Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

5-10-2024

Improving Automatic Refactoring Candidate Identification

Ryan Devoe
rmd9481@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation

Devoe, Ryan, "Improving Automatic Refactoring Candidate Identification" (2024). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Improving Automatic Refactoring Candidate Identification

by

Ryan Devoe

A thesis submitted in partial fulfillment of the
requirements for the degree of
Master of Science
in Software Engineering

B. Thomas Golisano College of Computing and
Information Sciences
Rochester Institute of Technology
Rochester, NY

May 10, 2024

Approved by:

Dr. Mohamed W. Mkaouer

Dr. Christian Newman

Dr. Ali Ben Mrad

Abstract

Extract method refactoring is pivotal for enhancing code readability, maintainability, and modularity by segmenting complex code into clearer, isolated methods. Identifying opportunities for such refactorings necessitates a deep understanding of the codebase's evolution and its intricate relationships. Current methodologies utilize developer commit messages, advanced graph analysis, and diverse machine learning approaches to automate this identification process.

This research delves into the application of deep learning-based Large Language Models (LLMs) to tackle the complexities inherent in extract method refactoring. We introduce innovative approaches, including the use of LLMs to cluster code blocks based on complex patterns and dependencies, and the analysis of developer commit messages to infer the intent behind refactorings. These methods aim to enhance the precision of identifying refactoring opportunities by leveraging historical code data and contextual insights.

Through rigorous experiments, we compare the efficacy of our proposed methods against traditional refactoring tools using metrics such as precision, recall, and F1-score. Our findings reveal the significant potential of integrating deep learning techniques into the refactoring workflow, enhancing the automation and efficacy of software maintenance.

This study not only validates the use of deep learning-based approaches for code refactoring but also paves the way for future research aimed at the continuous improvement of automated software maintenance tasks.

This thesis is dedicated to my parents, whose love and guidance are with me in whatever I pursue. They have given me the foundation of values which I will carry all my life, and have taught me the importance of hard work and a dedication to pursuing my dreams. Their sacrifices and unwavering support have shaped who I am today, and for this, I am eternally grateful.

With all my love and gratitude,

Ryan Devoe

Contents

1 Introduction **1**

2 Background **4**

2.1 Extract Method Refactoring 4

2.2 Code Smells 5

2.3 Large Language Models 6

2.4 Natural Language Processing 6

2.5 Code Embeddings 6

3 Research Objective **8**

3.1 Motivation and Contribution 8

3.2 Research Questions 9

4 Related Work **10**

4.1 Traditional Techniques for Refactoring Identification 10

4.2 Machine Learning Approaches to Refactoring 11

4.3 Advanced Code Representation Techniques 12

5 Methodology **14**

5.1 Dataset Generation 14

5.2 Feature Learning 17

5.3 Classification 18

5.4 Investigating Developer Intent 20

5.5 Investigating LLM Identified Clustering 22

5.6 Evaluation Metrics 24

6 Analysis & Discussion **25**

7 Threats to Validity **33**

8 Conclusion **36**

9 Acknowledgement **39**

List of Tables

5.1 Optimal Hyper-Parameter Values for Random Forest 20

6.1 Embeddings vs. Encoded Embeddings 27

6.2 Extracted Keywords for Different Refactoring Motivations . . . 29

Chapter 1

Introduction

Refactoring is a fundamental practice in software engineering that aims to enhance the structure of code without altering its underlying functionality. This practice is essential for maintaining the readability, flexibility, and testability of codebases, making them more adaptable to changing requirements and easier to maintain [1], [2], [3]. Among the various refactoring techniques, extract method refactoring is particularly significant due to its role in reducing code duplication and improving code organization.

Identifying when and where to apply extract method refactoring remains a complex issue that often relies heavily on a developer's experience and knowledge of the codebase's history. Traditional approaches to identifying refactoring opportunities are predominantly manual, making them both time-consuming and susceptible to human error. In some instances, developers may use automated tools to generate code quality metrics and identify code smells, but the results of these tools still require careful interpretation to distinguish

true refactoring opportunities and which refactoring techniques to apply [4].

Recent developments in machine learning have impacted the methods used to identify refactoring opportunities. Researchers have employed various machine learning techniques that utilize code properties and process metrics to train algorithms for different types of refactoring [5], [6], [7]. These metric-based approaches have demonstrated state-of-the-art performance in identifying candidates for refactoring [5]. An issue is that these metrics focus predominantly on structural elements like the number of lines, loops, and assignments, so they often fail to capture the more nuanced semantic and behavioral aspects of source code. These aspects are crucial for accurately classifying and effectively improving the refactoring process.

While metrics provide a quantifiable measure of code complexity and structure, they do not necessarily reflect the underlying semantics that dictate code behavior. This limitation has led researchers to explore additional methods to enhance the detection and classification of refactoring opportunities. For instance, studies have attempted to encode the contextual and syntactic characteristics of codebases using advanced embedding techniques such as Code2Vec [8]. The effectiveness of these approaches is heavily dependent on the quality of the embeddings produced, showing the need for sophisticated techniques that can deeply understand the semantics of the source code.

In this paper, we attempt to advance the work of Palit et al. [9]. We attempt to refine the process of automating the identification of extract method refactoring opportunities by classifying refactorings based on commit messages, which we believed would provide contextual insights that traditional metrics

may overlook, and we explored the use of a large language model to cluster identified refactorings, thereby enhancing our understanding of refactoring patterns and improving the classifier's accuracy by training it on these grouped refactorings.

Chapter 2

Background

This Background section provides an overview of five fundamental concepts instrumental to our research:

1. Extract Method Refactoring
2. Code Smells
3. Large Language Models
4. Natural Language Processing
5. Code Embeddings

2.1 Extract Method Refactoring

Extract Method Refactoring is a technique used to improve code maintainability and readability by isolating specific groupable code fragments into new

methods without altering functionality. This refactoring is beneficial for reducing code complexity, improving testability, and improving code organization, making future changes more manageable. Figure 2.1 shows an example of a method of pre- and post-extraction method refactoring.

Figure 2.1: Method before extract method refactoring (left)(truncated) and after extract method refactoring (right)

2.2 Code Smells

Code smells are an important concept to understand in the realm of extract method refactoring because they are one of the main motivations behind it. They are indicators of potenial issues in the code that may not fully break it, but can lead to further issues later in the development process. Design defects such as duplicate code, dead code, and long method are examples of code smells. To resolve code smells, these design defects that violate software design principles and decrease code quality should be removed [10].

2.3 Large Language Models

Large Language Models (LLMs) are advanced artificial intelligence systems designed to understand and generate human-like text by learning from large amounts of textual data. These models have significantly influenced natural language processing (NLP) tasks (defined in Section 2.4), ranging from translation and summarization to question-answering and text generation. Some popular LLMs include GPT (Generative Pre-trained Transformer) and BERT (Bidirectional Encoder Representations from Transformers), both of which are integral components to many current NLP applications due to their deep understanding of language nuances [11] [12].

2.4 Natural Language Processing

Natural language processing is the field of artificial intelligence that aims to enable computers to understand human language as naturally as humans, involving tasks such as sentiment analysis, speech recognition, and response generation [13].

2.5 Code Embeddings

Code Embeddings are numerical representations of source code that capture syntactic and semantic characteristics essential for various software engineering tasks, such as refactoring and code search. These embeddings enable machine learning models to process code similarly to natural language, using techniques

like code2vec to learn from the structure and naming in code fragments. By representing code snippets as vectors in a high-dimensional space, code embeddings facilitate the comparison of code parts based on functionality and meaning, which is crucial for automated refactoring tools [6] [7] [8].

The following figure from Alon et al. shows a clear example of how code embeddings can help extract core functionality from code blocks.

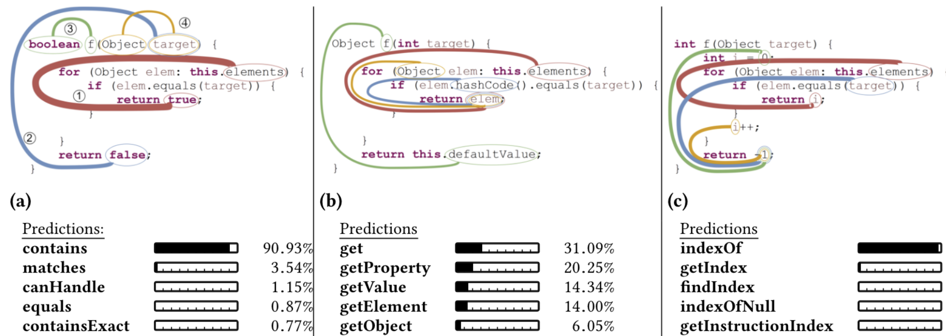

Figure 2.2: Code2vec Demonstration [8]

The three code blocks in Figure 2.2 all appear visually similar. When using traditional techniques for code analysis, which are typically more rule-based and syntactic in nature, the analysis might be limited to surface-level patterns and structural elements, leading to a lack of understanding of the code's context and functionality. Code2vec is able to extract the core functionality from each of the blocks of code, showing the correct predictions for each block.

