

VRIJE UNIVERSITEIT BRUSSEL

DOCTORAL DISSERTATION

---

**Extracting Library Features from  
Incomplete Code on Stack Overflow**

---

Camilo Velázquez-Rodríguez

*Dissertation submitted in fulfilment of the requirements  
for the degree of Doctor of Sciences*

*Promotor:*

Prof. Dr. Coen De Roover

*Jury:*

Prof. Dr. Viviane Jonckers, Vrije Universiteit Brussel  
Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel  
Prof. Dr. Ann Nowé, Vrije Universiteit Brussel  
Prof. Dr. Sam Verboven, Vrije Universiteit Brussel  
Prof. Dr. Davide Di Ruscio, Università degli Studi dell'Aquila  
Prof. Dr. Bin Lin, Radboud University

Faculty of Sciences and Bioengineering Sciences  
Department of Computer Science  
Software Languages Laboratory

March 29<sup>th</sup>, 2024

Alle rechten voorbehouden. Niets van deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook, zonder voorafgaande schriftelijke toestemming van de auteur.

All rights reserved. No part of this publication may be produced in any form by print, photoprint, microfilm, electronic or any other means without permission from the author.

Printed by

Crazy Copy Center Productions

VUB Pleinlaan 2, 1050 Brussel

Tel: +32 2 629 33 44

[crazycopy@vub.ac.be](mailto:crazycopy@vub.ac.be)

[www.crazycopy.be](http://www.crazycopy.be)

ISBN: 9789464948219

NUR: 965

THEMA: UMZ

# Abstract

It is common in contemporary software development to reuse features provided by third-party libraries. Reusing features instead of re-implementing them from scratch can reduce development time and may improve the overall system quality. However, selecting an appropriate library to reuse features from can be difficult for developers due to the lack of automated tool support. Some library indices propose rankings of libraries, but these are biased towards the number of library downloads, attributed stars, etc. This may lead developers to select the most popular library instead of the library with the feature they were hoping to reuse.

This thesis makes three contributions. The first, called LiFUSO, is an automated approach to enumerating and describing the features provided by a library based on publicly available information on social coding platforms such as Stack Overflow (SO) and GitHub. LiFUSO analyses library usages within SO posts and extracts usage patterns indicative of library features. To this end, it considers both the code snippets and the surrounding natural language within each SO post that discusses the library.

RESICO, the second contribution of the thesis, is an automated approach to resolving API type references within a code snippet to their corresponding fully-qualified name. As a learning-based text classification approach, RESICO needs to be trained on a corpus of programs for which a compiler has determined the correct type information. Once trained, it can take syntactically incorrect code snippets as input.

The final contribution combines LiFUSO and RESICO so the former is no longer limited in scope to SO posts that have been tagged with a library's name but can also extract information from posts in which it has recognised many library types. We evaluate the impact of broadening the scope of the analysis on the quantity and quality of the uncovered library features.

The contributions are relevant to the software engineering community at large. RESICO's type resolution can be adopted by tools that need to analyse potentially incomplete code snippets, or by tools that need to determine the libraries used within a code snippet —just like in our third contribution. LiFUSO paves the way for tool support for selecting a library from many alternatives. Finally, these contributions help understand the benefits and drawbacks of data-centric over algorithmic-centric solutions to software engineering problems.



# Samenvatting

In de hedendaagse softwareontwikkeling is het gebruikelijk om features van externe bibliotheken te hergebruiken. Features hergebruiken, in plaats van ze helemaal opnieuw te implementeren, kan de ontwikkeltijd verkorten en de algemene systeemkwaliteit verbeteren. Het selecteren van een geschikte bibliotheek om features uit te hergebruiken kan echter moeilijk zijn voor ontwikkelaars vanwege het gebrek aan geautomatiseerde toolondersteuning. Sommige bibliotheekindices stellen ranglijsten van bibliotheken voor, maar deze zijn bevooroordeeld op het aantal bibliotheekdownloads, toegekende sterren, enz. Dit kan ertoe leiden dat ontwikkelaars de populairste bibliotheek selecteren in plaats van de bibliotheek met de feature die ze hoopten te hergebruiken.

Deze thesis levert drie bijdragen. De eerste, genaamd LiFUSO, is een geautomatiseerde benadering voor het opsommen en beschrijven van de features die door een bibliotheek worden geboden, op basis van openbaar beschikbare informatie op sociale coderingsplatformen zoals Stack Overflow (SO) en GitHub. LiFUSO analyseert bibliotheekgebruik binnen SO-berichten en extraheert gebruikspatronen die indicatief zijn voor bibliotheekfeatures. Daartoe wordt zowel rekening gehouden met de codefragmenten als met de omringende natuurlijke taal binnen elk SO-bericht waarin de bibliotheek wordt besproken.

RESICO, de tweede bijdrage van de thesis, is een geautomatiseerde aanpak voor het omzetten van API-typerefereenties binnen een codefragment naar hun overeenkomstige, volledig gekwalificeerde naam. Als een op leren gebaseerde benadering van tekstclassificatie moet RESICO worden getraind op een corpus van programma's waarvoor een compiler de juiste type-informatie heeft bepaald. Eenmaal getraind kan het syntactisch onjuiste codefragmenten als invoer gebruiken.

De laatste bijdrage combineert LiFUSO en RESICO zodat de eerste niet langer beperkt is tot SO-berichten die zijn getagd met de naam van een bibliotheek, maar ook informatie kan extraheren uit berichten waarin veel bibliotheektypen zijn herkend. We evalueren de impact van het verbreden van de reikwijdte van de analyse op de kwantiteit en kwaliteit van de ontdekte bibliotheekfeatures.

De bijdragen zijn relevant voor de software-engineeringgemeenschap als geheel. De typeresolutie van RESICO kan worden overgenomen door tools die potentieel onvolledige codefragmenten moeten analyseren, of door tools die moeten bepalen welke bibliotheken binnen een codefragment worden gebruikt —net als in onze derde bijdrage. LiFUSO maakt de weg vrij voor toolondersteuning bij het selecteren van een bibliotheek uit vele alternatieven. Ten slotte helpen deze bijdragen bij het begrijpen van de voor- en nadelen van datagerichte dan wel algoritmische oplossingen voor software-engineeringproblemen.



# Acknowledgements

The path of a Ph.D. is not lonely, at least for me, it wasn't. I wouldn't be able to get to this point without the unconditional help of my family, friends and colleagues. I want to start by appreciating and thanking all jury members for their insightful and constructive comments during the private and public defences, which helped shape the dissertation better.

The moment I started my Ph.D. I was asking myself many things, but most of all, what kind of professional and person my promotor was. Luckily, Coen showed me his professionalism and I think more importantly, his good values. Despite his super-busy calendar, he always found time whenever I had a problem. I will always be in his debt because he allowed me to do research with top researchers and to grow as a scientist and person.

I wouldn't be here today if I didn't have remarkable people next to me. Especially my partner, my wife, my love, Indira. Her support, going through the good and more importantly, through the rough times, her stubbornness (more than mine) and her love towards me, make me feel like the luckiest husband.

On a special day like this, my father occupies a relevant place. I will always remember him as a brilliant man, but more importantly as the excellent father and paternal figure he was for me and my sister.

It is also a day with mixed feelings because I would have loved to have my family in Cuba as part of the audience. For my family, I would like to dedicate the following words in Spanish: *Má, Alla, que yo esté aquí es también gracias a ustedes, su apoyo, consejos y amor incondicional. Me hubiera gustado que estuvieran presentes físicamente, pero yo sé que están conmigo en la distancia y se sienten felices por mí. A mi sobrino, decirle que siga siendo lo imaginativo y creativo que es y que siga jugando con tío cuando lo visite. A Coello, le agradezco todas las veces que ha ayudado a mi madre y hermana en todos los momentos que pudo. A Fátima, porque aunque me viva jodiendo, no puedo pedir una cuñada mejor y también me alegro mucho de todos sus logros. Finalmente, también quisiera recordar a mis suegros, Arlene y Arnoldo por hacerme parte de la familia desde que llegué a su casa, y por siempre ser hermosas personas.*

I would also like to have a few appreciation words for Wolf. Even though he was not directly involved in my research, neither was he my promotor, he was very gentle to me since day one. His continuous pieces of advice to me and my wife about all sorts of topics, from transportation, politics, to a career, were very important for us as foreigners in Belgium.

I want to thank my co-authors who helped me tremendously during my Ph.D. Dario, you were my first office partner, and you were always a fun person to be around, very hard-working and helped me design my first paper submission. Eleni, thanks

for always being critical, asking me very difficult questions during our countless meetings and being there despite your busy agenda and life. Ahmed, I would also like to appreciate your hard-working compromise and all the nice conversations we had about research, life, sports, etc. Some words of appreciation also go to Maarten who volunteered to read the dissertation, gave me valuable feedback on the last two chapters, and has helped tremendously in organising the reception and dinner.

Many friends made it possible for me to be in Belgium and to start my life here in the best possible way. I would like to thank Roberto Becerra, for always being a friend since my student times in Cuba and for always teaching me a new thing when we met. Humberto, for starting my contact with Coen, helping me during all these years and making my life easier when I started. I want to also thank Scull for helping me with the paperwork for my Ph.D. application, during my first two years at Olguita's house and guiding me in Belgium. No quisiera dejar de mencionar a Olguita, Cirelda y Patrick que me acogieron como uno más de su familia cuando llegué. Son muchos los buenos momentos que compartimos y quisiera agradecerles su gentileza, valiosos consejos y compañía que impidieron que me sintiera lejos de casa.

I have gained really good friends in Belgium, and others stayed close despite being far in the distance. My good friends Elio, Alejandro (Paki) and Héctor; even though we are no longer together in Cuba, have continued our friendship, which is a daunting task in the distance. I met Jorge and later Jose here, and they became my friends. We have shared many good moments, which I truly hope to continue to be the case. Michael, thank you for all your help during these years, your delicious food and above all, your good company. Tatiana and Patrick, also took me in as family since we first met in Leuven, to them is my gratitude for all the nice moments together.

My previous mentors and professors also made it possible for me to be here today. I want to thank Fofi for introducing me to the research life and giving me the first opportunities as a researcher. To all my previous professors at Universidad de Holguín in Cuba, my deepest appreciation for their teachings during my career.

I didn't want to finish my acknowledgements without mentioning all my colleagues at the CAMP group and SOFT. I'm lucky to be part of the researchers at SOFT where many brilliant people gather. More importantly, during my years there I have also witnessed the goodness of the people willing to help at any moment about anything. To all of you, and all the persons mentioned before, thanks for being a part of my journey.



# Contents

<b>Abstract</b>	<b>iii</b>
<b>Samenvatting</b>	<b>v</b>
<b>Acknowledgements</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Statement . . . . .	3
1.2 Overview of the Approach . . . . .	5
1.3 Contributions and Supporting Publications . . . . .	6
1.4 Outline of the Dissertation . . . . .	7
<b>2 Background</b>	<b>9</b>
2.1 API Usages in Library Documentation, Messaging Platforms and Q&A Fora . . . . .	9
2.2 Stack Exchange and SOTorrent Stack Overflow Dataset Dumps . . . . .	14
2.3 Text Transformation into their Vector Representation . . . . .	15
2.3.1 TF-IDF . . . . .	16
2.3.2 Word2Vec . . . . .	16
2.4 Text Classification as a Natural Language Processing Problem . . . . .	18
2.4.1 Text Classification . . . . .	18
2.4.2 Machine Learning Algorithms Used for Text Classification . . . . .	18
K-Nearest Neighbours . . . . .	18
Random Forest . . . . .	19
Ridge Linear Classifier . . . . .	20
Support Vector Machines . . . . .	21
Multi-label Classification Algorithms . . . . .	23
2.5 Part-Of-Speech Tagging . . . . .	23
2.6 Hierarchical Clustering . . . . .	24
2.6.1 Static and Dynamic Tree Cutting . . . . .	24
<b>3 State of the Art</b>	<b>27</b>
3.1 Development Tools Incorporating Stack Overflow Information . . . . .	27
3.2 Program Analyses for Stack Overflow Code Snippets . . . . .	30
3.3 Embeddings for Source Code Analysis . . . . .	33
3.4 Library Usage Comprehension . . . . .	35
3.4.1 Feature Uncovering . . . . .	35
3.4.2 API Usage Analysis . . . . .	36
3.5 Limitations and Opportunities for Improvement . . . . .	38
3.5.1 Current Limitations in API Type Resolution . . . . .	38
3.5.2 Current Limitations in API Feature Discovery . . . . .	39
3.6 Conclusion . . . . .	40

<b>4</b>	<b>API Type Resolution for Incomplete Code Snippets on Stack Overflow</b>	<b>41</b>
4.1	Introduction	41
4.2	Motivation	42
4.3	RESICO: A Type Resolution Approach for Incomplete Code Snippets	44
4.3.1	A Prime on RESICO	44
4.3.2	An Overview of Eclipse JDT for Facts Extraction	45
4.3.3	Training Process	45
4.3.4	Resolution Process	49
4.3.5	Providing Top-K Recommendations	51
4.4	Evaluation	51
4.4.1	Datasets Collection	52
	Internal Dataset	52
	External Datasets	53
4.4.2	RQ1. What are the best hyperparameter combinations for the classifiers used within RESICO?	55
4.4.3	RQ2. How well do COSTER and the RESICO classifiers perform on instances extracted from the dataset used for training?	58
4.4.4	RQ3. What is the performance of the COSTER and RESICO classifiers when evaluated on unseen datasets?	61
4.4.5	RQ4. How much time is needed to train COSTER and the RESICO classifiers?	63
4.4.6	RQ5. To what extent do ambiguities in simple names influence the performance of the approaches?	64
4.5	Discussion	66
4.5.1	Context-based Approaches to API Type Resolution	66
4.5.2	Limitations	68
4.5.3	Potential Impact	69
4.6	Threats to Validity	69
4.6.1	Threats to Construct Validity	69
4.6.2	Threats to Internal Validity	69
4.6.3	Threats to External Validity	70
4.6.4	Threats to Conclusion Validity	70
4.7	Conclusion	70
<b>5</b>	<b>Uncovering Library Features from Stack Overflow Posts</b>	<b>73</b>
5.1	Introduction	73
5.2	Motivation	74
5.2.1	Support for Comparing Libraries	74
5.2.2	Support for Exploring Ecosystems	75
5.3	AutoCat: Automatic Library Categorisation	76
5.3.1	Evaluation	77
5.3.2	Limitations of Category-based Approaches to Feature Uncovering	78
5.4	MUTAMA: Multi-label Library Tagging	78
5.4.1	Evaluation	80
5.4.2	Discussion	84
5.4.3	Limitations of Tag-based Approaches for Features Discovery	85
5.5	LiFUSO: An Approach to Discover Features from API Usages on Stack Overflow	85
5.5.1	Data Collection (Steps 1-3)	86

5.5.2	Data Processing (Steps 4-5)	87
5.5.3	Data Transformation (Steps 6-8)	88
5.5.4	Clustering, Selecting and Naming (Steps 9-11)	89
5.6	Instantiation of the LiFUSO Approach	90
5.6.1	Implementation	90
5.6.2	Graphical User Interface	92
5.7	Evaluation	94
5.7.1	Selection of Libraries	94
5.7.2	Features Terminology	95
5.7.3	RQ1. Which combination of SO answer attributes produces the most cohesive clusters?	95
5.7.4	RQ2. How similar are the automatically uncovered features to documented tutorial features?	97
5.7.5	RQ3. To what extent do the uncovered features that do not match documented tutorial features correspond to actual API usage in client projects?	100
5.8	Discussion	102
5.8.1	Clusters as Features	102
5.8.2	Limitations	104
5.8.3	Potential Impact	104
5.8.4	Case Study	105
5.9	Threats to Validity	107
5.9.1	Threats to Construct Validity	107
5.9.2	Threats to Internal Validity	107
5.9.3	Threats to External Validity	107
5.9.4	Threats to Conclusion Validity	108
5.10	Conclusion	108
<b>6</b>	<b>Uncovering Library Features based on Resolved Code Snippets</b>	<b>109</b>
6.1	Introduction	109
6.2	GitHub API Usages as RESICO Training Data	110
6.3	Extending the LiFUSO Dataset with Additional SO Posts	112
6.3.1	Part I: Extraction of GitHub Data and Re-Training of RESICO	113
6.3.2	Part II: Library Usage Determination and Relatedness Rule Learning	117
6.4	Evaluation	122
6.4.1	Datasets Collection	123
6.4.2	RQ1. How well does the rule-based classifier perform on the manually labelled dataset?	125
6.4.3	Rule-based Model Application on the Manually-labelled Dataset	128
6.4.4	RQ2. What is the impact of the new SO answer dataset on the features uncovered by LiFUSO?	128
6.5	Discussion	130
6.5.1	Newly Discovered Features	131
6.5.2	Limitations	131
6.6	Threats to Validity	132
6.6.1	Threats to Construct Validity	132
6.6.2	Threats to Internal Validity	132
6.6.3	Threats to External Validity	133
6.7	Conclusion	133

<b>7</b>	<b>Conclusion and Future Work</b>	<b>135</b>
7.1	Summary . . . . .	135
7.2	Contributions . . . . .	137
7.2.1	RESICO: API Resolution for Incomplete Code Snippets . . . . .	137
7.2.2	LiFUSO: Uncovering Library Features from their Stack Over- flow Usage . . . . .	138
7.2.3	Feature Uncovering on Resolved Code Snippets . . . . .	138
7.2.4	Advantages and Limitations of Data-Driven Approaches . . . . .	139
7.3	Future Work . . . . .	140
7.3.1	API Type Resolution . . . . .	140
7.3.2	Uncovering Library Features . . . . .	140
7.4	Concluding Remarks . . . . .	141
<b>A</b>	<b>Appendix</b>	<b>143</b>
	<b>Bibliography</b>	<b>147</b>

# List of Figures

1.1	The integration the contributions of this dissertation. . . . .	2
1.2	A preview of our feature exploration and library comparison tool. . . .	5
2.1	Example of a question on the Stack Overflow Q&A forum. <sup>1</sup> . . . . .	12
2.2	Example of an answer to a question in Stack Overflow. <sup>2</sup> . . . . .	12
2.3	Schema of all XML files related to the Stack Overflow community. <sup>3</sup> . .	15
2.4	Word2Vec text transformation into a vector representation. . . . .	17
2.5	CBOV neural network weights after a training. . . . .	17
2.6	Phrase-structured tree of a sentence [32]. . . . .	24
2.7	Static (left) and dynamic (right) tree cutting of a dendrogram. . . . .	25
4.1	Example type resolutions computed by RESICO. . . . .	44
4.2	Training based on a corpus of programs. . . . .	46
4.3	Transformation step used by the training process. . . . .	47
4.4	Resolution process for API type references in code snippets. . . . .	50
4.5	Overview of our evaluation approach. . . . .	52
4.6	The three most and least frequent FQNs in the gathered dataset. . . . .	53
4.7	Hyperparameter optimisation for the classifiers considered in RESICO. .	57
4.8	Performance of the models on the internal dataset. . . . .	60
4.9	Performance of the models on the three external datasets. . . . .	62
4.10	Similar FQNs by their context vectors. . . . .	67
5.1	Comparison of two libraries based on popularity metrics. . . . .	75
5.2	The steps followed by tools following the MUTAMA approach. . . . .	79
5.3	Distribution of the number of tags for the 4,088 sampled libraries. . . .	81
5.4	A binary array example, its predictions and posterior classifications into <i>TP</i> , <i>TN</i> , <i>FP</i> , <i>FN</i> . . . . .	82
5.5	Overview of the approach to extract features from SO posts. . . . .	86
5.6	Collecting the public interface of a library for all its versions. . . . .	87
5.7	The API usages extracted by our island parser and its lightweight analysis for the code snippet on the left. . . . .	88
5.8	Vectorisation of the textual information. . . . .	89
5.9	A name for an uncovered feature suggesting operations on cache. . . .	90
5.10	Described approach implemented by the LiFUSO tool. The instantia- tion steps are within the red square. . . . .	91
5.11	Front page of the LiFUSO tool with the Search feature tab activated. . .	93
5.12	Shared features for the studied libraries. . . . .	105
5.13	Unique features for the studied libraries. . . . .	106
6.1	API element coverage of the LiFUSO library <i>Quartz</i> from its depen- dent GitHub repositories. . . . .	112

6.2	Comparison of API element coverage for each LiFUSO library between the 50K-C and GitHub-dependent datasets. . . . .	113
6.3	Approach to recognising posts related to a library - Part I. . . . .	113
6.4	The extraction of API usages from GitHub repositories. . . . .	114
6.5	The process to create contexts from the previously extracted API usage information. . . . .	115
6.6	Steps to (re)train the RESICO classification with the previously GitHub-dependent dataset. . . . .	116
6.7	Approach to classify posts related to a library - Part II. . . . .	117
6.8	Steps to extract characteristics from answers with classifications. . . . .	118
6.9	Steps to train and apply a rule-based model to improve RESICO classifications. Highlighted in red are the data elements related to the usage of LiFUSO libraries. . . . .	120
6.10	Integration of the new approaches into the LiFUSO pipeline. . . . .	122
A.1	API element coverage of the LiFUSO library <i>Guava</i> from its dependent GitHub repositories. . . . .	143
A.2	API element coverage of the LiFUSO library <i>HttpClient</i> from its dependent GitHub repositories. . . . .	144
A.3	API element coverage of the LiFUSO library <i>JFreeChart</i> from its dependent GitHub repositories. . . . .	144
A.4	API element coverage of the LiFUSO library <i>JSoup</i> from its dependent GitHub repositories. . . . .	145
A.5	API element coverage of the LiFUSO library <i>PDFBox</i> from its dependent GitHub repositories. . . . .	145
A.6	API element coverage of the LiFUSO library <i>POI-OOXML</i> from its dependent GitHub repositories. . . . .	146

# List of Tables

2.1	Comparison of the different sources of API usage. . . . .	13
3.1	Techniques used to extract API references from code snippets. . . . .	29
4.1	Extracted information from the method <code>getIMEI()</code> in Listing 4 by RESICO. . . . .	47
4.2	Transformed API references and contexts from Table 4.1 by the Word2Vec models. . . . .	49
4.3	The last transformation step in the RESICO process. The previous context vector is further averaged with the API vector, and FQNs are converted into numbers. . . . .	49
4.4	Datasets used for the external evaluation of COSTER and RESICO. . .	54
4.5	Hyperparameters of the classifiers and their search space configuration	56
4.6	Computational cost of the approaches. Time is measured in hours (h), minutes (m), seconds (s) and milliseconds (ms). . . . .	63
4.7	Ambiguity analysis for COSTER and RESICO-KNN trained models on the external datasets. . . . .	65
4.8	The accuracy of the best models per approach shown per library in the <i>StatType-SO</i> dataset. Highlighted in green and red are the largest successes and failures, respectively. . . . .	68
5.1	Scores achieved by the trained classifiers on the automatic classification of libraries. Highlighted in bold are the best models according to their F1 score. . . . .	77
5.2	Performance metrics of MUTAMA instantiated with different multi-label classifiers. The best results for each metric are highlighted in bold. . . . .	81
5.3	Multi-tag predictions made by the best trained multi-label model. . . .	84
5.4	SO code snippets making use of the libraries. . . . .	95
5.5	Top- <i>K</i> scores for all combinations of library attributes. Between parentheses is the obtained Silhouette score. . . . .	97
5.6	Analysis of the matched features per library. . . . .	99
5.7	Client projects information from GitHub. . . . .	101
5.8	Newly matched features from GitHub client projects. . . . .	101
5.9	Detailed overflow scores per library. . . . .	102
6.1	Extracted fields from the GitHub-dependent repositories and their descriptions. . . . .	115
6.2	Group of answer characteristics extracted to improve RESICO's classifications. . . . .	119
6.3	Dependent GitHub repositories for each LiFUSO library before and after the cleaning. . . . .	123

6.4	Performance metrics (Left) and confusion matrix (Right) of the trained rules on the manually labelled dataset. . . . .	127
6.5	Previous and newly collected SO answers that use a LiFUSO library. .	128
6.6	Comparison between old and new features. . . . .	129



# List of Listings

1	Examples of different forms of API usages from three libraries. . . . .	10
2	Javadoc documentation for a library method. . . . .	10
3	Real code snippets with different issues related to their incompleteness.	43
4	Running example for explaining RESICO. . . . .	46
5	Examples of features uncovered for JFreeChart. . . . .	103
6	Examples of features uncovered for PDFBox. . . . .	103
7	Examples of non-feature clusters uncovered. . . . .	104
8	Command to obtain all library dependencies from a Java repository into a specific folder. . . . .	114
9	Trained rules of the J48 decision tree based on the manually labelled dataset. . . . .	126
10	Some examples of newly discovered features for the LiFUSO libraries.	131
11	Examples of improved features for the LiFUSO libraries. . . . .	132



# List of Abbreviations

**AST:** Abstract Syntax Tree  
**API:** Application Programming Interface  
**AUG:** API Usage Graph  
**BoW:** Bag-of-Words  
**CDG:** Control Dependency Graph  
**CFG:** Control Flow Graph  
**CAN:** Convolutional Attention Network  
**CBOW:** Continuous Bags-Of-Words  
**CNN:** Convolutional Neural Network  
**DDG:** Data Dependency Graph  
**FQN:** Fully Qualified Name  
**GNN:** Graph Neural Network  
**GCN:** Graph Convolutional Network  
**HMM:** Hidden Markov Models  
**IDE:** Integrated Development Environment  
**IR:** Intermediate Representation  
**JAR:** Java ARchive  
**JSON:** JavaScript Object Notation  
**KNN:** K-Nearest Neighbours  
**LOF:** Local Outlier Factor  
**NLP:** Natural Language Processing  
**NN:** Neural Network  
**NP:** Noun Phrase  
**PDG:** Program Dependency Graph  
**POS:** Part Of Speech  
**PPA:** Partial Program Analysis  
**SO:** Stack Overflow  
**S2S:** Sequence-to-Sequence  
**SMO:** Sequential Minimal Optimisation  
**SVC:** Support Vector Classifier  
**SVM:** Support Vector Machines  
**RF:** Random Forest  
**RNN:** Recurrent Neural Network  
**RL:** Ridge Linear  
**TF-IDF:** Term Frequency-Inverse Document Frequency  
**TPE:** Tree of Parzen Estimators  
**UI:** User Interface  
**VB:** Verb Phrase



# Chapter 1

## Introduction

Libraries date back almost to the first conceptualisations of a digital programmable computer. An early pioneer in computer science such as Charles Babbage suggested in 1838 (and later published in 1888) a way of saving and *reusing* computer operations on separate cards [26]. Years later in 1947, von Neumann resumed this idea and proposed magnetic recordings to store code to be later reused [45]. This served as inspiration to Wilkes and his team to construct the Electronic Delay Storage Automatic Calculator (EDSAC) computer, released in 1949. Programs in EDSAC contained a main program and a sequence of subroutines copied from a cabinet storing punched tape libraries [143]. These early reused subroutines were physically stored, hence the *library* term is literally a reference to a physical place with external code.

