Automatic Refactoring Candidate Identification
Leveraging Effective Code Representation

Indranil Palit, Gautam Shetty, Hera Arif, Tushar Sharma
Dalhousie University, Canada
{indranil.palit, gautam.shetty, hr833194, tushar}@dal.ca

Abstract—The use of machine learning to automate the detection of refactoring candidates is a rapidly evolving research area. The majority of work in this direction uses source code metrics and commit messages to predict refactoring candidates and do not exploit the rich semantics of source code. This paper proposes a new approach for extract method refactoring candidates identification. First, we propose a novel mechanism to identify negative samples for the refactoring candidate identification task. We then employ a self-supervised autoencoder to acquire a compact representation of source code generated by a pre-trained large language model. Subsequently, we train a binary classifier to predict extract method refactoring candidates. Experiments show that our new approach outperforms the state of the art by 30% in terms of F1 score. The proposed work has implications for researchers and practitioners. Software developers may use the proposed automated approach to predict refactoring candidates better. This study will facilitate the development of improved refactoring candidate identification methods that the researchers in the field could use and extend.

Index Terms—extract method refactoring, deep learning, code representation

I. INTRODUCTION

Refactoring is a process and a set of techniques to improve source code’s structure and quality without impacting the behavior and functionality of the software, thus making the software more maintainable [1], [2]. It is a widely used technique among software developers as it improves code readability, testability, flexibility, and adaptability to introduce changes to meet new requirements [3].

One of the critical questions that a developer answers during software development is whether a source code entity (e.g., a method or a class) requires refactoring and if yes, what is the most appropriate refactoring in the context. Practitioners decide what to refactor according to their intuition and experience. They can also use automated tools to calculate code quality metrics and code smells [4]. However, metrics and smells focus on the problem aspect and provide little help to decide whether and what refactoring technique must be applied to remove the smell or improve the metrics.

To overcome this challenge, many studies propose techniques to predict refactoring candidates by analyzing source code properties [5], [6], [7]. However, existing efforts in this direction suffer from several deficiencies. Currently, the state-of-the-art research in detecting refactoring candidates, such as Aniche et al. [5], follows a metric-based approach, i.e., it collects code quality and process metrics, and trains a machine learning model using the collected metrics as features. Similarly, Xu et al. [8], use variable accesses in addition to code quality metrics. These approaches fail to capture the hidden contextual and syntactical characteristics of code that might contribute to better refactoring candidate identification. To overcome the issue, researchers have used code embeddings [7], [6] extracted from Code2Vec [9]. However, existing studies in this domain do not capture rich contextual and syntactical characteristics of code [10]; such information could significantly improve the performance of the identified refactoring candidates. Secondly, existing studies lack an appropriate mechanism to define and identify negative samples for their dataset in this context. A code snippet is considered a negative sample if it is not a candidate for the specific refactoring. Typically, studies use tools such as RefactoringMiner [11], [12] to identify positive code samples. However, to identify negative samples, researchers define unsound rule-based heuristics resulting in a low-quality noisy dataset [13]. Finally, most of the previous research in this field fails to consider the real-world ratio of positive and negative samples while evaluating the predictive models. Ignoring this guideline results in a model that works well in an experimental study but performs poorly when deployed in a real-world context [14], [15].

In this paper, we address the aforementioned deficiencies. We present an automated Deep Learning (DL)-based technique to identify candidates for extract method refactoring. The extract method refactoring isolates a code block from a larger method and generates a new method based on the extracted code snippet [2]. We kept our focus on extract method because it is one of the most commonly used refactoring [16]. We create our dataset from open-source Java repositories and prepare an effective code representation capturing syntactic and semantic properties of methods by combining GraphCodeBERT [17] and Autoencoder [18]. The representation is then used to identify extract method refactoring candidates.

Contributions of the study: We propose a novel mechanism to properly identify positive and negative samples for extract method refactoring. The mechanism helps us create a dataset containing 55,430 positive and negative samples that serves as a benchmark for automated refactoring candidate identification approaches properly. To study the effectiveness of the method representation generated using GraphCodeBERT in a binary classification task, we propose an Autoencoder-based approach to identify latent features.

Replication package: We made our code [19] publicly
available with our training data [20] for easier replication and use.

II. Approach

This section describes the experimental approach followed to investigate the potential of Large Language Models (LLMs) in determining the suitability of a method for extract method refactoring.