Chapter 3

Research Objective

3.1 Motivation and Contribution

Our research builds on the work of Palit et al. [9], which is motivated by the challenges inherent in extract method refactoring. Current approaches to identifying refactoring opportunities include heuristic and metric-based tools for automated code analysis [5] [6] [7], but often miss the semantics of the code and context. The identification processes tend to overlook the historical development of the codebase and the intricate semantic relationships, which can lead to suboptimal refactoring suggestions and a lack of actionable insights. Palit et al. does a good job addressing the shortcomings of other approaches by utilizing code embeddings to capture the semantics of code blocks.

The contributions of this research are the introduction of novel approaches that utilize large language models for the analysis of code segments to classify and cluster similar semantical pieces, and utilizing developer commit messages in order to identify refactorings based on their intent. Through extensive

experimentation, we seek to validate the efficacy of our models against the approach of Palit et al. and set the stage for future research in automating and enhancing software maintenance tasks.

3.2 Research Questions

- **RQ1:** ***Can deep learning-based natural language models effectively be used to cluster code blocks based on complex patterns and dependencies to improve the existing approach of identifying extract method refactoring opportunities?***

  This question evaluates the capability of deep learning models in understanding the semantic relationships within code for clustering code blocks together.

- **RQ2:** ***How can the integration of developer commit messages, reflecting the intent behind extract method refactorings, enhance the accuracy of the classifier in the existing approach?***

  This question evaluates how using the intent of a refactoring based on developer commit messages to cluster like refactorings together can effect the performance of the existing approach.

Chapter 4

Related Work

Many studies addressed challenges in software maintenance in general [14–122], and refactoring in particular. The landscape of software maintenance and the identification of refactoring opportunities have been extensively explored, with a particular focus on the methodologies for automating extract method refactoring and the identification of refactoring opportunities. Extract method refactoring is a vital maintenance activity that aims to improve code quality and readability, which are essential factors in proper software development practices.

4.1 Traditional Techniques for Refactoring Identification

Traditional techniques in refactoring identification primarily focus on heuristic rules and structural analysis of code. The literature review conducted by Al

Dallal [123] provides a foundational perspective on the current uses of metrics and patterns to detect code smells, which signal the need for refactoring. The work of Czibula and Czibula [124] exemplifies this category with their hierarchical clustering-based algorithm to suggest refactorings for improved software design, advocating for a systematic method restructuring.

Bavota et al. [125] uses semantic cohesion measures with structural metrics to detect refactoring opportunities, arguing for the significance of understanding method interrelations beyond mere structural attributes. Complementing this is the approach of Tsantalis and Chatzigeorgiou [126], which automates the identification of extract method refactoring opportunities by focusing on complete computation slices and object state slices, combining the concepts of functionality and state alteration.

4.2 Machine Learning Approaches to Refactoring

Machine learning introduces a novel lens to refactoring identification, with Aniche et al. [5] pioneering the use of supervised algorithms to predict refactoring needs across multiple granularities. Their work motivated many more studies where process and ownership metrics are as crucial as code metrics in guiding refactoring decisions. Van Der Leij et al. [127] built on this premise, applying machine learning models within a financial organization to demonstrate the real-world applicability and accuracy of such models in recommending refactorings.

The exploration by Di Nucci et al. [128] of the use of machine learning for the detection of multiple smell code smells addresses the limitations of

the subjectivity of previous tools, suggesting a more objective and learning-based approach to refactoring. The empirical analysis by Kumar et al. [129] on software metrics prediction further attests to the potential of machine learning to refine the precision of refactoring identification.

Priyadarshni et al. [130] offers a different perspective by analyzing commit messages alongside code metrics to predict refactoring activity. Their findings highlight the predictive power of commit messages in discerning method-level refactoring types, giving importance to the relevance of developers' intentions in the refactoring process.

4.3 Advanced Code Representation Techniques

Advancements in code representation techniques provide an additional dimension to refactoring identification. Kurbatova et al. [7] employ a path-based representation of code, leveraging machine learning to recommend move method refactorings with high accuracy. This progression shows a shift towards recognizing the importance of syntactic and semantic intricacies of code fragments.

Alon et al.'s [8] code2vec framework emerges as a significant step in representing code snippets as continuous vectors, effectively predicting method names and understanding semantic properties, establishing a linkage between code syntax and semantics. Their work not only enhances the capability to predict refactoring opportunities but also offers a nuanced understanding of code functionalities through semantic representation.

The exploration of automated refactoring methods and tools has evolved from rule-based heuristics to sophisticated machine learning models that con-

sider a wide array of metrics and developer intent. The introduction of semantic and structural representations of code further enriches the refactoring landscape, pointing toward more nuanced and context-aware refactoring tools. These diverse approaches underscore the ongoing efforts to refine software maintenance practices and the potential of emerging technologies to support this endeavor.

Chapter 5

Methodology

The following section will present our adopted methodology to identify candidates for extract method refactoring. We start by demonstrating the approach overview in Figure 5.1, we then discuss the details of each phase, and finally, we extend the proposed methodology to explore whether this model is better suited for specific refactoring intents. This methodology is adopted and extended from Palit et al. [9].

5.1 Dataset Generation

To train a classification model, we need to either use existing datasets or create our own. In the case of identifying extract method refactoring opportunities, the dataset must include both positive and negative examples of potential refactorings. Positive examples are typically derived from historical codebase changes, identifiable using tools such as RefactoringMiner. Identifying suitable

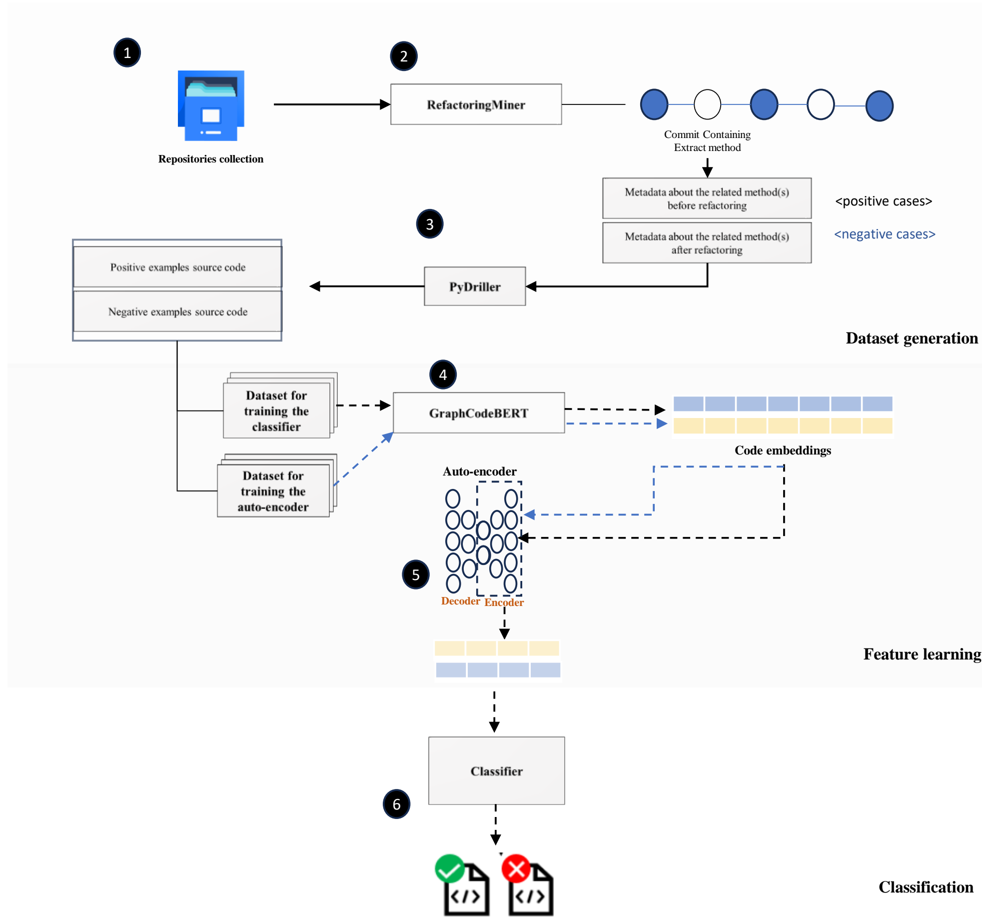

Figure 5.1: Methodology overview

negative examples is more challenging, as not all unchanged code fragments necessarily represent missed refactoring opportunities. Researchers often employ heuristic methods to classify these negative examples. Using pre-existing datasets thus involves inheriting the underlying heuristics that were previously established. We utilized the same approach as Palit et al., generating negative examples by designating a method as negative if it underwent extract method refactoring in the preceding commit [9]. The assumption is that a method which has just been refactored is unlikely to require immediate further refactoring.