The designers of the first programming languages also considered libraries as an important element. In these cases, a *library* is referred to software code outside the main program. COBOL, for example, included some capabilities for libraries [142], while FORTRAN introduced the concept of *subprogram* which resembles a library [144], and Ada incorporates *packages* into its design and implementation [29]. Contemporary developers frequently use libraries to collaborate on, distribute, and reuse code. Programming languages such as Java and JavaScript are two of the most popular ones. The number of libraries being supported by these languages is considerably large, e.g., currently above the 500,000.<sup>1</sup>

As the number of libraries increased throughout the years, easily-accessible library repositories become necessary. Examples of such repositories include Maven<sup>2</sup> for JVM-based programming languages (e.g., Java, Scala, Kotlin) and NPMJS<sup>3</sup> for JavaScript-based programming languages (e.g., JavaScript, TypeScript). Every day many libraries and library updates are uploaded to these repositories.

An ever-growing number of libraries signifies that developers have more options to choose from. The days where a developer had a single library for a task (e.g., *mocking*) are long gone. On the *MVNRepository*<sup>2</sup>, developers have 50 Java libraries at their disposal for the category *Mocking*.<sup>4</sup> These numbers will continue to increase as new libraries continue being developed and uploaded to the software repositories.

---

<sup>1</sup> <http://www.modulecounts.com/>

<sup>2</sup> <https://mvnrepository.com>

<sup>3</sup> <https://www.npmjs.com/>

<sup>4</sup> <https://mvnrepository.com/open-source/mocking>: accessed on December 6<sup>th</sup>, 2023.

The growth of libraries represents a problem for developers as they have to choose from multiple candidates. This selection is very often not properly performed as libraries are selected not by the features they offer, but by popularity metrics such as number of downloads. Therefore, we envision library repositories where the features offered by libraries are first-class citizens. For example, instead of ranking libraries based on the number of downloads<sup>5</sup> we picture library repositories where features are displayed and explored by developers according to their requirements. This dissertation contributes building blocks to realise this vision through automated approaches that mine API usages to compute an enumeration of library features.

The extraction of features requires a source from where to mine and analyse candidates of library features. We selected Q&A fora, and more specifically Stack Overflow as the mining source for features of a library (cf. Section 2.1). However, code snippets in Stack Overflow are very often incomplete and syntactically incorrect (cf. Section 4.2). We propose our API type resolution approach RESICO to resolve the fully qualified names within incomplete code snippets (cf. Chapter 4). A means to enumerate the features provided by a library is currently missing from library repositories. Such an enumeration could support developers in feature exploration and library comparison. To extract library features from Stack Overflow, we propose our LiFUSO approach and tool (cf. Chapter 5). LiFUSO focuses on a group of Stack Overflow posts that explicitly mention the name of a library in their tag list. We therefore also extend the dataset considered by LiFUSO by merging our previous approach RESICO into the former's pipeline (cf. Chapter 6). Figure 1.1 depicts how Chapter 6 integrates the RESICO and LiFUSO approaches introduced by Chapter 4 and Chapter 5 respectively.

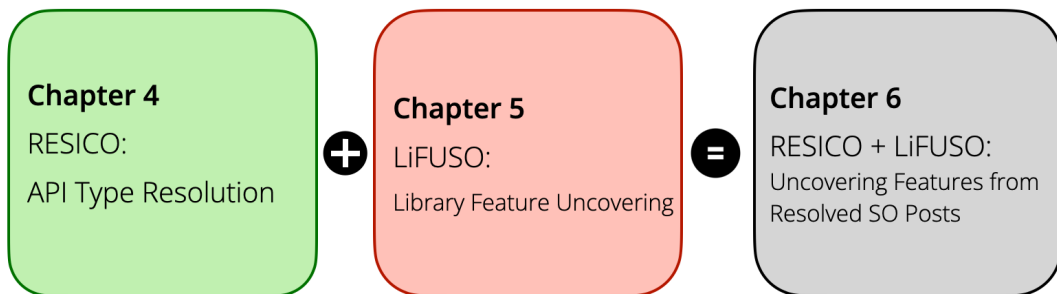


FIGURE 1.1: The integration the contributions of this dissertation.

The remainder of the Introduction is structured as follows. Section 1.1 defines the problem statement in the context of this dissertation through elements that explain the relation between the library feature mining from API usages and the analysis of incomplete code snippets. We highlight that feature discovery and exploration is required to avoid biased and unfair library selection and also to enable library comparisons according to their provided features. Section 1.2 explains our approaches and their research objectives. Section 1.3 respectively list the contributions of this dissertation and their supporting publications. Finally, Section 1.4 presents the outline of this dissertation.

<sup>5</sup> <https://mvnrepository.com/open-source/io-utilities>

## 1.1 Problem Statement

The selection criteria developers have to resort to when selecting a library from multiple candidates targetting the same domain (e.g., *Mocking*) suffers from two problems. First, developers tend to be based on the popularity of the library such as the number of downloads, the number of maintainers, etc. These criteria remain at the level of metadata and are biased towards the most popular libraries. Second, these criteria do not capture the suitability of the library for the task at hand. This dissertation provides initial steps towards using automatically enumerated library features as a selection criterion.

A **library feature** has been previously defined by Kanda et al. [64] as “*a set of frequent API calls, accompanied by a corresponding name*”. Such a definition consists of a code part, i.e., frequently invoked API methods, and a natural language part, i.e., the name of the feature. The extraction and analysis of features provided by libraries is a recent research area in software engineering. Previous works [3, 10, 51, 110] have explored uncovering the features provided by Android applications through an analysis of the user reviews. However, their analyses are limited to the natural language processing of the reviews and does not contribute to the code elements in the definition by Kanda et al. [64]. We investigate the extraction and analysis of library features from posts on the Stack Overflow (SO) platform. To overcome the incompleteness and syntactical incorrectness of the code in these posts, we also propose an approach capable of resolving API types in code snippets.

### Library Client Analysis for Feature Discovery

The analysis of clients that use the API of a library has been frequently employed to extract knowledge about the library. Examples include recommending API sequences for a certain functionality [48], providing similar API usage examples to familiarise with a library [47] or detecting API usage anomalies [93]. Natural language processing is required for the automatic uncovering of the features provided by a library. However, only two approaches combine the analysis of library clients with natural language processing to enumerate the features provided by a library. Kanda et al. [64] extract a set of co-occurring API usages, and label them manually to form features. Shen et al. [113] automatically process natural language information from SO posts, but, mine information about API usages from GitHub repositories and not from SO posts. It is evident that API usage analysis approaches have not been targetting feature discovery. Those proposing similar techniques in this research area have either not fully automated their approach (i.e., Kanda et al. [64]) or have not considered API usage focused on an individual feature such as those found within SO code snippets (i.e., Shen et al. [113]).

### Analysis of Incomplete Code

Incomplete code can be found in multiple sources such as commit diffs, video and webpage tutorials, and SO snippets. The analysis of incomplete code is relevant since it might be accompanied by other valuable information. This is indeed the case for SO code snippets as they relate to the natural language information in their surroundings. Analysing the API usage within SO code snippets can be challenging due to their often incomplete and syntactically incorrect nature. Nonetheless, several

works (e.g., [37, 72, 100, 128, 131]) have demonstrated that the content on the SO platform is very rich in information.

To mine information from SO posts such as API usage, other works have also proposed techniques to address incompleteness of code snippets (e.g., [36, 40, 95, 106, 118, 124, 150, 163]). Many of these works are based on the Eclipse JDT parser or on PPA [36], which only have access to the information provided in the code snippet itself. Other works, in contrast, leverage learning-based techniques to augment the snippets with its missing information (e.g., import statements to qualify simple names of API types). Example techniques include source-to-target translation [95], or context-based learning [106], etc. We deem context-based techniques more promising to resolve missing information as they can leverage the code surrounding the missing information. However, we noticed that the design space for the actual context to use is currently underexplored. This brings us to our problem statement:

*There is a need for automated approaches that uncover the library features provided by a library through the mining of public information regardless of code incompleteness and incorrectness*

## Towards Feature-based Library Selection

In a survey of 115 developers, Vargas et al. [133] found that developers rely on library metrics such as the number of downloads, the number of GitHub stars, etc. An automated approach for extracting the features provided by a library could bring many benefits. For example, library repositories could enable users to explore their libraries based on features provided. Features can be displayed as frequently API elements and combinations thereof described by a name or other natural language text facilitating their comprehension. In this way, developers can more easily select the library with the exact features that match their requirements. Additionally, similar features provided by different libraries can be compared based on their case of use, the number of lines of code, etc. This ultimately contributes to higher-quality library selection.

Figure 1.2 shows a preview of the user interface of our tool for feature exploration and library comparison. This tool is described in more detail in Section 5.6, and Section 5.8.4 demonstrates its utility with a use case. As observed in Figure 1.2, a list of features from various libraries is presented to users for them to explore. Each feature is shown with the name of the library it belongs to on top, followed by a description of the feature in terms of frequent verb-noun pairs and the most frequent corresponding API references required for reusing the feature. Next to the API references are frequency percentages.

A user can explore features by browsing through all features extracted by the tool. Moreover, a user can search for a feature by its description, for example, “*filter list*” or “*create chart*” which will reduce the list of available features to those matching the queried terms. Once a feature is found by a user, it can be included into a project by adding the corresponding import statements and the missing *glue code* (i.e., variable declarations, parameter additions, etc.).



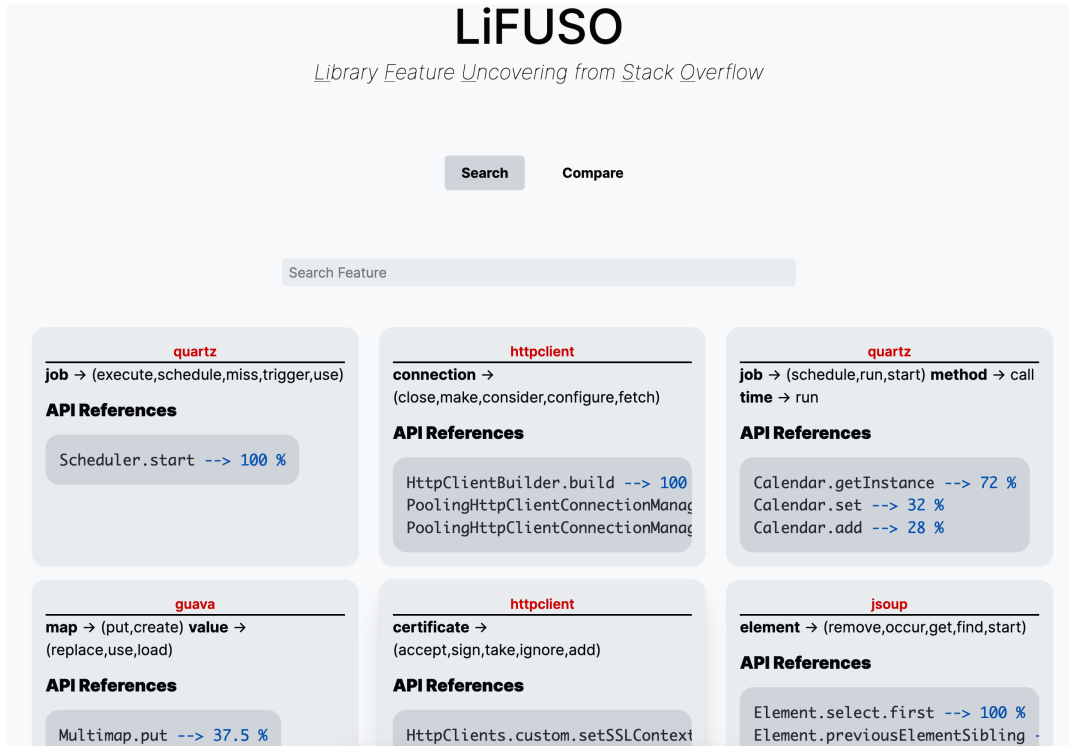


FIGURE 1.2: A preview of our feature exploration and library comparison tool.

## 1.2 Overview of the Approach

Our research goal is the automated uncovering of library features by mining and processing public information. To this end, we propose a number of related approaches to advance the state of the art in the domain:

1. We investigate the use of Maven repository categories as a coarse description of library features. We design an approach, called AutoCat, that learns from the bytecode implementation of libraries that have already been categorised and recommends a suitable category for any unseen library (cf. Section 5.3).
2. We extend the above approach to tags in the Maven repository as they represent a more fine-grained library features. This new approach, called MUTAMA, explores the capabilities of multi-label classification algorithms in recommending a set of tags for libraries (cf. Section 5.4).
3. As the former approaches only support coarse-grained descriptions of features (i.e., categories and tags) we design another approach that supports API-level feature descriptions. LiFUSO is architected as a series of steps analysing Stack Overflow posts that carry the name of the target library among their tags. LiFUSO parses the code snippet of such post using a lenient and robust parser to avoid parsing errors due to code incompleteness. The library's API usages extracted from those posts are clustered according to their similarity. Clusters are filtered according to the frequency of the API elements within them. The remaining clusters are named in order to form library features by processing the natural language text surrounding the clustered code snippets (cf. Section 5.5).

4. As our source of library feature information, SO code snippets have the advantage of being focused on the task mentioned in the corresponding question of the post. However, SO code snippets also have disadvantages such as code incompleteness and code incorrectness. We target the lack of import statements that would otherwise qualify the simple names of used API types through a dedicated API type resolution technique. The API type resolution problem is addressed as a text classification problem where an API type reference and the context into which appears are the input text, and the fully qualified name (FQN) of the referenced API type is the output class to predict. Our approach RESICO learns the most likely FQN of an API type when referenced in a particular context and is trained on the complete code within GitHub repositories. RESICO recommends FQNs based on the context similarity of API type references in incomplete snippets (cf. Chapter 4).
5. LiFUSO is limited to SO posts that carry the name of a library in their list of tags. However, this limitation can be overcome by using RESICO to recognise SO posts that use the library in their code snippet. We design a combined approach in which we retrain RESICO on a corpus of API usages of the LiFUSO libraries. The resulting model is applied to the entire set of Java SO code snippets to expand the number of posts considered by LiFUSO. As a result, LiFUSO is able to uncover more features and also improve their textual description (cf. Chapter 6).

### 1.3 Contributions and Supporting Publications

This dissertation presents contributions in two main research areas: i) **automated library feature uncovering** and ii) **API type resolution for incomplete code snippets**. Specifically, the contributions are supported by five international peer-reviewed publications [134–138]:

- Velázquez-Rodríguez, Camilo and De Roover, Coen, (2020). Automatic library categorization. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (SoHeal)*, pages 733-734. (Workshop paper)

This workshop paper proposes an automated approach to categorising libraries shared in the Maven repository index into predefined categories.<sup>6</sup> The approach supports existing as well as newly added libraries. Based on the embeddings of their code the machine learning model of the approach learns how to categorise libraries as a form of text classification (cf. Chapter 5). This paper presents the foundation for our later work about feature uncovering.

- Velázquez-Rodríguez, Camilo and De Roover, Coen, (2020). MUTAMA: An Automated Multi-label Tagging Approach for Software Libraries on Maven. In *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 243-247.

This paper presents a similar approach to the previous workshop paper. However, in this case, the target is the recommendation of multiple tags for the libraries in

<sup>6</sup> <https://mvnrepository.com/open-source>

the Maven repository.<sup>7</sup> Our MUTAMA approach (cf. Chapter 5) incorporates multi-label classification algorithms and word vectorisation. We considered and evaluated multiple algorithms which contributed to the design of our feature uncovering approach.

- Velázquez-Rodríguez, Camilo, Constantinou, Eleni and De Roover, Coen, (2022). Uncovering Library Features from API Usage on Stack Overflow. In *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 207-217.
- Velázquez-Rodríguez, Camilo, Constantinou, Eleni and De Roover, Coen, (2022). LiFUSO: A Tool for Library Feature Unveiling based on Stack Overflow Posts. In *Proceedings of the 38th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 489-493.

These two papers propose, implement and evaluate our automated library feature uncovering approach. The first paper presents the initial approach to uncovering features from their usages in SO posts. The approach is based on the clustering of similar library API usages and a natural language processing of the text surrounding them (cf. Chapter 5). An implementation of the approach is discussed in the second paper. This tool paper demonstrates the main use cases of LiFUSO and also presents a case study into its application to find similar and unique features across libraries within the same domain.

- Velázquez-Rodríguez, Camilo, Di Nucci, Dario and De Roover, Coen, (2023). A Text Classification Approach to API Type Resolution for Incomplete Code Snippets. In the *Science of Computer Programming Journal*, Volume 227, article 102941.

The approach to resolve simple names of API types to their FQN in incomplete code snippets from Stack Overflow is introduced in this paper. The approach, RESICO, is motivated by the fact that Java code snippets on Stack Overflow are often incomplete with missing import statements to qualify simple names. RESICO approaches API type resolution as a text classification problem by vectorising the surrounding context of an API reference that misses its FQN. Given an API reference, a machine learning classifier computes the most likely FQN for the context it resides. RESICO outperforms the state-of-the-art approach in an extensive evaluation (cf. Chapter 4).

## 1.4 Outline of the Dissertation

The remainder of this dissertation is structured as follows:

**Chapter 2: Background** We provide a detailed explanation of the main sources where API usages can be found as well as our reasons for selecting Stack Overflow as the source from where to mine for library features. We describe the main techniques used throughout the dissertation ranging from text transformation to hierarchical clustering.

---

<sup>7</sup> <https://mvnrepository.com/tags>

**Chapter 3: State of the Art** We survey the state of the art in program analyses for SO code snippets, machine learning embeddings for source code, and API usage analysis. The approaches proposed in this dissertation are inspired by previous work in these research areas and overcome the shortcomings of the state of the art.

**Chapter 4: API Type Resolution for Incomplete Code Snippets on Stack Overflow** We motivate the need for an approach that resolves missing information in incomplete code snippets. We describe RESICO, our machine-learning based approach, which learns the FQNs that simple names of API types resolve to in a given API usage context. The approach is trained on a publicly available dataset of Java programs and is subsequently evaluated. The evaluation is carried out thoroughly on internal and external datasets, and by comparing RESICO's resolutions to those made by the state-of-the-art approach COSTER which has also been trained on the same dataset. RESICO outperforms COSTER in the conducted experiments, although, it is slightly slower due to its context-learning nature. RESICO's type resolution capabilities are later employed in this dissertation to identify SO code snippets that use a particular library.

**Chapter 5: Uncovering Library Features from Stack Overflow Posts** An automated approach to library feature uncovering is proposed in this chapter. We motivate our LiFUSO approach by the need to support the selection of libraries and their comparison. Two approaches are initially proposed to automatically categorise libraries and recommend tags for them. These can be seen as more high-level descriptions of the features provided by a library. Later we introduce our API-level feature discovery approach built on top of many of the techniques described in the Background chapter. Our approach is also evaluated with respect to features of libraries as documented in tutorials and cookbooks. Finally, the chapter also describes an instantiation of LiFUSO into a working tool prototype that supports library feature exploration and comparison. We demonstrate its use through a case study.

**Chapter 6: Uncovering Library Features based on Resolved Code Snippets** The previous implementation of LiFUSO only considers SO posts that have been tagged with one of the libraries under analysis. To extend the analysis to the full set of SO Java code snippets, we propose an approach that integrates RESICO with LiFUSO. We first analyse the extent to which the API of the LiFUSO libraries is covered by the original dataset RESICO was trained on, and then retrain RESICO on a more focused dataset. As the retrained RESICO suffers from a limited vocabulary, we design a rule-based classifier to assist RESICO on ambiguous simple names. This rule-based approach is evaluated and confirmed to be helpful in assessing the correctness of RESICO's API type resolutions. Using RESICO complemented by the rule-based classifier, we collect a much larger dataset of SO answers for LiFUSO to process. As a result new features are uncovered and existing feature descriptions are improved.

**Chapter 7: Conclusion and Future Work** We conclude the dissertation by revisiting the problem statement and listing potential avenues for future work.

# Chapter 2

## Background

This chapter provides the background for the remainder of the dissertation. We start with a closer look at the problem and solution domains of our dissertation: the sometimes suboptimal documentation of software libraries and the presence of usage of their corresponding API in developer fora. Natural-language techniques for analysing and transforming texts are reviewed. Furthermore, supervised machine learning algorithms for automated text classification are explained in detail since they form the foundation for the API usage resolution contribution of this dissertation. Unsupervised machine learning algorithms for clustering are also reviewed as they form the foundation for the library features discovery contribution of the dissertation.

### 2.1 API Usages in Library Documentation, Messaging Platforms and Q&A Fora

It is common for reusable software to provide access to its features through a public interface called Application Programming Interface (API), while shielding users from its internal implementation details. In general, there are two main forms of APIs: I) endpoints exposed by web services<sup>1</sup>, and II) the public interface of libraries.<sup>2</sup> This dissertation focuses on the analysis of the latter form of APIs.

How to use the API from such a library can vary depending on multiple factors such as the overall design of the library, the type of features it offers for reuse, the programming paradigm and language is implemented in, etc. For example, some libraries such as the popular Google Guava<sup>3</sup> have an API for which a single method call suffices to perform a feature (e.g., the concatenation of collections). Other libraries, such as JFreeChart<sup>4</sup> require a sequence of API calls that together perform a task (e.g., draw a pie chart). Some of these libraries (e.g., Apache HttpClient<sup>5</sup>) provide so-called fluent API (i.e., chained method calls) to achieve an objective (e.g., send HTTP authentication credentials). Listing 1 illustrates client usages of these APIs.

<sup>1</sup> <https://github.com/public-api-lists/public-api-lists>

<sup>2</sup> <https://guava.dev/releases/snapshot-jre/api/docs/com/google/common/graph/package-summary.html>

<sup>3</sup> <https://guava.dev>

<sup>4</sup> <https://jfree.org/jfreechart>

<sup>5</sup> <https://hc.apache.org/httpcomponents-client-5.2.x>

```

1 // Guava multiple examples of single method calls
2 Iterators.concat(...);
3 Strings.repeat(...);
4 Graphs.reachableNodes(...);
5
6 // JFreeChart minimum group of calls to create a pie chart
7 DefaultPieDataset.setValue(...);
8 ChartFactory.createPieChart(...);
9
10 // HttpClient chain of method calls to send authentication credentials
11 CredentialsProviderBuilder.create().add(...).build();

```

LISTING 1: Examples of different forms of API usages from three libraries.

```

1 /**
2  * Returns the character at the specified index. An index
3  * ranges from 0 to length() - 1.
4  *
5  * @param index the index of the desired character.
6  * @return the desired character.
7  * ...
8  */
9 public char charAt(int index) {
10     ...
11 }

```

LISTING 2: Javadoc documentation for a library method.

Such examples of API usages could be illustrative for other library clients and may point them to proven recurring API usage patterns. In this dissertation, we collect and analyse such API usage patterns to not only support developers in reusing the features provided by a library, but also in comparing libraries based on the features they provide and the API through which it can be reused.

Information about the API of a library and its usage can be obtained from various sources. One of the most common sources is the documentation of the library itself. The documentation should provide a description in natural language of all interfaces, classes, methods, and fields that are part of the API. Library developers are supported in this task by documentation generators, such as Javadoc, which analyse the structured strings of documentation placed in the implementation above each library element.<sup>6</sup> Listing 2 provides an example.

Documentation provided by library developers has been the only source of information about the library's API usage for many years. In recent years, however, several alternative sources of API usage information have become available. These sources include README files shipped with open-sourced library implementations, messaging software, and Question and Answer (Q&A) fora for developers.

<sup>6</sup> <https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#javadocTags>

README files can often be found in the public version repository (e.g., GitHub) of a library. README files are intended to provide a general overview of the library as well as some insights into its installation, usage, features offered, etc. The files do not have a predefined structure (in contrast to auto-generated library documentation) and they can be as detailed as the library developers desire. Some READMEs include examples of the library's API usage, although it is not common to see the entire API detailed in a README file.

Messaging software for developers such as Discord<sup>7</sup>, Slack<sup>8</sup>, and Gitter<sup>9</sup> are another emergent source of API usage information. These applications are characterised by their support for extensive dialogues about a library-related topic between library developers and users. Discord and Slack support multiple communication channels. Their users might, therefore, assign each channel to a different topic about the library. Gitter, in contrast, supports only a single channel. The channel topic can be very general such as *programming*<sup>10</sup> or be dedicated to a specific library or technology such as *scikit-learn*<sup>11</sup>, *docker*<sup>12</sup>, etc.

Q&A fora such as Code Ranch<sup>13</sup>, Code Project<sup>14</sup> and Stack Overflow<sup>15</sup>, provide yet another source of information about a library and its API usage. Forums differ from the previous information sources in that their posts focus on individual questions users have about a particular feature of a library or a programming language. For example, many questions in fora have the format of “*How to do X with Y?*”, where X is a feature offered by library Y. Sometimes, the question includes a (partially-functional) code snippet to illustrate what the user is trying to achieve. Code snippets in answers posted by the community either complete the original snippet or illustrate the required API usage from scratch.

Figure 2.1 shows an example of a question about “*How to merge PDFs files?*” posted on the SO Q&A forum. Relevant parts of the posts have been highlighted in the colour red. The **Title** is one of the main parts and where a user states the topic of a post. The other correspond to the **publication** and **modification dates**, as well as the **number of views**. The **number of votes** depends on whether the community considers the question relevant (i.e., the higher the votes, the more relevant). The **tags** part of the post is a list of up to five tags from a number of forum-defined ones. Finally, the main content of the question consists of the **text part of the question**, and the **code part of the question**. As their names indicate, the text part refers to a natural language description of the problem a user has, and the code part (if any) exemplifies the previous text in any programming or markup language (e.g., Java, Python, HTML, XML, etc.). Multiple snippets could be present in a question, alternating text and code to exemplify better the user's intention.

