MaRV: A Manually Validated Refactoring Dataset

Henrique Nunes
Federal University of Minas Gerais
Belo Horizonte, Brazil
henrique.mg.bh@gmail.com

Tushar Sharma
Dalhousie University
Halifax, Canada
tushar@dal.ca

Eduardo Figueiredo
Federal University of Minas Gerais
Belo Horizonte, Brazil
figueiredo@dcc.ufmg.br

***Abstract*—Despite the existence of traditional refactoring tools that offer semi-automated assistance, machine learning-based models have shown significant potential to generate refactored code. A comprehensive, manually validated refactoring dataset could help the software engineering community to train such models for effective refactorings. However, the community lacks a manually validated refactoring dataset. This paper introduces the *MaRV* dataset containing 693 manually evaluated code pairs extracted out of 126 GitHub Java repositories, representing four types of refactoring. In addition, the metadata describing the supposedly refactored elements was collected. Each code pair was manually evaluated by two reviewers out of 40 participants. *MaRV* dataset is constantly evolving with a web-based tool available for evaluating refactoring representations. The potential application of this dataset is to improve the accuracy and reliability of state-of-the-art models in refactoring tasks (*e.g.,* refactoring candidate identification and refactoring code generation) by providing high-quality data.**

***Index Terms*—refactoring, dataset, manually validated, foundation models**

I. Introduction

Refactoring is an essential software development activity to reduce complexity and improve maintainability of source code, without affecting its behavior [1, 2]. Refactorings are commonly employed to remove code smells and to address technical debt. Researchers have explored various aspects related to refactoring including cataloging of refactoring techniques [3], refactoring candidate identification [4, 5], tools to automatically refactor code [6, 7], and measure the impact of refactoring on software maintainability [8, 9].

Conducting refactoring manually is costly and slow, while automated tools, although useful, are at-best semi-automated and often produce inaccuracies requiring human evaluation to ensure correctness [10]. The arrival of state-of-the-art artificial intelligence models (*e.g.,* large language models (LLMs), or more generally, foundation models), has enabled a new approach to solve software engineering problems, including code refactoring by using prompting the model. Such models tend to perform better when prompted with some examples (also referred as few-shot learning [11]). However, current LLM-based approaches exhibit limitations such as introducing new errors and unintended behaviors [12, 13, 14]. One way to improve the efficacy of these models is to provide high-quality refactoring samples. Tools such as RefactoringMiner [15] provides a foundation to build these datasets; however, automatically collected datasets are subject to ambiguities and are prone to false positives that may adversely affect model training [16, 17]. Existing studies have used such tools to create refactoring datasets without evaluating the quality of the tool output [18, 19].

In this study, we create and propose Manually Validated Refactoring (*MaRV*) dataset, containing code snippets before and after a change, along with metadata such as manually annotated refactoring techniques, affected code elements, and commit details. To prepare this dataset, we use RefactoringMiner to identify applied refactorings from 126 popular Java repositories from GitHub. We focus on four refactoring techniques (*i.e., rename method*, *rename variable*, *extract method*, and *rename parameter*) to strike a balance between covering a variety of refactoring techniques, ensuring a significant number of annotations, and keeping the manual effort manageable. The initial set of identified refactorings is shown to the human reviewers with the help of a web-based tool that we developed to annotate each code pair with the presence or absence of refactoring. We invited researchers, developers, and students by email to contribute to the dataset. At the time of writing this paper, 40 reviewers have evaluated 693 pairs of code. Each code pair is evaluated by two reviewers. We provide the dataset in the form of JSON file.

Unlike other datasets [18, 19], our work consists of code diffs capturing manually evaluated refactoring techniques. Our approach aims to reduce noise and improve data quality in refactoring datasets. Furthermore, we attempt to specify various design choices (including selection criteria or repositories and refactoring techniques), provide the used scripts and tools to ensure replicability and extendibility.

The contributions of this work are:

1) A manually validated refactoring dataset comprising 693 pairs of code snippets across four refactoring techniques, each manually evaluated by two reviewers.
2) A web-based tool designed to support easy manual validation of refactorings. The tool is open-source and provided in replication package allowing other researchers to extend this or create new similar datasets.
3) Raw output generated by RefactoringMiner for 126 repositories, each with more than $10,000$ commits, and the scripts to extract snippets of various refactoring techniques using the RefactoringMiner and *git* metadata.