The dataset compilation begins by selecting a subset of repositories-5% of the 11,149 open-source Java repositories analyzed in the study by Aniche et al. [5]. This selection is then ran through RefactoringMiner to inspect each commit's version control history for instances of extract method refactorings. Metadata, including the file path and the start and end lines of each method before and after refactoring, are collected to facilitate the extraction of both positive and negative examples.

Once the commit data, file paths, and specific method lines are identified, PyDriller is used to extract the actual source code corresponding to these examples. This procedure creates a dataset comprised of 55,430 positive and negative examples. To ensure the study's reproducibility, we modified the dataset provided by the original replication package by Palit et al., which involved the data cleansing of empty repositories and duplicates. The dataset was divided into two distinct sets for training and testing both the autoencoder and the classifier models, 27,634 and 27,796 respectively.

5.2 Feature Learning

We then utilize GraphCodeBERT (step 4), a pre-trained transformer model, to derive the semantic and syntactic properties of the source code [131]. First, the code undergoes tokenization to accommodate the model's maximum token length of 512, applying truncation as necessary. These tokenized input IDs are then processed through the model's 12 encoding layers. The resultant output, a 768-dimensional vector, is computed by averaging the final representations across the input tokens. This methodology has proven to be more effective than relying solely on the embedding of the [SEP] token.

We then use an encoder (step 5) both as an additional feature extractor and a technique for dimensionality reduction of the GraphCodeBERT embeddings [132]. The autoencoder's architecture includes three fully connected linear layers with ReLU activation, designed to compress the input dimensions down to a bottleneck layer of 128 units. This bottleneck layer captures the essential features of the input, which the decoder then uses to reconstruct the original 768-dimensional input. The autoencoder is trained on a subset of 27,634 examples from the initial dataset, using a 70:30 split for training and testing. The model's performance is assessed on the basis of the reconstruction loss, calculated using Mean Squared Error (MSE) loss, to evaluate the accuracy of the reconstructed outputs against the original inputs.

5.3 Classification

After training the auto-encoder, only the encoder layers are utilized to generate dense representations of a subset of the source code, which are then used to train a binary classification model (step 6). We then evaluate both a Random Forest-based classifier and a Neural Network-based classifier using a distinct subset of 27,796 examples. Palit et al.'s selection of the Random Forest classifier was based on its robust capability to handle non-linear relationships between features and its proven performance in various software engineering tasks, including refactoring identification [128] [133] [5] [127] [9].

The data used for classifier training are divided into training, validation, and test sets using a 70:20:10 stratified sampling approach. To optimize the classifier's performance, a GridSearchCV process is utilized to determine the most effective hyperparameters for each model.

The Random Forest Classifier employs an ensemble learning approach, which effectively captures the non-linear relationships between features. As illustrated in Figure 5.2, the training begins with an initial feature set of 128 dimensions, derived from the encoded data of the autoencoder in step 5. This results in a matrix of size n x 128, which is used across training, validation, or test datasets according to the 70:20:10 split. During the training phase, bootstrap sampling is used to generate varied datasets for different decision trees. This process involves sampling with replacement of the initial encoded data, which helps to understand the influence of each feature on predictive outcomes. The final decision-making process of the Random Forest involves aggregating the predictions from various trees through a majority voting system, enhancing

the overall prediction accuracy.

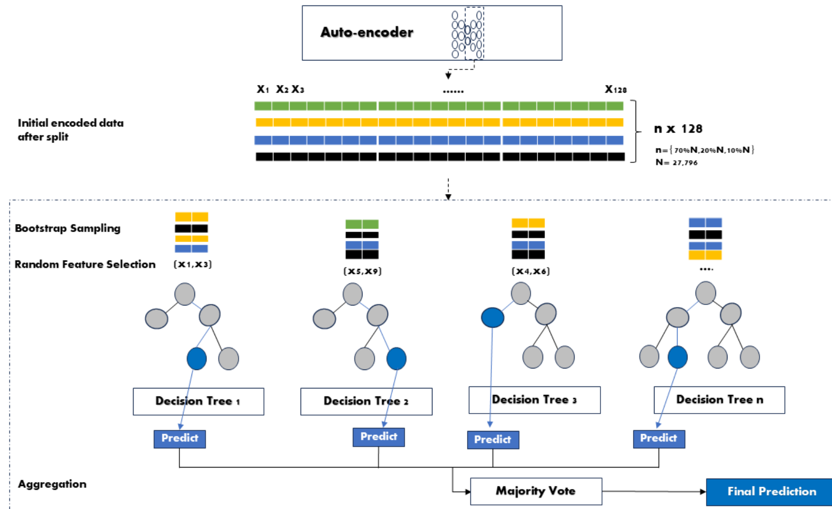

Figure 5.2: Classifying the extract method refactoring candidates using the encoded data and the Random Forest Model

During the GridSearch, we tune a set of important parameters that control the complexity and depth of the random forest. This helps prevent the model from over-fitting to the training data. We present the set of parameters as follows:

- **Maximum Number of Trees**: Maximum number of trees in the random forest.
- **Minimum Samples Split:** The minimum number of samples that we need at each internal node to split it.
- **Minimum Leaf Node Samples:** After splitting an internal node, the

Table 5.1: Optimal Hyper-Parameter Values for Random Forest

Parameter	Search Space	Best Value
Number of trees	[100, 200, 300, 1000]	1000
Minimum samples split	[8, 10, 12]	10
Minimum leaf node samples	[3, 4, 5]	3
Maximum features	[2, 3]	2
Maximum tree depth	[80, 90, 100, 110]	80

resulting nodes must contain at least the Minimum Leaf Node Samples.

- **Maximum Features:** The maximum number of features selected per tree during random feature selection.
- **Maximum Tree Depth:** The maximum depth of the decision trees.

We show in Table 5.1 the set of best values for the RandomForest model along with their search space.

For the Neural Network Model, the architecture is made up of two fully connected layers with ReLu activation function and a final Sigmoid activation function.

5.4 Investigating Developer Intent

As an extension of the proposed approach, our aim is to investigate the effect of developer's motivation when performing extract method refactoring and whether the approach is better suited for specific intents. We believe that if the classifier is trained separately on different intents (e.g. "extract method

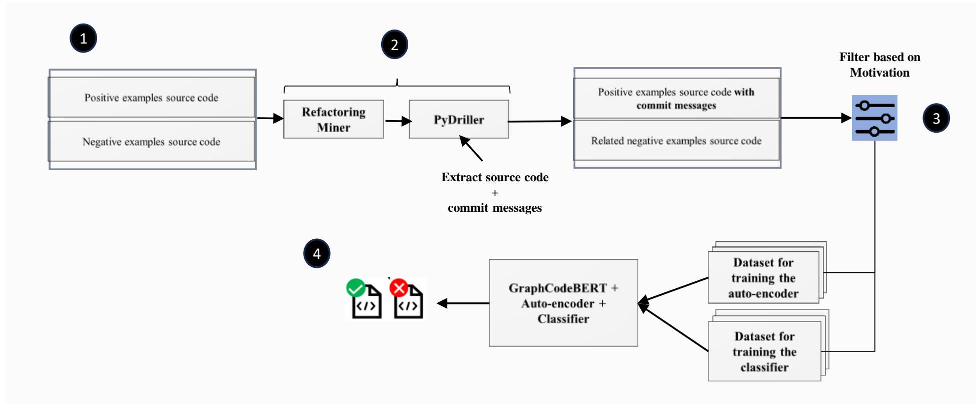

Figure 5.3: Proposed approach to investigating the effect of the intent of extract method refactoring on the performance of classification

refactoring for long methods"), it may perform better in real development scenarios. The proposed technique does not take into consideration these nuances in motivation. By examining the motivations behind extract method refactoring instances, we can assess whether model performance correlates with specific intents and whether our dataset comprehensively represents the range of refactoring motivations. Should any motivations be underrepresented, we also explore how the model performs in these cases.

