, appearing 32 times in the top-1 ranking.

Observation 6: *Class-level features play an important role in method-level and variable-level refactorings.*

C. Dedicated Models

1) DNNFFz:

III. Benefits of AI-Powered Refactoring

A. Improved Code Quality

B. Enhanced Maitainability

C. Reduction of Technical Dept

IV. Challenges and Limitations

A. Over-reliance on Automation

B. Potential for Unintended Consequences

C. Performance Concerns

V. Future Directions and Research Opportunites

A. Personalized Code Refactoring Suggestions

VI. Discussion

VII. Methodology

VIII. Conclusion and Outlook

References

[1] Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius HS Durelli. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, 48(4):1432–1450, 2020.

[2] Nils Baumgartner, Padma Iyenghar, Timo Schoemaker, and Elke Pulvermüller. Ai-driven refactoring: A pipeline for identifying and correcting data clumps in git repositories. *Electronics*, 13(9), 2024.

[3] Christopher M Biship. Pattern recognition and machine learning (information science and statistics). *Springer New York*, 2007.

[4] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[5] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20:273–297, 1995.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2018.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[8] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 576–585, 2002.

[9] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6):18–21, 2012.

[10] R. Leitch and E. Stroulia. Assessing the maintainability benefits of design restructuring using dependency analysis. In *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No.03EX717)*, pages 309–322, 2003.

[11] André Meyer-Vitali. Ai engineering for trust by design. In *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering-MBSE-AI Integration, Rome, Italy*, pages 21–23, 2024.

[12] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[13] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 48(3):930–950, 2022.

[14] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th international conference on software engineering*, pages 483–494, 2018.

[15] H Zhang. The optimality of naïve bayes. flairs2004 conference, 2014, 2014.