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 on developments in AI-assisted/Automated Code refactoring, how it can help the industry and which models work most efficiently to tackle different challenges code refactoring brings. This paper will cover various literature describing first the challenges at hand and afterwards discussing several possible solutions that got 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 [9], 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. [3].
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. [13]
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. [3]
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) [3] 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. [11] [15]
We will take a closer look at how exactly different models go about this task in section III

II. Refactoring

Fowler states refactoring describes changing the internal structure of code without changing its behaviour. [9] Refactoring can lead to improvements in code in various aspects. By reducing the code's complexity while keeping its behaviour identical, it becomes more human-readable and, therefore, more maintainable since it is easier to expand on it in the future. [12] Another great advantage of refactoring is the reduction of technical debt. This term refers to a metaphorical debt when subpar solutions get chosen to accelerate software production. While the chosen solutions might suffice they often do not take further developments into account and cause problems later. [14] And of course, the overall code quality can be improved. The main difficulty in code refactoring lies in finding places to optimize, otherwise called "bad smells". These bad smells can be found virtually anywhere in the source code. Refactoring can happen on different levels, like class-level refactoring describing the extraction of classes into one or more sub-classes, while variable-level refactoring could be something like executing an action inline instead of creating an unnecessary variable. [3] This makes it very difficult to locate these bad smells and as a consequence time consuming and expensive.

III. 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.

A. Large Language Models based on GPT Models

This section covers the experiments' results of the paper [5, 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 niche speciality to an almost omnipotent tool which finds applications from image creation to code generation. [16] 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. [5]

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 per cent, but the specificity only being 14 per cent. 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 per cent. 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 per cent, 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 per cent compared to 82 per cent of GPT-4. Surprisingly, however, providing no definitions about data clumps leads to the best results, reaching a median of 46 per cent. All other instruction types are 0 per cent each. The experiment covered in the next subsection, III-B, will compare 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 [3, 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) [6] is centred on combining input values using coefficient values to predict an outcome value.
- (Gaussian) Naive Bayes algorithms [24] use training data to compute the probability of each outcome based on the information extracted from the feature values.
- Support Vector Machines (SVMs) [8] search for the best hyper-plane to separate the training instances into their respective classes in high-dimensional space.
- Decision Trees [19] yield hierarchical models composed of decision nodes and leaves, presenting a partition of the feature space.
- Random Forest [7] is an algorithm using several decision trees with random subsets of the training data.
- Neural Networks [10] create an architecture that is similar to neurons and is 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 baseline.

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 [21], a tool for refactoring detection having an average precision of 99.6 percent [20].

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

RQ1 Observation 1: *Random Forest models are the most accurate in predicting software refactoring.* Its average accuracy for class, method, and variable-level refactorings are 0.93, 0.90, and 0.94 respectively.
Observation 2: *Random Forest was outperformed only a few times by Neural Networks.* Specifically, it outperformed Random Forest 4 times in terms of accuracy, and

in two opportunities with the difference always lying at around 1 per cent.

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.* 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 the number 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 occasional 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.* In the top-1 ranking for method-level refactoring models, 13 out of 17 features are class-level features. Variable-level shows the same for 11 of the features.

Observation 7: *Some features never appear in any of the rankings.*

RQ3 Observation 8: *Random Forest still presents excellent precision and recall when generalized, but smaller when compared to previous results.* Training Random Forest models with the GitHub dataset and testing it in Apache achieves a precision of 0.87 and recall of 0.84 and still performs reasonably well when trained on smaller datasets. However, Random Forest performs remarkably better when trained on the specific datasets.

Observation 9: *Method and variable-level refactoring models perform worse than class-level refactoring.* Using Random Forest models trained on the GitHub dataset and testing on the F-Droid dataset, the average precision and recall on the class level is 0.92. On the contrary, average precision and recall at the method-level are 0.77 and 0,72 respectively; at the variable level, 0.81 and 0.75 for precision and recall are observed.

Observation 10: *SVM outperforms Decision Trees when generalized.*

Observation 11: *Logistic Regression is still a somewhat good baseline.* Trained on the GitHub dataset and tested on the Apache dataset, it shows an average precision and recall of 0.84 and 0.83, with the worst results when trained on the Apache dataset and tested on F-Droid.

Observation 12: *Heterogeneous datasets might generalize better.* The Apache and F-Droid datasets present lower precision and recall when carried to other contexts. Cross-testing these datasets never went beyond precision and recall values beyond 0.78, which was not the case for GitHub, which is a more heterogeneous dataset.

5) Summary: The experiments main findings show that Random Forest models outperform other Machine Learning models in predicting software refactoring in almost all tests. Furthermore, process and ownership metrics seem to play a crucial role in the creation of better models. Finally, models trained with data from heterogeneous projects generalize better and achieve good performance.

