Phase 3 Report

REMS: Recommending Extract Method Refactoring Opportunities via Multi-view Representation of Code Property Graph

Manohar Reddy Uppula, Meghana Kalluri, Soham Deo

INTRODUCTION

Extract method refactoring is a common technique used in software engineering to improve the quality of code by breaking down large, complex methods into smaller, more manageable ones. However, identifying appropriate extraction points can be a challenging task due to the complexity of modern software systems, which often consist of large codebases and involve multiple programming languages and libraries.

Currently there are two broad categories of state-of-the-art techniques to identify Extract Method refactoring opportunities- non-machine-learning-based approaches built on heuristics, and machine learning-based approaches built on historical data. The first line of approaches relies on heuristics, and the extraction criteria are characterized by deriving software metrics from fine-grained code properties. However, in most cases, these metrics can be challenging to concretize, and their selections and thresholds largely rely on expert knowledge. The second line of approaches, machine learning-based techniques, leverages historical data to recommend refactoring opportunities. Most of these approaches still rely on software metrics and features derived from code properties.

To overcome these challenges, this paper propose REMS, a Representation based Extract Method refactoring recommender System, a novel approach that leverages the power of Code Property Graphs (CPGs), and multi-view representation to identify and recommend Extract Method refactoring opportunities in software code. To accomplish this, we combine various representations of abstract syntax tree, control flow graph, and program dependency graph from code property graph using compact bilinear pooling and train machine learning classifiers to guide the extraction of appropriate lines of code that can be refactored as new methods.

In summary, our approach removes the need for expert knowledge to define heuristic rules and thresholds by automatically extracting features from the code property graph. The use of multi-view representations and machine learning classifiers allows for the extraction of suitable lines of code as a new method, providing a promising approach for enhancing software quality and maintainability.

RELATED WORK

Automated Extract Method refactoring approaches can be broadly divided into two categories: non-machine-learning-based and machine-learning-based. Non-machine-learning-based approaches utilize program slicing, block structure, and single responsibility principle violation to identify Extract Method refactoring candidates. For instance, JDeodorant is a non-machine-learning-based approach that employs block-based program slicing for variables in assignment statements to extract relevant statements. SEMI is another approach that explores the coherence between statements to identify code fragments with high cohesion. These approaches do not rely on machine learning techniques but instead utilize program structure analysis and code quality principles to identify Extract Method opportunities.

On the other hand, machine-learning-based approaches employ software metrics and machine learning classifiers to identify Extract Method refactoring opportunities. One such approach is GEMS, which uses metrics such as complexity, cohesion, and coupling as features to train machine learning classifiers. These approaches are often more automated and require less human intervention in the selection of metrics. Recent studies have focused on using machine learning techniques to automatically extract features from code fragments for related refactoring tasks, such as code smell detection. For instance, Liu et al. [55] employ the word2vec technique [30] to extract features from identifiers/comments, while Hadj-Kacem et al. [56] derive features from abstract syntax trees with the Variational Auto-Encoder [57]. These studies have reduced the reliance on expert expertise in selecting metrics for machine learning-based approaches and have made the process more automated.

LIMITATIONS OF RELATED WORK

There are several limitations to the studies on automated Extract Method refactoring.

Firstly, some of the non-machine-learning-based approaches are limited in their ability to identify complex refactoring opportunities that involve a large number of statements or control structures. Secondly, the selection of appropriate metrics for machine learning-based approaches requires expert knowledge, which can be challenging to obtain. Moreover, the use of different metrics may lead to different recommendations, and the criteria for selecting the most appropriate metric can be subjective.

Additionally, some of the automated approaches may require additional input or configuration, such as specifying a slicing criterion or a threshold for a particular metric. This requirement may increase the complexity of the refactoring process and reduce the automation's potential benefits.

PROPOSED SOLUTION

The majority of previous approaches for automated Extract Method refactoring rely on code metrics and require expert knowledge to set heuristic rules and thresholds. In contrast, our approach utilizes embedding techniques to automatically extract features from the code property graph for recommendation of Extract Method refactoring. We first extract the code property graph from both training and testing samples. Then, we create multi-view representations such as tree-view and flow-view representations and fuse them using compact bilinear pooling. Finally, we train machine learning classifiers to guide the extraction of appropriate lines of code as a new method.