Replication package. The scripts and datasets are available online [20].

II. Dataset Construction

Figure 1 shows the study design to create *MaRV* dataset. We describe each step in the following sections.

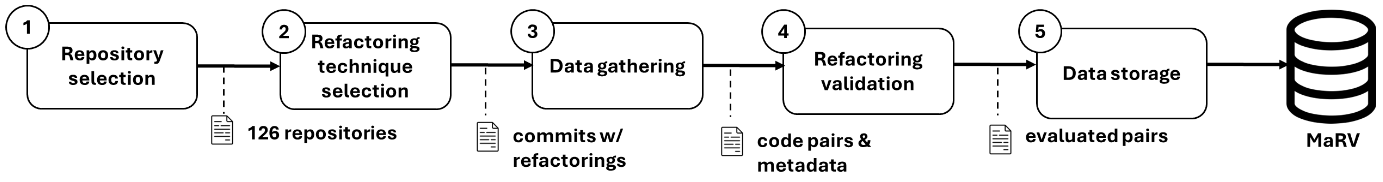


Fig. 1. Steps to create *MaRV* dataset.

A. Datasource

Repository selection: This is step 1 of our study design. We use the SEART GITHUB search [21] to select relevant repositories from GITHUB. We specify criteria to select repositories that are active, popular, and community-driven Java projects. Specifically, the criteria to select the repositories are (1) the programming language of the repository is Java — because it is one of the most used languages on GITHUB[1], (2) at least 10 contributors, (3) a large number of commits (at least $10,000$) [19], (4) at least $1,000$ stars [22]; and (5) at least one update in the last twelve months[2]. With these criteria, we identify 172 repositories.

We use RefactoringMiner [15] (version 3.0.7) to identify refactoring techniques applied in the selected repositories. We cloned each repository locally and analyzed them using RefactoringMiner on the default branch (*i.e., main* or *master*). RefactoringMiner generates a JSON file for each analyzed project. We successfully analyzed 126 repositories. We classify the 46 failed analyses into three categories: (1) *clone error*: 17 cases where our scripts were unable to clone the repositories automatically, (2) *RefactoringMiner execution error*: 16 cases where RefactoringMiner stopped due to tool-related errors and, (3) *commit analysis hang*: 13 cases where RefactoringMiner stalled for more than 6 hours on a specific commit, leading us to terminate its execution.

Despite the 26.74% failure rate, the 126 successfully mined repositories were sufficient to identify over 9 million refactorings across 102 different techniques.

Refactoring technique selection: This is step 2 of our study design. RefactoringMiner identifies 102 different refactoring techniques. We applied a filter to focus on a subset of it. We first obtained a distribution of refactoring techniques applied in a set of repositories. We selected the refactoring techniques in the top upper quartile (*i.e.,* in the top 75%). In addition, we selected only the method-level language agnostic traditional refactorings applied in Java and defined by Fowler [3].

With this selection criteria, we selected the following refactorings with their occurrence frequency: *change parameter type* ($563,098$), *add parameter* ($482,769$), *rename method* ($405,217$), *rename variable* ($303,547$), *extract method* ($285,153$), *rename parameter* ($282,396$), and *remove parameter* ($258,453$). Considering refactorings that researchers and programmers are frequently using [23, 24, 25], we focus on four refactoring techniques for this study: *rename method*, *rename variable*, *extract method*, and *remove parameter*.

[1]https://octoverse.github.com/2022/top-programming-languages
[2]September 2023 to September 2024.

B. Data gathering

This is step 3 of our study design. We created and used our `diff.py` script to parse the analysis reports generated by RefactoringMiner. The scripts filter refactoring techniques of the four selected types. We used the commit SHA and file path information for each refactoring instance to assess the code diff, extracting snippets before and after the commit for each identified refactoring. We identified added, modified, and deleted lines and stored this information. Furthermore, for each refactoring, we extracted additional metadata: (1) repository owner account and name, (2) refactoring technique, (3) affected source code elements by the refactoring (*e.g.,* method names, variables, parameters), (4) commit SHA, and (5) refactored file path.