More importantly: ML algorithms can be used to accurately model the refactoring recommendation problem.

C. Summary

Both of the covered experiments show great success with automating code refactoring. Out of the GPT-Models GPT-4 outperformed both the GPT-3.5 and GPT-3.5-Turbo variants. Out of the compared machine learning models, Random Forest models consistently outperform all other models in every field. Using classic machine learning models should always, if possible, be trained on the dataset or at least homogeneous datasets as they perform by far the best when trained and tested on the same datasets. GPT-Models do not have such a restriction but may overall perform slightly worse than Random Forest Models.

The next step in this field of research should be strictly comparing GPT-4 with Random Forest.

IV. Benefits of AI-Powered Refactoring

Logically, automated refactoring comes with all the benefits discussed in the prior chapter II, namely more readable, higher quality code that is more maintainable, but more importantly, it makes all this possible in a shorter amount of time. [17] Automating the refactoring process is not a new idea at all numerous tools have been developed and modern IDEs already support some refactorings. However, those are often small-scale and need input on what to refactor. [22] It could also be verified that automated refactorings bring a statistically significant time improvement to nearly every type of refactoring. [18] This of course means less cost in extension. The same study also found that currently automated refactoring is more often used for smaller changes since the refactoring is less error-prone. Using AI the possible use cases could be significantly expanded and lead to greater time savings.

V. Challenges and Limitations

A. Over-reliance on Automation

Especially when using applications like Chat GPT users often rely too much on the model understanding their problem and give insufficient and/or vague requests which have a lot of room for interpretation. This can then, unsurprisingly, lead to undesired results. [2]

B. Potential for Unintended Consequences

A big reason software developers often choose to not use atomated refactoring is the unpredictability of the result. "[...] If I cannot guess, I don't use the refactoring. I consider it not worth the trouble. [...]" [22] Since refactoring can span multiple files it can be hard for a developer to check all the changes after an automatic refactoring has occured. Especially generative AI solutions which do not only handle smell detection but also the generation of solutions have often

found to provide mixed results, which can lead to the behavior of the code being changed. This also leads to reduced trust to those tools by developers. [23]

C. Performance Concerns

While big strides in the technology are being made, it is still a present issue that some automatic refactorings do produce errors especially when misused [1] Especially when using AI based tools it is not always given that the behaviour of the code is indeed the same as before, making the refactoring itself faulty. Baqais and Alshayeb could find quite a lack in checking for bahavior in a multitude of research papers, leaving out a crucial step. [4]

D. Trust and knowledge gaps

Some developers stated, when using an AI refactoring tool, it would probably be useful to know how it works in the background to potentially get better results. [23] This would then require more focus on learning those Models. Also in another study Developers did not use certain automated refactorings of IDE's because they did not know of their existence. [18] In both of those cases better education about already existing tools alone could help improve performance.

VI. Future Directions and Research Opportunites

Due to these discussed limitations there are two main takeaways for problems that must be tackled for AI powered refactoring to be truly industry changing. Firstly there has to be more work put into the systems to avoid bahavioural changes of the refactored software. Secondly it will be very important to improve the communication between tools and develorpers as well as improve education about existing tool so that they can be used to their full potential.

A. Personalized Code Refactoring Suggestions

VII. Conclusion and Outlook

References

[1] Jehad Al Dallal and Anas Abdin. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(1):44–69, 2018.

[2] Eman Abdullah AlOmar, Anushkrishna Venkatakrishnan, Mohamed Wiem Mkaouer, Christian D. Newman, and Ali Ouni. How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations, 2024.

[3] 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.

[4] Abdulrahman Ahmed Bobakr Baqais and Mohammad Alshayeb. Automatic software refactoring: a systematic literature review. *Software Quality Journal*, 28(2):459–502, Jun 2020.

[5] 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.

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

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

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

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

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

[11] 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.

[12] Amandeep Kaur and Manpreet Kaur. Analysis of code refactoring impact on software quality. In *MATEC Web of Conferences*, volume 57, page 02012. EDP Sciences, 2016.

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

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

[15] 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.

[16] 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.

[17] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E Johnson, and Danny Dig. Using continuous code change analysis to understand the practice of refactoring. *University of Illinois*, 2012.

[18] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *ECOOP 2013–Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings 27*, pages 552–576. Springer, 2013.

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

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

[21] 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.

[22] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 233–243, 2012.

[23] Justin D. Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I. Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. Perfection not required? human-ai partnerships in code translation. In *Proceedings of the 26th International Conference on Intelligent User Interfaces*, IUI '21, pages 402–412, New York, NY, USA, 2021. Association for Computing Machinery.

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