BACKGROUND

A code property graph is a data structure that represents the source code of a software system as a directed graph. This is generated by combining Abstract syntax tree, Control flow graph and Program Dependency Graph. A brief background of this is provided below.

Abstract Syntax Tree (AST) is a way of representing a program's structure as a tree-like structure, where each node represents a language construct (such as a statement or an expression) and each edge represents the relationship between them. In the context of a property graph, the AST can be represented as a set of nodes and edges with labels and properties.

A Control Flow Graph (CFG) is a way of representing the flow of control in a program, showing all possible paths that code statements and conditions can take during program execution. The CFG property graph is denoted as PGcfg = (Vcfg,Ecfg, λcfg), where Vcfg represents the nodes in the graph and for each edge in Ecfg, λcfg assigns the label from ∑ cfg = {true, false, ϵ}.

A Program Dependency Graph (PDG) is a way of representing the dependencies between statements and predicates in a program, including both control and data dependencies. It captures all related statements and predicates that have an impact on the value of a particular variable at a specified statement.In the context of a property graph, the PDG can be represented as PGpdg = (Vcfg, Epdg, λpdg, μpdg), which means that it shares the same nodes as the Control Flow Graph (CFG), but has distinct edges. The edge labeling function λpdg assigns a label to each edge from the set Ppdg = {C,D}, where C and D represent control and data dependencies, respectively.

To put it simply, **Code Property Graph (CPG)** is a data structure that combines three important graphs in software analysis - Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependency Graph (PDG). It's a property graph, meaning it's made up of nodes and edges with properties assigned to them. Overall, the CPG provides a comprehensive view of the code structure and how different parts of the code are connected and depend on each other, which can be useful for analyzing, optimizing, and securing software systems.

STUDY DESIGN

The proposed approach is broadly divided into two phases.

Phase 1 – Representation Generation and feature extraction :

In this phase, for the given training and testing samples, the methods of the source code are transformed into code property graph using a parser(srcML) and a static analysis tool (Joern). The CPG represents the code as a graph, where the nodes are programming constructs like variables, functions, and classes, and the edges represent relationships between them. Embedding techniques (6 state-of-the-art relevant embedding techniques including CodeBERT , GraphCodeBERT, CodeGPT, CodeT5 , PLBART , and CoTexT) are used and are fused using compact bilinear pooling. Features are extracted from control-view and data-flow views to identify code patterns that may indicate opportunities for Extract Method Refactoring. These are then split into training and testing samples.

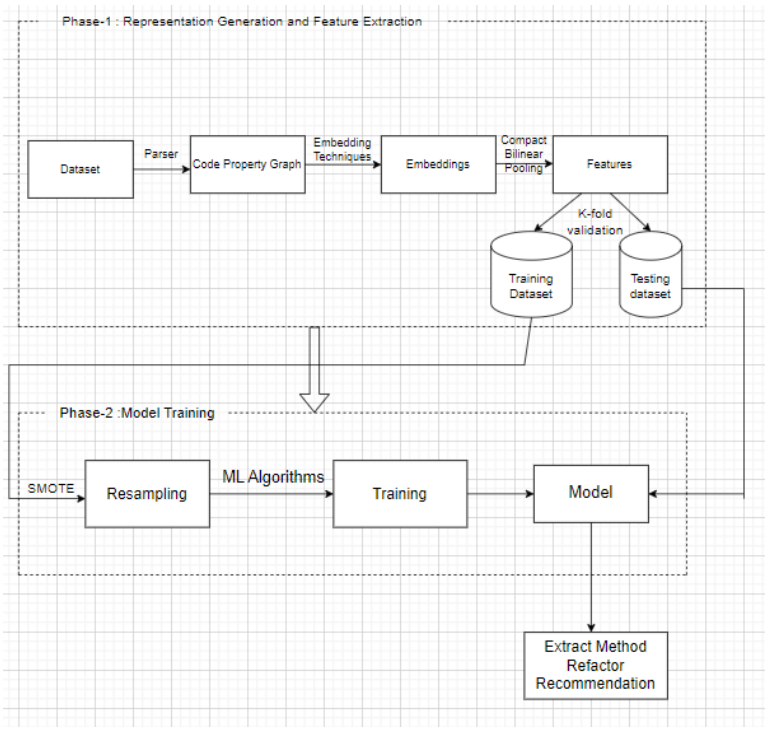


Figure 1: Approach Overview

Phase 2 – Model Training :