---

<sup>7</sup> <https://discord.com>

<sup>8</sup> <https://slack.com>

<sup>9</sup> <https://gitter.im>

<sup>10</sup> <https://gitter.im/chat-rooms/programming>

<sup>11</sup> <https://gitter.im/scikit-learn/scikit-learn>

<sup>12</sup> <https://gitter.im/docker/docker>

<sup>13</sup> <https://coderanch.com>

<sup>14</sup> <https://www.codeproject.com>

<sup>15</sup> <https://stackoverflow.com>

<sup>16</sup> <https://stackoverflow.com/questions/3585329>



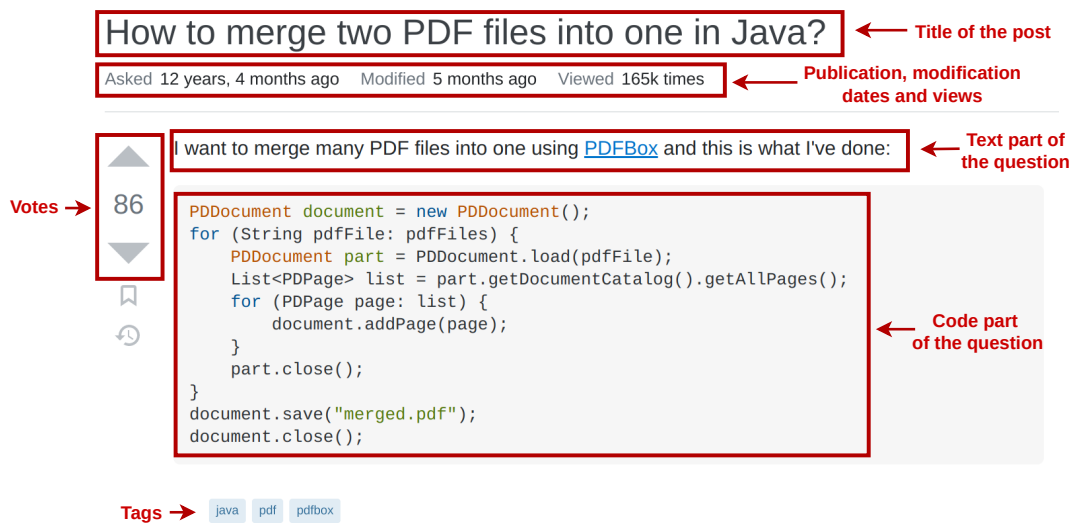


FIGURE 2.1: Example of a question on the Stack Overflow Q&A forum.<sup>16</sup>

Next to the question multiple answers can be part of a post. An answer has a similar structure to the question formulated in the post. The main purpose of an answer is not only to help the user who originally asked the corresponding question, but also to help other developers who encounter the same issue. As a result, SO has become a massive knowledge base for developers. Figure 2.2 displays an example of an answer to the previous question in Figure 2.1.

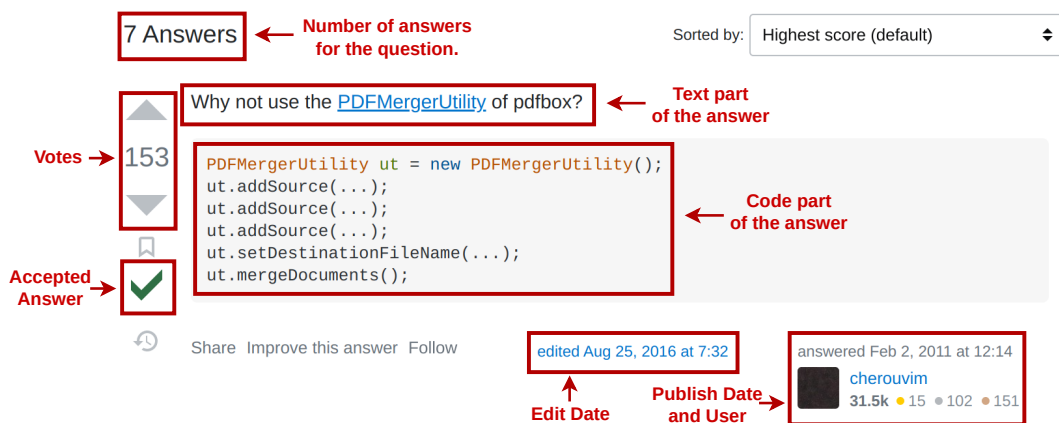


FIGURE 2.2: Example of an answer to a question in Stack Overflow.<sup>17</sup>

In total, this question has received seven answers. The main parts of the answer are similar to those of a question such as the **number of votes**, the **text** and **code** parts and the **publication** and **edit** dates. Unique to answers is an optional green check mark indicating whether the answer is the one among all posted answers that has been accepted by the user who made the question. This attribute functions as a form of both answer curation and stimulus.

<sup>17</sup> <https://stackoverflow.com/questions/4874334>



The text and code parts of the answer point to a solution for the posted question. They form the main source of information for the data-driven tool support for library users which we will develop in this dissertation.

API Source	Structured	Feature-focused	Updated	Ecosystem
Generated documentation	+	–	–	+
README files	–	–	+	–
Messaging software	–	+	+	–
Q&A fora	+	+	+	+

TABLE 2.1: Comparison of the different sources of API usage.

Table 2.1 compares the four sources of API usage information discussed above. Column **Structured** specifies whether the source is structured and can therefore easily be analysed through automated means. **Feature-focused** indicates those sources where a group of related API usages is likely to be found. **Updated** highlights those that are most likely kept up to date throughout time. Finally, column **Ecosystem** states the reach of the information source or how much of the ecosystem it covers.

Auto-generated library documentation is typically well-structured (cf. Listing 2). This structure is easily exploited for analysis. Moreover, most libraries feature such documentation as it is considered a best practice among library developers. However, the documentation of a library targets its public API at the finest granularity, describing each of its elements (i.e., methods, fields, classes, etc). This is convenient whenever help for a specific component is required. However, it rarely offers information about API elements that need to be used together in idiomatic ways due to its very fine-grained nature. In other words, library documentation was not designed to describe API usage sets. Moreover, documentation may not always be kept up to date for every library release as developers consider it a cumbersome task.

README files, as previously mentioned, do not have a defined structure since they are basically an open-format document for library installation and configuration. Due to their informative nature of providing general descriptions, it is rare to find library API usage patterns in them. Nonetheless, as library projects evolve, maintainers usually keep README files up to date. The reach of READMEs to an ecosystem-wide scope depends on the availability of libraries as open-source projects, which is currently not the case.

Messaging software, as shown in Table 2.1, is a less-structured source of library information. For example, it is not straightforward to retrieve all messages that belong to a particular conversation since conversation threads intertwine. Advanced techniques to untangle messages from different conversations are required. API usage patterns might appear in code examples within a conversation; however, it is very common to have conversations about a single API call.<sup>18</sup> Several libraries have started adopting Slack, Gitter or Discord to be closer to their users and provide feedback more quickly (e.g., Akka<sup>19</sup>, Pandas<sup>20</sup>, Django<sup>21</sup>).

<sup>18</sup> <https://discord.com/channels/632150470000902164/1008846246661849179>

<sup>19</sup> [https://matrix.to/#/#akka\\_akka:gitter.im](https://matrix.to/#/#akka_akka:gitter.im)

<sup>20</sup> <https://pandas.pydata.org/docs/dev/development/community.html#community-slack>

<sup>21</sup> <https://discord.com/invite/TP9yQBwyxF>

Q&A fora impose more structure on conversations as shown in Figure 2.1 and Figure 2.2. This structure facilitates extracting library information. Furthermore, API usage patterns are commonly found in Q&A fora since as conversations are structured as one or more answers to a well-scoped question. Therefore, the code snippets and the natural-language text within a post are related to the same topic. In terms of API updates, authors of both questions and answers can update their content to the latest version of an API, but, unfortunately, this is not a common practice. For example, Zerouali et al. [153] report that code snippets with import statements make use of an 80.7% of libraries with outdated versions. Finally, Q&A fora, like Messaging software, have the potential to host any conversation about any library in an ecosystem. Although this is not yet the case for Stack Overflow (the most popular Q&A forum), it already contains many questions and answers for numerous libraries in the Java ecosystem. For example, there is currently 2 million questions with the *java* tag.<sup>22</sup>

**Based on this information, we deem Q&A fora a promising source of API usage information. More specifically, we will use Stack Overflow as our information source because it is the most popular among its relatives, containing a vast number of posts (e.g., over 55 million) to be analysed.**

## 2.2 Stack Exchange and SOTorrent Stack Overflow Dataset Dumps

Stack Overflow started in 2008 as a company launched by Jeff Atwood and Joel Spolsky. Since then, many communities (e.g., AI, Bioinformatics, Space, Unix, etc.) have congregated into the umbrella fora called Stack Exchange<sup>23</sup>, which includes the Stack Overflow forum. These communities discuss topics related to their name, being Stack Overflow the most popular with over 55 million posts. Every three months a snapshot of the contents in all communities is created and made publicly available in the Internet Archive.<sup>24</sup> The snapshot contains compressed (i.e., 7zipped) XML files with an internal structure. Stack Overflow, as the rest of communities, also has a schema depicted in Figure 2.3.

The extracted schema reflects the relations between different files or tables. Moreover, the figure provides a description of the fields for each file and their types (e.g., field *Score* in file *Posts* of type *int*). In this dissertation we use the contents of the files *Posts* and *Tags*. The *Posts* file contains important fields for our research such as *PostTypeId*, indicating the type of post (e.g., question or answer); *Title*; *Tags*; *Body* of the post, among others.

Based on this dataset of publicly available information, SOTorrent<sup>26</sup> is proposed by Baltes et al. [12] with the goal of investigating the evolution of SO posts. SOTorrent has been referenced multiple times by previous works that analyse SO posts for software engineering tasks. In contrast to Stack Exchange compressed files, SOTorrent provides an already configured MySQL database where each table corresponds to

<sup>22</sup> <https://stackoverflow.com/tags>

<sup>23</sup> <https://stackexchange.com/tour>

<sup>24</sup> <https://archive.org/download/stackexchange>

<sup>25</sup> The figure is extracted from here: <https://sedeschema.github.io/>

<sup>26</sup> <https://empirical-software.engineering/sotorrent/>

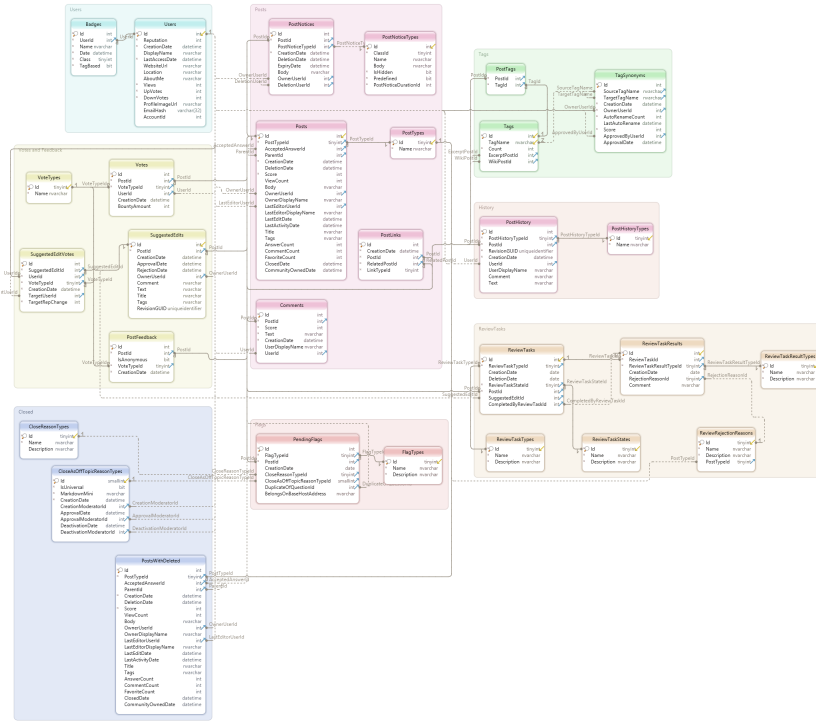


FIGURE 2.3: Schema of all XML files related to the Stack Overflow community.<sup>25</sup>

an *XML* file. In such a way, queries are much faster in the database than in a file with millions of rows and information to parse.

The research conducted in this dissertation uses both SOTorrent and Stack Exchange *XML* files. SOTorrent is used in Chapter 5 to extract features from SO posts. The Stack Exchange files are employed in Chapter 6 to extend the dataset of posts referring to a library usage. The reason for the selection of the latter in Chapter 6 is the access to the most recent posts, being SOTorrent discontinued. We had to implement the mechanisms for accessing and querying, which were already facilitated by SOTorrent.

SO posts are used throughout the whole dissertation. **Specifically, Chapter 4 proposes an API resolution approach for incomplete SO code snippets. Chapter 5 proposes an approach to uncover library features from API usages in SO posts. Chapter 6 proposes an approach that extracts SO posts characteristics to train a model that improves the API type predictions on incomplete code snippets.**

## 2.3 Text Transformation into their Vector Representation

Q&A fora feature text written in natural and programming languages alike. In order for this data to serve as input to analysis and learning algorithms, some text transformations are typically required as a preprocessing step. Examples of commonly-used

text transformations include Bag-of-Words (BoW) [54], Term Frequency-Inverse Document Frequency (TF-IDF) [116], and Word Embeddings [86, 87]. Each take a text document as input and produce an  $M \times N$  dimensional matrix as output.

### 2.3.1 TF-IDF

TF-IDF [116] is one of the earliest techniques used in natural language processing to transform text into a vector representation for the purpose of classifying documents. **The technique is based on the frequency of the words or terms in a document.** More specifically, TF-IDF is defined by the following equations:

$$TF(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \quad (2.1)$$

$$IDF(t, D) = \log \frac{N}{|d \in D : t \in d|} \quad (2.2)$$

$$TF-IDF(t, d, D) = TF(t, d) \cdot IDF(t, D) \quad (2.3)$$

The same terminology is shared across the above equations. Equation (2.1) determines the frequency of the term  $t$  in a document  $d$ . The frequency is simply defined as the number of occurrences of  $t$  in  $d$ :  $f_{t,d}$ , divided by the sum of the frequencies of all terms in  $d$ :  $\sum_{t' \in d} f_{t',d}$ . Analysing documents using the frequency of their words alone can be misleading since articles (e.g., *the*, *a*, etc.), and prepositions (e.g., *in*, *at*, etc.) occur more often than nouns and verbs.

TF-IDF therefore accounts for very common terms that might not be relevant to characterise a document as follows. Equation (2.2) calculates the logarithm of the number of documents  $N$ , divided by the number of documents where the term  $t$  appears:  $|d \in D : t \in d|$ . Equation (2.3) combines the term frequency of Equation (2.1) with the inverse document frequency of Equation (2.2). TF-IDF will generate a rectangular matrix  $D \times M$ , where  $M$  is the number of unique terms in all documents  $D$ .

### 2.3.2 Word2Vec

**The use of Word Embeddings (e.g., Word2Vec) [86, 87] has become popular due to its ability to capture semantic relations between words based on the surrounding context.** For instance, the words “dog” and “pet” may be related because of the similarity of their contexts.

In general, the Word2Vec approach follows the flow described in Figure 2.4.

First, the training data is constructed from the text passed as input. Training data is usually built by considering a *window* hyperparameter which extracts the surrounding words around a target word as its context. For example, consider the sentence “*The cat is on the mat*”. The context for the target word *on* comprises all the words adjacent to the selected target. Each word of its surrounding context forms the training dataset.

Once the training data has been constructed its words are transformed into one-hot encoded vectors. One-hot encoded vectors have all elements set to zero, except for

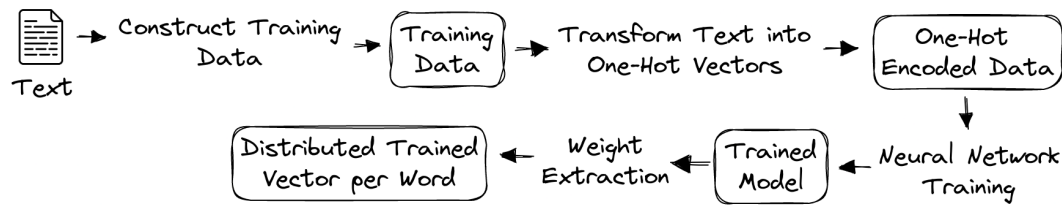


FIGURE 2.4: Word2Vec text transformation into a vector representation.

the element at the index corresponding to the word under transformation which is set to one. These vectors are of the size of the unique vocabulary in the text and form the input of a neural network (NN) for training.

The particular NN used has been described by Mikolov et al. [86, 87] and is shallow with only two hidden layers, and a linear activation function in the internal neurons for faster processing. The first hidden layer represents the embedding layer and the second one is the output layer which is compared to the target vector after the application of a sigmoid function. Given the vectors of the context words, the task of the NN is to predict the vector of the target word. Once the network has been trained, it is not used for future predictions, but the weights within the embedding layer are extracted. Thus, each word is associated with a learnt vector for it given its surrounding context. Two words might appear in many similar contexts, but since they might be used differently, their one-hot vectors might differ, hence resulting vectors are also dissimilar. The output is, therefore, a lookup table or a map data structure where the keys are words and the values are the learnt  $n$ -dimensional vectors, where  $n$  corresponds to the number of neurons in the first layer.

Two configurations can be used for Word2Vec: Continuous Skip-gram (Skip-gram) and Continuous Bag-Of-Words (CBOW) [86]. The former predicts the context around a word, whereas the latter tries to predict a word given its surrounding context. CBOW is used in this research as it is faster than the alternative Skip-gram [87] while resulting in similar efficiency. The following figure shows the weights of the selected architecture after a training epoch on the running example:

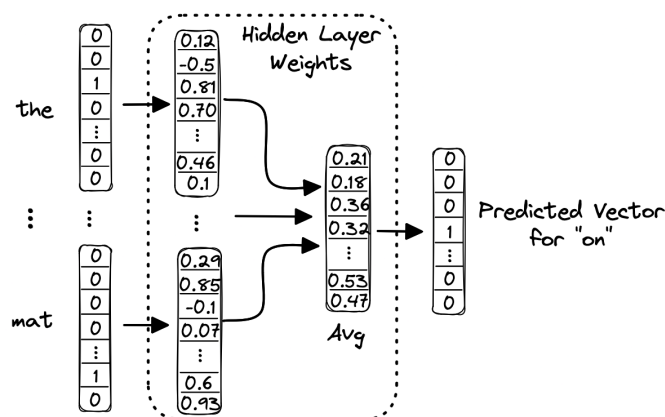


FIGURE 2.5: CBOW neural network weights after a training.

As depicted by Figure 2.5, the input of the CBOW NN are one-hot encoded vectors representing the target word's context. Two hidden layers are responsible for the training, with a first layer of neurons containing the weights for each input word individually, and a second layer with the aggregated (e.g., averaged) weights of all context words. The output layer of the network will predict the most likely word, encoded as a one-hot vector, and after an evaluation check, a backpropagation process calculates the needed adjustment of the weights. After the adjustment is made, the process repeats until a success criterion is met, e.g., number of epochs, goal achieved, etc.

The resulting weight vectors or embeddings of the first hidden layer are extracted for each of the context words, and eventually for all words in the vocabulary. These vectors effectively capture the context in which the target word appears.

Chapter 3 discusses recent applications of word embeddings within the state-of-the-art work in source code analysis (e.g., [6, 7, 57, 91, 92, 123]). Additionally, the techniques presented in this section are utilised in Chapter 4 and Chapter 5 as a processing step before a learning procedure (i.e., classification or clustering).

## 2.4 Text Classification as a Natural Language Processing Problem

### 2.4.1 Text Classification

**Text classification [14] is a natural language processing problem that requires assigning a correct label  $y_i$  to each set  $X_1, X_2, \dots, X_n$  of tokens that corresponds to a text from a given corpus.** For example, a paragraph describing movies, actors, directors, and scenes might be tagged with the label *cinema*. In contrast, a text about a forecast with snowfall, wind direction, and expected humidity could be labelled as *weather*. Supervised machine learning approaches for text classification train a learning model by extracting the relations between the tokens and the labels of a smaller representative set (i.e., the training set) to assign a correct label  $y'_i$  to a new set of tokens  $X'_1, X'_2, \dots, X'_n$  (i.e., the test set). Several approaches to training and predicting these labels have been proposed over the years (e.g., [23, 54, 107, 116, 149]).

### 2.4.2 Machine Learning Algorithms Used for Text Classification

**In classification problems, the feature vector resulting from feature extraction (e.g., through word embeddings or TF-IDF) is given to a classification algorithm for either training or class/label prediction.** This subsection briefly presents the ML-based classification algorithms employed for one of the main contributions of this dissertation.

#### K-Nearest Neighbours

**K-Nearest Neighbours (KNN) [38] classifies a new instance by analysing its  $K$  closest neighbours in the space of the independent features.** If these neighbouring instances belong to different classes, the assigned class will be the most common among the neighbours. Algorithm 1 details the KNN classification procedure.



**Algorithm 1:** K-Nearest Neighbours classifier pseudocode**Input :** Features  $X$ , Labels  $y$ , unlabelled vector  $j$ , neighbours  $k$ .**Output:** Label for vector  $j$ .

---

```

1  $distances \leftarrow \emptyset$ 
  /* Calculates the distances of all records in the training data to
   the unlabelled vector */
2 for  $i \leftarrow 0$  to  $n - 1$  do
3   |  $distances \cup \{calculateDistance(X_i, j)\}$ 
4 end
  // Computes the nearest  $k$  distances
5  $selectedDistances \leftarrow nearestDistances(distances, k)$ 
  // Returns the majority label of the  $k$  nearest neighbours
6 return  $majorityLabel(selectedDistances)$ 

```

---

The input of the KNN algorithm is the matrix  $X$  with the features for the documents in the corpus and their labels  $y$ . KNN also requires an unlabelled vector  $j$  for which the label will be predicted as the outcome of the learning process and the number of neighbours  $k$ . The algorithm first calculates the distance between each vector  $X_i$  and the unlabelled vector  $j$  (Lines 2-4). The  $k$ -nearest distances are selected (Line 5) to finally obtain the majority label from all labels  $y_1, y_2, \dots, y_k$  of the  $k$ -closest neighbours (Line 6). KNN is popular because of its fast classification and good results. Existing work [20, 111, 115] has applied KNN to text classification problems.

### Random Forest

Random Forest (RF) [24, 58] belongs to the family of machine learning classifiers categorised as ensemble algorithms. Ensemble classifiers group several machine learning algorithms that are considered efficient, yet lightweight. **In the case of Random Forest, the ensemble consists of several Decision Tree classifiers, hence the term Forest. Each tree is trained with a random selection of features, hence the term random.** This last characteristic prevents bias in the RF algorithm also avoiding overfitting especially on imbalanced datasets. At prediction time in a classification task, each of the trees performs a vote on the decision to be taken. The majority vote represents the final prediction of the trained model. On the other hand, for regression tasks, RF considers the averaged outcome of all decision trees in the forest. Algorithm 2 describes this ensemble algorithm into more detail.

The algorithm starts with an empty forest, or set of decision trees, which will be dynamically filled in. The number of decision trees comprising the forest is set using a hyperparameter. Each tree in the forest is trained with a bootstrapped (i.e., randomly sampled and replaced) version of the training dataset (Line 4).

Each decision node makes a split based on a random subset of features  $f' < f$  (Lines 8 and 11). This will ensure that the algorithm remains unbiased. Like KNN, RF has already been used with success for text classification tasks [31, 111].

**Algorithm 2:** Random Forest classifier pseudocode**Input** : Training data  $Xy$ , feature vector  $f$ , number of trees  $n$ .**Output:** Trained Random Forest  $RF$ .

---

```

1  $RF \leftarrow \emptyset$ 
  // Trains a tree and added it to the forest set
2 for  $i \leftarrow 0$  to  $n - 1$  do
3    $dataBoost \leftarrow \text{bootstrap}(Xy)$ 
4    $tree \leftarrow \text{RandomisedTreeLearn}(dataBoost, f)$ 
5    $RF \cup \{tree\}$ 
6 end
7 function  $\text{RandomisedTreeLearn}(dataset, features)$ :
8   // Constructs a Decision Tree considering the following:
9   for Each decision node do
10    // Selects a random subset of features
11     $featuresSubset \leftarrow \text{randomSubsetFeatures}(features)$ 
12    // Splits on the best feature of the subset
13     $\text{splitNode}(featuresSubset)$ 
14    /* Re-calculates information gain, entropy, etc., and
15       continues to other remaining nodes in the tree if any */
16  end
17  return The learnt tree
18 end
19 return  $RF$ 

```

---

**Ridge Linear Classifier**

**Linear Regression classifiers fit a line to the data that optimally splits the data into two categories or labels.** To use them on multi-class datasets (e.g., such as many for text classification), techniques have been proposed including One-vs-Rest, and One-vs-One. One-vs-Rest transforms a multi-class dataset in multiple binary datasets where one class is considered positively, and all other classes are treated negatively [89]. One-vs-One on the other hand, constructs multiple binary datasets with every possible pair of classes [21].

Regularised linear classifiers are a variation on classical linear classifiers since that introduces penalisation hyperparameters at training time. The latter have been used successfully for text classification problems [160]. We will instantiate our approach with the Ridge Linear (RL) regressor [101], which first transforms the multi-class data into a multi-output regression problem and then fits one regressor per target label. RL regressors introduce a regularisation parameter  $\alpha$  that determines the variance of the estimated weights for the model.<sup>27</sup> The regularisation parameter adds the linear classifier into the regularised classifier family.

The base of the classifier is an ordinary linear regression model (OLR) of the form detailed in Equation (2.4).

<sup>27</sup> [https://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.RidgeClassifier.html](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RidgeClassifier.html)



$$Model_{olr} = mx + n \quad (2.4)$$

The variables  $m$  and  $n$  will be adjusted to the training data features  $x$ . To optimise the model to the best possible  $m$  and  $n$ , a loss function is needed to report how well the trained model performs with its current variable values on the testing data. One of the most popular loss functions is the mean-squared error (MSE) function defined in Equation (2.5).

$$MSE(y, y') = \frac{1}{n} \cdot \sum_{i=1}^n (y_i - y'_i)^2 \quad (2.5)$$

Equation (2.5) is applied to the testing data of which  $n$  is the number of elements,  $y_i$  refers to the true label, and  $y'_i$  is the predicted label of the  $i^{th}$  element. Regularised classifiers add parameters to the loss function in order to control the variables in the model.

$$Reg\_MSE(y, y') = MSE(y, y') + \alpha \cdot \|\theta_i\|_1 \quad (2.6)$$

Equation (2.6) shows an additional summand to the original Equation (2.5) with the  $\alpha$  and the  $\theta$  parameters.  $\alpha$  will penalise the parameters  $\theta_i$  in case their values are too low or too high. The parameters  $\theta_i | i = 1 \dots n$  refer to all variables in the model being regularised. In this case, the variables to be adjusted are  $m$  and  $n$  from Equation (2.4). Finally, a L1-norm or absolute-value norm is applied to  $\theta_i$  which means  $\|\theta_i\|_1 = \sum |\theta_i|$ . The regularisation addition to linear classifiers is considered advantageous since it prevents model overfitting.

## Support Vector Machines

**The Support Vector Machines (SVM) classifier [34] searches for an optimal hyperplane that can effectively distinguish the classes based on their features.** For example, when the number of features in a dataset is two, SVM will try to find the optimal line dividing the data in a 2D space. If the number of features is three, the division becomes a plane trying to separate a 3D feature space. This concept can be generalised to datasets with  $n$  feature dimensions and a hyperplane with  $n - 1$  dimensions, which require considerable computational resources if  $n$  is high.