A. Overview

The study aims to develop a DL-based extract method refactoring candidate identification technique that addresses the deficiencies in the existing studies. Figure 1 presents an overview of our approach. We first pick a set of repositories to prepare our dataset. We use existing tools RefactoringMiner [11] and PyDriller [21], to segregate methods into positive and negative samples. Our approach utilizes GraphCodeBERT to generate embeddings for each sample. We employ a DL model based on Autoencoder [18] that is used for feature extraction and dimensionality reduction. We utilize the encoder component of the trained Autoencoder to generate a lower-dimensional latent space representation from the initially high-dimensional embedding input. The representation obtained from the bottleneck layer of Autoencoder is then used as a feature vector to train a Random Forest (RF) classifier on the extract method identification task. We formulate the following research questions.

RQ1 How does our proposed approach perform compared to the state-of-the-art?

By answering this research question, we intend to evaluate and validate the performance of the proposed approach as compared to the state of the art.

RQ2 How effectively does the autoencoder extracts features for the classification task?

In this research question, we aim to evaluate the effectiveness of the employed autoencoder-based model by extracting the learned features and using them for the classification task.

B. Dataset preparation

We utilized a subset of repositories (5%) from the 11,149 open-source Java repositories used by Aniche et al. [5], as working with the entire set required extensive computing infrastructure. This initial selection yielded 558 repositories, from which we excluded repositories that were no longer available on GitHub or lacked any instances of extract method refactoring throughout their history. After filtering, we obtained 410 repositories with at least one extract method refactoring performed. To leverage a trained autoencoder and using the trained encoder of the autoencoder for the classification task without data leakage, we divided this dataset into two parts: one for autoencoder pipeline with 208 repositories and the remaining 202 for the classification pipeline.

As shown in step 1 of Figure 1, we use RefactoringMiner, a state-of-the-art refactoring detection tool [11, 12] to prepare our dataset. This tool reports performed refactorings, if any, in each commit within a Java repository’s history. It provides essential metadata such as the code component involved (e.g., method start line and end line in the case of extract method refactoring) and the associated commit hash. Leveraging this information, we utilize PyDriller [21] to iterate through the identified commits and extract source code of the involved methods.

We identify positive samples where extract method refactoring has been applied following the mechanism described above. Identifying negative samples for extract method refactoring is a challenging task. Merely excluding methods not reported by RefactoringMiner is insufficient since the absence of refactoring does not guarantee a method is not a refactoring candidate. Previous work have proposed heuristics-based approaches to address this challenge. For instance, Aniche et al. [5] used a criterion based on the method’s modification history, while Yamanaka et al. [22] selected code portions that differ from actual extractions. However, these heuristics may introduce noise and create sub-optimal datasets by misclassifying potential extract method candidates.

We propose a new mechanism to identify negative samples for the study. A method is considered a negative sample in commit $C_n$ if it underwent extract method refactoring in its parent commit $C_{n-1}$. The rationale behind this idea is that it is highly unlikely that a method that underwent extract method refactoring will again go through the same refactoring. This reduces the risk of false negative detection and ensures a high quality dataset. The aforementioned approach of identifying positive and negative samples, resulted in 27,634 and 27,796 samples for training and evaluating the Autoencoder, and binary classifier respectively.

C. Data representation

In step 2, we use GraphCodeBERT to capture both syntactic and semantic information of code, providing a comprehensive representation of code snippets by using graph-guided masked attention function to incorporate the code structure. The initial step in processing the input code through the GraphCodeBERT model involves tokenization and encoding. To accomplish this, we utilize the pre-trained GraphCodeBERT tokenizer. To ensure the token sequence adheres to the model’s maximum length of 512, we truncate it if it surpasses this limit. Subsequently, we perform batch encoding on the token sequence, generating input_ids which represent the tokens numerically for the model.

To extract the embeddings, we pass this encoded input to GraphCodeBERT. During the forward propagation of the input, each of the 12 hidden layers of the model generates individual token embeddings based on the surrounding context. To get the condensed representation of the sequence of tokens, we use mean pooling. We conducted a pilot study and we found that this approach performs better than taking the embedding of the [SEP] token alone. This results in a single embedding vector of size 768 for each of the input sample. We consider this as our feature vector for the classification task.
D. Model training and classification

1) Autoencoder: We use the generated embeddings from GraphCodeBERT as the input for training the autoencoder model (step 3). The architecture of the autoencoder that we trained consists of an encoder with three fully connected linear layers and ReLU activation to learn the hidden representation that reduces the input dimension to a bottleneck layer of size 128. The decoder reverses this process to reconstruct the original input of size 768. The autoencoder model is trained on 70%. The rest 30% is used to validate the model. We calculate the reconstruction loss using Mean Squared Error (MSE) loss.