The samples derived are not completely representative and so to address this imbalance issue, SMOTE technique is used on the training data to adjust refactoring and non-refactoring samples. Different Machine Learning algorithms like Decision Tree, K-Nearest Neighbor, Logistic Regression, Naïve Bayes, Random Forest, Support Vector Machine are employed to train the model. These classifiers are selected as they are efficient and can be further integrated into IDE for interactive refactoring by users. Consequently, the testing data is given as an input to the model and the results are obtained.

LIMITATIONS OF REFERENCE PAPER'S APPROACH

The authors have used SMOTE resampling technique during the testing, which can impact the evaluation of the classifiers. Also, during the K-fold cross validation, applying SMOTE before k-fold split can result in presence of duplicates in test fold, which is sign of data leakage. To counter the negative impact of the above stated limitation, we made sure to apply SMOTE only on training folds of K-fold split.

EXPERIMENTATION

The evaluation is designed to answer the following research question:

RQ1: Which representations are the most effective in suggesting extract method refactoring opportunities?

The answer to this question would help us better understand how various representation combinations affect the performance of REMS. We carry out the experimentation in a way that reveals how the various ML algorithms are picking up knowledge from the various combinations of the data.

Experiment Setup:

We illustrate the used datasets, evaluation metrics, and experiment settings as follows:

Datasets:

We used the preprocessed data of Silva et al.'s dataset with the fused features to evaluate the impact of various multi-view representation combinations on the performance of REMS.

Evaluation Metrics:

We employed the same three frequently used evaluation metrics including Precision, Recall, and F1-Measure from the reference paper, which are defined as follows:

$$Precision = \frac{\#\ of\ correct\ recommended\ refactorings}{\#\ of\ recommended\ refactorings}$$

$$Recall = \frac{\#\ of\ correct\ recommended\ refactorings}{\#\ of\ correct\ refactorings}$$

$$F1\text{-}measure = \frac{2 * Precision * Recall}{Precision + Recall}$$

Precision is calculated as the ratio of correctly recommended refactoring candidates to the total number of recommended refactoring candidates. Recall is calculated as the ratio of correctly recommended refactoring candidates to the total number of correct refactoring candidates annotated by experts. F1-Measure is calculated as the harmonic mean of Precision and Recall value.

Experiment Settings:

We ran all the preliminary experiments on a 3.3GHz AMD Ryzen 9 6900HS laptop with 8 logical cores and 16GB of memory. We followed the default hyperparameters mentioned in the reference paper.

Experimental Process:

To reduce the bias caused by the experimental randomness, we repeat the 10-fold cross-validation 10 times (10×10) and compute the average value of evaluation metrics as results during these times. SMOTE is used on the training data to counter the imbalance of the target class. Grid search strategies are used to automatically tune the hyper-parameters of classifiers at each fold.