To achieve this, we follow the approach demonstrated in Figure 5.3. We utilize the existing dataset from our prior methodology (step 1), employing RefactoringMiner to regenerate the necessary metadata for identified extract method candidates. In step 2, we utilize PyDriller to retrieve both the method bodies and their associated commit messages. We only extract negative examples related to the positive examples that satisfy the filter.

The filter (step 3) will be established through the analysis of the commit messages associated with each case of extract method refactoring. The goal is to identify relevant keywords related to specific refactoring intents, which will be used later for filtering. After conducting the analysis, we will retest the approach on specific filters (step 4), re-assessing the strength and weaknesses of the technique based on this newly introduced dimension. Using this protocol, we were able to extract 53,260 positive and negative methods with their commit messages. These methods may come from commits that changed multiple files, which limits the precision of the commit description. To mitigate this, we further refined our dataset to include only those commits affecting a single file, resulting in a more manageable and focused dataset of 5,678 entries. This refinement ensures a more precise analysis of commit intents and their impact on refactoring practices.

5.5 Investigating LLM Identified Clustering

This section extends our existing methodologies by introducing a novel approach that leverages Large Language Models (LLMs) for autonomously clustering extract method refactoring instances. This approach enables the LLM to identify natural groupings in refactoring data based solely on the information extracted from the source code. It aims to enhance the classifier's ability to generalize across various scenarios without the constraint of label-induced overfitting. We will follow the approach shown in Figure 5.4.

This extension will be performed similar to the previous, but the filtering portion in step 4 will be the main difference in these two extensions. Uti-

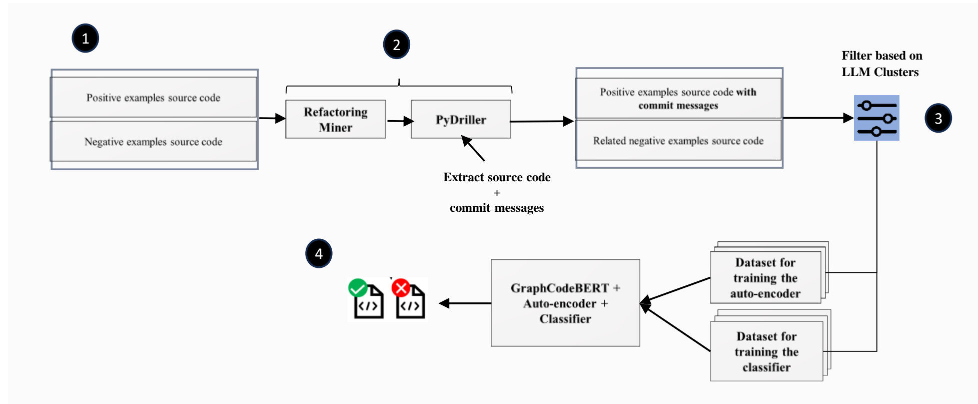

Figure 5.4: Proposed approach to investigating the effect of LLM-based clusters on the performance of classification

lizing an LLM such as OpenAI's GPT, we will allow the model to uncover subtle and complex patterns within the data in order to cluster refactorings into groups (step 4). The LLM-identified clusters are expected to capture various refactoring patterns that may not be explicitly defined by traditional labeling approaches. Once the clusters are formed, each cluster is treated as a distinct category within our dataset. This categorization allows us to train a specialized classification model for each cluster to determine the specific refactoring needs of new code snippets more accurately. By training classifiers on these dynamically identified clusters, our objective is to tailor the prediction models to the nuances of each cluster, potentially increasing the precision and relevance of the refactoring suggestions.

To validate the effectiveness of this approach, we will compare the performance of classifiers trained on LLM-identified clusters against those trained

with traditional label-based methods. Additionally, we will conduct a detailed analysis of the types of refactorings within each cluster to understand the characteristics that the LLM has used to form these groups. This analysis will help determine whether LLM-based clustering leads to more meaningful and actionable refactoring insights compared to conventional methods.

5.6 Evaluation Metrics

We evaluate the performance of the model using accuracy, precision, recall, and F1 score. Palit et al. initially kept the initial ratio of 1:1 positive to negative examples in their testing data, but a number of researchers [134] [128] identify this ratio as unrealistic because it results in very inflated performance of models that end up performing poorly in production [9]. To overcome this issue, they sampled 20 repositories at random from the set of repositories they acquired, which is also the approach we followed. We used RefactoringMiner again to retrieve the identified commits for which extract method refactoring was applied. For each commit, we calculate the ratio of the refactored methods to the total number of methods in the source code. To achieve this, the posCount option in RefactoringMiner is used to get the number of refactoring methods, and the totalCount in PyDriller is used to calculate the total method count. This value is then averaged across the commits to obtain a ratio of 15:85, which is more realistic. The test data are adapted to adhere to this ratio.

Chapter 6

Analysis & Discussion

RQ1: ***Can deep learning-based natural language models effectively be used to cluster code blocks based on complex patterns and dependencies to improve the existing approach of identifying extract method refactoring opportunities?***

We encountered significant challenges in our attempt to leverage Large Language Models (LLMs) for clustering code blocks to improve extract method refactoring identification. The biggest challenge was the model's requirement to comprehend all refactoring samples simultaneously to effectively cluster them, which proved impractical due to the extensive dataset size.

Initial trials involved feeding smaller subsets of the dataset into the LLM to manage data volume and complexity. This method required predefined categories for clustering, which introduced a high risk of overfitting as the model was unable to process enough data samples to form meaningful categories. Attempts to manage the dataset size by clustering encoded embeddings were also

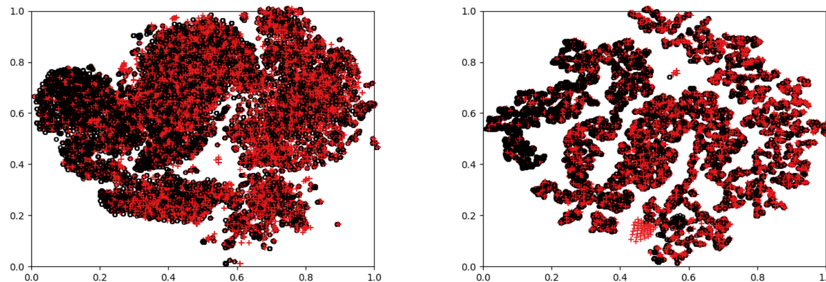

Figure 6.1: Class separation from Raw Embeddings (left) and from Autoencoded Embeddings (right)

considered. Encoded embeddings, being truncated, were expected to facilitate easier handling by the LLM. This leads to our biggest overall challenge when attempting to build off the work of Palit et al. [9]. We could not replicate the same performance enhancements using autoencoded embeddings shown in their study, and our results showed that models trained with autoencoded embeddings performed poorly compared to those trained with raw embeddings.

Figure 6.1 illustrates the t-SNE plots for the raw embeddings and the autoencoded embeddings, where it can be clearly seen that the intended bifurcation/ class separation with the encoded embeddings is not present. Also, when looking at the model performance in our replication, using the autoencoded embeddings provides worse results then the regular embeddings (Table 6.1). These discrepancies led us to abandon the approach of using encoded embeddings for clustering with the LLM. The inadequacy of the LLM to handle extensive data without significant reduction, and the subsequent degradation

Model	Accuracy	Precision	Recall	F1-score
Embeddings	0.93	0.93	0.93	0.93
Encoded embeddings	0.87	0.87	0.87	0.87

Table 6.1: Embeddings vs. Encoded Embeddings

in performance when using encoded embeddings, highlighted a critical limitation in applying LLMs for clustering code in the context of extract method refactoring.

Although this approach is theoretically promising, the practical implementation of LLMs to cluster code blocks for extract method refactoring identification limited us in these trials. The data handling limitations, risk of overfitting, and poorer performance of autoencoded embeddings significantly hindered the effectiveness of this approach. Moving forward, alternative strategies that either enhance the LLM's capacity to handle large datasets or improve the quality of autoencoded embeddings might be necessary to harness the potential of LLMs in this domain.

We encountered several significant challenges in our investigation into the application of large language models (LLMs) for clustering code blocks based on complex patterns and dependencies. Initially, the large size of our dataset proved unmanageable for the LLM, as the model's capacity to handle such extensive data without a reduction in context was limited. This limitation hindered our ability to perform clustering with a comprehensive understanding of all available data. Next, we attempted to input a smaller subset of the dataset to allow the LLM to identify labels based

on detected patterns and dependencies. This approach did not yield the desired results due to insufficient data input, which impeded our ability to derive accurate labels and effectively prevent model overfitting. Despite these setbacks, we recognize the potential of this method. If the challenges associated with the input data size can be addressed, this approach may significantly enhance the performance of the existing methodology for identifying extract method refactoring opportunities.