2) Binary classifier: After training the autoencoder, in step 4, we take the encoder part of the trained model and use it as our feature extractor for the binary classification dataset. We train two classifiers—a traditional machine learning model Random Forest and a DL-based feed forward neural network, and compare their performance. We chose to use Random Forest due to its ensemble learning method for classification and its ability to learn the non-linear relationship between the features, Random Forest has shown to perform very well in different software engineering tasks [15], [23].

To train our models, we first split the 27,796 samples into train, validation, and test sets in 70:10:20 ratio using stratified sampling. We use GridSearchCV to select the optimal hyper-parameters for Random Forest. The optimal set of hyperparameter values along with their search space is reported in Table I. The neural network classifier consists of two fully connected layers with ReLU activation and a final sigmoid activation layer.

3) Evaluation: To evaluate our models, we calculate the accuracy, precision, recall, and F1 score.

Initially, the test split of our classification dataset, contains positive and negative samples in equal proportion. However, it has been argued [14], [15] that a test set not representative of the real-world may show good performance while experimentation but do poorly when deployed in a real-world scenario. To address this issue, we identify the ratio of positive and negative samples in the following manner. First, we sample 20 repositories from our dataset randomly. For each of the selected repositories, we identify the commits in which extract method refactoring has been applied using RefactoringMiner along with the count of such methods (posCount). Using PyDriller, we identify the count of total methods present in the source code for that commit (totalCount). Then we compute the ratio posCount/totalCount and take the mean across all identified commits to find a real-world ratio of extract method refactoring candidates. We modify the test set to represent the computed ratio (85:15) and then perform the evaluation.

III. EXPERIMENTAL RESULTS

RQ1: How does our proposed approach perform compared to the state-of-the-art?

In this research question, we compare our two approaches M1 (i.e., neural network-based classification) and M2 (i.e., Random Forest-based classification), where both the models utilize GraphCodeBERT and Autoencoder to generate latent representation. We compare the results from our models against state-of-the-art approach from Aniche et al. [5]. Though the baseline study compare many machine learning techniques, we chose to compare our models with only their Random Forest model because it reported the best results in that study. All of the models we tested using the same test split.

Table II presents results of our experiments. From the results it is evident that our M2 model outperforms the baseline as well as the M1 model. We observe that both M1 and M2 outperform the baseline. Specifically, we see that M2...
outperform the Random Forest used by Aniche et al. by nearly 50% in terms of precision. At the same time, our model M2 exhibits a good recall rate of 0.87. Consequently, we see that our model performs significantly better than the considered baseline model by approximately 30% in terms of F1 score.

RQ1 Summary: Our results show that our Random Forest-based model outperforms the baseline model significantly (by 30%, in terms of F1 score). The results indicate that our code representation is successfully capturing syntactic and semantic characteristics of code necessary to identify extract method refactoring candidates.

RQ2 Summary: Autoencoder-encoded code representations from GraphCodeBERT significantly improve classification performance compared to GraphCodeBERT representations alone.

RQ2: How effectively does the autoencoder extracts features for the classification task?

We train an autoencoder model and use the trained encoder part of it as a feature extractor. We do so to reduce the vectors’ dimensionality and extract relevant features from the embeddings generated from GraphCodeBERT.

To measure the performance and usefulness of the trained autoencoder model as a feature extractor, we first analyze the t-SNE plots of the embeddings generated by GraphCodeBERT and those generated by the combination of GraphCodeBERT and Autoencoder. We do so to study the class separability. The distinguishability of classes in the t-SNE space is assessed through a clear separation and quantified by calculating the Euclidean distance between the centroids of each class. Figure 3 shows a reasonably clear bifurcation between the classes with an euclidean distance of 0.367 as compared to 0.122 for Figure 2. We can infer that a classification model trained on the encoded embedding will perform better than the one trained on embeddings alone.

To further investigate the effectiveness of these representations, we perform an ablation study where we train classifier with and without the autoencoder representation of the embeddings. We report the performance results in Table III. The Random Forest classification model, when trained with the encoded representation of the embedding provided by Autoencoder, outperforms the one trained with only the GraphCodeBERT embedding vector as features. These findings support our claim that the autoencoder extracts features and reduces dimensionality effectively in the context of refactoring candidate identification.