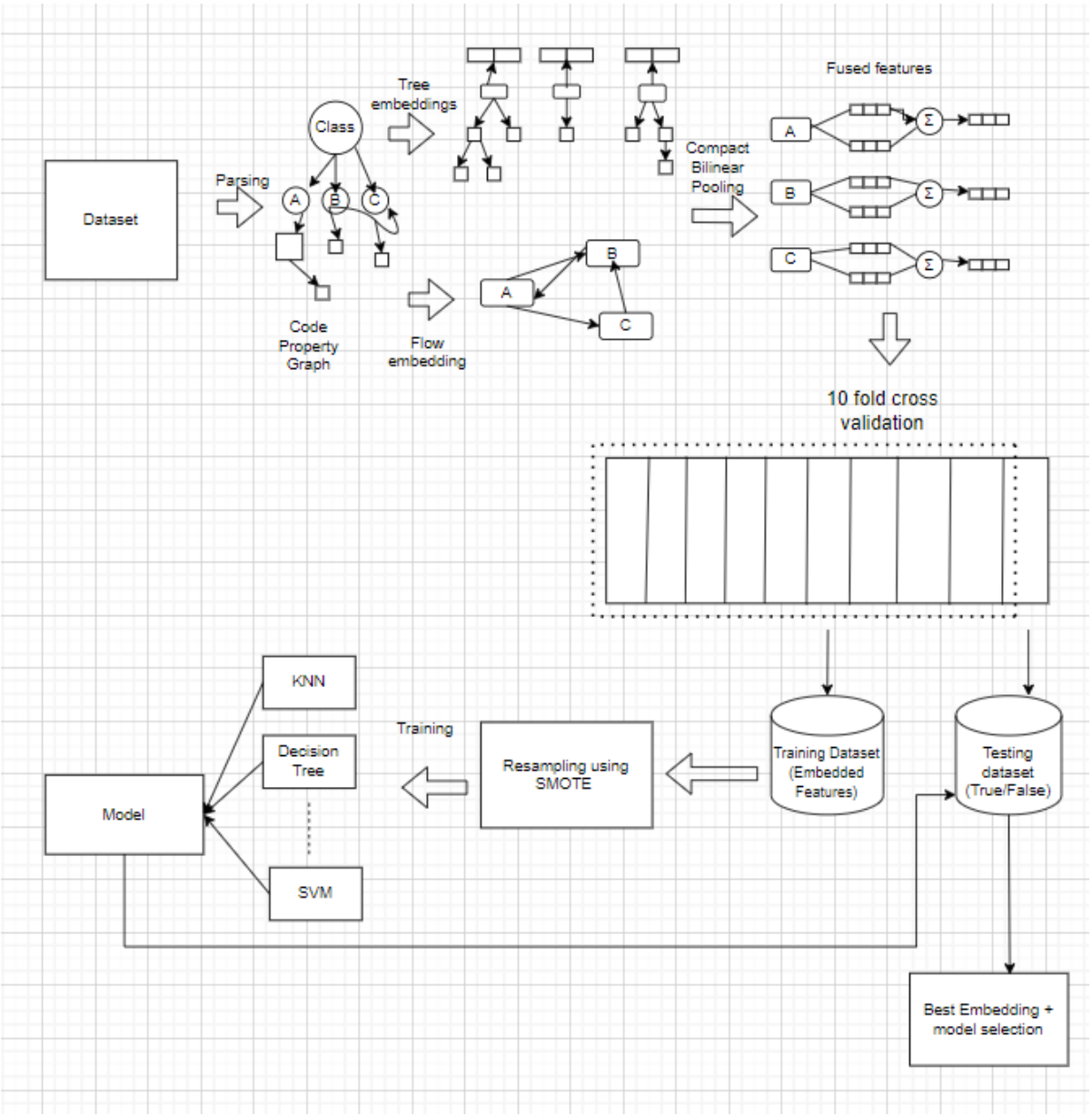


Figure 2 : The overall workflow of REMS

Experiment Results

Note: *Refer to attached excel report for detailed representation of results.*

Performance graphs of multiple Machine Learning classifiers across all the 42 combinations are as follows:

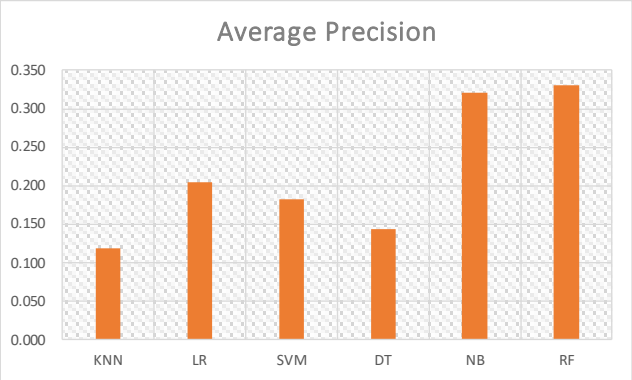


Figure 3 : Average Precision across combinations

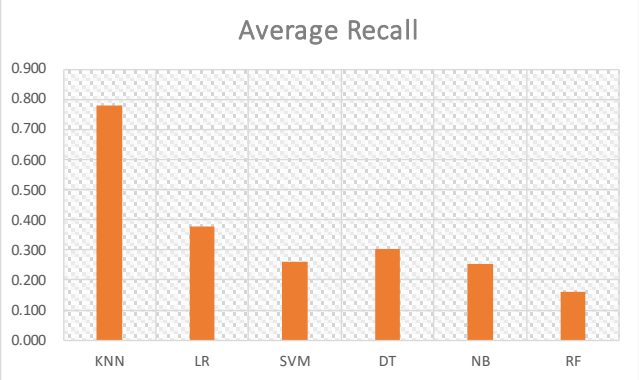


Figure 4 : Average Recall across combinations

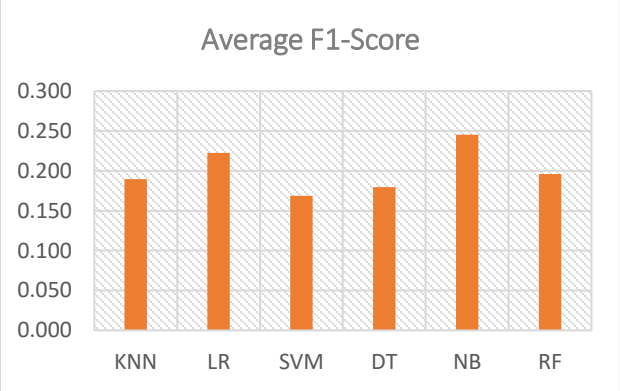


Figure 5 : Average F1-Score across combinations

From Figure 3,4,5 we see that Random Forest classifier performs well with respect to precision and fails in the case of recall. In contrast, KNN performs well with respect to recall but is not good with precision metric. To counter effect this bias created, we use the average of these metrics across the classifiers for each combination to get the best combination representation.

Table I presents the systematic comparison of 42 combinations of 6 tree-view representations and 7 flow-view representations, where rows and columns are labeled by tree-view and flow-view representations respectively. We name each representation with the corresponding embedding technique. It presents the average precision, recall, and f1-measure score of various classifiers on each combination.

For each flow-view representation, we highlight the greatest precision, recall, and f1-measure results with tree-view representations using pink color. Furthermore, we highlight the greatest results of all possible combinations using a thick border for the pink colored cell.

Tree_Views	CodeBERT			CodeGPT			CodeT5			CoTexT			GraphCodeBERT			PLBART		
Flow_Views	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
deepwalk_cg	0.275	0.521	0.288	0.291	0.462	0.274	0.228	0.483	0.223	0.252	0.500	0.245	0.287	0.495	0.278	0.242	0.382	0.237
grarep_cg	0.357	0.358	0.312	0.353	0.376	0.254	0.247	0.270	0.193	0.272	0.285	0.223	0.269	0.285	0.236	0.291	0.304	0.256
line_cg	0.089	0.196	0.052	0.062	0.217	0.061	0.069	0.208	0.050	0.066	0.205	0.056	0.051	0.199	0.050	0.063	0.228	0.056
node2vec_cg	0.260	0.546	0.291	0.246	0.473	0.263	0.237	0.467	0.214	0.252	0.436	0.252	0.280	0.510	0.285	0.288	0.425	0.271
prone_cg	0.195	0.240	0.151	0.194	0.286	0.119	0.141	0.212	0.101	0.152	0.221	0.113	0.133	0.222	0.110	0.145	0.187	0.101
sdne_cg	0.237	0.349	0.249	0.245	0.393	0.249	0.155	0.358	0.186	0.178	0.377	0.206	0.201	0.321	0.204	0.202	0.308	0.204
walklets_cg	0.324	0.463	0.303	0.306	0.427	0.269	0.203	0.435	0.194	0.241	0.442	0.234	0.239	0.420	0.239	0.284	0.443	0.263

Table I : Revised performance of various combinations on Silva et al.'s dataset

Table II represents the experimental results of the reference paper.