Different *kernels* can be used to fit the model to the data. Examples of kernel functions include linear, polynomial, sigmoid, etc. SVMs have also been used successfully for text classification problems [62, 71].

Algorithm 3 is a version<sup>28</sup> of the original SVM that has been simplified for didactic purposes. For example, the  $\lambda$ s (i.e., trainable penalisation parameters) in the original SVM are calculated using a Sequential Minimal Optimisation (SMO) algorithm [96]. SMO is used to speed up calculations as well as to explore more regions on the solution space. The simplified version depicted in Algorithm 3 uses the simpler bounded calculations instead.

<sup>28</sup> A Python implementation can be found at: [https://github.com/fbeilstein/simplest\\_smo\\_ever/blob/master/simple\\_svm.ipynb](https://github.com/fbeilstein/simplest_smo_ever/blob/master/simple_svm.ipynb)

**Algorithm 3:** Support Vector Machine classifier pseudocode**Input :** Features  $X$ , Labels  $y$ , kernel  $k$ , regularisation  $C$ , max. iter.  $m$ .**Output:** Trained SVM.

---

```

1  $\lambda \leftarrow \emptyset$ 
2  $K \leftarrow k(X, X) \cdot y' \cdot y$  //  $y'$  is  $y$  with an extra dimension
  // Trains the  $\lambda$ s
3 for  $\_ \leftarrow 0$  to  $m - 1$  do
4   for  $i \leftarrow \text{length}(\lambda)$  do
5      $j \leftarrow \text{random}(\lambda)$ 
6      $Q \leftarrow K[[[i, i], [j, j]], [[i, j], [i, j]]]$ 
7      $v_0 \leftarrow \lambda[[i, j]]$ 
8      $k_0 \leftarrow 1 - \sum \lambda \times K[[i, j]]$ 
9      $u \leftarrow [-y_j, y_i]$ 
10     $t_{\max} \leftarrow k_0 \cdot u / (Q \cdot u^2 + \sigma)$ 
11     $\lambda_{[i, j]} \leftarrow v_0 + u \times \text{restrict}(t_{\max}, v_0, u)$ 
12  end
13 end
14 function  $\text{restrict}(t, v_0, u)$ :
15   // Bounds the results to defined limits
16   return  $\text{bound}(v_0 + t \times u, 0, C)$ 
17 end
  // Obtains the indices supporting  $\lambda$  at a minimum  $\sigma$ 
18  $\text{indices} \leftarrow \text{index}(\lambda > \sigma)$ 
  // Calculates the bias  $b$ 
19  $b \leftarrow (\sum(1 - \sum K_{\text{indexes}} \times \lambda) \times y) / \text{length}(\text{indices})$ 
  // Computes and returns the model
20 return  $\sum(k(X, X) \times y \times \lambda) + b$ 

```

---

Algorithm 3 takes as input the training data  $X$  and  $y$ , the kernel  $k$ , the regularisation  $C$  and the maximum number of iterations  $m$ . Some variables are initially bound to certain values, for example, the  $\lambda$ s (Line 1) start with an empty set and  $K$  (Line 2) stores the application of the kernel  $k$  on the features multiplied by their labels. The nested for loop (Lines 3-13) iterates until the maximum iteration  $m$  is reached for all  $\lambda$ s in each outer cycle. The variables  $i$  and  $j$  are used in the internal loop for various operations to finally modify the value of  $\lambda$  at a specific location. The `restrict` function (Lines 14-16) limits the calculated results to the maximum parameter  $C$ . Indices are extracted (Line 17) where the calculated  $\lambda$ s are higher than a defined threshold  $\sigma$ . The threshold is usually a very small number (e.g.,  $1E - 15$ ). Finally, the extracted indices and their  $\lambda$ s are used to compute the bias  $b$  (Line 18) and the SVM model (Line 19).

The classification algorithms described above are used in Chapter 4 to suggest a likely FQN in an incomplete code snippet. The algorithms are evaluated according to their performance in solving the API type resolution as a text classification problem.

## Multi-label Classification Algorithms

In general, classifiers can be trained on three types of datasets. The first type of dataset is called a binary dataset as it contains only two classes to predict. This explains why classifiers trained on binary datasets can be very effective. The second, more complex, type is called a multi-class dataset as it includes more than two classes to predict. Classifiers trained on multi-class datasets can still achieve good performance. For both types of datasets, classifiers only need to predict one class per instance in the dataset. **The third and most complex type of dataset is called a multi-label datasets as it includes more than one class to predict per data instance. In order to perform well, classifiers on a multi-label dataset need to fit several distinct data distributions which increases the complexity of both training and evaluation.**

We discuss the multi-label classifiers that performed best in an extensive survey and empirical comparison by Madjarov et al. [82]. Note that multi-label classifiers need to be instantiated with a base classifier. That is, a multi-label classifier cannot make predictions on its own without consulting a base classifier. The previously discussed SVM algorithm has, for instance, been used as a base classifier for multi-label classifiers.

### Binary Relevance Classifiers

The Binary Relevance (BR) classifier [129] trains one classifier per label to predict. It assigns one class for a label and another class for the rest of the labels. In other words, it simplifies the multi-label problem to a single-label one by binarising the dataset.

Like BR, Classifier Chaining (CC) [103] also uses  $n$  binary classifiers. The algorithm “chains” these classifiers together. Each binary classifier is assigned a label  $j$  the relevance of which will be learned. To compute the remaining relevance of the  $j + q \mid q > 0$  labels, prior knowledge is used.

### Ensemble Method Classifiers

The RANdom  $k$ -LABELsets (RAkEL) ensemble classifier [130] forms groups of labels and trains a single classifier for each group. It does consider relations among the labels within a group. Being an ensemble algorithm, predictions are made by majority vote from the classifiers for each group.

The Ensemble Multi-label (EML) classifier proposed by Read [102] combines several multi-label classifiers in an ensemble way. It can, for instance, be instantiated with the aforementioned Binary Relevance (BR) and Classifier Chaining (CC) classifiers.

## 2.5 Part-Of-Speech Tagging

**Phrase-structured trees are a common way to represent the structure of sentences.** Figure 2.6 shows the tree representation of the sentence “*The man hit the ball*”. The sentence comprises several constituents, each of them having a distinct grammatical role in the sentence structure.

Initially, the tree in Figure 2.6 divides the sentence into two elements: a Noun Phrase (NP) and a Verb Phrase (VP). The NP is further decomposed into the determiner (T)

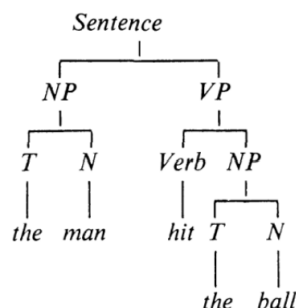


FIGURE 2.6: Phrase-structured tree of a sentence [32].

*the* and the noun (N) *man*. On the other hand, the VP comprises the verb *hit* and NP comprising another determiner *the* and the noun *ball*.

The tags alongside the constituents forming a sentence (e.g., NP, VP, etc.), are known as part-of-speech (POS) tags. POS tags help identifying specific elements in sentences and assist in cases where relation-based characteristics or patterns need to be extracted, e.g., NP formed by determiners and nouns. Automatic approaches for POS tagging have been introduced [126, 127] and their models have been made publicly available to enable more research on the topic. The techniques in Chapter 5 use POS tagging as a way of automatic feature naming. All sentences surrounding code snippets are analysed to extract the pairs of nouns and verbs to can help in naming a frequent pattern into a library feature.

## 2.6 Hierarchical Clustering

**Hierarchical clustering [83] is a technique that clusters elements based on the similarity of their attributes, and results in a hierarchy of clusters (often visualised as a dendrogram) so that elements in a child cluster also belong to the parent cluster.** Intuitively, each layer in the hierarchy is characterised by its own maximum distance among the elements in its cluster. Bottom-up (or agglomerative) approaches initially form one cluster per element and merge these clusters repeatedly according to their similarity. The cluster at the root of the hierarchy encompasses all elements. Top-down (or divisive) approaches initially group all elements in a single cluster and repeatedly split until every element is on its own.

### 2.6.1 Static and Dynamic Tree Cutting

The Hierarchical clustering algorithm results in a tree of clusters that needs to be cut at a given height to obtain a unique clustering of the dataset. In static tree cutting (left part of Figure 2.7), each child below the given height cut off forms a separate cluster. However, an appropriate height threshold might be very challenging to determine.

A suboptimal clustering can result, especially for height thresholds at which there are many similar elements and hence nested clusters. Alternatively, Langfelder et al. [68] propose to cut the tree branches dynamically (right part of Figure 2.7) based on its shape using either a top-down or a bottom-up tree-cutting algorithm. These have been shown to outperform static cutting in bioinformatics applications, and are amenable to complete automation, rather than manual selection of the cutting

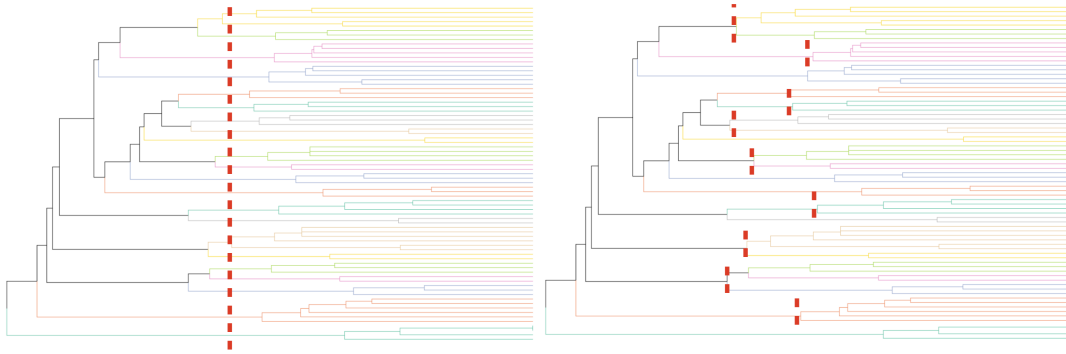


FIGURE 2.7: Static (left) and dynamic (right) tree cutting of a dendrogram.

threshold. We will use this technique in Chapter 5 for clustering the API usage in code snippets and form patterns of usages that will eventually be library features.



## Chapter 3

# State of the Art

This chapter describes the current state of the art in data-driven tooling based on SO information. The chapter starts with a survey of tools that consult Stack Overflow information to offer developers support on various software engineering problems. Next, we analyse how these tools approach the problems of SO code snippets being incomplete and ambiguous. Furthermore, as word embeddings applied to code are an integral part of the thesis contributions, this chapter also explores the latest approaches in this direction. As we use SO to provide tool support for library usage and selection, this chapter also includes a major survey on library recommendation and feature uncovering.

### 3.1 Development Tools Incorporating Stack Overflow Information

Several software engineering tools have been proposed throughout the years that use SO posts as a source of information. In general, these tools aim to leverage the multitude of advantages provided by Q&A platforms (detailed in Chapter 2) by analysing a selection of code snippets within SO answers that have been curated by the developer community.

RecoDoc [37], for instance, identifies API references within text in the library documentation and in library-selected answers on Q&A websites to improve their learning resources. To identify API references, RecoDoc performs multiple steps including snippet parsing and Partial Program Analysis (PPA) [36]. PPA introduces changes to the input program with the goal to produce complete and correct code ready to be compiled and run by the end user. For example, a code snippet can be incorrectly typed and PPA would attempt to correct variable declaration types. However, none of the code snippets presented in the PPA paper is syntactically incomplete, nor do they miss package imports statements. Assuming that PPA manages to make the code snippet compile, it will infer partial type information for the expression declarations in the snippet, but not for the types for which it misses a declaration (e.g., those imported from a library). When faced with incomplete code, RecoDoc is limited by these limitations of PPA.

Prompter [100] proactively retrieves SO posts that are relevant for the current context of the developer within the IDE. The approach establishes a ranking model based on different types of similarities between the context in the IDE and the SO posts to retrieve. The ranking model is based on eight features that capture the relations

between the mentioned entities. The considered features are textual similarity, code similarity, and API types similarity, question score, accepted answer score, user reputation and tags similarity. To calculate the API types similarity, Prompter relies on the outcome of the Eclipse JDT parser, which means that the input to the parser has to be at least syntactically correct.

SISE [128] augments library documentation with relevant API sentences in SO posts. To find insightful sentences in the corpus of SO information, SISE's search is based on regular expressions which are matched against the body and title of posts. The goal is to detect API elements within natural language and possibly augment the documentation of the referenced API with community-based knowledge.

CodeTube [99] enables querying the contents of software development video tutorials and complements the results with relevant discussions. The tool identifies the frames in the video that feature Java code to subsequently extract the region in the frame with the code. An Optical Character Recognition (OCR) tool is used to recognise and extract the text within the video frame. An island parser [11, 88] is used to obtain the Java code within the frame.

BIKER [28] takes a natural language query about a library as input (e.g., *"run linux commands in Java code"*) and returns related SO posts and API methods. To identify API references in SO posts, BIKER (like the aforementioned SISE [128]), uses regular expressions that match API types by their fully qualified names (FQNs).

POME [72] analyses SO posts to synthesise the community's opinion about a library. Once again, the identification step of the referenced API elements in SO code snippets heavily depends on the usage of regular expressions as some of the previous works. The POME approach includes an additional regular expression that matches the occurrences of method invocations by their parent classes. Sentences containing matches are considered related to the corresponding library and subsequently classified through sentiment analysis as positive or negative.

PostFinder [105] recommends SO posts based on the developer's context in the IDE by constructing queries to an already populated Lucene database. PostFinder tries to resolve API type references by parsing a code snippet using the Eclipse JDT, and then consulting the Maven Dependency Graph [16] to obtain the likely FQN for simple names within the Abstract Syntax Tree (AST). When multiple candidate types share the same simple name (i.e., ambiguous cases), PostFinder computes the Levenshtein distance between the candidates and the SO post title, the question body, and the answer. However, this heuristic does not ensure the accurate type resolution for an ambiguous simple name for the following reasons:

1. Levenshtein distance calculates the number of edits needed to transform a text sequence into another at the character level [70]. PostFinder takes the maximum distance (cf. Line 163 - Line 178 in its GitHub source code<sup>1</sup>), meaning that the selected FQN is the most dissimilar to the post text corpus. We did not find a rationale behind this decision.
2. SO code snippets might contain API types that are not related to the topic under discussion, yet are entangled with those types that are.

<sup>1</sup> <https://github.com/MDEGroup/PostFinder/blob/master/tools/src/main/java/soRec/Uutils/Jdt.java>



Opiner [131] mines the API usage of libraries in SO posts with the objective of producing API documentation. The approach combines several techniques to parse and filter the information in the posts, as well as to learn based on the proximity of API references. More specifically, the code snippets are parsed through a combination of ANTLR [94] and island parsers [88]. Each line is processed first by the Java ANTLR parser. In case of a parse error, the island parser will process it instead. Since many code snippets are syntactically incorrect, the technique can work in most of the cases. However, incomplete code snippets where there is no reference to the FQN of the used API element might represent a problem for Opiner as it is not able to resolve API types.

ATTACK [146] generates API tags for fragments of SO tutorial posts. Using question and answer pairs from SO and their corresponding tag sets, ATTACK trains an attention-based deep neural network that prioritises the most relevant parts of a post based on its tags. The trained network is able to recommend tags per API fragment and to provide more insights about the feature of an API usage. The technique employed by ATTACK is hybrid since it combines Eclipse JDT and PPA [36]. The approach works similarly to Opiner [131]: in case of failure from the first tool (Eclipse JDT in this case), the second one is used (PPA).

Table 3.1 summarises the techniques used by these state-of-the-art tools to extract (and resolve) API references from SO code snippets.

Approach	Eclipse JDT	PPA	Regex	ANTLR	Island Parser
RecoDoc [37]		✓			
Prompter [100]	✓				
SISE [128]			✓		
CodeTube [99]					✓
BIKER [28]			✓		
POME [72]			✓		
PostFinder [105]	✓				
Opiner [131]				✓	✓
ATTACK [146]	✓	✓			

TABLE 3.1: Techniques used to extract API references from code snippets.

Table 3.1 shows that many software engineering tools that leverage SO information rely on the Eclipse JDT or simple regular expressions. Eclipse JDT is not able to parse a syntactically incorrect code snippet. In other words, the code passed as input needs to compile without errors, which is also the case for PPA (e.g., RecoDoc [37], ATTACK [146]) and ANTLR (e.g., Opiner [131]). This is particularly problematic for SO code snippets as the Q&A platform has no restriction (i.e., syntax checks) on how a contributor *should* include code in a post.

The regular expressions used in the discussed approaches have been crafted to match particular parts of the text information in SO posts. For example, they target the title or the body of the post and retrieve specific forms of API references such as prototypical method calls and field accesses therein. However, syntactically more complex API references such as chained method calls known from libraries that feature a fluent API, might escape the hand-crafted regular expression. Island parsers

[88], in contrast, could be configured with the complete and official grammar for any language. This ensures that the parsing step covers all intricacies of the language. Moreover, statements and declarations that are unrelated to API references can be declared as *water* whereas the expressions that are to be extracted (e.g., method calls) can be declared as *islands*. Only two of the discussed approaches (e.g., CodeTube [99] and Opiner [131]) rely on island parsers [88]. This renders their extraction of API references robust. However, other means are required to resolve fully qualified names.

## 3.2 Program Analyses for Stack Overflow Code Snippets

Several algorithms for API type resolution in code snippets have been proposed over the years.

PARSEWeb [124] is a tool supporting developers unfamiliar with a library in identifying an API method sequence to obtain the object of a type they need. To this end, it constructs Method Invocation Sequences (MIS) from direct acyclic graphs constructed from code examples where the nodes represent statements and the edges represent the control flow between two statements. Next, a clustering technique is applied to group similar sequences. The resulting clusters are used to recommend completions of partial method invocation sequences to developers. The approach is limited to a single library under analysis at a time, while developers often use APIs from multiple libraries. Moreover, the method invocations do not necessarily form contiguous sequences, but may be scattered throughout a snippet.

PPA [36] has been used in several approaches that require a robust technique for extracting type-related information from incomplete code snippets. For instance, this is the case for RecoDoc [37] and ATTACK [146], as shown in Table 3.1. PPA is based on the Eclipse JDT parser and aims to recover types from partial programs. The goal of PPA is to infer type information (i.e., member declaration and subtyping information) for referenced types of which the type declaration (i.e., a Java class or some file with a class or interface that declares the type) is in otherwise complete and syntactically correct programs. As PPA is based on the Eclipse JDT parser it fails to analyse syntactically incorrect code and code that does not form a compilation unit on its own.

Baker [118, 119] traverses an AST constructed by the Eclipse JDT parser or the Esprima parser for JavaScript to collect type information at variable declaration nodes. It then associates a list of candidate FQNs for these nodes by consulting a database populated from the JAR files of candidate libraries in the case of Java code snippets. These lists are iteratively refined for every method invocation of which the receiver expression is a known variable reference or method invocation. The computed lists satisfy the semantic constraints imposed by using the API-related variables in the snippet. The deductive reasoning approach taken by Baker is therefore limited by I) the syntactical correctness of the snippet, II) the extent to which it contains candidate-reducing API usages, and III) the library implementations for which its database has been populated.

Yang et al. [150] investigate the usability of code snippets in Stack Overflow for four programming languages: C#, Java, Python and JavaScript. Usability, in this case, is defined as whether a code snippet can be successfully parsed, compiled, and run

using the language’s default tools. Their findings indicate that few snippets are usable as they are in the posts, especially in the case of Java with less than 4% parsing and 1% compilation success rates. Their heuristics for repairing code snippets (e.g., addition of wrapping classes, methods, and semicolons) slightly improves these results, but still, a large percentage of snippets (more than 80%) remains unusable. As pointed out by Yang et al. [150], almost 63% of errors are due to missing symbols, which reinforces the need for tools capable of analysis incomplete code snippets.

CSnippEx [122] is an Eclipse plug-in for repairing Java code snippets and converting them into compilable source code files. CSnippEx takes a feedback-based approach where the Eclipse Quick Fix tool is continuously queried for errors in the snippet under repair. First, the tool discovers the likely merging of multiple code snippets in a post. Second, it resolves the import dependencies by relying on a database that maps FQNs to the latest version of the JAR that defines them and by exploiting their *clustering hypothesis*. This hypothesis refers to the observation that classes in a Java file are frequently used together, hence, import declarations might share common package names and thus, create clusters. Third, CSnippEx considers the compilation errors of the Eclipse Quick Fix tool to identify repairs that could render the code into a compilable source. CSnippEx [122] reports the successful repair of 40,410 code snippets from a total of 242,175 for a success rate of around 17%, which is still testament to the difficulty of the problem.

Grapa [163] is a tool and approach also targeting partial programs to enable their successful analysis. Grapa is slightly different to similar techniques since it leverages other approaches for completing programs to analyse incomplete ones. The approach first explores previous versions of the partial program to extract the majority of its declared types, which are referred as the *context version* of the partial program. Using PPA, Grapa extracts the ASTs from the partial program including unknown types, which might be resolved in a following step. Using a custom inference algorithm, and the extracted context, Grapa resolves the missing types in the partial program. The approach concludes with an ambiguity analysis to decide upon shared simple names at resolution time. The concept of versions being syntactically correct and containing types to use in incomplete programs, is not applicable to code snippets in SO.

StatType [95] uses statistical machine translation to resolve FQNs. The approach translates sentences from a source language (i.e., of partially qualified API names) to a target language (i.e., of FQNs). It requires training a language model and a separate mapping model on a corpus of projects that use the API of the targeted libraries. These projects are simultaneously translated into StatType’s source and target language by traversing their methods’ ASTs. Sentences in the source language capture information about the simple name of each referenced API element. In contrast, sentences in the target language convey the corresponding FQN and the syntactic construct through which the API element was referenced. Extracting this information requires the input projects to compile. However, the approach is entirely data-driven because it is agnostic to program semantics. Resolutions learnt by combining the language and mapping model are refined based on the local context surrounding the name to be resolved. StatType achieved higher accuracy than the previously discussed Baker [118, 119]; however, training its models may be computationally expensive.

COSTER [106] takes an information retrieval approach to the problem of resolving

FQNs in incomplete code snippets. Like StatType [95], it relies on the wisdom-of-the-crowd obtained from a corpus of compilable projects. Extracted information from the corpus is used to compute the likelihood that textual tokens surrounding the non-qualified API reference co-occur in the project corpus when the reference resolves to the corresponding FQN. The immediately surrounding tokens are referred to as the local context of the API reference. The local context is extended with its global context. It relies on the names of methods called on the variable to which the reference is assigned or the names of methods to which this variable is passed as an argument. In a snippet, context and name similarities are subsequently used to refine the candidates returned from this so-called occurrence likelihood dictionary for the queried name and context. COSTER has outperformed StatType and Baker in many aspects of the evaluation conducted in Saifullah et al. [106].

JCoffee [50] is another PPA-based tool with the goal of transforming code snippets into compilable Java programs. As previous PPA approaches, JCoffee assumes the syntactical correctness of the code snippets. Nonetheless, in this case, the approach uses the errors triggered by the Java compiler as feedback to transform the code snippets and finally make them compilable. The transformation is based on the automatic creation of Java classes, their methods and fields including *unknown* types.

Ahmed et al. [1] use deep neural networks (NNs) with the goal of repairing code snippets from student solutions and SO fragments. The usage of classical parsers such as Eclipse JDT is immediately identified as impractical by Ahmed et al. [1], due to the multiple issues incomplete code has, such as missing semicolons, braces, brackets, external types, presence of ellipsis, etc. Therefore, alternative strategies are adopted to first parse incorrect code and second repair all identified errors in the snippets. Three transformer-based neural networks are proposed to this end. The first, BlockFix, has been trained to repair the block structure mistakes such as missing curly braces. The second, FragFix, learns repairs for syntactically incorrect fragments of code. The third, TypeFix, repairs code snippets by adding missing import statements and types. TypeFix, the most related to this dissertation, has been trained on 50 correct GitHub Java projects. Code elements with no types are hidden with a special marker (e.g., ~). Random fragments of code are created consisting of pairs of Java code and the described typed code. In this way the transformer NN learns how to “tag” Java code with types based on the local information of their usages.

SnR [40] infers FQNs based on constraints in SO code snippets. The approach constructs a knowledge base from libraries consisting of relations between types of fields and methods. As in CSnippEx [122], SnR tries first to repair a code snippet and to form a compilation unit from which to extract an AST and infer API types. The type inference step uses Datalog, given the current code snippet constraints and the knowledge base, to provide reachable nodes in a dependency graph. Nodes in the dependency graph represent the variables or constants in a program and edges represent a relation (e.g., references) between the nodes. In this way, the graph captures the reachable relations of all typed elements in a program. SnR ranks type candidates and selects those at the top to create import statements and to include them in the repaired code. SnR is compared against COSTER [106] using the public *StatType-SO* dataset.

### 3.3 Embeddings for Source Code Analysis

As previously stated in Chapter 2, our research makes use of embeddings [86, 87] to transform text into its vector representation. More specifically, RESICO, our API-type resolution approach, employs embeddings to convert API references and their contexts into a format suitable as input to machine learning algorithms. Applications leveraging similar context-based embeddings in software engineering tasks have been increasingly prevalent in the literature. Approaches considering the tokens of programs as input to machine learning processes were pioneers in the application of ML for software engineering.

Allamanis et al. [5] propose a language model for code text capable of suggesting a name for a method or a class based on the features of their implementation. Class and method names of which the vector embedding is close in distance denote similar features according to their contexts. For the same task as in the previous research, Allamanis et al. [4] have trained a Convolutional Attention Network (CAN) (i.e., similar to a Convolutional Neural Network, but focused on the attention mechanism) to summarise source code into a method name. Based on the sequence of (sub)tokens that are in the method's implementation, the CAN learns the most relevant ones related to the real name of the method which, in this case, serves as ground truth. The attention mechanism used in the latter work is the major difference with the former proposed approach.

Iyer et al. [60] introduce Code-NN, a Long-Short Term Memory NN with attention to summarise source code into short sentences. Code-NN is based on the information Stack Overflow posts provide such as the natural language features surrounding code snippets. Code-NN leverages the available information in SO posts, and makes use of embeddings in its learning process.

Henkel et al. [57] attempt to learn trace-oriented embeddings for source code. These are obtained by applying abstracting transformations to a lightweight form of intra-procedural symbolic execution. Similarly, Wang and Su [140] propose LiGer, a neural network that can learn from a mixture of symbolic and concrete traces. The blended trace represents the input to an encoder-decoder NN architecture which outperforms code2seq [6] in the task of predicting method names. The use of symbolic execution for the embeddings is considered infeasible for the research in this dissertation, given the sheer number of libraries and library usages on SO.