RQ2: ***How can the integration of developer commit messages, reflecting the intent behind extract method refactorings, enhance the accuracy of the classifier in the existing approach?***

The integration of developer commit messages into the classification model for identifying extract method refactorings can significantly enhance the model's accuracy by providing context that directly reflects the developer's intent. Our analysis of commit messages revealed that while these messages could specify the broader motivation behind code changes, they were not detailed enough to categorize refactoring intents granularly. Even with this, by grouping commit messages into broader categories such as "Adding Functionality/Enhancements," "Fixing Issues," and "Refactor," we gained valuable insights into the typical motivations for refactorings (Table 6.2).

For instance, developers might perform extract method refactoring to "Fix Issues" without explicitly mentioning it as a refactoring, suggesting that it is part of broader bug-fixing activities. Similarly, refactorings aimed at "Adding Functionality" might not be recognized by developers as explicit refactorings, affecting how models trained on code metrics interpret the need for refactoring.

By integrating these categorized intents, the model can be more finely tuned to understand the context of changes, leading to improved prediction accuracy.

Table 6.2: Extracted Keywords for Different Refactoring Motivations

Adding	**Fixing Issues**	**Refactor**
Add	Fix	Simplify
Adding	fixed	simplifying
Added	fixes	Refactoring
Implement	fixing	refactored
Implemented	fixing up	re-factored
Support	Issues	factored out
Introduce	Wrong	Duplication
Enhance	Exception	duplicated
Support	Bugfix	duplicate
Enable		Reduced
Allow		Clean up
Handle		cleanup
Update		cleans up
Updated		cleaned
refined		clean
		cleaner
		Better
		Duplicated
		Perfomance
		Move
		Extract
		Readability
		Improve
		reorg
		complexity

For example, if a commit message includes terms like "clean up" or "simplify," it signals a clear intent for structural improvement, indicating a strong candidate for extract method refactoring. This contextual understanding can help distinguish between necessary refactorings and mere code alterations that do not improve code design.

Consider these two commit examples that illustrate the variability in how developers describe their refactoring activities:

1. **Poorly explained commit**:
    - **repo name:** vert.x
    - **repo url:** https://github.com/eclipse-vertx/vert.x
    - **commit message:** *"Cleanup handshaking code"*
2. **Well explained message**:
    - **repo name:** react-native
    - **repo url:** https://github.com/facebook/react-native
    - **commit message:** *"Refactor: Introduce methods to show/hide DevLoadingView in DevSupportManagerBase*

      Summary:

      Rationale: Throughout DevSupportManagerBase, we show/hide the DevLoadingView and simultaneously write to the ‘mDevLoadingViewVisible‘ boolean. This diff pulls all those boolean writes into methods, so that subclasses of DevSupportManagerBase can show/hide the DevLoadingView without accessing the boolean directly.

The second commit provides a clear, well-documented rationale for the refactoring, which could help the classifier understand and categorize the refactoring intent accurately, thus improving prediction outcomes. The first commit is a more common example of what we found, not providing enough information to classify the refactoring into a specific type. We have understood from these trials that if developers could leave informative commit messages in more cases, they would be able to improve classifier performance in specific scenarios.

In future work, we aim to deepen the analysis by exploring more granular sub-categories of refactoring based on commit messages. This could involve developing a more sophisticated categorization framework that can distinguish between different types of refactorings beyond the basic intents. By re-evaluating the model with these refined categories, we can assess whether specific types of refactorings, such as those aimed at improving readability or reducing complexity, are better suited to certain contexts or codebases.

By integrating the intent reflected in commit messages, the classifier's accuracy in identifying valid refactoring opportunities is expected to improve, leading to more precise recommendations for developers and, ultimately, to higher quality software maintenance practices.

While exploring the impact of developers' intent on identifying extract method refactoring opportunities, we encountered a notable limitation in the quality of commit messages. Our analysis revealed that commit messages are frequently not descriptive enough to conclusively determine the intent behind the changes. Through keyword analysis of these commit

messages, we revealed insights into how developers might not explicitly recognize or label specific changes as refactoring. This lack of clarity could potentially lead classifiers to misidentify refactoring opportunities. Despite these challenges, our findings suggest that a more robust dataset, characterized by guaranteed descriptive commit messages, could enhance the effectiveness of our proposed methodology. By using keyword analysis to categorize refactorings based on developer intent, we anticipate an improvement in the classifier's ability to accurately predict refactoring opportunities. The promise shown by this approach warrants further investigation, particularly with datasets that ensure the richness and descriptiveness of commit messages.

Chapter 7

Threats to Validity

Construct validity in this study refers to the accuracy with which the tools and measurements, namely RefactoringMiner and PyDriller, reflect the properties they are intended to measure. One of the biggest threats to the validity of this study is the quality of the positive and negative samples extracted using these tools. In order to mitigate this threat, a random subset of the identified negative samples was selected and manually evaluated to ensure they were of good quality.

The approach presented in this paper uses a dataset that is approximately 5% the size of the dataset used in the state-of-the-art approach, which can also be considered a threat. This study design decision was made because of the large amount of computing resources required to run the full dataset. The other reason this smaller dataset was used was to explore the feasibility and effectiveness of this approach, as with a much larger dataset it may be less feasible. Repeating the experiments in this study with a larger dataset could

mitigate this threat.

Internal validity concerns the degree to which the results of this study can be attributed to the conditions set out in the experimental design rather than external factors. In this study, internal validity is potentially compromised by several factors, particularly relating to the use of the autoencoder and the large language model.

Firstly, the performance of the autoencoder is critical, as it is responsible for reducing the dimensionality of the input data without significant loss of information. There's a risk that the autoencoder may not capture essential features or might overfit to the training data. Such issues were evident from the variable performance outcomes observed when comparing encoded versus raw embeddings, which suggested that the dimensionality reduction might sometimes strip away useful information.

The use of LLMs to cluster refactoring instances encountered significant hurdles due to the size and complexity of the dataset. Computational constraints meant that the LLM could not process the entire dataset simultaneously, which likely led to incomplete learning and suboptimal clustering performance. This also raised concerns about the scalability of this approach and its dependence on the quality of encoded embeddings.

Experimental reproducibility is another internal validity concern, highlighted by difficulties in replicating Palit et al.'s previous studies' performance improvements with autoencoded embeddings [9]. This discrepancy could stem from variations in experimental setups or differences in parameter tuning, which underscores the need for precise documentation and consistency in ex-

perimental procedures.

The **external validity** of this study refers to the generalizability and repeatability of the produced results. A threat to this paper is that this approach is specific to extract method refactoring, and it is not generalizable to other forms of refactoring. It would take extensive reworking of this approach in order to make it apply to other forms of refactoring. An example of this threat would be a move method refactoring-this approach does not have the code to collect the refactored commit as the method would be moved to another class.

Chapter 8

Conclusion

This study has explored the feasibility and effectiveness of using large language models and developer commit messages to improve the identification of extract method refactoring opportunities. Through rigorous experimentation, we have identified both the potential and the limitations of these advanced machine learning techniques in the context of software refactoring.

The use of LLMs to cluster code blocks based on complex patterns and dependencies showed promise but faced significant challenges due to the large size and complexity of the dataset. Our findings indicate that while LLMs have the potential to detect nuanced patterns in code that might be indicative of refactoring needs, the practical application of these models is currently limited by computational constraints and the risk of overfitting, particularly when using autoencoded embeddings.

The integration of developer commit messages into the classification process aimed to enhance the precision of the refactoring identification by providing

context on the developer's intent. This approach proved beneficial for understanding the motivations behind refactoring activities, which could potentially improve model accuracy. However, the variability in how developers describe their refactoring actions poses challenges for consistently categorizing intent and applying it effectively in a predictive model.

Given these insights, future work could take several directions:

Enhancing Autoencoder Performance: Investigating different autoencoder architectures or training strategies to improve their ability to capture essential information without losing important details. Scalable LLM Applications: Addressing the computational challenges associated with applying LLMs to large datasets could involve research into more efficient model architectures or incremental clustering techniques that can handle data in manageable chunks without losing the context necessary for effective clustering. Refined Intent Analysis: Developing a more sophisticated framework for categorizing refactoring intents based on commit messages. This framework might include natural language processing techniques to extract and categorize intents more precisely, potentially using unsupervised learning to discover new categories of intent. Generalization to Other Refactorings: Expanding the current approach to include other types of refactorings, such as move method or inline method, could significantly increase the utility of the developed models. Each refactoring type may require adjustments or extensions to the existing methodology to address its unique characteristics. Integration with Development Environments: Integrating the refactoring identification models directly into development environments as plugins would allow developers to receive real-time

suggestions for refactoring opportunities, tailored to their current coding activities and specific project contexts.

