Mining Software Repositories to Improve Refactoring Assistants

Diogo Faria

Supervisor: Ademar Aguiar
Co-Supervisor: Sara Fernandes

17/07/2024

What is Refactoring?

Refactoring is “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.”[1]

1	Identify an Issue
2	Select Refactoring
3	Apply the Refactoring

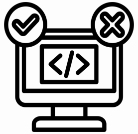
1. Fowler, M. Refactoring: Improving the Design of Existing Code. Pearson Education, 2018. https://books.google.pt/books?id=2H1_DwAAQBAJ.

Motivation

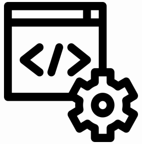
Traditional Refactoring can be seen as

- Error Prone
- Tedious
- Time Consuming

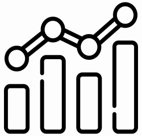
Solutions



Automated Testing



Automated Refactoring Tools



Refactoring Metrics

Problem

Threshold-Based Refactoring Tools Problems

01

Old Data

Thresholds based on the data available at the time of the tool's creation

02

Subjective to Creator

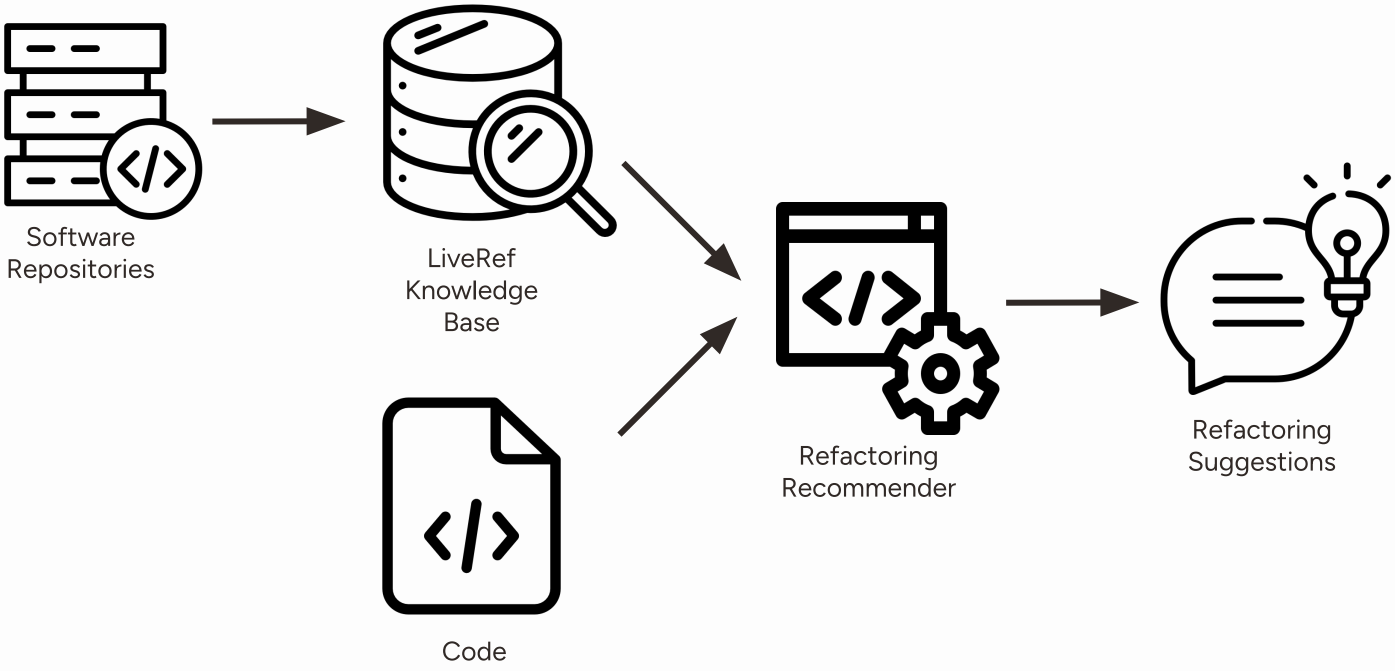
Transmitting the author's possible biases

03

Unable to adapt

Stuck in the moment and software contexts of its creation

Objective



Software
Repositories
LiveRef
Knowledge
Base
Code
Refactoring
Recommender
Refactoring
Suggestions

Problem Statement

"A refactoring recommendation system based on a dynamic classification model, built with real-life data, will lead to more accurate suggestions when compared to threshold-based methods."