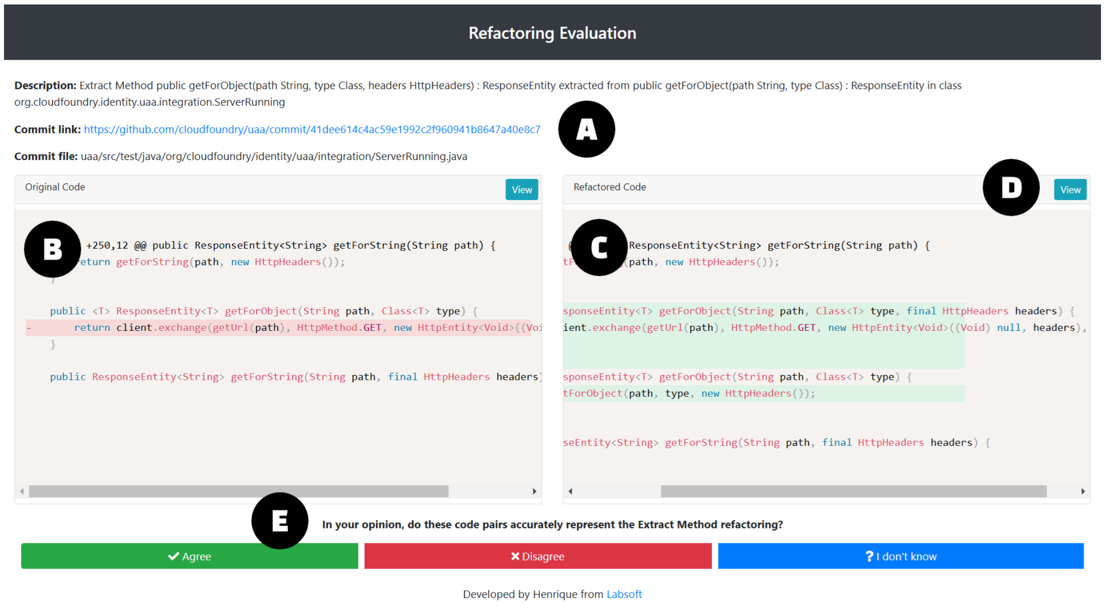
Keeping extensibility in mind, our script processes one refactoring technique at a time, that can be specified as an input parameter. The command `python3 diff.py 'Extract Method'` illustrates the execution of our script in a command prompt to perform the data gathering step for *extract method*. This allows researchers to use our script for specific refactoring techniques and add support to other refactoring techniques. The script creates a directory for each repository and subdirectories named with the commit SHA within it. Each commit could have one or more refactorings. The script generates three files for each refactoring:

1) `original_refactor<counter>.java`: snippet before the change.
2) `refactored_refactor<counter>.java`: snippet after the change.
3) `metadata_refactor<counter>.txt`: refactoring metadata.

We used our `filter_snippets.py` script to generate an SQL file that links the paths of the 'before' and 'after' snippets along with their metadata. Three filters are applied during parsing: (1) snippets with at least 100 lines (considering their suitability for manual evaluation), (2) code pairs with non-empty files, and, (3) pairs of snippets with at least one method declaration. This SQL file is used to populate the database that feeds the manual validation tool.

C. Refactoring validation

This is step 4 of our study design. We developed a Web tool for manual validation and classification of the identified



Description: Extract Method public getForObject(path String, type Class, headers HttpHeaders) : ResponseEntity extracted from public getForObject(path String, type Class) : ResponseEntity in class org.cloudfoundry.identity.uaa.integration.ServerRunning

Commit link: https://github.com/cloudfoundry/uaa/commit/41dee614c4ac59e1992c2f960941b8647a40e8c7

Commit file: uaa/src/test/java/org/cloudfoundry/identity/uaa/integration/ServerRunning.java

Original Code

```
 +250,12 @@ public ResponseEntity<String> getForString(String path) {
 eturn getForString(path, new HttpHeaders());


    public <T> ResponseEntity<T> getForObject(String path, Class<T> type) {
-       return client.exchange(getUrl(path), HttpMethod.GET, new HttpEntity<Void>((Voi
    }

    public ResponseEntity<String> getForString(String path, final HttpHeaders headers)
```