By pursuing these avenues, future work can build on the foundation laid by this study to enhance the accuracy and applicability of machine learning models in the identification of software refactoring opportunities, ultimately contributing to more maintainable and high-quality software systems.

Chapter 9

Acknowledgement

I would like to express my sincere appreciation to my faculty advisor, Dr. Mohamed Wiem Mkaouer, and for his support and guidance throughout my research. His profound knowledge in code quality, refactoring, and artificial intelligence helped to guide me through this process, providing direction and inspiration. Dr. Mkaouer's constant encouragement and confidence in my abilities motivated me to extend the boundaries of my academic pursuits and strive for excellence in every facet of my work, allowing me to explore an emerging technology and develop a passion for it. Dr. Mkaouer has greatly enriched my educational experience, making my academic journey not only achievable but also deeply enriching.

Additionally, I extend my sincere thanks to Taha Draoui, whose collaboration was crucial in replicating and expanding upon the work of Palit et al.. Our joint efforts have been instrumental in exploring new dimensions of this research.

This thesis is not only a reflection of my own efforts but also a testament to Dr. Mkaouer's dedication to his students and his unwavering commitment to advancing academic inquiry. I am profoundly thankful for his mentorship and the opportunities I have been afforded under his guidance.

Ryan Devoe

Bibliography

[1] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[2] K. Beck J. Brant W. Opdyke M. Fowler, P. Becker and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

[3] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[4] A survey on software smells. *Journal of Systems and Software*, 138:158–173, 2018.

[5] Maurício Aniche, Erick Maziero, Rafael Durelli, and Vinicius Durelli. The effectiveness of supervised machine learning algorithms in predicting software refactoring, 2020.

[6] Chitti Babu Karakati and Sethukarasi Thirumaaran. Software code refactoring based on deep neural network-based fitness function. *Concurrency and Computation: Practice and Experience*, 35(4):e7531.

[7] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. Recommendation of move method refactoring using path-based representation of code, 2020.

[8] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code, 2018.

[9] Indranil Palit, Gautam Shetty, Hera Arif, and Tushar Sharma. Automatic refactoring candidate identification leveraging effective code representation. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 369–374, 2023.

[10] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software*, 171:110821, 2021.

[11] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[13] Taweh Beysolow II and SpringerLink (Online service). *Applied Natural Language Processing with Python: Implementing Machine Learning and Deep Learning Algorithms for Natural Language Processing*. Apress, Berkeley, CA, 1st 2018.;1; edition, 2018.

[14] Wajdi Aljedaani, Mohamed Wiem Mkaouer, Anthony Peruma, and Stephanie Ludi. Do the test smells assertion roulette and eager test impact students' troubleshooting and debugging capabilities? *arXiv preprint arXiv:2303.04234*, 2023.

[15] Christian D Newman, Michael J Decker, Reem Alsuhaibani, Anthony Peruma, Mohamed Mkaouer, Satyajit Mohapatra, Tejal Vishoi, Marcos Zampieri, Timothy Sheldon, and Emily Hill. An ensemble approach for annotating source code identifiers with part-of-speech tags. *IEEE Transactions on Software Engineering*, 2021.

[16] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1303–1313, 2021.

[17] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. Behind the scenes: On the relationship between developer experience and refactoring. *Journal of Software: Evolution and Process*, page e2395, 2021.

[18] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology*, 140:106675, 2021.

[19] Eman Abdullah AlOmar, Ben Christians, Mihal Busho, Ahmed Hamad AlKhalid, Ali Ouni, Christian Newman, and Mohamed Wiem Mkaouer. Satdbailiff-mining and tracking self-admitted technical debt. *Science of Computer Programming*, 213:102693, 2022.

[20] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. How do i refactor this? an empirical study on refactoring trends and topics in stack overflow. *Empirical Software Engineering*, 27(1):1–43, 2022.

[21] Eman Abdullah AlOmar, Jiaqian Liu, Kenneth Addo, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Zhe Yu. On the documentation of refactoring types. *Automated Software Engineering*, 29(1):1–40, 2022.

[22] Eman Abdullah Alomar, Tianjia Wang, Vaibhavi Raut, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. Refactoring for reuse: an empirical study. *Innovations in Systems and Software Engineering*, pages 1–31, 2022.

[23] Anthony Peruma, Emily Hu, Jiajun Chen, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Christian D Newman. Using grammar pat-

terns to interpret test method name evolution. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 335–346. IEEE, 2021.

[24] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Mining and managing big data refactoring for design improvement: Are we there yet? *Knowledge Management in the Development of Data-Intensive Systems*, pages 127–140, 2021.

[25] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif Ghallab, and Stephanie Ludi. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, pages 170–180, 2021.

[26] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. On the use of information retrieval to automate the detection of third-party java library migration at the method level. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 347–357. IEEE, 2019.

[27] Hussein Alrubaye, Mohamed Wiem Mkaouer, Igor Khokhlov, Leon Reznik, Ali Ouni, and Jason Mcgoff. Learning to recommend third-party library migration opportunities at the api level. *Applied Soft Computing*, 90:106140, 2020.

[28] Hussein Alrubaye, Stephanie Ludi, and Mohamed Wiem Mkaouer. Comparison of block-based and hybrid-based environments in transferring programming skills to text-based environments. *arXiv preprint arXiv:1906.03060*, 2019.

[29] Anthony Peruma, Mohamed Wiem Mkaouer, Michael John Decker, and Christian Donald Newman. Contextualizing rename decisions using refactorings and commit messages. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 74–85. IEEE, 2019.

[30] Christian D Newman, Mohamed Wiem Mkaouer, Michael L Collard, and Jonathan I Maletic. A study on developer perception of transformation languages for refactoring. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 34–41, 2018.

[31] Hussein Alrubaye and Mohamed Wiem Mkaouer. Automating the detection of third-party java library migration at the function level. In *CASCON*, pages 60–71, 2018.

[32] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Anthony Peruma. Variability in library evolution: An exploratory study on open-source java libraries. In *Software Engineering for Variability Intensive Systems*, pages 295–320. Auerbach Publications, 2019.

[33] Montassar Ben Messaoud, Ilyes Jenhani, Nermine Ben Jemaa, and Mohamed Wiem Mkaouer. A multi-label active learning approach for mobile

app user review classification. In *International Conference on Knowledge Science, Engineering and Management*, pages 805–816. Springer, 2019.

[34] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. Migrationminer: An automated detection tool of third-party java library migration at the method level. In *2019 IEEE international conference on software maintenance and evolution (ICSME)*, pages 414–417. IEEE, 2019.

[35] Deema Alshoaibi, Kevin Hannigan, Hiten Gupta, and Mohamed Wiem Mkaouer. Price: Detection of performance regression introducing code changes using static and dynamic metrics. In *International Symposium on Search Based Software Engineering*, pages 75–88. Springer, 2019.

[36] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[37] Licelot Marmolejos, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On the use of textual feature extraction techniques to support the automated detection of refactoring documentation. *Innovations in Systems and Software Engineering*, pages 1–17, 2021.

[38] Eman AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Can refactoring be self-affirmed? an exploratory study on how developers document

their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 51–58. IEEE, 2019.

[39] Alex Bogart, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Increasing the trust in refactoring through visualization. In *2020 IEEE/ACM 4th International Workshop on Refactoring (IWoR)*, 2020.

[40] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Marouane Kessentini. How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation. *Expert Systems with Applications*, 167:114176, 2021.

[41] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357. IEEE, 2021.

[42] Eman Abdullah AlOmar, Anthony Peruma, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. On the relationship between developer experience and refactoring: An exploratory study and preliminary results. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 342–349, 2020.

[43] Eman Abdullah AlOmar, Philip T Rodriguez, Jordan Bowman, Tianjia Wang, Benjamin Adepoju, Kevin Lopez, Christian Newman, Ali Ouni, and Mohamed Wiem Mkaouer. How do developers refactor code to improve code reusability? In *International Conference on Software and Software Reuse*, pages 261–276. Springer, 2020.

[44] Eman Abdullah AlOmar, Tianjia Wang, Raut Vaibhavi, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. Refactoring for reuse: An empirical study. *Innovations in Systems and Software Engineering*, pages 1–31, 2021.

[45] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. An exploratory study on the refactoring of unit test files in android applications. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 350–357, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the dis-

tribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 193–202, USA, 2019. IBM Corp.