RQ1

"How do threshold-based refactoring recommendations compare to the refactoring practices developers employ in real-life contexts?"

RQ2

"Is a classification model based on real data able to improve on the refactoring recommendations when compared to a threshold method?"

Mining Software Repositories (MSR)

Approaches

- Repositories contain a myriad of information suited to many purposes, including software maintainability and refactoring.[2]
- Tools such as SemDiff[3] and SysRepoAnalysis[4] analyse the repositories to provide change recommendations or static analyses.

Improvements

- DISDRILLEY can increase performance in data extraction in MSR.[5]
- Reverted commits have a high impact on noise data in refactoring detection.[6]

2. Mário André de F. Farias, Renato Novais, Methanias Colaço Júnior, Luís Paulo da Silva Carvalho, Manoel Mendonça, and Rodrigo Oliveira Spínola. A systematic mapping study on mining software repositories, 2016.
3. Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. ACM Trans. Softw. Eng. Methodol., 20(4):Article 19, 2011.
4. Armando Sousa, Gisele Ribeiro, Guilherme Avelino, Lincoln Rocha, and Ricardo Britto. Sysrepoanalysis: A tool to analyze and identify critical areas of source code repositories, 2022.
5. Martin Steinhauer and Fabio Palomba. Speeding up the data extraction of machine learning approaches: a distributed framework, 2020.
6. Fengcai Wen, Csaba Nagy, Michele Lanza, and Gabriele Bavota. Quick remedy commits and their impact on mining software repositories. Empirical Software Engineering, 27(1):14, 2021.

Refactoring Activity Detection

Mining Commit Logs

- Developers report their own refactorings in the commit logs.
- Approaches identify and Self Affirmed Refactoring (SAR) patterns that indicate refactoring activity.[7]

Mining the Source Code

- Analysing the version of the code before and after modifications to identify refactorings.
- Either through the use of code metrics that change in specific ways.[8]
- Or through defined code similarity thresholds.[9]

(1) Modif*	(15) Fix* quality flaw
(2) Simplif*	(16) Remov* dependency
(3) Polish* code	(17) Code improvement*
(4) Chang* design	(18) Fix* quality issue
(5) Us* less code	(19) Renam* consistency
(6) Simplif* code	(20) Reorganiz* structure
(7) Pull* some code	(21) Fix* technical debt
(8) Us* better name	(22) Remov* unused classes
(9) Code cosmetic*	(23) Remov* redundant code
(10) Delet* old stuff	(24) Improv* code quality
(11) Simplif* design	(25) Mov* more code out of
(12) Fix* code smell	(26) Fix* naming convention
(13) Nam* improvement	(27) Chang* package structure
(14) Modulariz* class	(28) Improv* naming

(1) Typo	(15) Formatted
(2) Tidy*	(16) Cleaned up
(3) Spell* code	(17) Code clean
(4) Tidied	(18) Get rid of
(5) Polish*	(19) Getting rid of
(6) Clarif*	(20) Meaningful
(7) Separat*	(21) Modulariz*
(8) Optimiz*	(22) Pulled out
(9) Organiz*	(23) Cleaning up
(10) Clean-up	(24) Better name
(11) Pull out	(25) Pulling out
(12) Structur*	(26) New structure
(13) Correct*	(27) Duplicate code
(14) Normaliz*	

Fig 1. Lists of SAR patterns[7]