Refactored Code

```
ResponseEntity<String> getForString(String path) {
tF (path, new HttpHeaders());


sponseEntity<T> getForObject(String path, Class<T> type, final HttpHeaders headers) {
ient.exchange(getUrl(path), HttpMethod.GET, new HttpEntity<Void>((Void) null, headers),


sponseEntity<T> getForObject(String path, Class<T> type) {
tForObject(path, type, new HttpHeaders());


seEntity<String> getForString(String path, final HttpHeaders headers) {
```

In your opinion, do these code pairs accurately represent the Extract Method refactoring?

Developed by Henrique from Labsoft

Fig. 2. Example of refactoring manual evaluation screen.

snippets in the previous step (Section II-B). The tool is built using PHP and MySQL and is hosted with a publicly accessible URL. Due to double-blind review guidelines, we do not list the URL in this version of the paper. We provide the source code of the Web tool in our replication package.

When a reviewer (of the snippets) accesses the tool, the first page collects demographic information including their expertise in software development, Java, and refactoring as well as email and optionally name. After this registration, a reviewer can proceed with evaluations of refactorings.

Figure 2 shows an example of a refactoring evaluation page. At the top Ⓐ, it displays the refactoring metadata, provided by RefactoringMiner: (1) a description of the refactoring technique and affected elements, (2) a link to the commit webpage associated with the refactoring, and (3) the file path where the refactoring was applied. In the main panel, two blocks of code are presented: on the left side Ⓑ, the snippet before the change, and on the right side Ⓒ, the snippet after the change. Removed and added lines are highlighted in red and green, respectively. Each snippet box includes a *View* button Ⓓ that allows the whole file to be displayed in full-screen mode. At the bottom Ⓔ, there is the question *"In your opinion, do these code pairs represent the [refactoring technique]?"* followed by three options: *Agree*, *Disagree*, and *I don't know*. When a reviewer selects an answer, the corresponding label is recorded in the database, and a new random pair of code snippets is shown to the reviewer. There is no limit to the number of reviews each participant (verified by email) can evaluate, but our analysis indicates that no reviewer is responsible alone for more than 15% of the total evaluations. The tool ensures that two distinct reviewers evaluate each code pair. Additionally, the tool monitors the number of responses for each refactoring technique to maintain a balanced distribution: Extract Method (172), Remove Parameter (174), Rename Method (173), and Rename Variable (174).

We invited researchers, developers, and students by email to participate in our manual validation. The message we sent provides detailed explanations on participating in the evaluation and an email address to clarify any doubts. Overall, 40 participants evaluated 693 refactorings — we exclude five refactorings that were evaluated exclusively by one participant.

Table I shows the votes distribution. *Consensus* votes combine cases in which both reviewers agree [agree, agree] or disagree [disagree, disagree] with refactoring. A total of 321 $(46,32\%)$ refactorings are considered true positive by both reviewers and 84 $(12,12\%)$ are considered false positive, totaling 405 $(58,44\%)$ votes. *Conflict* votes are the cases in which reviewers vote differently, totaling 285 $(41,12\%)$ refactorings — [disagree, agree] $(31,31\%)$, [I don't know, disagree] $(3,89\%)$, [I don't know, agree] $(5,91\%)$. For only 3 $(0,43\%)$ refactorings both reviewers answered *'I don't know'*.

D. Data storage

This is step 5 of our study design. Our data are stored in the replication package. The dataset is provided as `MaRV.json`. The snippets are provided as follows:

- `Extract_Method.tar.gz`
- `Remove_Parameter.tar.gz`
- `Rename_Method.tar.gz`
- `Rename_Variable.tar.gz`

Furthermore, RefactoringMiner outputs are available online[3].

III. Potential Research Applications

A manually validated refactoring dataset has several potential research applications. We list some of them below.