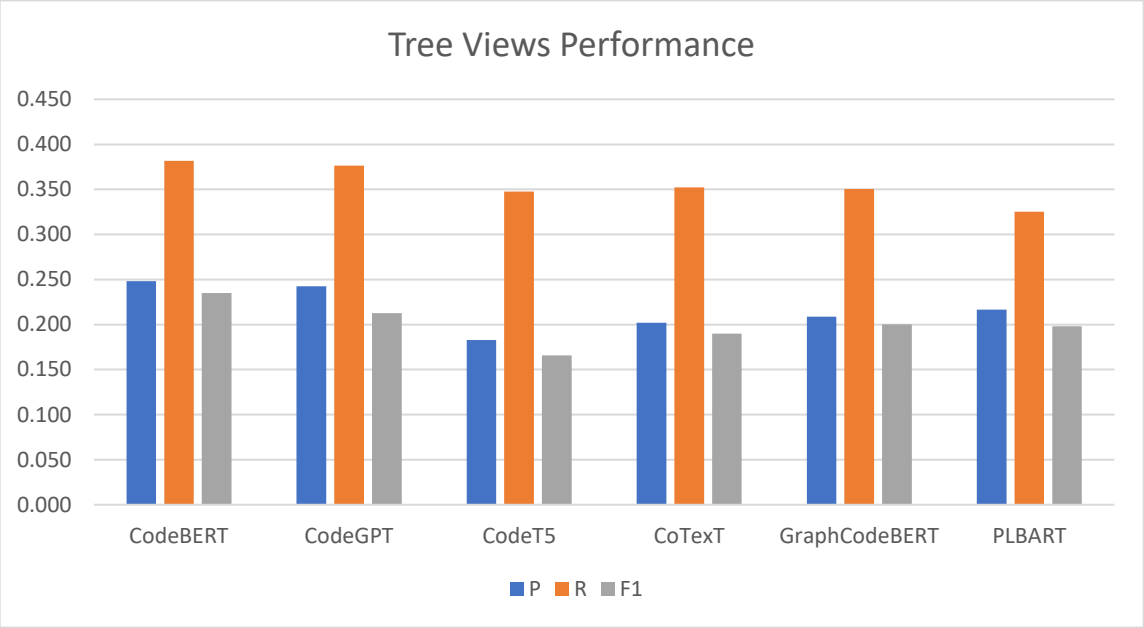
	CodeBERT			GraphCodeBERT			CodeT5			CodeGPT			PLBART			CoTexT		
Embedding	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
DeepWalk	.745	.668	.672	.761	.635	.661	.71	.721	.681	.76	**.627**	.658	.767	.631	.665	.731	.665	.655
GraRep	**.771**	.668	**.693**	**.783**	.642	.676	.713	**.725**	**.682**	**.809**	.6	**.672**	.762	**.683**	**.687**	.725	**.728**	**.69**
Line	.745	.625	.645	.75	.625	.653	.698	.659	.64	.741	.573	.62	.76	.569	.623	.729	.619	.634
Node2Vec	.765	.613	.651	.693	**.735**	**.687**	.714	.711	.676	.766	**.627**	.662	.763	.635	.667	.73	.668	.659
ProNE	.758	.584	.641	.734	.628	.65	.721	.604	.63	.75	.587	.639	**.77**	.531	.618	.726	.61	.637
SDNE	.726	**.716**	.685	.722	.721	**.687**	.712	.713	.673	.756	.62	.652	.752	.666	.671	.718	.714	.678
Walklets	.74	.693	.688	.763	.645	.674	**.762**	.584	.64	.799	.603	.67	.767	.644	.676	**.762**	.585	.641