Refactoring type	Rule
Change Method Signature m_a to m_b	$\exists(M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_a.b, m_b.b) \mid m_a \in M^- \land m_b \in M^+ \land m_a.c = m_b.c \land$ $\boxed{m_a.n \neq m_b.n \Rightarrow \text{Rename Method}}$ ① $(U_{T_1} = \varnothing \land U_{T_2} = \varnothing \land \texttt{allExactMatches}(M)) \lor$ ② $(\lvert M\rvert > \lvert U_{T_1}\rvert \land \lvert M\rvert > \lvert U_{T_2}\rvert \land \texttt{locationHeuristic}(m_a, m_b) \land \texttt{compatibleSignatures}(m_a, m_b)) \lor$ ③ $(\lvert M\rvert > \lvert U_{T_2}\rvert \land \texttt{locationHeuristic}(m_a, m_b) \land \exists\,\texttt{extract}(m_a, m_x)) \lor$ ④ $(\lvert M\rvert > \lvert U_{T_1}\rvert \land \texttt{locationHeuristic}(m_a, m_b) \land \exists\,\texttt{inline}(m_x, m_b))$
Extract Method m_b from m_a	$\exists(M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_a.b, m_b.b) \mid (m_a, m_{a'}) \in M^= \land m_b \in M^+ \land m_a.c = m_b.c \land \neg\texttt{calls}(m_a, m_b) \land \texttt{calls}(m_{a'}, m_b) \land \lvert M\rvert > \lvert U_{T_2}\rvert$
Inline Method m_b to $m_{a'}$	$\exists(M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_b.b, m_{a'}.b) \mid (m_a, m_{a'}) \in M^= \land m_b \in M^- \land m_{a'}.c = m_b.c \land \texttt{calls}(m_a, m_b) \land \neg\texttt{calls}(m_{a'}, m_b) \land \lvert M\rvert > \lvert U_{T_1}\rvert$
Change Class Signature td_a to td_b	$\exists(td_a, td_b) \mid td_a \in TD^- \land td_b \in TD^+ \land (td_a.M \supseteq td_b.M \lor td_a.M \subseteq td_b.M) \land (td_a.F \supseteq td_b.F \lor td_a.F \subseteq td_b.F)$ $\boxed{td_a.p \neq td_b.p \Rightarrow \text{Move Class}}$ $\boxed{td_a.n \neq td_b.n \Rightarrow \text{Rename Class}}$
Move Method m_a to m_b	$\exists(M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_a.b, m_b.b) \mid m_a \in M^- \land m_b \in M^+ \land m_a.c \neq m_b.c \land \lvert M\rvert > \lvert U_{T_1}\rvert \land \lvert M\rvert > \lvert U_{T_2}\rvert \land$ $(td_a, td_{a'}) \in TD^= \land m_a \in td_a \land (td_b, td_{b'}) \in TD^= \land m_b \in td_{b'} \land (\texttt{importsType}(td_{a'}, m_b.c) \lor \texttt{importsType}(td_b, m_a.c))$ $\boxed{\texttt{subType}(m_a.c, m_b.c) \Rightarrow \text{Pull Up Method}}$ $\boxed{\texttt{subType}(m_b.c, m_a.c) \Rightarrow \text{Push Down Method}}$
Move Field f_a to f_b	$\exists(f_a, f_b) \mid f_a \in F^- \land f_b \in F^+ \land f_a.c \neq f_b.c \land f_a.t = f_b.t \land f_a.n = f_b.n \land$ $(td_a, td_{a'}) \in TD^= \land f_a \in td_a \land (td_b, td_{b'}) \in TD^= \land f_b \in td_{b'} \land (\texttt{importsType}(td_{a'}, f_b.c) \lor \texttt{importsType}(td_b, f_a.c))$ $\boxed{\texttt{subType}(f_a.c, f_b.c) \Rightarrow \text{Pull Up Field}}$ $\boxed{\texttt{subType}(f_b.c, f_a.c) \Rightarrow \text{Push Down Field}}$
Extract m_b from m_a & Move to $m_b.c$	$\exists(M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_a.b, m_b.b) \mid (m_a, m_{a'}) \in M^= \land m_b \in M^+ \land m_a.c \neq m_b.c \land$ $\neg\texttt{calls}(m_a, m_b) \land \texttt{calls}(m_{a'}, m_b) \land \lvert M\rvert > \lvert U_{T_2}\rvert \land (td_a, td_{a'}) \in TD^= \land m_a \in td_a \land \texttt{importsType}(td_{a'}, m_b.c)$
Extract Supertype td_b from td_a	$\exists(td_a, td_b) \mid (td_a, td_{a'}) \in TD^= \land td_b \in TD^+ \land \texttt{subType}(\texttt{type}(td_{a'}), \texttt{type}(td_b))$ $\boxed{\exists\,\texttt{pullUp}(m_a, m_b) \mid m_a \in td_a \land m_b \in td_b \lor \exists\,\texttt{pullUp}(f_a, f_b) \mid f_a \in td_a \land f_b \in td_b \Rightarrow \text{Extract Superclass}}$ $\boxed{\exists(m_a, m_b) \mid m_a \in td_a \land m_b \in td_b \land \texttt{identicalSignatures}(m_a, m_b) \land m_b.b = \texttt{null} \Rightarrow \text{Extract Interface}}$
Change Package p_a to p_b	$\exists(p_a, p_b) \mid \texttt{path}(p_a) \in D^- \land \texttt{path}(p_b) \in D^+ \land \exists\,\texttt{MoveClass}(td_a, td_b) \mid td_a.p = p_a \land td_b.p = p_b$