[3]https://zenodo.org/records/14395034

TABLE I
CHARACTERISTICS OF THE COLLECTED ANNOTATIONS

	Votes	Count	Total
Consensus	[disagree, disagree]	84	405
	[agree, agree]	321	
Conflict	[disagree, agree]	217	285
	[I don't know, disagree]	27	
	[I don't know, agree]	41	
Other	[I don't know, I don't know]	3	3
Total			**693**

Refactoring identification benchmark: Refactoring identification tools aim to spot introduced refactorings in a commit (such as RefactoringMiner). The proposed dataset can be used to validate such tools with the help of manually validated refactorings in code snippets.

Refactoring candidate identification: The *MaRV* dataset can be used as a source for training models to detect refactoring candidates. By providing manually evaluated examples, *MaRV* allows the improvement of models to identify refactoring candidates with greater precision.

Refactored code generation: Models trained on *MaRV* can improve the quality of refactored code generation. Real examples of code before and after refactoring can improve the models capability to automatically apply program transformations while preserving the code behavior.

IV. Related Datasets

Manually validated datasets. Hegedus *et al.* [26] used a manually validated dataset to evaluate software maintainability. The study used software metrics and the *relative maintainability index* (RMI) on 145 true-positive refactoring diffs out of 627 manually evaluated snippets, comparing these metrics with a non-validated dataset. The non-validated dataset identified 2–4 metrics strongly associated with refactorings, while the validated dataset proved more reliable, identifying 3–6 such metrics. Nandani *et al.* [16] proposed a manually annotated code smells dataset constructed from ten Java repositories. The study first identify metrics thresholds for code smells with the help of 110 participants. The participants used a web-based tool and provided 17,869 annotations to classify snippets between smelly and benign. Subsequently, the study used 82 participants to classify 5,192 samples that are deemed subjective based on the derived metric thresholds in first phase of their study. Madeyski and Lewowski [17] created a dataset of four types of code smells extracted from GITHUB repositories with industrial relevance. Twenty-six developers from a company used a non-public visual tool to evaluate 4,770 code samples from 523 Java projects. The dataset includes the file path, the link to the sample, the smell type along with its severity.

Refactoring datasets. Aniche *et al.* [18] collected metrics to use as features for evaluating the performance of six machine learning algorithms in predicting opportunities for 20 techniques of refactoring. The dataset contains Java classes that are not validated for the presence of refactorings. Li and Zhang [19] proposed *RefT5*, an approach to detect five techniques of refactoring opportunities in multilingual systems. Their study built a non-validated dataset from 60 Java and 50 Python projects hosted on GITHUB, comprising 17,278 samples that include commit messages, edit sequences, refactoring techniques, and code refactoring diffs.

Unlike related works, our study perform a manual validation process with a publicly accessible tool, enabling collaborative annotations from contributors. This approach ensures not only the reliability of the dataset but also its scalability, as additional refactoring techniques and annotations can be integrated.

V. Threats to Validity

Internal validity. The tools and scripts used in this study may present a potential threat to internal validity. For instance, we could not analyze 46 repositories (i.e., 27%) due to script cloning errors or execution errors in RefactoringMiner. Given that each code pair is independent of other pairs, a partially failed analysis does not impact the rest of the dataset. Our scripts only include snippets with at least one method declaration and no more than 100 lines of code. While these criteria might exclude valid refactorings, they ensure the feasibility of a manual analysis. Finally, the manual validation process relies on reviewer expertise, which may introduce personal biases. To mitigate this potential issue, we ensured that each code pair was evaluated by two reviewers.

External validity. Our dataset is limited to four refactoring techniques detected in 126 open-source Java projects from GITHUB. Therefore, our results may not generalize, for instance, to other refactoring techniques or programming languages. We mitigate these threats by focusing on active, popular, community-driven Java projects with high numbers of commits and stars. We also target the most prevalent refactoring techniques in the third quartile (Q3) due to the effort required to manually classify a sufficient number of instances. However, it is important to emphasize that our scripts are prepared for the inclusion of new refactoring techniques; a dataset extension we plan for the upcoming months.

VI. Conclusion and Future Work

By selecting 126 repositories, focusing on a four of relevant refactoring techniques, and implementing a robust data gathering process followed by a manual evaluation, we created high-quality *MaRV* dataset, with 693 refactoring entries. This dataset serves as a valuable resource for further research and development in the field of code refactoring, especially in the context of automatic refactoring and LLMs.