The import2vec [123] framework learns library usage representations based on the co-occurrence of their corresponding import statements in client projects. Trained word embeddings are suitable for recommending imports with a usage context similar to the queried one. The embeddings have shown that they capture meaningful semantic relations such as which library packages are often used together in particular problem domains (e.g., data science code).

Lehmann and Pradel [69] propose SnowWhite, an approach and tool that predicts high-level types from low-level types and their contexts in WebAssembly (WA) binaries. A trained sequence-to-sequence (S2S) model generates high-level types from WA binaries and supports developers in auditing possible security issues (e.g., supply chain attacks).

An alternative to token-based embeddings are those based on information derived from ASTs. For instance, code2vec [7] predicts source code properties such as method



names provided their bodies. It uses a neural network for learning embeddings that effectively model the correspondence between the code in the method body and its associated label. The network learns to aggregate syntactic paths from the AST of the code into a vector. Similarly, code2seq [6] uses AST paths and an encoder-decoder architecture to learn code representations which are subsequently used in code summarisation and captioning.

Zhang et al. [158] propose an AST-based Neural Network (ASTNN) to capture syntactical information at the statement level. The approach is based on a Recurrent NN (RNN) and the statement trees that are extracted from the AST. Statement vectors are learnt in a similar manner as word embeddings in RNN. The nodes in the statement trees and their hidden states in the RNN influence the weights of the final statement embedding.

InferCode [27] also focuses on the analysis of subtrees extracted from the AST. In this case, the approach uses a self-supervised technique (i.e., identified subtrees should be predicted by previous subtrees) in conjunction with a Tree-Based Convolutional Neural Network. InferCode improves upon previous approaches (e.g., [7], [6], [158]) in tasks such as code clustering, clone detection, etc.

Lin et al. [73] present a novel method called Block-wise Abstract Syntax Tree Splitting (BASTS) to improve code summarisation. BASTS constructs the Control Flow Graph of a method and then removes cycles and conditional edges. The remaining subgraphs are encoded and represent the input to a transformer-based approach. Code summaries are generated based on the comments of the methods and the learnt subtree encoded representations.

Graph-based representations have also been used in ML-based support for software engineering tasks. For example, ProGraML [35] constructs an enhanced graph from an initial Control Flow Graph by adding the information flow in Data Flow and Call graphs. A graph-based machine learning approach leverages the previously enhanced graph to learn downstream tasks such as device (e.g., CPU or GPU) mapping. Similarly, Liu et al. [75] propose a representation of the code via Graph Neural Networks (GNN) called UniCoRN. The approach is based on a control and data flow representation of the code at the method level with additional enhancements such as AST node types and variable usage relations (e.g., first, or last use). Graph-LSTMs presented by Jiang et al. [61] are yet another proposed representation proposal relying on program dependence graphs (PDGs). The goal of Graph-LSTMs is not only to learn from local syntactic dependencies, but also, to explore global relations of different sub-trees which are extracted similarly to UniCoRN [75]. The difference of Graph-LSTM approaches to Tree-LSTM-based ones is that the latter's internal structure depends on the parent relationship, whereas the former's can be generalised to any relationship between nodes.

Approaches that learn hybrid representations have been proposed as well. Zhang et al. [157] combine paths of the PDG, intermediate representation (IR) instructions (i.e., from the Jimple IR), and natural language comments in a representation to train a model for suggesting method names. Vagavolu et al. [132] use paths of the AST, CFG and PDG to also name methods of C programs. Tian and Treude [125] hypothesise that external context (e.g., traces, developer activities, etc.) could help code representations and the approaches based on them. As a proof-of-concept they employ the call hierarchy as context, thereby, enhancing the results compared to same approach without it.

## 3.4 Library Usage Comprehension

This section discusses automated approaches that support developers in understanding what features are offered by a library and their usage.

### 3.4.1 Feature Uncovering

Kanda et al. [64] present an approach to uncovering functional features from the source code of an Android application. Functional features are defined as a set of related API calls with a textual description. The approach exploits the assumption that apps with similar features make similar calls to the Android SDK. It is semi-automated as call sequences that are frequent across apps need to be described manually as a feature.

Zhang and Hou [161] analyse forum discussions in search of problematic APIs. Their tool Haystack identifies sentences with a negative sentiment containing API tokens. Through the Stanford NLP toolkit, it recommends API feature names (e.g., “*resize jscrollpane*”).

Other work focused on features as natural language concepts without an associated code implementation. Guzman et al. [51] and Shah et al. [110] uncover features from app reviews using NLP techniques. The former use techniques such as collocation finding, aggregation, topic modeling, and sentiment analysis to provide single-feature reviews to developers and ease the burden of inspecting possibly thousands of them. The latter approach by Shah et al. [110] employs the same techniques but goes one step further by comparing competing apps (e.g., fitness applications) according to the sentiment description of their respective reviews. In other words, the apps are compared in relation to their ratio of positive and negative reviews from users. Similarly, Sarro et al. [108] empirically studied the lifecycles of features in app stores. Their feature extraction pipeline focuses on the processing of app descriptions also through NLP techniques.

Al-Subaihin et al. [3] group apps based on the features extracted from their descriptions. A hierarchical clustering algorithm is employed to form groups of similar apps based on their most shared tuple of bi-gram or tri-gram words (i.e., featurelets).

Guo et al. [49] extract features from the UI elements in mobile applications. Their goal is to identify redundant features and to investigate the extent of feature redundancy in installed apps on a smartphone. Redundancy is defined as two or more installed applications sharing common features (e.g., *schedule*, *app management*). Their extraction process includes the identification of relevant words from the XML files that describe the graphic elements of apps. Subsequent steps in the process involve text processing of the extracted terms in the visual descriptions. Their study shows that more than 85% of smartphones contain redundant features and that all studied features are redundant on at least half of the analysed smartphones.

The NLI2Code framework proposed by Shen et al. [113] bridges the natural language and functional perspectives on features. The framework consists of three components, a functional feature extractor, a pattern miner, and a synthesizer. The functional feature extractor mines SO posts with the name of the target library in the tag list. Verb phrases are extracted from the surrounding text after a natural language filtering application. API candidate elements are retrieved and processed (e.g., camel-case split) to associate them to the previous verb phrases. Selected API

candidates are mined from client projects on GitHub and a data flow graph is constructed from their usage. A frequent graph pattern mining algorithm (e.g., gSpan) is used to obtain the frequent sub-graphs as API patterns. Finally, a synthesizer is used to incorporate information into the incomplete patterns (e.g., local variable names) and suggest a complete API usage to the developer.

FeatCompare [10] employs user reviews to extract global and local features from apps. Local features are related to the category of a particular app, whereas global features are not bound to any domain. Supported by the advantages of word embeddings, the outcome of FeatCompare is a comparison table based on the discovered features from user reviews.

SIRA [141] is also based on the reviews of apps as many of the already mentioned approaches. However, SIRA focuses on the negative sentences that might represent problematic features. The approach uses a pre-trained language model (i.e., BERT-CRF) to identify problematic phrases to later cluster them and detect the troublesome features. The pre-trained model is trained with the local data in the study, its hyperparameters tuned, and the resulting model renamed to BERT-Attr-CRF. The newly trained model improves the identification of problematic features from user reviews.

### 3.4.2 API Usage Analysis

Gu et al. [48] propose a learning-based approach for generating API sequences starting from a natural language description. A RNN architecture is used where the encoder is the natural language description of library APIs in their Javadoc documentation, and the decoder represents the sequence of associated API elements. The designed network is trained on a dataset of over seven million pairs of API elements and their respective document annotation.

Ghafari et al. [44] gather unit tests to generate API usage examples. This is particularly useful in cases where libraries are not popular, are new additions to a software ecosystem, or their status is a private one. The approach mines test scenarios to first identify all methods under test and then synthesise API usages based on data and control dependencies of all expressions in a test. Approaches such as the one by Ghafari et al. [44] could boost the features' discovery of lesser-known libraries and extend feature-based comparisons in a software ecosystem.

Gu et al. [47] propose CodeKernel to cluster API usage graphs by embedding them into a continuous space using a graph kernel and applying spectral clustering. Their approach uses a kernel function to transform graphs to their vector representation under different similarity metrics. After the transformation and clustering, CodeKernel selects the most representative graph in each cluster by relying on metrics of centrality and specificity.

Zhang et al. [159] study the adaptations of SO code snippets into development scenarios such as those in GitHub repositories. Their empirical analysis shed light on the different kinds of variations code snippets must pass through to be usable for developers. For example, many of the adaptations to make are related to changes in method calls (e.g., signature of the method or the receiver type) and renaming of variables, fields, and methods to fit adequately into the target program. Based on their empirical analysis on SO snippets and their GitHub equivalents, the designed the ExampleStack recommendation tool. ExampleStack, when given a code snippet,



proposes a list containing similar GitHub code snippets and their possible adaptations.

Zhong and Mei [162] found that single-type usages (i.e., of a single API class) are as common as multi-type usages (i.e., involving more than one API class). Additionally, static methods are found to be frequently called and methods (both static and virtual) are more commonly called than fields. API usages in SO code snippets follow the above findings. Code snippets form in many cases, clusters with a handful of related classes whereas in others the diversity of API types is much higher.

A special place in API analysis is occupied by those approaches targeting API misuses. The detection of misuses has been (and still is) an active research area in the community. The following works span different ranges of API misuse detection: from techniques to target programming languages and platforms.

MAPO [148] extracts method call sequences and extracts API usage patterns by clustering similar sequences and mining frequent patterns in each cluster. Similarly, UP-Miner [139] uses the same technique of clustering and mining, however in this case, UP-Miner performs a two-step clustering process. Call sequences are clustered first to determine clusters of similar sequences. A frequent closed pattern mining algorithm is applied to each cluster to extract API usage patterns. This two-step approach reduces some redundant and repetitive patterns produce by previous approaches such as MAPO [148].

GrouMiner [93] is frequency-based approach to finding API patterns. It is founded on graphs called Groums which include API call and conditional control nodes, as well as temporal order and data dependency edges. The approach transforms the graphs into characteristic vectors to avoid the computational cost of graph isomorphisms. Graphs with the same vector can be considered isomorphic and therefore, increase the occurrence frequency of the graph candidate. If the selected candidate exceeds a predefined threshold, it can be considered as an API pattern.

A statistical approach to recommending API sequences based on Groums is proposed by Nguyen et al. [90]. The approach uses Hidden Markov Models (HMM) to recommend subsequent API calls following a given call. Call sequences extracted from Groums built for Android applications, are the input to the HMM which learns the most likely API to recommend based on preceding API calls.

Amann et al. [8] survey several API usage miners intended for API misuse detection, including GrouMiner [93]. They differ in the techniques used to extract and subsequently analyse API usage for frequent patterns. Their findings indicate a low percentage of detecting precision, a high presence of false positives, and the inability to capture differentiation details. Nonetheless, the analysis also indicates that studied detectors are capable of identifying misuses when they are explicitly provided with correct usages to mine. Moreover, highlighted shortcomings can pave the way for future detectors on improving previous research.

Sven et al. [120] present MuDetect, an API misuse detector based on graphs similar to those used by previous approaches (e.g., GrouMiner [93]). MuDetect constructs API Usage Graphs (AUGs) from control and dependency information about the program under analysis. AUGs can be considered a more detailed variant of Groums which includes more specialised edges (e.g., receiver, parameter, definition, etc.) in its representation. The extraction of API patterns in MuDetect employs the Apriori

frequent sub-graph mining algorithm and requires users to set a frequency threshold.

CPAM proposed by Liu et al. [76] approaches API misuse detection differently. CPAM extracts patterns (also based on frequency) from code changes in commits and thus has a mapping for code that might be highly related to misuses to its repairs. Lastly, the ALP by Kang and Lo [65] approaches the problem of misuse detection as a form of active learning. ALP extracts the previously described AUGs [120] and identifies discriminative sub-graphs from the extracted graphs. Discriminative sub-graphs are a set of sub-graphs from an AUG having the peculiarity that some contain misuses, whereas other extracted sub-graphs are correct. The identification of discriminative sub-graphs enables mapping them to a one-hot vector (i.e., only ones and zeros) where the  $i$ th element with a one indicates the presence of a misuse corresponding to the  $i$ th sub-graph. This vector representation allows the training of a machine learning algorithm that ultimately identifies API misuses.

## 3.5 Limitations and Opportunities for Improvement

Discussed approaches have limitations which can be considered as opportunities to improve their current research. Specifically, shortcomings can be found in two main areas of research, API type resolution and feature discovery from API library usage.

### 3.5.1 Current Limitations in API Type Resolution

The resolution of API types has been the target of many approaches as previously discussed. These approaches rely on parsers that may fall short in handling incomplete and syntactically incorrect code snippets. Therefore, these limitations are also inherited by approaches leveraging them. For example, Table 3.1 shows a considerable number of techniques leveraging Eclipse JDT and PPA (e.g., [37, 100, 105, 146]). Their limitation to handle incomplete code comes from the inflexibility of the Eclipse JDT parser. Another limitation related to the PPA-based solution is that they rely on all available program information extracted from the code to infer types. Such information may not be explicitly referenced in the code of incomplete snippets.

More robust parsers such as ANTLR and island parsers are not frequently used by the described approaches. The former is only used once by Udin et al. [131], whereas the latter only two times (e.g., [99, 131]). From the two mentioned parsers, island parsers [88] have more flexibility towards the analysis of incomplete and incorrect code snippets. These parsers present a more robust and lenient approach to the extraction of relevant information from a text.

A more robust parsing could guarantee the extraction of information (e.g., method calls) from syntactically incorrect code snippets. However, in addition to parsing, a resolution step may be required to incorporate missing information (e.g., FQNs) in an incomplete snippet. As previously mentioned, PPA-based techniques cannot recommend external information from the types in the code. Therefore, there is an opportunity for recommendation-based algorithms to resolve missing information in a code snippet.

In summary, we stress the need for an approach that can extract information from syntactically incorrect code snippets. Such an approach should also resolve missing external API references in incomplete code snippets. The extracted information and the API reference resolution may improve the current SO code processing. Discussed approaches have not addressed these problems commonly found on SO code snippets.

### 3.5.2 Current Limitations in API Feature Discovery

Regarding the automatic discovery of features from API usages, presented approaches in this chapter also have limitations. For example, the work Kanda et al. [64] study a set of automatically extracted API sequences on Android applications and manually label them as features. Inspired in this work, we consider that a fully-automated approach can replace the need for human-labelling intervention. Moreover, we are not aware of similar research on libraries, which could lead the way to more feature-based applications.

Android applications have been more investigated according to the features they might offer. Guzman et al. [51], Sarro et al. [108], Shah et al. [110], Assi et al. [10], and Wang et al. [141] uncover and analyse features from app reviews and descriptions. This same process is not applicable to external libraries. Additionally, library features are more subscribed to the definition by Kanda et al. [64], e.g., API patterns associated to a natural language description. Therefore, the exploration of only the text surrounding API usages, as previously investigated, may not be sufficient to discover library features.

The work by Shen et al. [113], focuses on the functional features described by Kanda et al. [64]. Their approach, called NLI2Code, leverages the natural language text of SO posts to extract feature descriptions and pair them with API references in the posts. Pair references are mined on GitHub to extract API usages, construct a data flow graph and later through frequent graph mining extract API patterns related to the descriptions. NLI2Code is aligned with the feature discovery research on this dissertation. However, the approach uses GitHub for the mining code API patterns instead of relying on SO code snippets. In many cases, the information about API references in SO code snippets is more focused to their text description. Additionally, many studies (e.g., [59, 77, 78, 151]) have shown that SO code has been reused in GitHub projects with additional information from the projects.

Many of the previous works target Android applications. Features in Android applications are more clearly defined due to their relation to UI elements, e.g., “*weather forecast*”. Stack Overflow stands out as the source of information from where to uncover features from libraries. We only found the work by Shen et al. [113] exploring the feature research domain while leveraging SO post information. However, that work only extracts the natural language information in SO posts and not the API patterns within code snippets. Instead, they explore GitHub API usages as an alternative to SO code snippets. SO code snippets are more focused on the API references of interest, unlike code on GitHub repositories.

To summarise, there is a need for an approach that extracts API patterns from library usages in SO code snippets. This approach should process the natural language text and automatically name the extracted patterns. Unlike Shen et al. [113], the

approach should investigate patterns from SO snippets since they are more focused to the task discussed on the post.

## 3.6 Conclusion

This chapter presented the current state of the art on several works related to the research on this dissertation. Approaches discussed in this chapter include development tools that incorporate SO information, program analyses for SO code snippets and works proposing embeddings for source code analysis. In addition, research on library usage comprehension is also explored consisting of feature discovery and API usage analysis works. Explored approaches in this chapter have limitations related to code snippet analysis and the discovery of features from library API usages. Some detected limitations are inherited by the adopted technology (e.g., Eclipse JDT), while others are determined by the research domain (e.g., feature discovery from API usages). These limitations present research opportunities to improve the current state of the art. In the next chapters we will target the discussed limitations through our approaches, API type resolution for incomplete code snippets and feature discovery based on API usages from SO posts.

## Chapter 4

# API Type Resolution for Incomplete Code Snippets on Stack Overflow

### 4.1 Introduction

Developers may consult several sources of information online. Stack Overflow is a platform where users can post questions answered by others with expertise in the domain. SO posts often contain code snippets, e.g., to illustrate how to use the API of a library. However, such code snippets may miss the type declarations and package import statements that constitute a compilation unit. They may contain API usages without any syntactic reference to the library that provides the API. Even if the library name is mentioned in the text surrounding the code snippet (e.g., Guava), the fully-qualified name (FQN) of the types to import (e.g., `com.google.common.*`) can be challenging to resolve.

Determining which library types to import for a given code snippet is a problem shared by several tools that rely on SO. For example, many of the works discussed in Chapter 3 (e.g., [95, 106, 118, 119, 122, 150, 163]) propose a solution to resolve missing information, such as FQNs, in incomplete code snippets. RESICO differs from each of the previous approaches either in the technique or in the problem formulation and its respective solution.

This chapter presents RESICO (RESolution in Incomplete COde), a new learning-based *text classification* approach to the problem of resolving API types in incomplete code snippets. This approach will assist our feature extraction approach which is proposed in the next chapter by being part of its pipeline in Chapter 6. RESICO's prediction capabilities will assist our feature extraction approach by extending the number of SO posts related to a library with those that are not necessarily tagged with the name of the library. The resolved answers by RESICO will be processed by our feature extraction approach (cf. Chapter 5).

**RESICO embraces the hypothesis that the API elements used within a snippet and the context in which this usage occurs often suffice to resolve the simple name of a type within the snippet to its fully qualified one.** Word2Vec plays a fundamental role in RESICO, where it is used to learn vector representations for the API elements and their usage contexts within a dataset, so relying on a suboptimal predetermined one can be avoided. Once vector representations or word embeddings have been

learnt, RESICO vectorises the dataset and trains a multi-class (i.e., each FQN corresponds to a class) machine learning algorithm on the resulting vectors. RESICO supports using any supervised machine learning classifier to train on the vectorised dataset and considers FQNs as the label to predict. At resolution time, RESICO returns the most likely FQNs for all simple names in a given code snippet, regardless of whether the snippet is incomplete or syntactically incorrect.

Like StatType [95] and COSTER [106], RESICO is applicable in contexts where the crowd’s wisdom is to be preferred over the deductive reasoning of Baker [118, 119] (cf. Chapter 3), e.g., when snippets are small and contain few expressions that impose semantic constraints on ambiguous API names. Although RESICO is a machine learning approach such as StatType, it features a different approach to solving the problem. StatType considers API type resolution as a *sequence-to-sequence* task. In contrast, RESICO relies on a classification procedure where a learned context influences the class prediction (i.e., the FQN of the API reference). COSTER considers the surroundings of API references like RESICO, but it does not perform any learning to improve context comparisons further. We chose COSTER as the baseline to compare to as previous studies showed that it is similar or superior in performance to StatType. Furthermore, despite repeated attempts, we could not obtain an implementation of StatType from the authors.

We evaluate RESICO and COSTER extensively on four datasets: one gathered from a corpus of 50K compilable GitHub projects and three datasets that serve as external validators for the trained models. Our approach is more complex than COSTER since it involves training several machine learning models; hence, it consumes more computational resources during training. Despite being slower to train, RESICO outperforms COSTER in all experiments we conducted. We also performed a root cause analysis of the type resolution failures of the two approaches. More specifically, we measured how many of the failures were due to simple names being ambiguous.

The remainder of the sections in the chapter is structured as follows. First, a motivation section provides arguments towards the necessity of context-based approaches for API resolution. Second, the approach to resolving missing FQNs is presented in detail. Third, a thorough evaluation is performed on RESICO in comparison to the state-of-the-art tool and approach COSTER. Fourth, more results of the evaluation are further discussed and analysed to subsequently, present the limitations of RESICO and its potential impact. Threats to the validity of the approach and evaluation are also discussed to finally conclude the chapter about learning-based type resolution.

## 4.2 Motivation

Each example in Listing 3 illustrates a real-world incompleteness issue that can be encountered on SO. The issues in the code snippets of Listing 3 could be considered as intractable for classic parsers and static analysis tools. Unfortunately, this is very frequent on the SO platform, which represents a reuse problem for the developer community. The incompleteness issues that code snippets in Listing 3 experience are the following:

### Incomplete Structures

Java compilers expect compilation units that group type declarations together with

```

1 // 1st code snippet
2 // URL: https://stackoverflow.com/questions/21936577
3 Objects.toString(gearBox, "")
4 Objects.toString(id, "")
5
6 // 2nd code snippet
7 // URL: https://stackoverflow.com/questions/8897384
8 Ordering<Map.Entry<Key, Value>> entryOrdering =
9     ↪ Ordering.from(valueComparator)
10    .onResultOf(new Function<Entry<Key, Value>, Value>() {
11        public Value apply(Entry<Key, Value> entry) {
12            return entry.getValue();
13        }
14    })
15    .reverse();
16
17 // 3rd code snippet
18 // URL: https://stackoverflow.com/questions/2319126
19 Iterator<?> i = queue.iterator();
20 ...
21 Object next = i.next();
22 i.remove();

```

LISTING 3: Real code snippets with different issues related to their incompleteness.

import statements. On Q&A platforms, in contrast, users frequently post sequences of standalone Java statements or expressions that do not form a compilation unit. This is the case for all code snippets in Listing 3, which is an issue for compilers and most code analysis tools alike.

### Missing Variable Declarations

Another source of incompleteness are references to undeclared variables. This is the case for `gearBox` and `id` in Snippet 1, `valueComparator` in Snippet 2, and `queue` in Snippet 3. The natural language semantics of their names or the Q&A text around them could provide hints about their types, but the information itself is missing from the code.

### Missing Import Statements

Similarly, library types might be referenced by their simple name while an explicit import statement for the corresponding declaration is missing. `Ordering`, `Map`, `Function`, and `Entry` in the second snippet are examples of library types (e.g., from Google Guava) that are referenced in the snippet without their declaration being imported.

### Name Ambiguities

Missing import statements cause another problem typical of incomplete code snippets: name ambiguities. Many libraries could share the same simple name for types and methods, encumbering the resolution of a simple name to a fully qualified one.



For example, the Function name in snippet 2 could be a reference to either of the interfaces `java.util.function.Function` or `com.google.common.base.Function`. This is a problem for both users and tools needing to reuse or reason about the snippets.

Despite their incompleteness, code snippets on Q&A platforms are an important source of information for both developers and tools. To fully realise their potential, a reliable and effective approach to resolving the simple name of API types in a code snippet to their FQNs is needed. The next section proposes such an approach, capable of predicting the most likely FQN for simple names of API types by leveraging similarities in name usage contexts among other snippets on the Q&A platform.

## 4.3 RESICO: A Type Resolution Approach for Incomplete Code Snippets

### 4.3.1 A Prime on RESICO

RESICO (RESolution in Incomplete Code) is a new learning-based *text classification* approach to the problem of resolving API types in incomplete code snippets. The approach embraces the hypothesis that the API elements used within a snippet and the context in which this usage occurs often suffice to resolve the simple name of a type within the snippet to its fully qualified one. Word2Vec (cf. Section 2.3.2) plays a fundamental role in RESICO, where it is used to learn vector representations for the API elements and their usage contexts. Once vector representations or word embeddings have been learnt, RESICO vectorises the dataset and trains a multi-class (i.e., each FQN corresponds to a class) machine learning algorithm on the resulting vectors. RESICO supports using any supervised machine learning classifier to train on the vectorised datasets and considers FQNs as the label to predict. At resolution time, RESICO returns the most likely FQNs for all simple names in a given code snippet, regardless of whether the snippet is incomplete or syntactically incorrect, thanks to the custom island parser designed to extract information from snippets.

Figure 4.1 shows examples of the output that can be expected. Lines of code from the input snippet are depicted in black. For this snippet, we removed all import statements from which the FQNs could be resolved. We annotated the API references to be resolved by RESICO with a rectangle. The line in red and italics depicts the computed FQN for each API reference.

```
[org.hibernate.cfg.Configuration]
[Configuration] c = new Configuration();

[-]
[c.configure]("hibernate.cfg.xml");

[org.hibernate.SessionFactory]
[SessionFactory] se = [c.buildSessionFactory]();
[org.hibernate.cfg.Configuration]

[org.hibernate.Session]
[Session] session = [se.openSession]();
[org.hibernate.SessionFactory]

[org.hibernate.Transaction]
[Transaction] tran = [session.beginTransaction]();
[org.hibernate.Session]
```

FIGURE 4.1: Example type resolutions computed by RESICO.

In all cases but one, the predicted FQN was correct compared to previously removed import statements. RESICO could precisely resolve the FQN for most API references despite multiple candidates with the same simple name. For example, the



simple names `Configuration`, `SessionFactory`, `Session` and `Transaction` ambiguously denote 44, 4, 67 and 14 different FQNs respectively in one of our datasets.

For the simple name not annotated (i.e., a dash is depicted instead), RESICO could not correctly predict the FQN of the variable `c`, which was correctly predicted for the line above (i.e., `Configuration`). However, if all other predicted FQNs are added via import statements, the missing FQN will not preclude compilation any more. Before detailing how RESICO resolves simple names to their FQN, we briefly describe the Eclipse JDT as it features prominently in the remainder of the chapter.

### 4.3.2 An Overview of Eclipse JDT for Facts Extraction

**Eclipse Java Development Tools (JDT)**<sup>1</sup> is a set of plugins developed for the Eclipse platform that enable lightweight static analysis of Java programs. The Eclipse JDT comprises five components: APT, Core, Debug, Text, and UI. Each component is independent and specialised in a different purpose. For example, COSTER [106] and RESICO use the Eclipse JDT Core component for their fact extraction from compilable Java programs (e.g., information about method invocations and variable declarations). It can be used headless without the Eclipse IDE and provides, among others, ASTs and symbol and type hierarchy information for Java programs. As we use the Eclipse JDT Core component frequently and none of the other components, we refer to the former as Eclipse JDT throughout the dissertation.