$\texttt{matching}(T_1, T_2)$ returns a set of matched statement pairs (M) between the trees T_1 and T_2 representing method bodies, and two sets of unmatched statements from T_1 (U_{T_1}) and T_2 (U_{T_2}), respectively
$\texttt{indexOf}(m, td)$ returns the position of m inside type declaration td $\texttt{typeDecl}(c)$ returns the type declaration of type c $\texttt{type}(td)$ returns the qualified name of type declaration td
$\texttt{locationHeuristic}(m_a, m_b) = \lvert\texttt{indexOf}(m_a, \texttt{typeDecl}(m_a.c)) - \texttt{indexOf}(m_b, \texttt{typeDecl}(m_b.c))\rvert \leq \lvert M_c^- - M_c^+\rvert$ $\texttt{importsType}(td, t)$ returns true if type declaration td depends on type t
$\texttt{compatibleSignatures}(m_a, m_b) = m_a.P \supseteq m_b.P \lor m_a.P \subseteq m_b.P \lor \lvert m_a.P \cap m_b.P\rvert \geq \lvert(m_a.P \cup m_b.P) \setminus (m_a.P \cap m_b.P)\rvert$ $\texttt{calls}(m_a, m_b)$ returns true if method m_a calls m_b
$\texttt{subType}(c_a, c_b)$ returns true if c_a is a direct or indirect subclass of c_b or implements interface c_b $\texttt{path}(p)$ returns the directory path for package p

Fig 2. Refactoring detection rules in RMiner[9]

7. E. AlOmar, M. W. Mkaouer, and A. Ouni. Can refactoring be self-affirmed? An exploratory study on how developers document their refactoring activities in commit messages. In 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR), pages 51-58.
8. Raimund Moser, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A model to identify refactoring effort during maintenance by mining source code repositories. In Andreas Jedlitschka and Outi Salo, editors, Product-Focused Software Process Improvement, pages 360-370. Springer Berlin Heidelberg
9. Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history, 2018.

Refactoring Recommendation

Analysing the Source Code

- Analysing the structure of the code, like the Abstract Syntax Tree (AST), to identify possible need for refactoring.