Table II : Performance of various combinations on the Silva et al.'s dataset

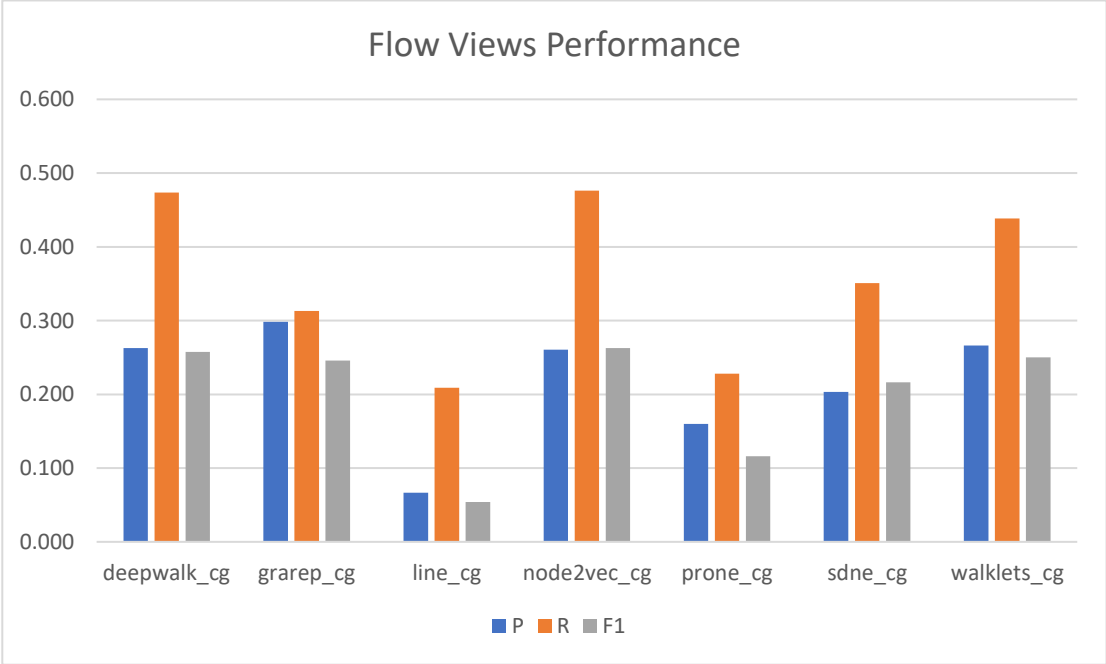
As presented in Table II, the authors of the reference paper observed that the combinations of CodeBERT+GraRep, CodeGPT+GraRep, and GraphCodeBERT+Node2Vec outperform other representation combinations. CodeGPT+GraRep achieved the highest precision score, whereas GraphCodeBERT+Node2Vec achieved the highest recall score. CodeBERT+GraRep presents the greatest scores in f1-measure. Two of the three combinations contained the flow-view representation: GraRep. A possible explanation devised by the authors was that GrapRep may comprehensively capture topology structures of control flows and program dependencies and thus improves the recommendation performance.

In contrast to the results from the reference paper, our experimentation resulted in combinations of CodeBERT+GraRep, CodeBERT+node2vec, and CodeBERT+GraRep as top performers in precision, recall and f1-measure respectively. Similar to the reference paper's results, two of the three combinations contain the flow-view representation: GraRep. However, CodeBERT dominated as the tree-view representation unlike the results of the reference paper.

Graphs of performance of the representations across different classifiers is given below.



From this we observe that CodeBERT and CodeGPT have similar performance. However, on an overall basis, CodeBERT outperforms other tree view representations (across all flow view representations and classifiers).



Considering recall, node2vec has high recall value. However, grarep performs better than other flow view representations as it has better precision and f1 score (across all tree view representations and classifiers).

THREATS TO VALIDITY

1. The tool is only trained on java-based programs and hence may not be useful for constructs of other programming languages.
2. For training the machine learning classifiers, we have used preprocessed data from the reference paper. This preprocessing is assumed to have been accurately representing Silva et al.'s dataset.

CONCLUSION

In this paper, we propose an approach to automatically recommend Extract Method refactoring opportunities named REMS. Our method involves using a code property graph to generate multiple representations of the code, including tree and flow views, and then fusing these views using compact bilinear pooling. We incorporate machine learning algorithms to train classifiers that can guide the identification of the most suitable lines of code to extract into a new method.