### 4.3.3 Training Process

Figure 4.2 depicts the steps in the training process.

This subsection uses the snippet depicted in Listing 4 as a running example.

In the first step of the training process, depicted as action node 1, the Eclipse JDT is used to I) compile the Java programs in the training corpus, and II) extract the simple names of API references and their FQNs as follows:

- for variable declarations, the simple name and FQN of the declared type.
- for variable and field accesses, the simple name of the accessed type, the name of the field, and the FQN of the accessed type.
- for method invocations, the simple name and FQN of the statically-declared type of the receiver expression, and the identifier of the invoked method.
- the line numbers for each previous construct.

The information provided by the JDT is used to perform this extraction step. For example, for the method invocation on Line 18 of Listing 4, RESICO first collects `TelephonyManager` (the simple name or type of the receiver variable `tm`), `getDeviceId` (the method identifier), and `android.telephony.TelephonyManager` (the FQN of the receiver type). The latter represents the label to predict by RESICO, whereas the former is part of the training data.

Figure 4.2 shows the information collected for each program as data node *B*. The gathered data so far is further augmented with the context surrounding the API

<sup>1</sup> <https://www.eclipse.org/jdt/>

```

1 import org.apache.cordova.DroidGap;
2 import android.context.Context;
3 import android.telephony.TelephonyManager;
4 import android.webkit.WebView;
5 import android.webkit.JavascriptInterface;
6
7 public class GetNativeTelephonyManager {
8     private WebView mAppView;
9     private DroidGap mGap;
10
11     public GetNativeTelephonyManager(DroidGap gap, WebView view) {
12         mGap = gap;
13         mAppView = view;
14     }
15     @JavascriptInterface
16     public String getIMEI() {
17         TelephonyManager tm = (TelephonyManager)
18             ↪ mGap.getSystemService(Context.SERVICE);
19         String imeiID = tm.getDeviceId();
20         return imeiID;
21     }
22 }

```

LISTING 4: Running example for explaining RESICO.

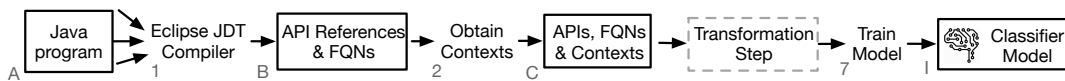


FIGURE 4.2: Training based on a corpus of programs.

reference. We define the context of an API reference as the information extracted for all other API references in the same method body but without their FQNs. In this way, RESICO can extract the same information from complete and incomplete code snippets, since the FQNs are used neither in the API reference nor in the contexts. Finally, RESICO collects the vocabularies for both API references and contexts which will serve as input to train Word2Vec models later in the process. The vocabulary for contexts is obtained by adding all contexts into a single set of words. On the other hand, the vocabulary for the API references comprises the set of API references themselves.

To continue with our running example, from the API reference `tm.getDeviceID` on Line 18, its surrounding context is the following:

TelephonyManager, DroidGap, getSystemService, Context, SERVICE, String, String

Within Figure 4.2, the context extracted for each API reference is depicted as data node C in addition to the previously collected information. Table 4.1 shows for every API reference (column *API Ref.*), the line numbers where they are located (column *#*), their contexts (column *Context*) and their respective FQNs (column *FQN*) that RESICO is able to extract from the method `getIMEI()` from Listing 4. As observed, many API references could be present

on the same line, enlarging the collected dataset even in small methods such as `getIMEI()`.

#	API Ref.	Context	FQN
17	TelephonyManager	DroidGap, getSystemService, Context, SERVICE, String, TelephonyManager, getDeviceID, String	android.telephony.TelephonyManager
17	DroidGap, getSystemService	TelephonyManager, Context, SERVICE, String, TelephonyManager, getDeviceID, String	org.apache.cordova.DroidGap
17	Context, SERVICE	TelephonyManager, DroidGap, getSystemService, String, TelephonyManager, getDeviceID, String	org.android.Context
18	String	TelephonyManager, DroidGap, getSystemService, Context, SERVICE, TelephonyManager, getDeviceID, String	java.lang.String
18	TelephonyManager, getDeviceID	TelephonyManager, DroidGap, getSystemService, Context, SERVICE, String, String	android.telephony.TelephonyManager
19	String	TelephonyManager, DroidGap, getSystemService, Context, SERVICE, String, TelephonyManager, getDeviceID	java.lang.String

TABLE 4.1: Extracted information from the method `getIMEI()` in Listing 4 by RESICO.

Figure 4.3 zooms in on the following transformation step. We use Word2Vec [86, 87] to vectorise API references and contexts, whereas label categorisation is used to convert FQNs to label numbers.

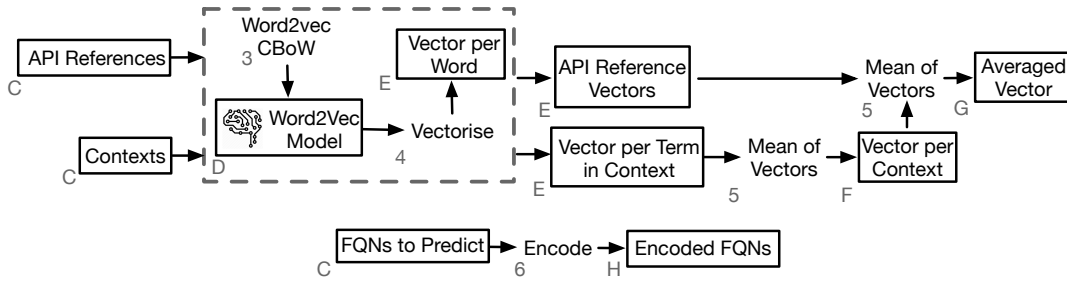


FIGURE 4.3: Transformation step used by the training process.

Word2Vec is used to learn the best possible vector representation of API usages and contexts (columns *API Ref.* and *Context* of Table 4.1) and to obtain a word embedding per word in both cases. Each API reference is considered as a single word. API reference records with two words (e.g., `DroidGap, getSystemService` on line 17) are concatenated with a dot (e.g., `“.”`) as it is done for a field or method call. However, words in context records are kept separate and considered individually in the training phase. This decision was made because API references can take the form of variable declarations as well as method calls and field accesses. The former only comprises a single relevant word (e.g., the name of the API type), while the latter shall consist of two (e.g., the field or method’s name along with the name of the API type). Contexts, in contrast, comprise the simple names stemming from surrounding API references and each word is therefore considered equally relevant.

Step 3 iterates once over all extracted API references and contexts while using the Word2Vec CBoW neural network architecture to train the algorithm (cf. Section 2.3). We used the implementation of the Word2Vec model in

the Golang programming language<sup>2</sup> to take advantage of its Goroutines. We configured the Word2Vec training for the API usages and contexts, similarly, relying on the CBOW architecture, a batch size of 1000 words, 20 generated features, negative sampling as the optimiser, five words of window size, a minimum occurrence of a word equals to one (i.e., considering all words), and five iterations for the training of the model. The number of features was set to 20 to enable us to perform the following steps (e.g., hyperparameter optimisation) more efficiently without reducing the ability to capture the semantic differences between the labels to predict. The number of iterations for the learning was set to five to keep the training process as simple as possible. The batch size considers numerous words per training batch with a window size of five to take short segments around the word to learn. The latter is also the default window size for the Word2Vec implementation in the Python library Gensim as it is the selection of negative sampling as optimiser.<sup>3</sup> No other changes specific to the domain were made.

The resulting models or word embeddings were saved in external files and are depicted as the data node *D*, one for the API references and another one for the contexts. Once the word embeddings have been learnt, step 4 transforms the input data into the learnt vectors. The vectors obtained for each word (data node *E*) are kept in the case of the API references and further averaged in the case of contexts in step 5. Data node *F* depicts the averaged vectors per context.<sup>4</sup>

To illustrate the transformation described above, Table 4.2 depicts the vector representation of the information in Table 4.1. In this case, column *API Vector* denotes a transformed API reference, whereas column *Cont. Vect.* denotes the corresponding vector for each word in its context. The averaged vector of all vectors in a context is shown in column *Av. Cont. Vect.*. This vector is stored in data node *F*, which will be used in the following steps. The column *FQN* remains the same, and it will be transformed in a subsequent step.

A final averaged vector is obtained in data node *G* by considering each vector of API references and its corresponding averaged context vector. In parallel to the vectorisation, step 6 encodes the FQNs collected in data node *B*. A natural number is assigned to each FQN, denoting the label to be predicted by the classifier. Table 4.3 exemplifies this last transformation step for our dataset. Column *Av. Embedding Vector* contains the final averaged vector, and column *FQN* contains the transformed FQNs as numbered labels. In machine learning concepts, the former is the training dataset, while the latter corresponds to the class to predict.

The last step in the training process (action node 7) is responsible for training a supervised machine learning classifier (data node *I*) and for saving the

<sup>2</sup> <https://github.com/ynqa/wego>

<sup>3</sup> <https://radimrehurek.com/gensim/models/word2vec.html>

<sup>4</sup> Please remember that contexts are composed of several words.

#	API Vector	Cont. Vect.	Av. Cont. Vect.	FQN
17	[0.684, ..., 0.457]	[-0.158, ..., -0.378], ..., [-0.728, ..., -0.629]	[-0.332, ..., 0.643]	android.telephony.TelephonyManager
17	[-0.154, ..., 0.254]	[0.029, ..., -0.916], ..., [0.904, ..., 0.601]	[0.378, ..., -0.173]	org.apache.cordova.DroidGap
17	[-0.804, ..., 0.915]	[0.807, ..., 0.092] ..., [0.277, ..., 0.592]	[0.155, ..., 0.791]	org.android.Context
18	[0.218, ..., 0.613]	[0.167, ..., -0.805], ..., [0.767, ..., 0.466]	[-0.464, ..., 0.608]	java.lang.String
18	[0.349, ..., 0.505]	[0.092, ..., 0.397], ..., [-0.336, ..., 0.719]	[0.047, ..., 0.927]	android.telephony.TelephonyManager
19	[0.218, ..., 0.613]	[0.167, ..., -0.805], ..., [0.767, ..., 0.466]	[-0.464, ..., 0.608]	java.lang.String

TABLE 4.2: Transformed API references and contexts from Table 4.1 by the Word2Vec models.

#	Av. Embedding Vector	FQN
17	[-0.756, ..., 0.628]	0
17	[0.118, ..., 0.112]	1
17	[0.860, ..., -0.145]	2
18	[0.253, ..., 0.183]	3
18	[0.092, ..., -0.792]	0
19	[0.253, ..., 0.183]	3

TABLE 4.3: The last transformation step in the RESICO process. The previous context vector is further averaged with the API vector, and FQNs are converted into numbers.

resulting model into a file. RESICO supports the use of any supervised machine learning classifier with the data extracted at this point. For our experiments, we selected four machine learning classifiers previously used in text classification tasks: KNN, Random Forests, Ridge Linear, and Support Vector Machines (cf. Chapter 2). **The outcome of the training process is a model to predict a numbered FQN given an averaged vector.** The averaged vector is the mean of a vector  $A$  for an API reference and a vector  $B$  which is the mean vector of all vectors in the surrounding context of the mentioned API reference.

#### 4.3.4 Resolution Process

The resolution process for code snippets depicted in Figure 4.4 resolves all API type references within a code snippet to their FQNs. The process has to parse a snippet in a *non-standard* way due to the many issues a SO code might have (cf. Section 4.2). RESICO uses a custom island parser [88] (action node 8) configured to parse SO code snippets regardless of their syntactical

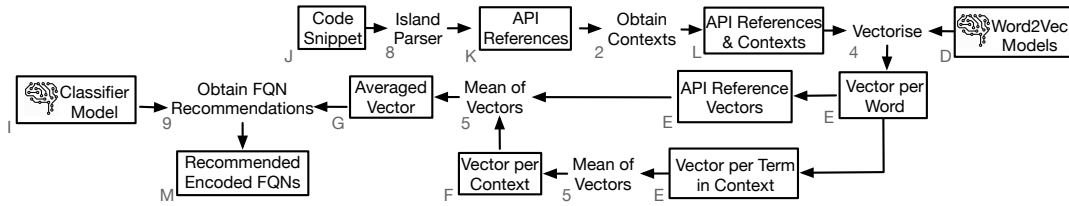


FIGURE 4.4: Resolution process for API type references in code snippets.

correctness. Such parsers focus on the constructs of interest (i.e., API references in our case) and consider the remainder of the code as water. It is implemented using the framework *Parboiled*<sup>5</sup> for Java.

API references are gathered in data node *K*, for which our supervised machine learning models will predict missing FQNs. The contexts of API references are also collected similarly as previously described for a complete code example.

After gathering API references and their respective contexts in an incomplete code snippet, RESICO will transform the input to make it suitable for the resolution phase. The vectorisation of API references and their contexts relies on the formerly trained Word2Vec models saved in external files (data node *D*), respectively. The availability of each API reference word and its context words are checked in the trained Word2Vec models. If the API reference word cannot be found (i.e., the API Word2Vec model was not trained with it), the process stops since API references are fundamental for the prediction. However, if one or multiple context words cannot be found (i.e., the context Word2Vec model was not trained with them), the resolution process continues with other context words as this scenario is more likely to happen, and contexts are usually composed of several words. Nonetheless, if there are no context words, RESICO will also likely fail since in this case, it only depends on the API reference word.

Each vector is processed to obtain an averaged vector that will serve as input to the trained models in data node *I*. These trained models are loaded from the previously saved files. Classifiers have different ways of predicting the class for an unseen input; for example, KNN calculates the closest neighbours and takes the majority class among them. The outcome of the machine learning model therefore depends on how the classifier calculates the most likely class for its final prediction. In general, their output takes the form of a number that corresponds to a particular FQN and needs to be mapped back to the original FQN.

<sup>5</sup> <https://github.com/sirthias/parboiled>



### 4.3.5 Providing Top-K Recommendations

Most supervised machine learning classifiers predict the class with the highest probability for a certain input vector. This probability is calculated based on the similarity of the input vector to already trained vectors in the model. By default, trained models make a Top-1 prediction, returning the class that achieved the highest probability for a certain input vector. However, it is common for implementations to provide access to the internal probabilities for all predicted classes. For example, in the Python Scikit-Learn framework, method `predict_proba`<sup>6</sup> is provided by most implementations. RESICO will therefore return a Top-K with  $1 \leq K \leq N\_Classes$  for its resolutions of API type references. If the actual FQN of the reference is among the Top-K resolutions returned by RESICO, the resolution is considered successful; otherwise, the resolution represents a failure of the trained model.

## 4.4 Evaluation

We describe the design and results of the empirical evaluation conducted to assess the RESICO machine learning classifiers (from now onwards, RESICO classifiers) for API type resolution. The evaluation compares our approach to COSTER [106], an information retrieval-based approach, which outperforms in several circumstances earlier methods such as StatType [95] and BAKER [119]. We contacted COSTER’s authors to verify that we configured and used the tool correctly before conducting our evaluation.

Our study aims to answer the following research questions:

- RQ<sub>1</sub> What are the best hyperparameter combinations for the classifiers used within RESICO?
- RQ<sub>2</sub> How well do COSTER and the RESICO classifiers perform on instances extracted from the dataset used for training?
- RQ<sub>3</sub> What is the performance of the COSTER and RESICO classifiers when evaluated on unseen datasets?
- RQ<sub>4</sub> How much time is needed to train COSTER and the RESICO classifiers?
- RQ<sub>5</sub> To what extent do ambiguities in simple names influence the performance of the approaches?

Figure 4.5 depicts a graphical overview of the steps we took to answer these research questions. Our evaluation started by gathering the datasets needed to train and evaluate RESICO and COSTER. We collected one dataset to train and evaluate the models, and three additional datasets to analyse the prediction capabilities of the trained models. Given the imbalanced nature of the dataset used to train the models, we applied data balancing. The same balanced dataset is also used to optimise the hyperparameters of the RESICO

<sup>6</sup> [https://github.com/scikit-learn/scikit-learn/blob/f3f51f9b6/sklearn/neighbors/\\_classification.py#L256](https://github.com/scikit-learn/scikit-learn/blob/f3f51f9b6/sklearn/neighbors/_classification.py#L256)

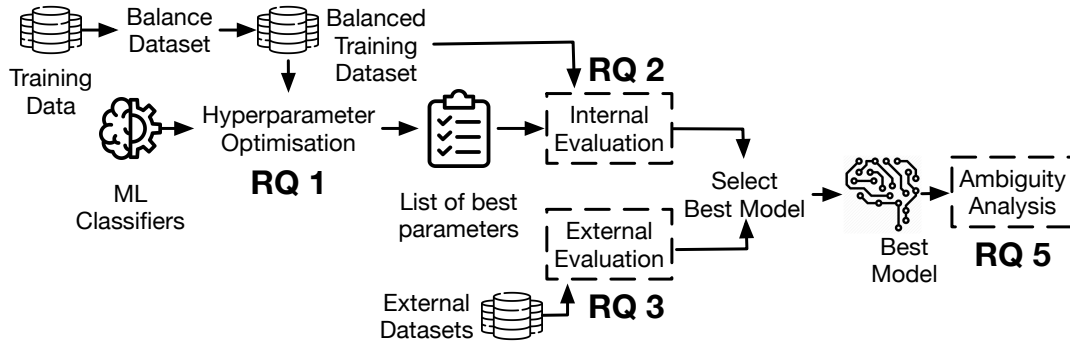


FIGURE 4.5: Overview of our evaluation approach.

classifiers in *RQ1*. In *RQ2*, we use the best configuration of hyperparameters for each RESICO classifier to conduct an internal evaluation with the balanced dataset and compare our approach against COSTER. We also perform an external evaluation in *RQ3* to assess the performance of the two approaches on other datasets. For *RQ4*, we present the performance results in terms of training time required by each approach. Finally, we perform an ambiguity analysis in *RQ5* on the best RESICO and COSTER models.

#### 4.4.1 Datasets Collection

In this section, we report on the datasets used in our empirical study: the dataset used to train and tune models and to evaluate their performance on similar data in *RQ2*, and the three datasets used to evaluate the obtained models on different data in *RQ3*.

##### Internal Dataset

We relied on the 50K-C dataset to answer *RQ1*, *RQ2* and train the models (i.e., COSTER and all RESICO classifiers) that are further evaluated on external datasets in *RQ3*. The dataset was extracted from a collection of 50K compilable Java projects mined from GitHub [85]. We followed the same extraction process previously used by Saifullah et al. [106] for COSTER, using the same Eclipse JDT extractor configuration for the internal datasets of the two approaches. We did it this way to ensure that the internal datasets on which COSTER and RESICO are trained and evaluated were constructed with the same API references and to avoid bias in the evaluation towards either approach. After the API references are extracted, we build the surrounding contexts for each and store the API reference, its context and its corresponding FQN.

We consider not only the 100 most frequent libraries as reported in Saifullah et al. [106] but all 5,356 libraries provided by the 50K-C dataset. From an initial pool of 50,000 projects, we extracted the API references, contexts, and FQNs for 48,951 (i.e., 98%). The resulting dataset contains 19,088,813 records, each representing an API reference.



### Data Balancing

The dataset gathered in the previous step is highly imbalanced. As can be expected, it contains more instances related to some commonly used FQNs (e.g., `java.lang.String` with more than 2M occurrences) and lacks instances of some rarely used FQNs (e.g., `java.sql.Statement[]` with only one occurrence). Figure 4.6 shows a fragment of the data distribution for the three most and three least frequent types out of the 39,643 unique FQNs in the dataset.

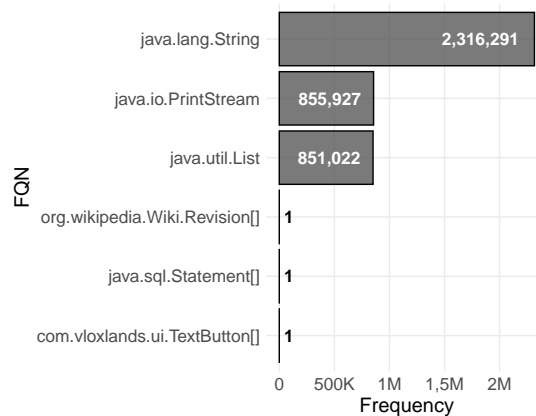


FIGURE 4.6: The three most and least frequent FQNs in the gathered dataset.

In this highly unbalanced setting, we decided to sample the previously extracted dataset to balance the training data and, thus, avoid introducing a bias towards any FQN to be predicted. We selected a threshold of 50 occurrences, as this threshold was previously selected in COSTER [106]. The FQNs with fewer occurrences than the defined threshold are not considered for the training phase and are therefore excluded. Those records with more FQNs than the threshold are randomly sampled into 50 instances. The resulting balanced dataset consists of 4,860 unique FQNs with 50 instances per FQN, amounting to 243,000 records. The new balanced dataset constitutes the internal training dataset, and it will be also used for the internal evaluation of both approaches.

### External Datasets

**Three external datasets are used to increase the generalisability of the results and to answer RQ3.** Table 4.4 shows the characteristics of the datasets considered for the evaluation. Eclipse JDT can parse and compile the snippets in these datasets to (I) extract the referenced API types and their surrounding contexts and (II) use their FQNs as ground truth.

Two of the datasets *COSTER-SO* [106] and *StatType-SO* [95] contain 401 and 245 code snippets respectively (col. *Snippets*), which have been previously used to assess COSTER.<sup>7</sup> After processing the code snippets, the number of

<sup>7</sup> <https://zenodo.org/record/7244690>

Dataset	Snippets	Fs	I-Fs	E-Fs	U-Fs	UI-Fs	UE-Fs
COSTER-SO	401	1,373	1,330	43	30	20	10
StatType-SO	245	1,827	431	1,396	167	39	128
RESICO-SO	371	1,741	596	1,145	215	47	168

TABLE 4.4: Datasets used for the external evaluation of COSTER and RESICO.

records extracted from *COSTER-SO* and *StatType-SO* were 1,373 and 1,827 respectively (col.*Fs*), including API references, their contexts and FQNs.

A closer look at the FQNs of *COSTER-SO* raises concerns about their distribution in the dataset. 1,330 out of the 1,373 references (96.7%) belong to the Java standard library (col.*I-Fs*) (i.e., the default Java Runtime Environment). Simple names with FQN prefixes starting with `java.lang` or `java.io` are considered as always observable<sup>8</sup> and thus, less significant for the potential users of the approaches. Therefore, a more diverse dataset with a prevalent number of non-default FQNs is desirable.

Alternatively, the *StatType-SO* dataset contains an increased number of external FQNs (76.4%) (col.*E-Fs*) with regards to *COSTER-SO*. Such a behaviour is also reflected in the number of unique external FQNs (col.*UE-Fs*) with only 10 for *COSTER-SO* and 128 for *StatType-SO*. To further increase the evaluation setup, we created another dataset (*RESICO-SO*) that replicates the distribution of FQNs in the *StatType-SO* dataset and possibly improves the number of unique external FQNs compared to previous datasets. This new dataset represents the third dataset considered to verify the generalisability of the results.

We randomly selected 371 code snippets from Stack Overflow referencing the same 11 libraries as *COSTER-SO* and *StatType-SO*. The 371 code snippets represent a statistically significant sample from the 11,047 Java code snippets with import statements in the SOTorrent dataset [13] dated March 15<sup>th</sup>, 2020 (95% Confidence Level and 5% Confidence Interval). We manually ensured that the snippets could be parsed and compiled using Eclipse JDT. The new dataset, named *RESICO-SO*, comprises 1,741 API references, their contexts, and FQNs. Most records (1,145 i.e., 65.8%) are references to external FQNs. At the same time, *RESICO-SO* exhibits a more extensive number of unique FQNs (215) with a larger number of unique internal (47) and external (168) FQNs. These aspects make the new dataset more challenging to predict than previous ones and might reinforce the results obtained for the *StatType-SO* with which it shares a similar FQN distribution.

<sup>8</sup> <https://docs.oracle.com/javase/specs/jls/se8/html/jls-7.html#jls-7.4.3>

### 4.4.2 RQ1. What are the best hyperparameter combinations for the classifiers used within RESICO?

#### Design

This research question investigates the best hyperparameter configuration on the selected machine learning classifiers for the RESICO approach (cf. Chapter 2).

All considered classifiers have default hyperparameters in their implementation. These default parameters should have an average performance in various applications and are not optimised for any particular task or dataset. Each classifier has a different set of hyperparameters with a range of possible values to choose from, representing a *search space*.

The search space for some hyperparameters is strictly limited to a group of options in a list, e.g., the hyperparameter *weights* for the KNN classifier restricts the possibilities to 'uniform' and 'distance' to calculate the space between neighbours.<sup>9</sup> In other cases, this limitation does not exist; therefore, the search space for such hyperparameters could be infinite. For instance, the parameter *alpha* of the Ridge linear classifier<sup>10</sup> can take any possible float as value. Additionally, a set of hyperparameters usually consists of more than one parameter, rendering searching for the best parameters a multi-objective optimisation problem.

We relied on the optimisation libraries Optuna<sup>11</sup> and HyperOpt<sup>12</sup> to perform the multi-objective search. Optuna was used for those machine learning algorithms that do not heavily demand computer resources, such as KNN, the Ridge linear classifier (RL) and the linear Support Vector Classifier (SVC). We took advantage of the distributed hyperparameter optimisation of HyperOpt to search for the best parameters for the Random Forest classifier without incurring memory overflow issues. The setup of the two libraries was similar, with 200 trials for each classifier and using the Tree of Parzen Estimators (TPE) [19]. Using TPE is recommended over other search space algorithms such as Random Search [18].

In addition to the similar setup, we also defined a similar goal for each search. More specifically, the goal was to minimise a loss function defined as  $1 - F1$ . In other words, the search for the best hyperparameter tried to minimise the difference between the maximum  $F1$  score (e.g., 1) and the obtained score. When the difference reached a minimum, the optimal parameters were found.

Table 4.5 shows the selected machine learning classifiers, their hyperparameters, a brief description, and their configured search space.