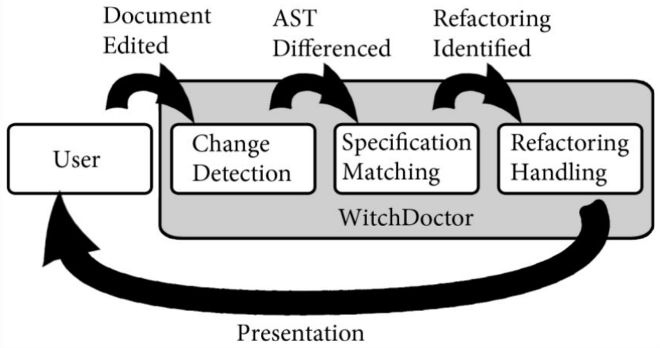


Fig 3. Graphical representation of WitchDoctor's workflow[10]

Code Quality Metrics

- Define code quality metrics which, depending on their values, for example using thresholds, may indicate the need for a specific refactoring.

Type of Metric	Code Quality Metric
File Metrics	Number of Lines of Code, Number of Comments, Number of Classes, Number of Methods, Average Number of Long Methods, Average Lack of Cohesion, Average Cyclomatic Complexity
Class Metrics	Number of Fields, Number of Public Fields, Number of Methods, Number of Long Methods, Class Lack of Cohesion, Average Cyclomatic Complexity
Method Metrics	Number of Parameters, Number of Lines of Code, Number of Comments, Number of Statements, Method Lack of Cohesion, Cyclomatic Complexity, Halstead Length, Halstead Vocabulary, Halstead Volume, Halstead Difficulty, Halstead Effort, Halstead Level, Halstead Time, Halstead Bugs Belivered, Halstead Maintainability

Fig 4. Code quality metrics supported by LiveRef[11]

10. S. R. Foster, W. G. Griswold, and S. Lerner. Witchdoctor: Ide support for realtime auto-completion of refactorings. In 34th International Conference on Software Engineering (ICSE), International Conference on Software Engineering, pages 222- 232, 2012. Foster, Stephen R. Griswold, William G. Lerner, Sorin 0270-5257.
11. Sara Fernandes, Ademar Aguiar, and André Restivo. A live environment to improve the refactoring experience, 2022.