We will extend this work by expanding the dataset with new manual evaluations and additional refactoring techniques. We also plan to use this dataset to fine-tune state-of-the-art models so they can refactor source code more efficiently. Additionally, we aim to explore the cases where reviewers disagreed in their evaluations, as this may reveal interesting insights into the nuances of code refactoring and model performance.

Acknowledgements. This research was partially supported by Brazilian funding agencies: CNPq (Grant 312920/2021-0), CAPES, and FAPEMIG (Grant APQ-01488-24).

References

[1] T. Mens and T. Tourwé, “A survey of software refactoring,” *IEEE Transactions on software engineering*, 2004.

[2] Y. Golubev, Z. Kurbatova, E. A. AlOmar, T. Bryksin, and M. W. 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*, 2021.

[3] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[4] 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*. IEEE, 2001.

[5] T. Tourwé and T. Mens, “Identifying refactoring opportunities using logic meta programming,” in *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.* IEEE, 2003.

[6] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, “Ten years of jdeodorant: Lessons learned from the hunt for smells,” in *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2018.

[7] F. A. Fontana, M. Mangiacavalli, D. Pochiero, and M. Zanoni, “On experimenting refactoring tools to remove code smells,” in *Scientific workshop proceedings of the XP2015*, 2015.

[8] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, “A quantitative evaluation of maintainability enhancement by refactoring,” in *International Conference on Software Maintenance, 2002. Proceedings.* IEEE, 2002.

[9] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, “An experimental investigation on the innate relationship between quality and refactoring,” *Journal of Systems and Software*, 2015.

[10] M. Schnappinger, A. Fietzke, and A. Pretschner, “Defining a software maintainability dataset: collecting, aggregating and analysing expert evaluations of software maintainability,” in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020.

[11] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni, “Generalizing from a few examples: A survey on few-shot learning,” *ACM computing surveys (csur)*, 2020.

[12] D. OBrien, S. Biswas, S. M. Imtiaz, R. Abdalkareem, E. Shihab, and H. Rajan, “Are prompt engineering and todo comments friends or foes? an evaluation on github copilot,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024.

[13] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, “Asleep at the keyboard? assessing the security of github copilot’s code contributions,” in *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[14] H. Nunes, E. Figueiredo, L. Soares, S. Nadi, F. Ferreira, and G. Esteves, “Evaluating the effectiveness of llms in fixing maintainability issues in real-world projects,” in *32th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2025.

[15] 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*, ser. ICSE ’18. New York, NY, USA: ACM, 2018.

[16] H. Nandani, M. Saad, and T. Sharma, “Dacos—a manually annotated dataset of code smells,” in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR).* IEEE, 2023.

[17] L. Madeyski and T. Lewowski, “Mlcq: Industry-relevant code smell data set,” in *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, 2020.

[18] M. Aniche, E. Maziero, R. Durelli, and V. H. Durelli, “The effectiveness of supervised machine learning algorithms in predicting software refactoring,” *IEEE Transactions on Software Engineering*, 2020.

[19] T. Li and Y. Zhang, “Multilingual code refactoring detection based on deep learning,” *Expert Systems with Applications*, 2024.

[20] H. Nunes, T. Sharma, and E. Figueiredo, “MaRV Scripts and Datasets,” Dec. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.14450098

[21] O. Dabic, E. Aghajani, and G. Bavota, “Sampling projects in github for MSR studies,” in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021.* IEEE, 2021.

[22] H. Borges and M. T. Valente, “What’s in a github star? understanding repository starring practices in a social coding platform,” *Journal of Systems and Software*, 2018.

[23] I. Palit, G. Shetty, H. Arif, and T. Sharma, “Automatic refactoring candidate identification leveraging effective code representation,” in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023.

[24] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, “Code smells and refactoring: A tertiary systematic review of challenges and observations,” *Journal of Systems and Software*, 2020.

[25] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” *IEEE Transactions on Software Engineering*, 2011.

[26] P. Hegedűs, I. Kádár, R. Ferenc, and T. Gyimóthy, “Empirical evaluation of software maintainability based on a manually validated refactoring dataset,” *Information and Software Technology*, 2018.