<sup>9</sup> <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>

<sup>10</sup> [https://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.RidgeClassifier.html](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RidgeClassifier.html)

<sup>11</sup> <https://optuna.org/>

<sup>12</sup> <https://hyperopt.github.io/hyperopt/>

Classifier	Hyperparameter	Brief description	Search Space
K-Nearest Neighbors (KNN)	<i>n_neighbours</i>	Number of neighbours.	$2 \leq x \leq 1E3$
	<i>weights</i>	Weight function.	['uniform', 'distance']
	<i>algorithm</i>	Algorithm to compute the nearest neighbours.	['ball_tree', 'kd_tree', 'brute']
	<i>leaf_size</i>	Leaf size passed to BallTree or KDTree.	$2 \leq x \leq 1E3$
Random Forest (RF)	<i>n_estimators</i>	Number of trees in the forest.	$10 \leq x \leq 200$
	<i>criterion</i>	Function to measure the quality of a split.	['gini', 'entropy']
	<i>min_samples_leaf</i>	Minimum number of samples to be at a leaf node.	$1 \leq x \leq 20$
	<i>min_samples_split</i>	Minimum number of samples to split an internal node.	$0.0 \leq x \leq 1.0$
Ridge Linear (RL)	<i>alpha</i>	Regularization strength.	$1.0 \leq x \leq 1E10$
	<i>solver</i>	Solver to use to compute the Ridge coefficients.	['auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga']
Support Vector Classifier (SVC)	C	Regularization parameter.	$1.0 \leq x \leq 1E10$

TABLE 4.5: Hyperparameters of the classifiers and their search space configuration

Note that we had to bound some hyperparameters with unbounded limits to sufficiently large limits when tuning the classifiers. Also, note that some intervals are floating point ranges while others consist of integer numbers. When the limits are shown as floats in Table 4.5, the range of possible values belongs to the former case, whereas integer values indicate the latter case.

## Results

The overview of the hyperparameter optimisation process for the classifiers is shown in Figure 4.7. Categorical parameters are encoded as numbers.

For example, for KNN, the values of the hyperparameter *algorithm* are transformed as follows *ball\_tree*  $\rightarrow 0$ , *brute*  $\rightarrow 1$ , *kd\_tree*  $\rightarrow 2$  and the values of the hyperparameter *weights* are converted as *distance*  $\rightarrow 0$ , *uniform*  $\rightarrow 1$ .

For the RidgeLinear classifier the values of *solver* are changed to *auto*  $\rightarrow$  0, *cholesky*  $\rightarrow$  1, *lsqr*  $\rightarrow$  2, *sag*  $\rightarrow$  3, *saga*  $\rightarrow$  4, *sparse\_cg*  $\rightarrow$  5, *svd*  $\rightarrow$  6. Lastly, the values of the RF hyperparameter *criterion* are transformed to *gini*  $\rightarrow$  0, *entropy*  $\rightarrow$  1.

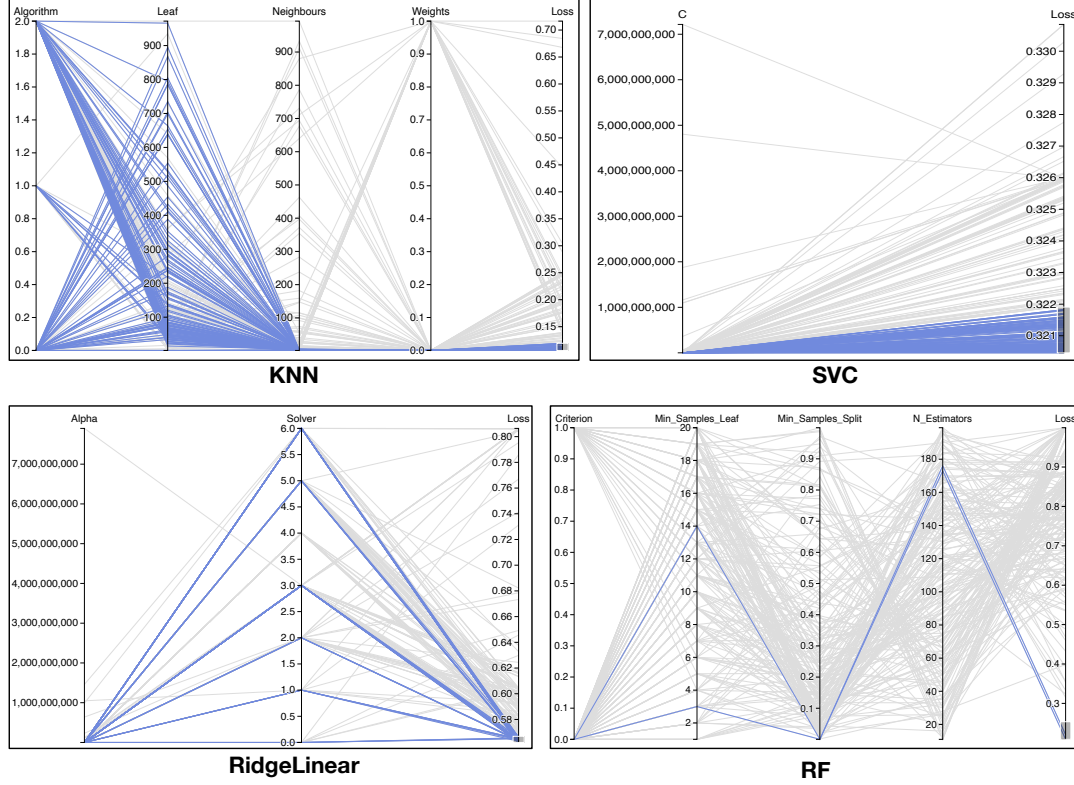


FIGURE 4.7: Hyperparameter optimisation for the classifiers considered in RESICO.

Figure 4.7 highlights in blue those parameter values resulting in the lowest and therefore most optimal loss after 200 trials. In some cases, the figure indicates a convergence towards a particular parameter for the best result, such as *weights* for the value 0 (i.e., 'distance') in the K-Nearest Neighbors classifier. Other best hyperparameters have a majority indicating the likely best selection as is for the value 2 (i.e., *kd\_tree*) for the parameter *algorithm* in KNN. Nonetheless, there are some cases where value changes in the parameter do not seem to influence the loss. Examples of the former are *solver* in the Ridge Linear classifier, where all optimal losses contain values from all possible solvers and *C* in the Support Vector Classifier, where the loss does not improve significantly regardless of the selected hyperparameter value.

The best hyperparameter configuration overall for each classifier is:

**KNN**  $n\_neighbours = 2$ ,  $weights = distance$ ,  $algorithm = kd\_tree$ ,  $leaf\_size = 63$ .

**RF**  $n\_estimators = 173$ ,  $criterion = gini$ ,  $min\_samples\_leaf = 14$ ,  $min\_samples\_split = 3E - 4$ .

**RL**  $alpha = 6473.18$ ,  $solver = sag$ .

**SVC**  $C = 11.14$ .

We will use the optimal values for each hyperparameter for training the models that will be evaluated in the next research questions. The optimised parameters allow us to obtain models tailored to predict FQNs from incomplete code snippets.

The machine learning algorithms employed within RESICO provide default values for hyperparameters that might be suboptimal for the problem at hand. However, several parameters can be tuned effectively, while the values for others do not influence the loss minimisation. The optimal values resulting from the hyperparameter optimisation are used to train the models used in the remaining research questions.

#### 4.4.3 RQ2. How well do COSTER and the RESICO classifiers perform on instances extracted from the dataset used for training?

##### Design

**This research question investigates the performance of COSTER and each of the RESICO classifiers on the internal dataset described in Section 4.4.1.**

To this end, we trained and evaluated the approaches using a 10-fold cross-validation technique [117], where nine folds are considered for training, and the remaining fold is considered for evaluation. This process iterates over each of the folds until they are all evaluated. Furthermore, we adopted a more reliable partition technique called stratified cross-fold validation [109]. This technique improves the folding partition of the data by ensuring that each fold contains approximately the same distribution of labels to predict. In such a way, the training and evaluation processes avoid (I) training a label without an evaluation and (II) evaluating a label not part of the training data. We use the implementation of the stratified cross-validation technique provided by the Python Scikit-learn library.<sup>13</sup>

For COSTER, we use its implementation for training and evaluation. We kept the configuration of COSTER's parameters as provided, only disabling the parameter concerning the minimum number of required contexts (named `fqnThreshold` in the command-line options). This parameter is set to 50 by default and heavily influences the selection of FQNs, establishing a high mark many FQNs cannot reach. These instances would be consequently excluded from the training and evaluation processes; therefore, we set COSTER to consider all FQNs in the balanced dataset described in Section 4.4.1. We configure RESICO to use the machine learning classifiers described in previous sections with the optimised hyperparameters obtained in Section 4.4.2.

<sup>13</sup> [https://scikit-learn.org/stable/modules/generated/sklearn.model\\_selection.StratifiedKFold.html](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html)



For each evaluated fold, we queried the Top- $K$  predictions related to the most likely FQN with  $K$  equals to 1, 3, and 5. If the actual value is among the Top- $K$  predictions, we counted the prediction as a success; otherwise, as a failure. This process allowed us to build sets of actual and predicted instances for all top predictions per fold.

Once the prediction data is gathered for each fold, we evaluate the performance of COSTER and each RESICO-trained model using Precision, Recall and F1-Score for each Top- $K$ . For example, considering a particular FQN as  $FQN_A$ , for each of them we define precision as the number of correctly predicted FQNs out of the number of predicted  $FQN_A$  (Equation (4.1)). Similarly, we define recall as the number of correctly predicted  $FQN_A$  out of the number of actual  $FQN_A$  (Equation (4.2)). Lastly, for each  $FQN_A$ , its F1-Score is the harmonic mean of both Precision and Recall defined in Equation (4.3).

$$Precision = \frac{\# \text{ correctly predicted } FQN_A}{\text{total } \# \text{ predicted } FQN_A} \quad (4.1)$$

$$Recall = \frac{\# \text{ correctly predicted } FQN_A}{\text{total } \# \text{ actual } FQN_A} \quad (4.2)$$

$$F1\text{-Score} = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \quad (4.3)$$

Since our labels are multi-class, we adopted the “micro” averaging approach, which is recommended for multi-class problems when the focus is on the general performance instead of rare classes [74]. Micro-average precision and recall are computed as follows, whereas the micro-average F1-Score is computed similarly to Equation (4.3) but taking into account micro-average precision and recall.

$$\text{Micro-average Precision} = \frac{\sum_{i=1}^k \# \text{ correctly predicted } FQN_i}{\text{total } \# \text{ predicted } FQN_s} \quad (4.4)$$

$$\text{Micro-average Recall} = \frac{\sum_{i=1}^k \# \text{ correctly predicted } FQN_i}{\text{total } \# \text{ actual } FQN_s} \quad (4.5)$$

Finally, we average the metrics for all folds and report them.

## Results

Figure 4.8 shows the averaged results for the trained models on the internal dataset for the Top-1, 3, and 5 predictions. Each row of bar charts represents

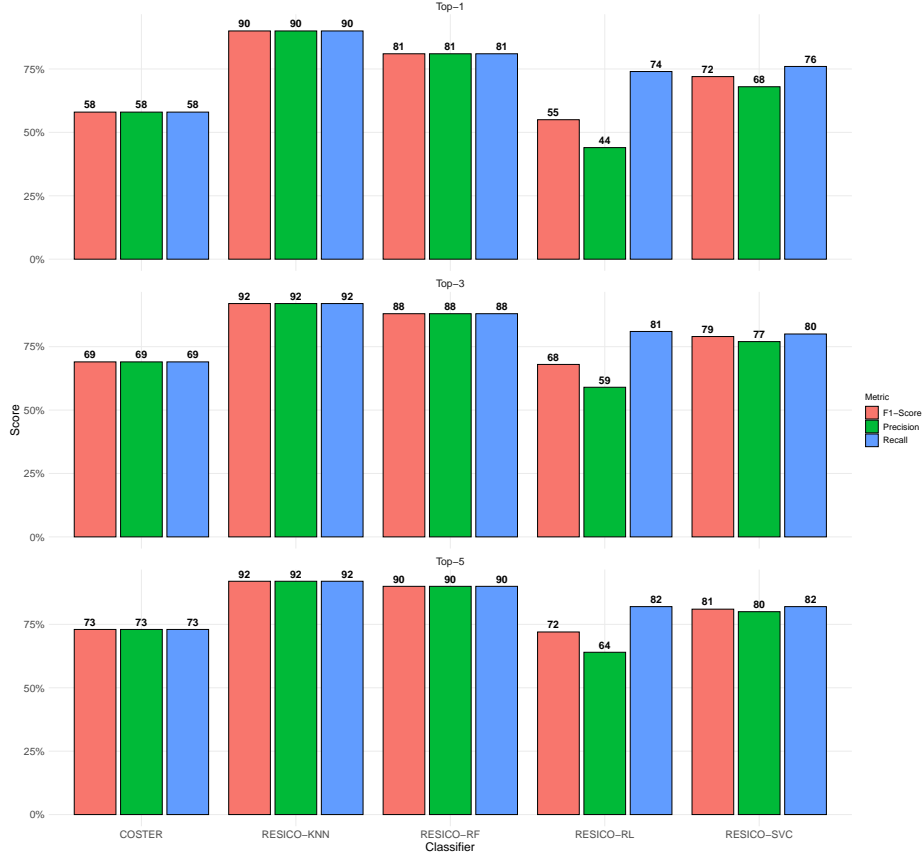


FIGURE 4.8: Performance of the models on the internal dataset.

the Top- $K$  evaluation on the internal dataset for a particular value of  $K$ . The individual bars within a bar chart depict the performance of the particular classifier denoted on the horizontal axis. A cell  $(i, j)$  on the intersection of row  $i$  and column  $j$  denotes the performance of the classifier  $j$  considering the Top- $K$  in  $i$ . We selected F1-Scores as performance comparison metric as they represent the harmonic mean between precision and recall; thus, we can observe the average performance of classifiers.

The performance of the two approaches increases along with the number of provided recommendations (i.e.,  $K$ ). In the specific case of COSTER, it starts with a reported Top-1 performance of 58% according to the F1-Score. This performance increases substantially (11% more) for the Top-3, whereas it grows only a 4% from the Top-3 to the Top-5 recommendations. Nonetheless, we note that overall, COSTER performs well, predicting most of the FQNs for this internal dataset.

Most RESICO classifiers (3 out of 4) outperform COSTER in all Top- $K$  configurations. The KNN classifier performs best in all three Top- $K$ , starting with an excellent performance of 90% for the Top-1 and slightly improving 2% in the remaining cases. The RF classifier ranked second with good scores, especially in Top-3 and Top-5, where it can correctly predict 88% and 90% of the FQNs, respectively. In the case of SVC, lower scores were achieved w.r.t. the previous classifiers but still superior to COSTER's scores. Lastly, the



RL classifier was the least performing classifier of all considered approaches. However, it is worth noting that the slight F1-Score differences compared to COSTER are due to the low precision of the RL since it performs similarly to SVC in terms of recall.

We conclude that the best classifier of our approach (i.e., KNN) is more effective than COSTER on this dataset. The API references and their surrounding contexts were correctly captured and learned by the Word2Vec and classification processes allowing to improve the prediction results of the RESICO classifiers.

COSTER achieves good performance on the balanced internal dataset with a maximum of 73% for the Top-5 recommendations. RESICO presents excellent results, with the best Top-1 recommendations of 90% and Top-5 of 92% for the KNN classifier, thus, outperforming COSTER on the internal dataset.

#### 4.4.4 RQ3. What is the performance of the COSTER and RESICO classifiers when evaluated on unseen datasets?

##### Design

This research question investigates the performance of COSTER and all RESICO models previously trained on the internal dataset and applied to external and unseen datasets.

We trained the approaches on the full version of the balanced internal dataset to answer this research question. Once the models are trained on the data, we use them to predict the extracted types from the external datasets *COSTER-SO*, *StatType-SO*, and *RESICO-SO*. Please note that the snippets in the datasets must be compilable because a ground truth of FQNs is needed to verify the effectiveness of the trained models. FQNs of API references are challenging to determine in incomplete code snippets, hindering obtaining the ground truth needed for an accurate evaluation of the models.

All snippets in the three datasets are compilable; hence, information about API references, their surrounding contexts, and FQNs can be extracted using Eclipse JDT. For each API reference, their specific context is extracted either using the configuration of COSTER (cf. Saifullah et al. [106]) or RESICO (cf. Section 4.3.3). When leveraging RESICO, the extracted information has to be transformed into a vector (cf. Section 4.3.4) before sending it as input to the trained model.

We match the predicted FQNs to the true FQNs, similarly to the previous research question. For each API reference and context (e.g., 1, 3, or 5), the likely FQNs are predicted and checked against the true FQN. The prediction is successful when the actual value is present; otherwise, it represents a failure. Finally, we compute the Top-K micro Precision, Recall, and F1-Score for each external dataset and report them.

## Results

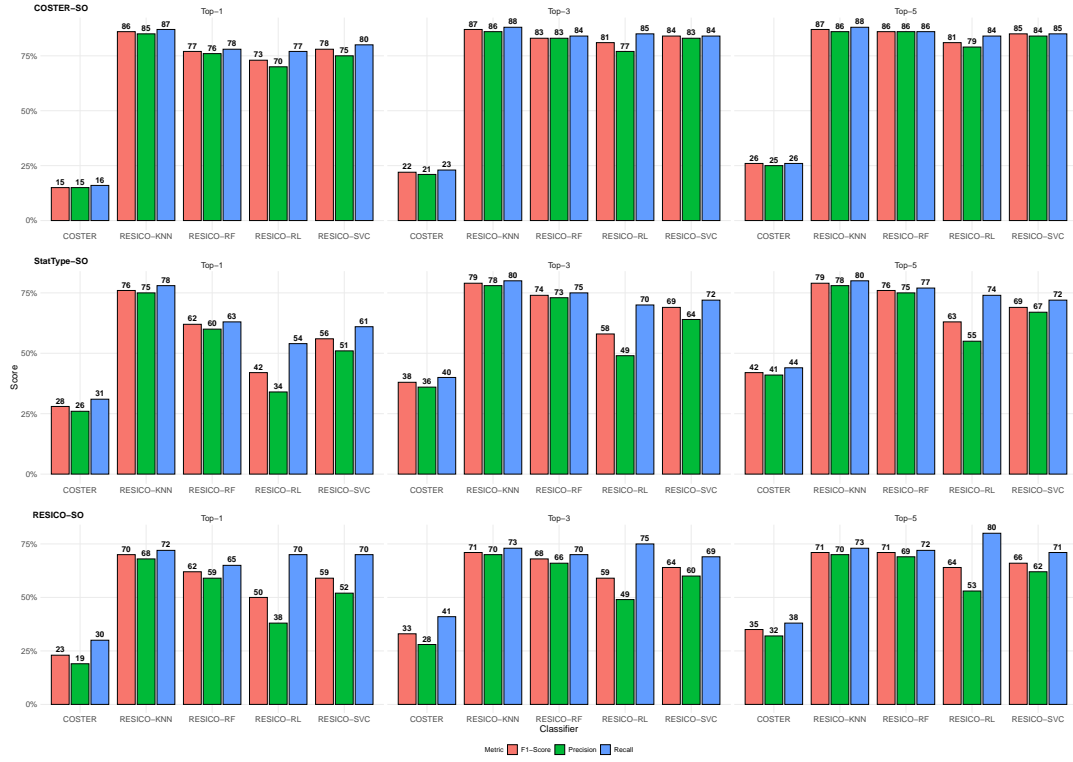


FIGURE 4.9: Performance of the models on the three external datasets.

Figure 4.9 depicts the results of COSTER and the RESICO trained models on the external datasets. Here, each row corresponds to one external dataset. Each bar chart within a row corresponds to the particular Top-K being considered denoted at the top. Each bar within a bar chart corresponds to the precision, recall, or F1-score for the approach denoted on the horizontal axis.

RESICO outperforms COSTER on all three datasets. The difference in F1-Score is considerable for some configurations. For example, there is a 71% performance difference between the F1-Scores of COSTER and RESICO-KNN when analysing the first predictions (i.e., Top-1) on the *COSTER-SO* dataset.

The best performance overall was RESICO-KNN, as in the previous research question (cf. Figure 4.8). This model showed a relevant performance starting with 87% F1-Score for the Top-1 recommendations in the *COSTER-SO* dataset. Although inferior to the *COSTER-SO* performance, RESICO-KNN still predicts with good accuracy the FQNs in the *StatType-SO* and *RESICO-SO* datasets. The F1-Score metrics are above 70% for the last two datasets considering all Top-Ks.

The performance difference of the models across the three external datasets can be explained by the number of records per FQN (e.g., 50) and the balancing of the training dataset. In the data balancing step (cf. Section 4.4.1), we limited the number of records per FQN to 50. The results in Figure 4.9 show

that COSTER might need more occurrences to improve its performance. On the other hand, most of the RESICO classifiers have good generalisability with this limited number of examples per FQN. Additionally, data balancing allowed us to harmonise the importance of each FQN, hence, widespread types such as `java.lang.String` have the same relevance as other less frequent FQNs. In such a way, the trained models are not biased towards any FQN, enabling better performance than other methods, such as COSTER, without this feature.

In the three external datasets considered to evaluate the generalisability of the performance, RESICO-trained models outperform COSTER with a notable difference in some cases. Some decisions made when designing the machine learning models, such as data balancing and hyperparameter optimisation, allow RESICO to achieve better predictive capabilities than the COSTER approach.

#### 4.4.5 RQ4. How much time is needed to train COSTER and the RESICO classifiers?

##### Design

This research question investigates the computational cost of COSTER and RESICO. The experiments were conducted on a Dell PowerEdge R730 with 2 Intel Xeon 2637 CPUs, each with four cores at 3.5 GHz with HyperThreading and 256 GB of RAM.

For each approach, we measured the time employed to extract the information from the projects in the initial corpus [85] and the time to train the model in the balanced internal dataset (cf. Section 4.4.3). Additionally, for RESICO, we measure the embedding time, i.e., the time required to transform the API references and contexts into vectors suitable for a machine learning algorithm. We do not consider the encoding time of FQNs to label numbers since it is a simple mapping whose execution time is negligible.

##### Results

Table 4.6 depicts the time measurements for all experiments.

Approach	Token Extraction	Context Embedding	Model Training	Total
COSTER	11h 49m 4s	-	25s	11h 49m 29s
RESICO-KNN		3m 43s	671ms	11h 52m 48s
RESICO-RF			29m 6s	12h 21m 53s
RESICO-RL			52m 16s	12h 45m 3s
RESICO-SVC			4m 43s	11h 57m 30s

TABLE 4.6: Computational cost of the approaches. Time is measured in hours (h), minutes (m), seconds (s) and milliseconds (ms).

The extraction time (col. *Extraction*) is the same for the two approaches. As COSTER and RESICO rely on the same extraction procedure built on top of Eclipse JDT with slight differences related to the handling of contexts, for every processed FQN, two outputs were written to two different datasets.

The total time (col. *Total*) required by COSTER is lower than that required by RESICO. In Table 4.6, we show the time needed to extract the tokens, the time needed to embed the contexts, if any, and the time needed to train the models using the sampled data. COSTER does not need embeddings, saving the time needed to craft them.

Fact extraction takes the longest, with nearly 12 hours for processing the data of 50K projects. After sampling and balancing the dataset, the remaining training data consists of 243,000 records. Note that embedding the API references takes less than a second, whereas embedding their contexts takes almost 4 minutes. These running times can be considered efficient for the specified dataset, however, with a more complex dataset, embedding times could increase drastically. The learned embeddings for the API references and their surrounding contexts allowed most RESICO classifiers to outperform COSTER both on the internal dataset (cf. Figure 4.8) and the three external datasets (cf. Figure 4.9).

Finally, concerning the training times for the RESICO classifiers and the COSTER approach, the fastest overall is RESICO-KNN with less than a second used for its training on the balanced dataset. COSTER also trained quickly with only 25 seconds, followed by RESICO-SVC, requiring almost 5 minutes. The slowest approaches are RESICO-SVC, with nearly 30 minutes and RESICO-RL, with more than 50 minutes to complete their respective training. For SVC, its training time agrees with its design since it is not an easy task to search for an optimal hyperplane in high-dimensional data. A hypothesis for why the RL model takes the longest to train might be the challenges in linearly differentiating the classes in a multi-class data scenario.

The fact extraction times of COSTER and RESICO are the same as they both rely on the same Eclipse JDT Core extension. On the dataset considered for training, embedding the tokens does not take considerable time while it does improve the predictions, as shown in the results of RQ2 and RQ3. The classifiers that take the longest to train are those of which characteristics of our internal dataset pose challenges to their design and implementation (e.g., multi-class and high-dimensional data).

#### 4.4.6 RQ5. To what extent do ambiguities in simple names influence the performance of the approaches?

##### Design

Name ambiguities affect the type resolution conducted by COSTER and RESICO. **This research question analyses the resolution failures and how ambiguities could have impacted them.** Consider an incomplete code snippet

having the simple name `Element`. It could resolve to `org.jdom.Element` or `org.jsoup.nodes.Element` within an incomplete code snippet. There are 17 FQN candidates in the internal balanced dataset for this simple name alone. The more ambiguous a simple name is, the more challenging it is for a resolver to predict its exact FQN effectively.

We analyse the erroneous type resolutions made by the *COSTER* and *RESICO* models, which were trained on the internal balanced dataset, by investigating whether the root cause for their incorrect resolutions produced on the external datasets is an ambiguous simple name. Specifically, we consider the models of *COSTER* and *RESICO-KNN* for this research question, the former being the approach to compare with and the latter the best *RESICO* model overall. We only consider the Top-1 predictions from the models on the external datasets since they have most of failures compared to the other two (i.e., Top-3 and Top-5).

Model	Dataset	# Uniq. Mis.	# Mis.	# Amb.	% Amb.
COSTER	COSTER-SO	27	1,173	4	0.34
	StatType-SO	155	1,355	37	2.73
	RESICO-SO	197	1,412	16	1.13
RESICO-KNN	COSTER-SO	18	205	31	15.12
	StatType-SO	96	458	108	23.58
	RESICO-SO	112	553	73	13.2

TABLE 4.7: Ambiguity analysis for *COSTER* and *RESICO-KNN* trained models on the external datasets.

## Results

The results of the ambiguity analysis are depicted in Table 4.7. The number of unique misclassifications (col. *Uniq. Mis.*) for the *COSTER-SO* dataset is lower compared to the other datasets. However, those numbers increase when counting the total number of misclassifications regardless of their uniqueness (col. *# Mis.*). The *COSTER* model produces more misclassifications than *RESICO-KNN* on all datasets, as previously reported in RQ3.

Interestingly, only a minority of the type resolution failures are due to ambiguous simple names. Column *# Amb.* of Table 4.7 demonstrates that in all datasets but one, less than 100 misclassifications correspond to such cases. Only a tiny percentage (col. *% Amb.*) of FQNs could have been misclassified because other FQNs share the same simple name. The *StatType-SO* dataset with the *RESICO-KNN* model combination has the highest percentage of ambiguous misclassifications compared to other combinations. A closer look indicates that only 31 unique misclassifications occurred but were repeated multiple times. For example, the FQN `org.hibernate.Session` is predicted as `org.hibernate.classic.Session` and as `javax.websocket.Session`, 35 and 3 times respectively.