LiveRef Knowledge Base Development Strategy

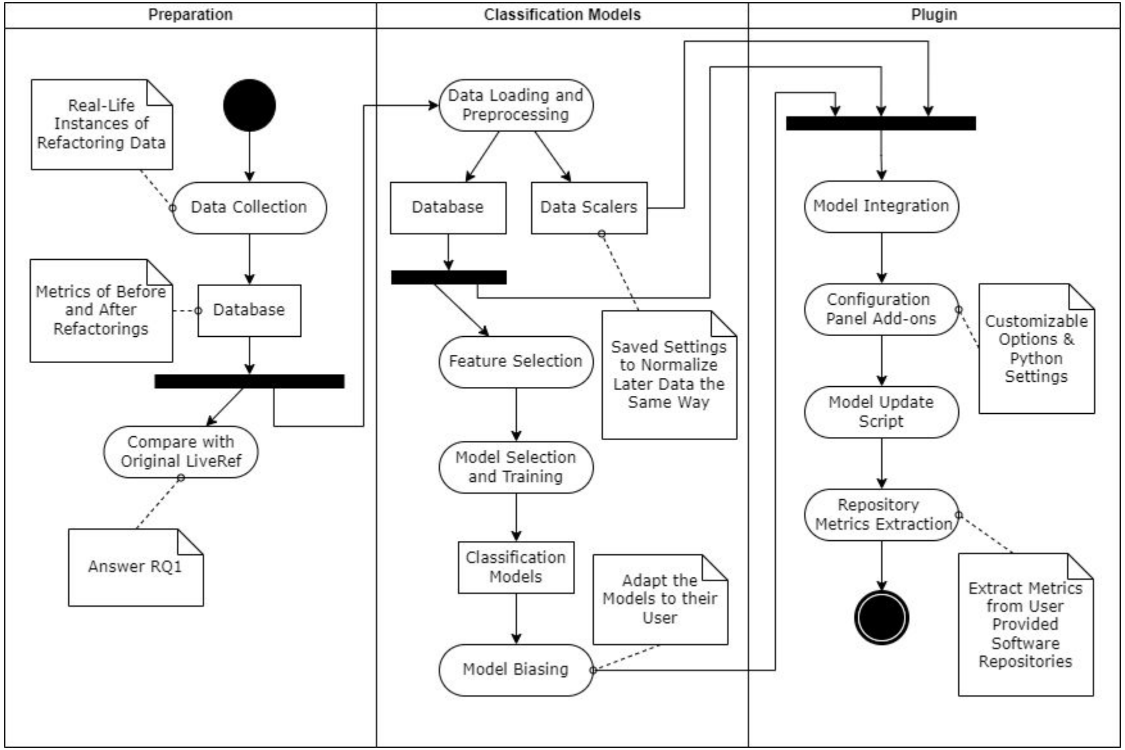


Fig 5. Activity diagram for solution development

Data Collection

Real-life instances of refactoring operations.

Extract Method:

- ≅ 25,000 rows
- 18 metrics

Extract Class:

- ≅ 2,500 rows
- 20 metrics

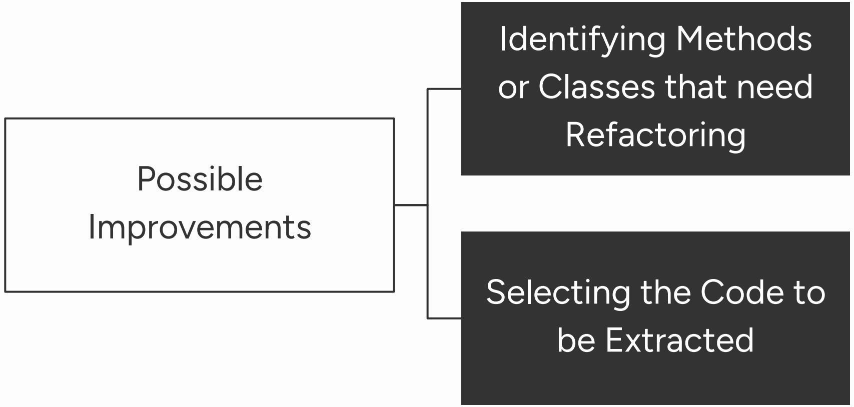
Metric	Extract Method	Extract Class
Number of Lines of Code	✓	✓
Number of Comments	✓	
Number of Blank Lines	✓	
Total Lines	✓	
Number of Parameters	✓	
Number of Statements	✓	
Halstead Length	✓	✓
Halstead Vocabulary	✓	✓
Halstead Volume	✓	✓
Halstead Difficulty	✓	✓
Halstead Effort	✓	✓
Halstead Level	✓	✓
Halstead Time	✓	✓
Halstead Bugs Delivered	✓	✓
Halstead Maintainability	✓	✓
Cyclomatic Complexity	✓	✓
Cognitive Complexity	✓	✓
LCOM	✓	
Number of Properties		✓
Number of Public Attributes		✓
Number of Public Methods		✓
Number of Protected Fields		✓
Number of Protected Methods		✓
Number of Long Methods		✓
Number of Methods		✓
Number of Constructors		✓

Fig 6. Collected Metrics

Threshold Model Baseline

Comparison of collected data with LiveRef:

- 56% of opportunities found by LiveRef
- Worse end result by LiveRef



Metric	Real-Life	LiveRef
Total Lines	-13.38	-9.75
Halstead Length	-11.54	-9.98
Halstead Vocabulary	-87.54	-77.02
Halstead Volumne	-85.74	-75.02
Halstead Difficulty	-2.86	-2.60
Halstead Effort	-2763.38	-2440.33
Halstead Level	0.06	0.03
Halstead Time	-153.52	-135.57
Halstead Maintainability	4.94	4.03
Cyclomatic Complexity	-2.13	-1.90
Cognitive Complexity	-42.96	-13.69
LCOM	-0.04	0.001

Fig 7. Comparison of Average Differences for Various Metrics when Compared to Baseline Code

Model Development

Data Loading and Preprocessing

- Cleaning faulty data;
- Normalizing the data.