In contrast to the reference paper, we have taken care in not applying SMOTE to the test data during the K-fold cross validation, as this could lead to data leakage and have a negative impact on the final outcomes. Based on our experimental results, the combination of CodeBERT+GraRep achieved the highest precision score, while CodeBERT+node2vec had the highest recall score. Additionally, CodeBERT+GraRep yielded the highest f1-measure, which is contrary to the findings reported in the reference paper. The average values of the evaluation metrics are comparatively lower than the values reported in the original paper. Furthermore, the combination of CodeBERT+GraRep can be claimed to be the best representation, as it exhibits superior precision and f1-score performance.

REFERENCES

[1]. S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, and P.Avgeriou, "Identifying extract method refactoring opportunities based on functional relevance," IEEE Transactions on Software Engineering, vol. 43, no. 10, pp. 954–974, 2016.

[2]. 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). IEEE, 2017, pp. 24–34.

[3]. H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, "Deep learning based code smell detection," IEEE transactions on Software Engineering, 2019

[4]. T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," arXiv preprint arXiv:1301.3781, 2013

[5]. M. Hadj-Kacem and N. Bouassida, "Deep representation learning for code smells detection using variational autoencoder," in 2019 International Joint Conference on Neural Networks (IJCNN). IEEE, 2019, pp. 1–8.

[6]. F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," Empirical Software Engineering, vol. 23, no. 3, pp. 1188–1221, 2018.

[7]. Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang et al., “Codebert: A pretrained model for programming and natural languages,” arXiv preprint arXiv:2002.08155, 2020

[8]. D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu et al., “Graphcodebert: Pre-training code representations with data flow,” arXiv preprint arXiv:2009.08366, 2020

[9]. S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang et al., “Codexglue: A machine learning benchmark dataset for code understanding and generation,” arXiv preprint arXiv:2102.04664, 2021

[10]. Y. Wang, W. Wang, S. Joty, and S. C. Hoi, “Codet5: Identifier-aware unified pre-trained encoderdecoder models for code understanding and generation,” arXiv preprint arXiv:2109.00859, 2021

[11]. W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “Unified pre-training for program understanding and generation,” arXiv preprint arXiv:2103.06333, 2021

[12]. L. Phan, H. Tran, D. Le, H. Nguyen, J. Anibal, A. Peltekian, and Y. Ye, “Cotext: Multi-task learning with code-text transformer,” arXiv preprint arXiv:2105.08645, 2021.

[13]. (2022) SRCML. [Online]. Available: https://www.srcml.org

[14]. (2022) Joern. [Online]. Available: https://github.com/joernio/joern

[15]. K. Maruyama, “Automated method-extraction refactoring by using block-based slicing,” in Proceedings of the 2001 symposium on Software reusability: putting software reuse in context, 2001, pp. 31–40.

[16]. N. Tsantalis and A. Chatzigeorgiou, “Identification ofextract method refactoring opportunities for the decomposition of methods,” Journal of Systems and Software, vol. 84, no. 10, pp. 1757–1782, 2011

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

[18]. D. Silva, R. Terra, and M. T. Valente, “Jextract: an eclipse plug-in for recommending automated extract method refactorings,” arXiv preprint arXiv:1506.06086, 2015.

[19]. D. Silva, R. Terra, and M. T. Valente, “Recommending automated extract method refactorings,” in Proceedings of the 22nd International Conference on Program Comprehension, 2014, pp. 146–156.

[20]. J. Hubert, “Implementation of an automatic extract method refactoring,” Master’s thesis, 2019.

[21]. J. Yamanaka, Y. Hayase, and T. Amagasa, “Recommending extract method refactoring based on confidence of predicted method name,” arXiv preprint arXiv:2108.11011, 2021.

22. H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, “Deep learning based code smell detection,” IEEE transactions on Software Engineering, 2019.

23. M. Hadj-Kacem and N. Bouassida, “Deep representation learning for code smells detection using variational autoencoder,” in 2019 International Joint Conference on Neural Networks (IJCNN). IEEE, 2019, pp. 1–8.

24. F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, “On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation,” Empirical Software Engineering, vol. 23, no. 3, pp. 1188–1221, 2018.