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 suffers 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 ❶ 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 ❷, 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.

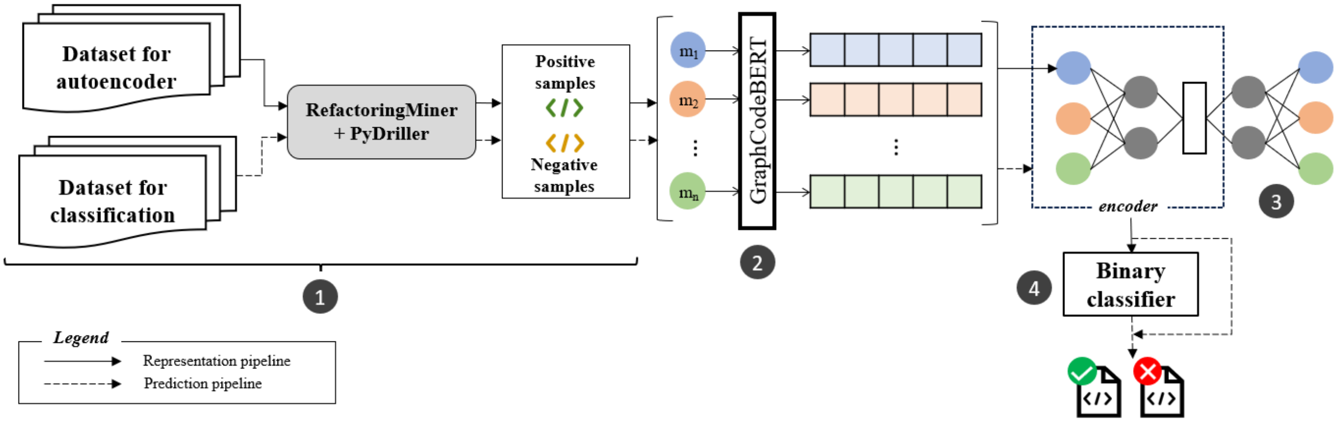

Fig. 1. Overview of the proposed approach

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] including refactoring identification [5], [24].

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 hyper-parameter 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 $\frac{posCount}{totalCount}$ 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.

TABLE I
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

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`

TABLE II
EXPERIMENTAL RESULTS FOR RQ1

Models	Accuracy	Precision	Recall	F1-score
M1 (GraphCodeBERT + Autoencoder + Neural Network)	0.57	0.71	0.57	0.63
M2 (GraphCodeBERT + Autoencoder + Random forest)	**0.87**	**0.90**	**0.87**	**0.88**
Baseline (with random forest)	0.84	0.44	**0.87**	0.58

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: *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*.

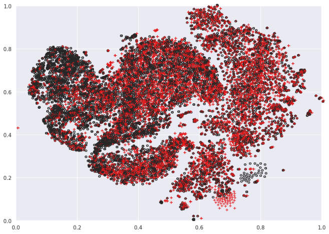

Fig. 2. Class separation with embeddings (from *GraphCodeBERT*)

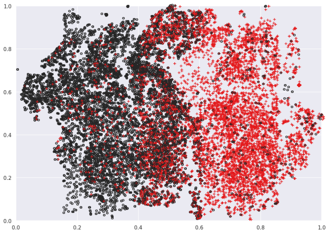

Fig. 3. Class separation with encoded embeddings (from Autoencoder)

To measure the performance and usefulness of the trained autoencoder model as a feature extractor, we first analyze the `t-SNE` [25] 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.

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

TABLE III
ENCODED EMBEDDING PERFORMANCE

Models	Accuracy	Precision	Recall	F1-score
Embeddings	0.57	0.71	0.57	0.63
Encoded embeddings	**0.87**	**0.90**	**0.87**	**0.88**

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 analyses keywords from commit messages and a source code-based model that analyses 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 picked 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

[1] W. F. Opdyke, "Refactoring: A program restructuring aid in designing object-oriented application frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[2] M. Fowler, P. Becker, K. Beck, J. Brant, W. Opdyke, 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*, vol. 30, no. 2, pp. 126–139, 2004.

[4] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158 – 173, 2018.

[5] M. F. Aniche, E. G. Maziero, R. S. Durelli, and V. H. S. Durelli, "The effectiveness of supervised machine learning algorithms in predicting software refactoring," *IEEE Transactions on Software Engineering*, vol. 48, pp. 1432–1450, 2020.