Feature Selection

- Removing features based on covariance.

Metric	Extract Method	Extract Class
Number of Lines of Code	✓	
Number of Statements	✓	
Halstead Effort	✓	✓
Halstead Length	✓	✓

Fig 8. Removed Features for each Refactoring Type

Model Development

Model Selection & Training

- There is only data for a single class, when a refactoring is meant to occur;
- Selected 3 one-class classification models, all effective in high dimensional datasets;
- Elliptic Envelope performed better in both scenarios, though One Class SVM was also kept for further consideration.

Model	Extract Method	Extract Class
One Class SVM	0.904	0.903
Isolation Forest	0.897	0.899
Elliptic Envelope	**0.989**	**0.988**

Fig 9. Comparison of Recall of the Different Models

Model Integration

- Models were created with Python, thus requiring integration with Java;
- User is required to provide Python path;
- LiveRef is now using created models to identify methods/classes that require refactoring;
- Models are updated after a certain amount of executed refactorings.

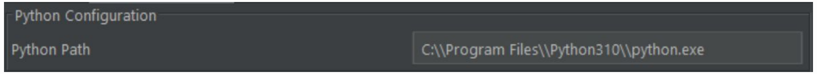


Fig 10. Set Python Path in LiveRef Configuration Panel

Algorithm 1 Get Extractable Fragments for Extract Method from Source File

```
function GETEXTRACTABLEFRAGMENTS(sourceFile)
    fragments ← empty list
    for each metrics in Values.before.methodMetrics do
        if sourceFile.name does not contain metrics.methodName then
            if metrics.method.body is not null then
                if PredictionModel.predictEM(metrics, editor.project) then
                    statements ← metrics.method.body.statements
                    for each statement in statements do
                        fragments.add(getFragmentsFromStatement(statement, metrics))
                    end for
                end if
            end if
        end if
    end for
    return fragments
end function
```

Fig 11. Pseudocode for GetExtractableFragments function

Model Biasing

- Large amount of training data impedes the model from being attuned to the user in a reasonable timeframe;
- Sample weights allow for the process to become quicker;
- Allows for the creation of profiles, such as individual, team, and organisation.

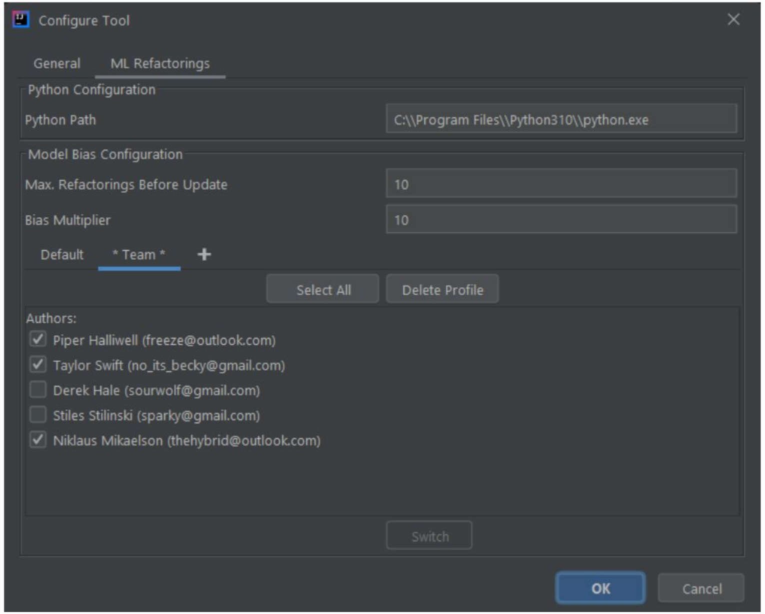


Fig 12. ML Refactorings tab in Configuration Panel

Repository Metrics Extraction

- Another way to attune the model to the user;
- Allows them to provide a repository, from which the refactoring data will be extracted;
- Updates the models and the authors for the bias profiles.