[48] Sirine Gharbi, Mohamed Wiem Mkaouer, Ilyes Jenhani, and Montassar Ben Messaoud. On the classification of software change messages using multi-label active learning. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1760–1767, 2019.

[49] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–45, 2015.

[50] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 331–336, 2014.

[51] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using nsga-iii. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1263–1270, 2014.

[52] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering*, 21(6):2503–2545, 2016.

[53] Mohamed Wiem Mkaouer, Marouane Kessentini, Mel Ó Cinnéide, Shinpei Hayashi, and Kalyanmoy Deb. A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering*, 22(2):894–927, 2017.

[54] Rafi Almhana, Wiem Mkaouer, Marouane Kessentini, and Ali Ouni. Recommending relevant classes for bug reports using multi-objective search. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 286–295. IEEE, 2016.

[55] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, pages 193–202, 2019.

[56] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering*, 29(1):1–61, 2022.

[57] Wajdi Aljedaani, Mona Aljedaani, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Stephanie Ludi, and Yousef Bani Khalaf. I cannot see you—the perspectives of deaf students to online learning during covid-19 pandemic: Saudi arabia case study. *Education Sciences*, 11(11):712, 2021.

[58] Islem Saidani, Ali Ouni, and Wiem Mkaouer. Detecting skipped commits in continuous integration using multi-objective evolutionary search. *IEEE Transactions on Software Engineering*, 2021.

[59] Marwa Daaji, Ali Ouni, Mohamed Mohsen Gammoudi, Salah Bouktif, and Mohamed Wiem Mkaouer. Multi-criteria web services selection: Balancing the quality of design and quality of service. *ACM Transactions on Internet Technology (TOIT)*, 22(1):1–31, 2021.

[60] Nuri Almarimi, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. csdetector: an open source tool for community smells detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1560–1564, 2021.

[61] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Bf-detector: an automated tool for ci build failure detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1530–1534, 2021.

[62] Oumayma Hamdi, Ali Ouni, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. A longitudinal study of the impact of refactoring in android applications. *Information and Software Technology*, 140:106699, 2021.

[63] Oumayma Hamdi, Ali Ouni, Eman Abdullah AlOmar, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. An empirical study on the impact of refactoring on quality metrics in android applications. In *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 28–39. IEEE, 2021.

[64] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Fabio Palomba. On the impact of continuous integration on refactoring practice: An exploratory study on travistorrent. *Information and Software Technology*, 138:106618, 2021.

[65] Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud. Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Information and Software Technology*, 135:106566, 2021.

[66] Fan Fang, John Wu, Yanyan Li, Xin Ye, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. On the classification of bug reports to improve bug localization. *Soft Computing*, 25(11):7307–7323, 2021.

[67] Makram Soui, Mabrouka Chouchane, Narjes Bessghaier, Mohamed Wiem Mkaouer, and Marouane Kessentini. On the impact of aesthetic defects on the maintainability of mobile graphical user inter-

faces: An empirical study. *Information Systems Frontiers*, pages 1–18, 2021.

[68] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357. IEEE, 2021.

[69] Moataz Chouchen, Ali Ouni, Raula Gaikovina Kula, Dong Wang, Patanamon Thongtanunam, Mohamed Wiem Mkaouer, and Kenichi Matsumoto. Anti-patterns in modern code review: Symptoms and prevalence. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 531–535. IEEE, 2021.

[70] Xin Ye, Yongjie Zheng, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. Recommending pull request reviewers based on code changes. *Soft Computing*, 25(7):5619–5632, 2021.

[71] Hussein Alrubaye, Deema Alshoaibi, Eman Alomar, Mohamed Wiem Mkaouer, and Ali Ouni. How does library migration impact software quality and comprehension? an empirical study. In *International Conference on Software and Software Reuse*, pages 245–260. Springer, 2020.

[72] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Whoreview: A multi-objective search-based

approach for code reviewers recommendation in modern code review. *Applied Soft Computing*, 100:106908, 2021.

[73] Moataz Chouchen, Ali Ouni, and Mohamed Wiem Mkaouer. Androlib: Third-party software library recommendation for android applications. In *International Conference on Software and Software Reuse*, pages 208–225. Springer, 2020.

[74] Nuri Almarimi, Ali Ouni, Moataz Chouchen, Islem Saidani, and Mohamed Wiem Mkaouer. On the detection of community smells using genetic programming-based ensemble classifier chain. In *Proceedings of the 15th International Conference on Global Software Engineering*, pages 43–54, 2020.

[75] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software*, 171:110821, 2021.

[76] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Web service api anti-patterns detection as a multi-label learning problem. In *International Conference on Web Services*, pages 114–132. Springer, 2020.

[77] Bader Alkhazi, Andrew DiStasi, Wajdi Aljedaani, Hussein Alrubaye, Xin Ye, and Mohamed Wiem Mkaouer. Learning to rank developers for bug report assignment. *Applied Soft Computing*, 95:106667, 2020.

[78] Motaz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Recommending peer reviewers in modern

code review: a multi-objective search-based approach. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 307–308, 2020.

[79] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology*, 128:106392, 2020.

[80] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. tsdetect: an open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1650–1654, 2020.

[81] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. On the prediction of continuous integration build failures using search-based software engineering. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 313–314, 2020.

[82] Nuri Almarimi, Ali Ouni, and Mohamed Wiem Mkaouer. Learning to detect community smells in open source software projects. *Knowledge-Based Systems*, 204:106201, 2020.

[83] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Aymen Saied. Towards automated microservices extraction using muti-objective evolu-

tionary search. In *International Conference on Service-Oriented Computing*, pages 58–63. Springer, Cham, 2019.

[84] Nuri Almarimi, Ali Ouni, Salah Bouktif, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Mohamed Aymen Saied. Web service api recommendation for automated mashup creation using multi-objective evolutionary search. *Applied Soft Computing*, 85:105830, 2019.

[85] Makram Soui, Mabrouka Chouchane, Mohamed Wiem Mkaouer, Marouane Kessentini, and Khaled Ghedira. Assessing the quality of mobile graphical user interfaces using multi-objective optimization. *Soft Computing*, 24(10):7685–7714, 2020.

[86] Nasir Safdari, Hussein Alrubaye, Wajdi Aljedaani, Bladimir Baez Baez, Andrew DiStasi, and Mohamed Wiem Mkaouer. Learning to rank faulty source files for dependent bug reports. In *Big Data: Learning, Analytics, and Applications*, volume 10989, page 109890B. International Society for Optics and Photonics, 2019.

[87] Vahid Alizadeh, Marouane Kessentini, Mohamed Wiem Mkaouer, Mel Ocinneide, Ali Ouni, and Yuanfang Cai. An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering*, 46(9):932–961, 2018.

[88] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J Decker, and Christian D Newman. An empirical investigation of how and why devel-

opers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 26–33. ACM, 2018.

[89] Makram Soui, Mabrouka Chouchane, Ines Gasmi, and Mohamed Wiem Mkaouer. Plain: Plugin for predicting the usability of mobile user interface. In *VISIGRAPP (1: GRAPP)*, pages 127–136, 2017.

[90] Ian Shoenberger, Mohamed Wiem Mkaouer, and Marouane Kessentini. On the use of smelly examples to detect code smells in javascript. In *European Conference on the Applications of Evolutionary Computation*, pages 20–34. Springer, Cham, 2017.

[91] Mohamed Wiem Mkaouer. Interactive code smells detection: An initial investigation. In *International Symposium on Search Based Software Engineering*, pages 281–287. Springer, Cham, 2016.

[92] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, and Mel Ó Cinnéide. A robust multi-objective approach for software refactoring under uncertainty. In *International Symposium on Search Based Software Engineering*, pages 168–183. Springer, Cham, 2014.

[93] Mohamed Wiem Mkaouer and Marouane Kessentini. Model transformation using multiobjective optimization. In *Advances in Computers*, volume 92, pages 161–202. Elsevier, 2014.

[94] Mohamed W Mkaouer, Marouane Kessentini, Slim Bechikh, and Daniel R Tauritz. Preference-based multi-objective software modelling.

In *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pages 61–66. IEEE, 2013.

[95] Taryn Takebayashi, Anthony Peruma, Mohamed Wiem Mkaouer, and Christian D Newman. An exploratory study on the usage and readability of messages within assertion methods of test cases. *arXiv preprint arXiv:2303.00169*, 2023.

[96] Eman Abdullah AlOmar, Salma Abdullah AlOmar, and Mohamed Wiem Mkaouer. On the use of static analysis to engage students with software quality improvement: An experience with pmd. *arXiv preprint arXiv:2302.05554*, 2023.