[6] C. B. Karakati and S. Thirumaaran, "Software code refactoring based on deep neural network-based fitness function," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 4, p. e7531.

[7] Z. Kurbatova, I. Veselov, Y. Golubev, and T. Bryksin, "Recommendation of move method refactoring using path-based representation of code," *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020.

[8] S. Xu, A. Sivaraman, S.-C. Khoo, and J. Xu, "Gems: An extract method refactoring recommender," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017, pp. 24–34.

[9] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019.

[10] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?" in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1332–1336.

[11] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 483–494.

[12] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022.

[13] A. Trautsch, J. Erbel, S. Herbold, and J. Grabowski, "What really changes when developers intend to improve their source code: a commit-level study of static metric value and static analysis warning changes," *Empirical Software Engineering*, vol. 28, no. 2, p. 30, 2023.

[14] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, “Code smell detection by deep direct-learning and transfer-learning,” *Journal of Systems and Software*, vol. 176, p. 110936, 2021.

[15] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. 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)*, 2018, pp. 612–621.

[16] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[17] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, “Graphcodebert: Pre-training code representations with data flow,” *CoRR*, vol. abs/2009.08366, 2020.

[18] C.-Y. Liou, W.-C. Cheng, J.-W. Liou, and D.-R. Liou, “Autoencoder for words,” *Neurocomputing*, vol. 139, pp. 84–96, 2014.

[19] I. Palit, G. Shetty, H. Arif, and T. Sharma, “Extract method identification.” [Online]. Available: https://github.com/SMART-Dal/extract-method-identification

[20] I. Palit, G. Shetty, H. Arif, and T. Sharma, “Dataset for Automatic Refactoring Candidate Identification Leveraging Effective Code Representation,” Jul. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8122619

[21] D. Spadini, M. Aniche, and A. Bacchelli, “PyDriller: Python framework for mining software repositories,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. ACM Press, 2018, pp. 908–911.

[22] J. Yamanaka, Y. Hayase, and T. Amagasa, “Recommending extract method refactoring based on confidence of predicted method name,” *ArXiv*, vol. abs/2108.11011, 2021.

[23] 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)*, 2019, pp. 1–7.

[24] D. van der Leij, J. Binda, R. van Dalen, P. Vallen, Y. Luo, and M. Aniche, “Data-driven extract method recommendations: A study at ing.” Association for Computing Machinery, 2021, p. 1337–1347.

[25] L. Van der Maaten and G. Hinton, “Visualizing data using t-sne.” *Journal of machine learning research*, vol. 9, no. 11, 2008.

[26] J. Al Dallal, “Identifying refactoring opportunities in object-oriented code: A systematic literature review,” *Information and software Technology*, vol. 58, pp. 231–249, 2015.

[27] B. Du Bois, S. Demeyer, and J. Verelst, “Refactoring-improving coupling and cohesion of existing code,” in *11th working conference on reverse engineering*, 2004, pp. 144–151.

[28] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin, “Automated support for program refactoring using invariants,” in *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, 2001, pp. 736–743.

[29] I. G. Czibula and G. Czibula, “Hierarchical clustering based automatic refactorings detection,” *WSEAS Transactions on Electronics*, vol. 5, no. 7, pp. 291–302, 2008.

[30] G. Serban and I.-G. Czibula, “Restructuring software systems using clustering,” in *2007 22nd international symposium on computer and information sciences*, 2007, pp. 1–6.

[31] G. Bavota, A. De Lucia, and R. Oliveto, “Identifying extract class refactoring opportunities using structural and semantic cohesion measures,” *Journal of Systems and Software*, vol. 84, no. 3, pp. 397–414, 2011.

[32] N. Tsantalis and A. Chatzigeorgiou, “Identification of extract method refactoring opportunities,” in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 119–128.

[33] J. Gerling, “Machine learning for software refactoring: a large-scale empirical study,” Master’s thesis, Delft University of Technology, 2020.

[34] L. Kumar, S. M. Satapathy, and L. B. Murthy, “Method level refactoring prediction on five open source java projects using machine learning techniques,” in *Proceedings of the 12th Innovations on Software Engineering Conference*, 2019.

[35] P. S. Sagar, E. A. AlOmar, M. W. Mkaouer, A. Ouni, and C. D. Newman, “Comparing commit messages and source code metrics for the prediction refactoring activities,” *Algorithms*, vol. 14, no. 10, 2021.