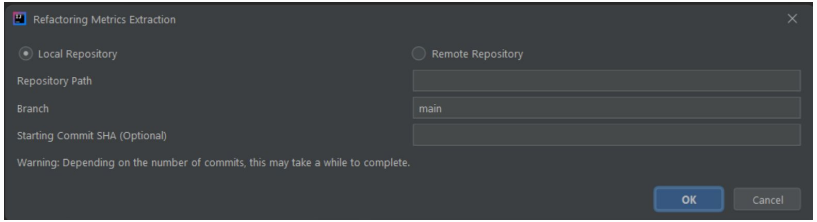


Fig 13. Pop-up for Repository Metrics Extraction

Validation

Extract Method

- 13% increase with Elliptic Envelope;
- 5% increase with One-Class SVM.

Method	Percentage of Opportunities
Original Plugin	55.98%
One-Class SVM	60.43%
Elliptic Envelope	69.08%

Fig 14. Comparison of Percentage of refactoring opportunities found for Extract Method

Extract Class

- 17% increase with Elliptic Envelope;
- 12% increase with One-Class SVM;

Method	Percentage of Opportunities
Original Plugin	5.75%
One-Class SVM	17.12%
Elliptic Envelope	22%

Fig 15. Comparison of Percentage of refactoring opportunities found for Extract Class

Conclusions

RQ1

Threshold-based refactoring recommendation tool missed a large number of refactorings and provided worse quality recommendations.

RQ2

When compared to real-life data, the classification model missed less suggestions than the threshold-based method.

Main Contributions

Literature Review

State-of-the-art research performed.

Plugin

Updated version of LiveRef.

Refactoring data

Training and testing dataset.

Future Work

Adding Refactoring Types

Profile Synchronisation

Diversifying the Data

Extended Testing

References

1. Fowler, M. Refactoring: Improving the Design of Existing Code. Pearson Education, 2018. https://books.google.pt/books?id=2H1_DwAAQBAJ.
2. Mário André de F. Farias, Renato Novais, Methanias Colaço Júnior, Luís Paulo da Silva Carvalho, Manoel Mendonça, and Rodrigo Oliveira Spínola. A systematic mapping study on mining software repositories, 2016.
3. Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. ACM Trans. Softw. Eng. Methodol., 20(4):Article 19, 2011.
4. Armando Sousa, Gisele Ribeiro, Guilherme Avelino, Lincoln Rocha, and Ricardo Britto. Sysrepoanalysis: A tool to analyze and identify critical areas of source code repositories, 2022.
5. Martin Steinhauer and Fabio Palomba. Speeding up the data extraction of machine learning approaches: a distributed framework, 2020.
6. Fengcai Wen, Csaba Nagy, Michele Lanza, and Gabriele Bavota. Quick remedy commits and their impact on mining software repositories. Empirical Software Engineering, 27(1):14, 2021
7. E. AlOmar, M. W. Mkaouer, and A. Ouni. Can refactoring be self-affirmed? An exploratory study on how developers document their refactoring activities in commit messages. In 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR), pages 51–58.

References

8. Raimund Moser, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A model to identify refactoring effort during maintenance by mining source code repositories. In Andreas Jedlitschka and Outi Salo, editors, Product-Focused Software Process Improvement, pages 360–370. Springer Berlin Heidelberg
9. Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history, 2018.
10. S. R. Foster, W. G. Griswold, and S. Lerner. Witchdoctor: Ide support for realtime auto-completion of refactorings. In 34th International Conference on Software Engineering (ICSE), International Conference on Software Engineering, pages 222– 232, 2012. Foster, Stephen R. Griswold, William G. Lerner, Sorin 0270-5257.
11. Sara Fernandes, Ademar Aguiar, and André Restivo. A live environment to improve the refactoring experience, 2022.

Thank you!
Any Questions?