<table>
<thead>
<tr>
<th>Models</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1 (GraphCodeBERT + Autoencoder + Neural Network)</td>
<td>0.57</td>
<td>0.71</td>
<td>0.57</td>
<td>0.63</td>
</tr>
<tr>
<td>M2 (GraphCodeBERT + Autoencoder + Random forest)</td>
<td>0.87</td>
<td>0.90</td>
<td>0.87</td>
<td>0.88</td>
</tr>
<tr>
<td>Baseline (with random forest)</td>
<td>0.84</td>
<td>0.44</td>
<td>0.87</td>
<td>0.58</td>
</tr>
</tbody>
</table>

IV. RELATED WORK

A. Refactoring candidate identification using traditional techniques

Software developers can manually decide what to refactor according to their intuition and past experiences [26]. Often, they use automated tools to calculate code quality metrics and code smells to identify refactoring candidates [3], [26]. Another method to identify refactoring candidates is to define a set of preconditions or compliance rules. If a code did not follow these rules, it was considered a candidate for refactoring. Studies by Bois et al. [27] and Kataoka et al. [28] used such compliance rules.

Another technique considers creating clustering algorithms to identify if code needs refactoring. Czibula et al. [29] and Serban et al. [30], created clusters based on the distance between methods and attributes within and outside of classes to identify numerous refactoring candidates. Similarly, Bavota et al. [31] suggest a graph-based approach that uses weighted graphs instead of abstract syntax trees to identify methods that can be extracted. Finally, Tsantalis et al. [32] used code slices to identify extract method candidates.

B. Detecting refactoring with Machine Learning techniques

Many studies have explored ways to identify refactoring candidates automatically using machine learning. Typically, such studies use source code metrics or commit messages to train a model. For example, Aniche et al. [5] predict 20 kinds of refactorings at the method, class, or variable level. They
use a large number of code, process, and ownership metrics to train six supervised machine learning algorithms. The study reports that Random Forest model performs the best among the compared models. Gerling [33] conducted an empirical study as an extension of Aniche et al.’s study. They focused on improving the data collection process in Aniche et al.’s study to create a high quality large-scale refactoring dataset.

Similarly, Van Der Leij et al. [24] analyze five machine learning models to predict extract method refactoring and compare the results with industry experts. They collect 61 code metrics and analyze Random Forest, Decision Tree, Logistic Regression, Linear SVM, and Gaussian Naive Bayes algorithms. They found Random Forest as the best performing model. Kumar et al. [34] perform a study to predict method-level refactoring and analyze 10 machine learning classifiers.

Sagar et al. [35] considers the problem of refactoring candidate prediction as a multi-class classification problem. Their study uses source code quality metrics and commit messages as features to predict six method-level refactorings. They compare two machine learning models: a text-based model that analyzes keywords from commit messages and a source code-based model that analyzes 64 code quality metrics. Kurbatova et al. [7] use code embeddings generated from Code2Vec [9] to train their machine learning model to predict the move method refactoring.

V. THREATS TO VALIDITY

Construct validity: Construct validity measures the degree to which tools and metrics actually measure the properties that they are supposed to measure. It concerns the appropriateness of observations and inferences based on measurements taken during the study. The quality of positive and negative samples extracted from RefactoringMiner and PyDriller poses a threat to validity. Though RefactoringMiner and PyDriller are state-of-the-art tools that are widely used and considered accurate, to mitigate the threat, we randomly pick up a subset of the identified negative samples and manually evaluated the quality of the samples.

Our approach uses a significantly smaller dataset (approximately 5%) compared to the dataset used in the state-of-the-art approach. Though the chosen repositories are representative, it can be considered a threat to validity. We chose the smaller dataset because of the computing resources required for the full-size dataset. Additionally, in this study, we aimed to explore the feasibility and effectiveness of the proposed approach. In the future, we aim to repeat the experiment with a larger dataset.

External validity: External validity concerns the generalizability and repeatability of the produced results. One of the threats to validity in this paper is that the approach proposed is exclusive to extract method refactoring. Using our approach for another kind of refactoring is challenging and requires extensive reworking of the approach used. For example, move method refactoring moves a method to an appropriate class [2]. We cannot apply the same approach that we adopted to create extract method dataset for move method refactoring because we will not have the code to collect in the refactored commit since the method would have moved to another class. A similar challenge is expected when considering other refactoring types, such as pull up method and push down method. In the future, we would like to address this challenge and propose an effective and generic dataset-creation approach for different refactoring types.

VI. CONCLUSION AND FUTURE WORK

To summarize, in this paper, we presented a technique for identifying extract method refactoring candidates using source code representation generated using GraphCodeBERT. We proposed a new technique for preparing an effective dataset for the refactoring candidate identification problem. Our experimental results demonstrated that our approach, that use source code embeddings, outperforms state-of-the-art machine learning-based technique trained on source code metrics.

In the future, we would like to extend the scope of our approach in terms of other kinds of refactorings, programming languages, and industry-based codebases. We also aim to increase the dataset size for extensive evaluation. In addition, our future studies will explore diverse code embeddings techniques to measure their effectiveness in this context.

REFERENCES