[97] Wajdi Aljedaani, Mona Aljedaani, Mohamed Wiem Mkaouer, and Stephanie Ludi. Teachers perspectives on transition to online teaching deaf and hard-of-hearing students during the covid-19 pandemic: A case study. In *Proceedings of the 16th Innovations in Software Engineering Conference*, pages 1–10, 2023.

[98] Wajdi Aljedaani, Mohamed Wiem Mkaouer, Anthony Peruma, and Stephanie Ludi. Do the test smells assertion roulette and eager test impact students' troubleshooting and debugging capabilities? *arXiv preprint arXiv:2303.04234*, 2023.

[99] Deema Adeeb Al Shoaibi and Mohamed Wiem Mkaouer. Understanding software performance challenges an empirical study on stack overflow.

In *2023 International Conference on Code Quality (ICCQ)*, pages 1–15. IEEE, 2023.

[100] Wajdi Aljedaani, Mohammed Alkahtani, Stephanie Ludi, Mohamed Wiem Mkaouer, Marcelo M Eler, Marouane Kessentini, and Ali Ouni. The state of accessibility in blackboard: Survey and user reviews case study. In *20th International Web for All Conference*, pages 84–95, 2023.

[101] Waleed Alhindi, Abdulrahman Aleid, Ilyes Jenhani, and Mohamed Wiem Mkaouer. Issue-labeler: an albert-based jira plugin for issue classification. In *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 40–43. IEEE, 2023.

[102] Marwa Daaji, Ali Ouni, Mohamed Mohsen Gammoudi, Salah Bouktif, and Mohamed Wiem Mkaouer. Bpel process defects prediction using multi-objective evolutionary search. *Journal of Systems and Software*, page 111767, 2023.

[103] Ali Ouni, Islem Saidani, Eman Alomar, and Mohamed Wiem Mkaouer. An empirical study on continuous integration trends, topics and challenges in stack overflow. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 141–151, 2023.

[104] Moataz Chouchen, Ali Ouni, Jefferson Olongo, and Mohamed Wiem Mkaouer. Learning to predict code review completion time in modern

code review. *Empirical Software Engineering*, 28(4):82, 2023.

[105] Ali Ouni, Eman Abdullah AlOmar, Oumayma Hamdi, Mel Ó Cinnéide, Mohamed Wiem Mkaouer, and Mohamed Aymen Saied. On the impact of single and co-occurrent refactorings on quality attributes in android applications. *Journal of Systems and Software*, 205:111817, 2023.

[106] Wajdi Aljedaani, Mohammed Alkahtani, Stephanie Ludi, Mohamed Wiem Mkaouer, Marcelo M Eler, Marouane Kessentini, and Ali Ouni. The state of accessibility in blackboard: Survey and user reviews case study. In *Proceedings of the 20th International Web for All Conference*, pages 84–95, 2023.

[107] Wajdi Aljedaani, Rrezarta Krasniqi, Sanaa Aljedaani, Mohamed Wiem Mkaouer, Stephanie Ludi, and Khaled Al-Raddah. If online learning works for you, what about deaf students? emerging challenges of online learning for deaf and hearing-impaired students during covid-19: a literature review. *Universal access in the information society*, 22(3):1027–1046, 2023.

[108] Deema Alshoaibi, Ikram Chaabane, Kevin Hannigan, Ali Ouni, and Mohamed Wiem Mkaouer. On the detection of performance regression introducing code changes: Experience from the git project. In *2022 IEEE 29th Annual Software Technology Conference (STC)*, pages 206–217. IEEE, 2022.

[109] Wajdi Aljedaani, Furqan Rustam, Mohamed Wiem Mkaouer, Abdullatif Ghallab, Vaibhav Rupapara, Patrick Bernard Washington, Ernesto Lee, and Imran Ashraf. Sentiment analysis on twitter data integrating textblob and deep learning models: The case of us airline industry. *Knowledge-Based Systems*, 255:109780, 2022.

[110] Wajdi Aljedaani, Ibrahem Abuhaimed, Furqan Rustam, Mohamed Wiem Mkaouer, Ali Ouni, and Ilyes Jenhani. Automatically detecting and understanding the perception of covid-19 vaccination: a middle east case study. *Social Network Analysis and Mining*, 12(1):128, 2022.

[111] Eman Abdullah AlOmar, Anton Ivanov, Zarina Kurbatova, Yaroslav Golubev, Mohamed Wiem Mkaouer, Ali Ouni, Timofey Bryksin, Le Nguyen, Amit Kini, and Aditya Thakur. Just-in-time code duplicates extraction. *Information and Software Technology*, 158:107169, 2023.

[112] Deema ALShoaibi, Hiten Gupta, Max Mendelson, Ilyes Jenhani, Ali Ben Mrad, and Mohamed Wiem Mkaouer. Learning to characterize performance regression introducing code changes. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 1590–1597, 2022.

[113] Wajdi Aljedaani, Mohamed Wiem Mkaouer, Stephanie Ludi, Ali Ouni, and Ilyes Jenhani. On the identification of accessibility bug reports in open source systems. In *Proceedings of the 19th international web for all conference*, pages 1–11, 2022.

[114] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering*, 29(1):21, 2022.

[115] Deema Alshoaibi, Mohamed Wiem Mkaouer, Ali Ouni, AbdulMutalib Wahaishi, Travis Desell, and Makram Soui. Search-based detection of code changes introducing performance regression. *Swarm and Evolutionary Computation*, 73:101101, 2022.

[116] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. An exploratory study on refactoring documentation in issues handling. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 107–111, 2022.

[117] Anthony Peruma, Eman Abdullah AlOmar, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. Refactoring debt: myth or reality? an exploratory study on the relationship between technical debt and refactoring. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 127–131, 2022.

[118] Eman Abdullah AlOmar, Salma Abdullah AlOmar, and Mohamed Wiem Mkaouer. On the use of static analysis to engage students with software quality improvement: An experience with pmd. *arXiv preprint arXiv:2302.05554*, 2023.

[119] Eman Abdullah AlOmar, Anton Ivanov, Zarina Kurbatova, Yaroslav Golubev, Mohamed Wiem Mkaouer, Ali Ouni, Timofey Bryksin, Le Nguyen, Amit Kini, and Aditya Thakur. Anticopypaster: extracting code duplicates as soon as they are introduced in the ide. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4, 2022.

[120] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Automating source code refactoring in the classroom. *arXiv preprint arXiv:2311.10753*, 2023.

[121] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. How is software reuse discussed in stack overflow? *arXiv preprint arXiv:2311.00256*, 2023.

[122] Eman Abdullah AlOmar, Anton Ivanov, Zarina Kurbatova, Yaroslav Golubev, Mohamed Wiem Mkaouer, Ali Ouni, Timofey Bryksin, Le Nguyen, Amit Kini, and Aditya Thakur. Just-in-time code duplicates extraction. *Information and Software Technology*, 158:107169, 2023.

[123] Jehad Al Dallal. Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and Software Technology*, 58:231–249, 2015.

[124] Istvan Czibula and Gabriela Czibula. Hierarchical clustering based automatic refactorings detection. *WSEAS Transactions on Electronics*, 5:291–302, 01 2008.

[125] Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *The Journal of systems and software*, 84(3):397–414, 2011.

[126] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *The Journal of systems and software*, 84(10):1757–1782, 2011.

[127] David van der Leij, Jasper Binda, Robbert van Dalen, Pieter Vallen, Yaping Luo, and Maurício Aniche. Data-driven extract method recommendations: A study at ing. ESEC/FSE 2021, page 1337–1347, New York, NY, USA, 2021. Association for Computing Machinery.

[128] Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–621, 2018.

[129] Lov Kumar, Shashank Mouli Satapathy, and Lalita Bhanu Murthy. Method level refactoring prediction on five open source java projects using machine learning techniques. In *Proceedings of the 12th Innovations in Software Engineering Conference (Formerly Known as India Software Engineering Conference)*, ISEC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[130] Suresh S. Priyadarshni, Abdulah A. Eman, Mohamed W. Mkaouer, Ali Ouni, and Christian D. Newman. Comparing commit messages and source code metrics for the prediction refactoring activities. *Algorithms*, 14(10):289, 2021.

[131] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow, 2021.

[132] Cheng-Yuan Liou, Wei-Chen Cheng, Jiun-Wei Liou, and Daw-Ran Liou. Autoencoder for words. *Neurocomputing*, 139:84–96, 2014.

[133] S. Delphine Immaculate, M. Farida Begam, and M. Floramary. Software bug prediction using supervised machine learning algorithms. In *2019 International Conference on Data Science and Communication (IconDSC)*, pages 1–7, 2019.

[134] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software*, 176:110936, 2021.