These findings indicate that the main reason for failures from the models is not the presence of ambiguous simple names. A likely reason for the mispredictions might be the close similarity of the contexts around different API references. The closer the contexts around API references are, the more prone the models might be to recommend distinct FQNs as similar.

The number of unique mispredicted FQNs is the highest for both models on the *StatType-SO* and the *RESICO-SO* datasets, illustrating their challenging nature. Only a tiny percentage of incorrect type resolutions is due to ambiguous simple names. The main reason for the mispredictions might not be ambiguous simple names, but rather, closer contexts could mislead the trained models towards failures.

## 4.5 Discussion

This section discusses the results, limitations, and potential impact of RESICO.

### 4.5.1 Context-based Approaches to API Type Resolution

The approaches considered in the previous evaluation section (cf. Section 4.4) are based on the contexts surrounding API references to resolve API types. Indeed, both RESICO and COSTER capture the contexts around API usages. Then, they recommend a FQN based on similar contexts in incomplete code snippets. Despite this similarity between the approaches, they have many differences, such as their context definitions and the usage of machine learning versus information retrieval techniques.

COSTER captures the API references, their surrounding contexts, and the FQNs of the referenced API element, and stores them into a Lucene<sup>14</sup> database to be queried later. RESICO, in contrast, starts three learning processes after a similar extraction process to improve its final prediction (cf. Section 4.3.3). After the learning processes, a classifier uses the learned embeddings in the form of vectors to learn to distinguish different contexts corresponding to FQNs. The embeddings and classifier models are stored in files for posterior use at resolution time.

We noticed that even though our approach takes slightly more time (around 3 minutes) and thus more computational resources than COSTER (cf. Table 4.6), it can resolve API types more effectively (cf. Figure 4.8, Figure 4.9).

Concerning type resolution failures, our experiments found that most are due to the contexts in which the same or different simple names occur being overly similar. For example, Figure 4.10 depicts different FQNs close to each other because of the similarity of their contexts. The figure was built by extracting the averaged vectors between API references and contexts of 14

<sup>14</sup> <https://lucene.apache.org/>





FIGURE 4.10: Similar FQNs by their context vectors.

randomly selected misclassified FQNs by RESICO. Afterwards, we reduced the dimensions of these vectors to 2 using TSNE<sup>15</sup> to facilitate their plotting. Figure 4.10 shows the context similarity of different FQNs. The colours and the text on the points identify each FQN in the 2-dimensional cartesian space. The points corresponding to the vectors of `org.joda.time.DateTimeZone` and `org.joda.time.DateTime` are close to each other since they might be used in a similar environment. The same happens to the points of the Apache and Java collections, such as `MultiMap` from the former and `List` and `Map` from the latter. Lastly, `PeriodFormatterBuilder`, `PeriodFormatter`, and `PeriodType` from the `joda` library share context similarities and the trained models struggle to distinguish them in some cases effectively.

In addition, we made an analysis based on the effectiveness of the approaches at the library level for the external datasets evaluated on RQ3. Table 4.8 shows how COSTER and RESICO-KNN (our best model) perform per library for the *StatType-SO* dataset. The analysis for the remaining datasets can be found in our online repository.<sup>16</sup>

For a total of 12 libraries in the *StatType-SO* dataset, the green entries in Table 4.8 show that RESICO-KNN outperforms COSTER on 10 libraries. For only 2 libraries, the approaches achieved the same score. In the case of the `Org-JSON` library, no approach could correctly resolve any of the seven API types. In the case of `KSoap2`, the approaches successfully predicted all five types.

Table 4.8 also shows RESICO-KNN scores more success than failure cases in all but two libraries (`Org-JSON` and `Apache-Log4J`). COSTER, in contrast, has a majority of failures instead of successes except for `XStream`, `Apache-Commons` and `KSoap2`. Interestingly, most failures for both approaches are located in the

<sup>15</sup> <https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html>

<sup>16</sup> <https://github.com/softwarelanguageslab/resico-paper>

Library	Total	COSTER		RESICO-KNN	
		Success	Failure	Success	Failure
JDK	665	46	619	453	212
GWT	350	149	201	303	47
Hibernate	319	65	254	242	77
Joda-Time	250	56	194	170	80
XStream	173	139	34	144	29
Apache-Http	29	1	28	26	3
Apache-Commons	15	10	5	14	1
Apache-Struts	7	0	7	7	0
Org-JSON	7	0	7	0	7
Dom4J	5	1	4	4	1
KSoap2	5	5	0	5	0
Apache-Log4J	2	0	2	1	1

TABLE 4.8: The accuracy of the best models per approach shown per library in the *StatType-SO* dataset. Highlighted in green and red are the largest successes and failures, respectively.

JDK library, with a very low success rate for COSTER (7%) and a relatively low one for RESICO (32%).

### 4.5.2 Limitations

RESICO cannot handle multiple versions of the same library, just like previous approaches (e.g., COSTER). They recommend the old and the new FQNs when a class is moved to another package.

The approach should be equally applicable to other languages with explicit import statements (e.g., Python). Some relatively small extensions will be required to support import statements with wildcards, which import several API types simultaneously without analysing the corresponding library files.

To support languages featuring import statements that load a library definition at run time will require more extensive work. For such languages, we envision run-time analysis of the libraries. Note that we are not gauging the modifications required based on whether the language is dynamically or statically typed but on the type of import statements that would occur in its snippets.

Finally, RESICO is based on word embeddings. Like other approaches relying on this technique, it suffers from *Out-Of-Vocabulary* errors when the model must predict a term that is not in the training set. RESICO will fail to resolve the API type for such a term. Nonetheless, the NLP community has proposed mitigation strategies such as enriching the vectors with subword



information [22, 81], which are not currently included in the implementation and are part of our future agenda.

### 4.5.3 Potential Impact

As revealed by our experiments, RESICO scores high on all considered datasets. Researchers and practitioners can use our approach to design and implement tools that analyse SO posts, e.g., API usages and the natural language around the code. Our approach and trained model provide type resolution to client analyses and tools requiring type information, even for syntactically incorrect and incomplete code snippets. One example tool could, for instance, help developers who rely on SO solutions and frequently copy their code into an IDE by providing import statements for all referenced library types.

## 4.6 Threats to Validity

We now discuss the threats that might affect the validity of our study for learning-based API type resolution in incomplete code snippets.

### 4.6.1 Threats to Construct Validity

RESICO relies upon a Word2Vec implementation provided by the Golang programming language to obtain the word embeddings and four classifiers from the Scikit-Learn framework for building and evaluating the machine learning models. RESICO uses these libraries extensively, so their inclusion constitutes a threat to validity. Nevertheless, they are among the most popular open-source machine-learning libraries.

### 4.6.2 Threats to Internal Validity

We chose the *Continuous Bag of Words* (CBOW) architecture for the Word2Vec algorithm instead of the alternative Skip-gram. Our selection was motivated by the faster processing times of the former approach compared to the latter. CBOW tries to infer a word given its context, whereas Skip-gram attempts to infer a context given a word. Skip-gram might produce slightly different results; however, as reported by Mikolov et al. [87], CBOW can also produce reliable vector representations while reducing the learning time.

Data balancing was used to equalise the internal dataset extracted from the 50K-C corpus, thereby avoiding biased training towards imbalanced classes. Data balancing should in general be applied to the training dataset and not to the testing dataset. In Section 4.4.3, we balanced the full dataset before the stratification and posterior training and testing, to verify whether our model could detect FQNs from a dataset similar to the training one. The possible implications of this decision are minimal for a two-fold reason. On the one hand, all evaluated methods (e.g., COSTER and all RESICO classifiers)

use the same balanced dataset for training and testing, making the evaluation fair. On the other hand, we conducted a second evaluation on external datasets, showing similar results.

### 4.6.3 Threats to External Validity

An external threat might be due to the comparisons between both approaches, COSTER [106] and RESICO. For an unbiased comparison, we first extracted information from the same API references in the 50K-C dataset [85]. Second, the training dataset is balanced to avoid bias towards any particular label, and the approaches are trained on this balanced dataset. Third, we evaluated both approaches on the datasets initially employed for assessing the quality of COSTER (e.g., *COSTER-SO* and *StatType-SO*). Lastly, we created a third external dataset (e.g., *RESICO-SO*) of code snippets that reference the same set of libraries present in the previous datasets.

Another external threat concerns the classifiers and the word embedding algorithm (i.e., Word2Vec) used in RESICO. We chose four different classifiers as instantiations of RESICO which have been used in ML-based software engineering solutions [39, 52] as well as in previous text classification works [31, 71, 111, 160]. Likewise, Word2Vec has been successfully used in other studies [57, 91, 152].

### 4.6.4 Threats to Conclusion Validity

The metrics used to evaluate our approach (i.e., Precision, Recall, and F1-Score) are widely used among recommenders, including the reference work (i.e., COSTER). To evaluate to what extent programs in the 50K-C dataset can be used to predict fully-qualified names with similar context, we adopted a 10-fold stratified cross-validation [117] after a previous balancing of the dataset. For datasets which are not completely balanced, the number of instances per fold will correspond to a similar distribution of the full dataset. Since we have a balanced dataset, it is not only ensured that each fold follows the general distribution but also that each of them contains the same number of instances per class.

## 4.7 Conclusion

This chapter proposes a new learning-based approach to resolving the fully-qualified name of API types referenced by their simple name in code snippets from online Q&A platforms such as Stack Overflow. The approach, called RESICO, extracts API references, their surrounding contexts and their associated FQNs from a dataset of 50K compilable GitHub projects. A data balancing technique is applied to balance the dataset from where COSTER and RESICO train and learn to distinguish API types in incomplete code. RESICO uses Word2Vec to transform API references and their contexts into

vector representations and class categorisation to convert FQNs into numbers. The vectors of the API references and contexts are combined into the input of machine learning classifiers, whereas the numbered FQNs are the labels to predict. The approach is instantiated with four machine learning classifiers, namely KNN, Random Forests, Ridge Linear, and Support Vector Machines. Before evaluating the resulting models, hyperparameter optimisation is applied to find an optimal configuration for these classifiers.

We have compared the RESICO machine learning models and COSTER extensively on four datasets: one gathered from a corpus of 50K compilable GitHub projects and three datasets that serve as external validators for the trained models.

Our approach is more complex than COSTER since it involves training several machine learning models; hence, it consumes more computational resources during training. More specifically, RESICO involves training two Word2Vec models to learn embeddings and one classification algorithm. Our best classifier (KNN) is slightly slower to train than COSTER. However, once deployed, most of the RESICO-trained models outperform COSTER in the 10-fold cross-validation on an internal dataset and on three external datasets. Finally, we showed that incorrect type resolutions produced by RESICO and COSTER might not be due to ambiguous simple names but mainly to similar contexts around usages.



## Chapter 5

# Uncovering Library Features from Stack Overflow Posts

### 5.1 Introduction

Contemporary software development commonly uses features from third-party libraries [15] to reduce development time and potentially improve overall system quality [2]. Each library targets at least one particular domain (e.g., graphical user interface, persistence, etc.) and offers features to client systems through its API, facilitating the implementation of a particular task (e.g., displaying a dialog, serialising to JSON, etc.).

Software ecosystems such as Maven or NPM host an ever-increasing number of software libraries<sup>1</sup>. As the number of libraries in software ecosystem grows, some are bound to offer similar features. For example, Maven currently has 81 different libraries providing reusable implementations of collection datatypes.<sup>2</sup> From the brief description of each library, one can infer that they provide different kinds of collections (e.g., maps, sets, queues, stacks, etc.) but the range of features offered is not immediately clear (e.g., persistent collections, specific utility methods for sorting or reversing collections), neither how these features can be used (e.g., a call to a single static method, a call to an instance method of a class that needs to be instantiated, three methods that need to be called together), nor how each library compares to other libraries with similar features. Ultimately, developers need to select the most suitable library from the available ones for the task at hand.

When selecting a library to reuse from a vast ecosystem, it becomes essential for developers to know the features offered by each library. Previous works in Chapter 3 propose definitions that are aligned with our idea of a feature. For example, Kanda et al. [64] consider a set of API calls accompanied by a corresponding name as a feature. Additionally, Antoniol and Guéhéneuc [9] define a feature as *“a set of data structures (i.e., fields and classes) and operations (i.e., functions and methods) participating in the realisation of the functionality”*. We adopt their definition in this dissertation

<sup>1</sup> <http://www.modulecounts.com/>

<sup>2</sup> <https://mvnrepository.com/open-source/collections>

so that *features* comprise the API elements that realise them, as well as their textual description.

This chapter presents the second contribution of the dissertation, namely the discovery of features, their relevance, and potential influence on the usability of libraries. First, a motivation section will motivate the need for automated approaches that describe a library in terms of the feature it offers. Second, our early works AutoCat and MUTAMA, towards discovering features from libraries are presented and discussed. AutoCat and MUTAMA are not focused on fine-grained features, but target more coarse forms of features such as categories and tags. Limitations of these coarse-grained feature descriptions are also discussed. Third, our approach to uncovering features from SO posts is presented. We describe the implementation of our approach in a tool called LiFUSO and demonstrate its usage in a case study. Next, we conduct an extensive empirical evaluation on the quality of the extracted features compared to those found in official library tutorials and usage cookbooks. Sixth, a discussion section explores the results obtained from our approach and its current limitations and impact. Validity threats for both our approach and tool are discussed to conclude the chapter.

The approach proposed in this chapter extracts features from SO code snippets. One of the requirements of our approach LiFUSO is that the name of the analysed library must be part of the tags list in an SO post. Although this decision might restrict SO answers to those certainly using a library, it might also miss other posts not correctly tagged. Therefore, LiFUSO will be further extended in Chapter 6 with the assistance of our RESICO approach (cf. Chapter 4).

## 5.2 Motivation

We highlight some motivational arguments towards the development and adoption of automated approaches to produce descriptions of the features offered by a library.

### 5.2.1 Support for Comparing Libraries

Developers rely on metrics such as the number of downloads, votes, stars on GitHub, open and closed issues, release frequency, etc., when comparing libraries to use [42, 133]. Several ecosystem indices such as NPMCompare<sup>3</sup>, NPMTrends<sup>4</sup> or NPMS<sup>5</sup> have therefore been proposed. These enable comparing libraries in terms of metrics and quality scores. Figure 5.1 depicts an example of such a comparison provided by NPMCompare.

The focus on popularity-based metrics, however, tends to bias the selection process towards the more popular and older libraries [156]. In addition,

<sup>3</sup> <https://npmcompare.com/>

<sup>4</sup> <https://www.npmtrends.com/>

<sup>5</sup> <https://npms.io/>

newly-introduced libraries with superior features or with an API requiring less boilerplate code to use might take a while to be adopted due to their low *popularity* metrics. Furthermore, none of the software ecosystems is supported by tools that enable comparing libraries based on the features offered, nor on the API usage that is required to use them. A desired tool would make library comparisons based on the features that libraries propose and not on the popularity-derived metrics.



	 <b>Angular</b> 1.8.2	 <b>React</b> 17.0.2
Monthly Downloads	2,370,845	51,039,615
Open Issues	464	867
Open Pull Requests	74	231
Stargazers	59,598	176,410
Subscribers	3,908	6,663
Forks	28,451	35,664

FIGURE 5.1: Comparison of two libraries based on popularity metrics.

### 5.2.2 Support for Exploring Ecosystems

Likewise, tool support for exploring an ecosystem is lacking, with most official ecosystem indices being limited to browsing through community-curated or maintainer-provided categories and library tags. Some ecosystem indices support natural language queries against the short description of the indexed libraries, but such documentation might be lacking and there is no standard format for documenting the functional features provided by each library. With an automated means for uncovering library features, ecosystem indices could provide a richer browsing experience. For instance, users could inspect a list of features that are commonly or rarely implemented by the libraries in a selected category.

Ecosystem indices could also support queries against the uncovered textual descriptions, as well as the API elements that realise each feature. For instance, the query “resize an image”, could return the API usage required for different libraries that support this task. Users could inspect the API usage required for a particular library and immediately navigate to a competing library for which less boilerplate code is required to use the same feature or of which the API is more aligned with the project’s own coding conventions (e.g., a fluent or a regular API). Even more, developers that need multiple features, ideally want to inspect to what extent each library supports these features, e.g., developers might need to “resize an image” and “convert image to pdf”. Current tool support does not allow developers to efficiently evaluate and compare candidate libraries with respect to all the desired features.

Given the previous arguments, there is a need for automated approaches that describe a given library in terms of its features. The features, however,

can be described at several levels of granularity, from the artifact level to the code level. We propose three approaches of increasing granularity. The first, called AutoCat, automatically categorises a library into one of the top-level library categories used by ecosystem indices. The second, MUTAMA, describes a library in terms of a collection of tags used on such indices. The third, LiFUSO, describes a library in terms of features consisting of the API elements that implement them and a natural language description.

### 5.3 AutoCat: Automatic Library Categorisation

For the Maven software ecosystem, there are at least two web-based views that have been developed to assist developers in the library selection process: *SonaType*<sup>6</sup> and *MVNRepository*<sup>7</sup>. The former supports search based on the matched queried text to the name or to the description of libraries. The latter supports two types of searches for exploring the vast number of libraries in the ecosystem:

**Metadata-based search** A developer searches for an artifact providing information such as `groupId`, `artifactId` or `version`.

**Category-based search** A developer searches for a specific category of artifacts related to a search criterion.

Metadata-based searches are of little help without knowledge of the precise metadata of the artifact. Category-based searches assist users who do not have a specific library in mind, but are rather exploring the ecosystem or inspecting alternatives to an already adopted library.

*MVNRepository* features 150 library categories covering various domains such as *Databases*, *I/O Utilities* or *Mocking*. Therefore, categories could also be seen as very coarse descriptions of the features offered by libraries. For example, libraries in the *I/O Utilities* category have features related to I/O. The total number of libraries in these 150 categories is 37,934. However, considering that the number of libraries reported by [17] is 73,653, over 30,000 libraries in the Maven ecosystem remain uncategorised.

**We propose an approach called AutoCat to automatically assign libraries to an appropriate category by analysing the bytecode of their implementation.** A new “uncategorised” library can be automatically classified into one of the existing categories, the libraries of which it will share similar features with.

The proposed approach is based on text classification machine learning algorithms trained and evaluated on a corpus of text extracted from the libraries. We obtain this corpus of text by extracting the identifiers of public

---

<sup>6</sup> <https://search.maven.org/>

<sup>7</sup> <https://mvnrepository.com/>



classes and methods from a library’s JAR file using the Apache BCEL<sup>8</sup> library. For those identifiers following the *CamelCase* naming convention (e.g., `getAccountNumber`), we separate the identifier into distinct words (e.g., `get`, `Account`, `Number`).

In order to feed the extracted corpus of text to a machine learning algorithm, we vectorise it using Word2Vec [86]. Fixed-length vectors are generated for each word in the vocabulary (i.e., words from class and method identifiers extracted through the previous step). The generated vectors capture the context around a word (e.g., a window size of five tokens); hence it is possible to relate different words by the surrounding context. Default parameters for the Word2Vec process are selected for its training.

The task of the machine learning algorithm is to learn and predict a discrete label for each vector that corresponds to the *MVNRepository* category to which the library belongs. We consider five machine learning algorithms to instantiate our approach: Gaussian Naive Bayes (GNB) as well as Bernoulli Naive Bayes (BNB) [63], Support Vector Machines (SVC) [56], K-Nearest Neighbors (KNN) [38] and Random Forest (RF) [24, 58] (cf. Section 2.4.2).

### 5.3.1 Evaluation

We train and evaluate our approach on five *MVNRepository* categories: *Collections*, *Dependency Injection*, *Http Clients*, *Compression* and *JSON* libraries. From each category, we select 15 libraries (i.e., JAR files). These 15 libraries are divided into 10 libraries for training and evaluation, and 5 for validation.

Classifier	Precision	Recall	F1-Score
GNB	0.75	0.79	0.77
BNB	0.88	0.91	0.89
SVC	0.95	0.96	0.95
RF	0.97	0.98	<b>0.97</b>
KNN	0.96	0.98	<b>0.97</b>

TABLE 5.1: Scores achieved by the trained classifiers on the automatic classification of libraries. Highlighted in bold are the best models according to their F1 score.

Table 5.1 depicts the results of the used 10-fold stratified cross-validation [109] method according to precision, recall and F1 metric on each trained model. All models produced good to excellent results with KNN and RF achieving the highest scores with 97% for the F1 metric each. This small experiment (i.e., only 15 libraries) returns promising results regarding the automatic classification of libraries based on their implementation. **Moreover,**

<sup>8</sup> <https://commons.apache.org/proper/commons-bcel/>

it demonstrates that libraries within a category are likely to share common public tokens and features. Despite the encouraging results shown in Table 5.1, the AutoCat approach also fails to meet our goal of feature uncovering due to the limitations discussed below.

### 5.3.2 Limitations of Category-based Approaches to Feature Uncovering

The library categories maintained by ecosystem indices such as *MVNRepository* can be considered very coarse-grained descriptions of the features provided by a library. However, it is clear that two libraries within the same category can still offer different features. The *MVNRepository* only maintains 150 categories<sup>9</sup> for all its libraries. Furthermore, a library can provide more than a single feature for reuse. Moreover, a category alone does not convey which API types comprise the feature.

## 5.4 MUTAMA: Multi-label Library Tagging

As discussed above the categories maintained by indexing platforms for libraries describe the features provided by a library in a too coarse-grained manner. The *MVNRepository* indexing platform therefore also describes libraries in terms of platform-wide tags.

Tags on *MVNRepository* are intended to correspond to the features of a library. The Apache library *Commons-CLI*<sup>10</sup>, for example, has been labeled with the tags *command-line*, *cli* and *parser*. The library does indeed provide reusable features for parsing command line arguments. Unfortunately, not all libraries indexed by *MVNRepository* have been tagged this precisely. This is often the case for libraries that have only recently been contributed to the ecosystem or libraries not enjoying widespread use. An automated approach to suggesting feature tags for a software library could overcome this problem and thereby facilitate ecosystem search.

We now describe the details of our learning-based approach MUTAMA (Multi-label TAGging in MAven) to assigning tags for libraries in a software ecosystem. The approach is trained on a dataset of libraries that have already been tagged with two or more tags, corresponding to the coarse-grained features offered by the library.

Figure 5.2 shows the steps comprising the training phase of MUTAMA. The *Main Pipeline* box at the top of the figure depicts how a trained model is obtained by training a multi-label classifier on the corpus of tagged libraries. The first step entails transforming this group into a format that is suitable for such a classifier.

<sup>9</sup> This was consulted on January 10<sup>th</sup>, 2024.

<sup>10</sup> <https://mvnrepository.com/artifact/commons-cli/commons-cli>

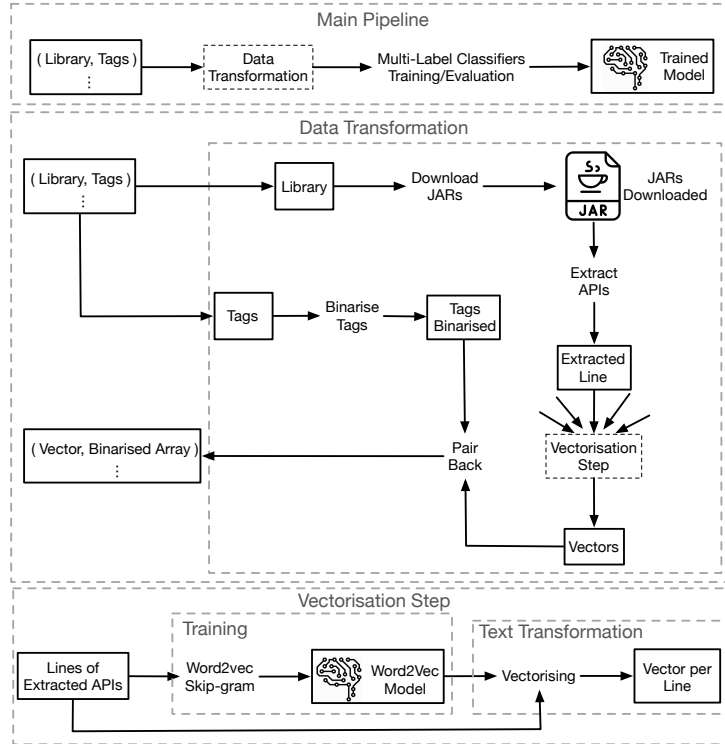


FIGURE 5.2: The steps followed by tools following the MUTAMA approach.

The *Data Transformation* step is depicted in the middle of the figure. For each library version, it takes as input a pair of a triplet `groupId - artifactId - version` and the set of associated tags. It then downloads the binary corresponding to each library version triplet. Next, the downloaded binaries are processed using the ByteCode Engineering Library (BCEL)<sup>8</sup> to extract the class and method names of their public API. This ensures that the machine learning classifier will learn patterns in the public interface of the libraries that have been tagged similarly, which is the same interface clients have access to. To help generalise the data, we split camel cased names into their constituent tokens. The resulting pieces of texts are appended into a single line per library version triplet.

Next, the extracted lines with textual information for all libraries are vectorised. We use Word2Vec [86, 87] due to its ability to recognise words that are semantically similar because they often occur in close contexts. Two neural network architectures can be used for Word2Vec. The CBOW (cf. Chapter 2) architecture predicts a target token from the context, whereas Skip-gram predicts context from a token. In this case, a token is one of the obtained words after the camel case split and its context is the surrounding five tokens selected as a default hyperparameter. We selected the CBOW architecture described by Mikolov et al. [86]. The vectorisation itself requires two steps, depicted at the bottom of Figure 5.2. The first training step takes a corpus of documents (i.e., the lines extracted for each of the libraries) and iterates over a shallow neural network until a threshold or convergence is reached.

The learnt weights correspond to vectors for each of the unique tokens in the corpus. As a second step, we transform each document in the corpus by mapping the tokens at each line to their corresponding vectors. All vectors corresponding to tokens in a line are then averaged into a single vector per line.

Concurrently, in the middle of Figure 5.2, the set of tags associated with each line of library text is transformed into a binary array. For each library, MUTAMA constructs a zeroed vector (i.e., a vector only containing zeros) with a length equal to the total number of tags in the dataset. It then replaces zeros with ones at those positions corresponding to a tag of the library. In such a way, the transformed dataset is ready to be used by multi-label machine learning classifiers.

Tying everything together, at the top of Figure 5.2, the Word2Vec vectors stemming from each library's bytecode represent the input to a multi-label machine learning classifier. The binary arrays stemming from each library's tags on *MVNRepository* represent the classes to predict by the classifier. The result is a trained model which can be used to automatically tag those untagged libraries in the Maven ecosystem.

### 5.4.1 Evaluation

The performance of MUTAMA depends on the multi-label classifier it is instantiated with, and on the quality of the dataset of tagged libraries this classifier is trained on. We will therefore first describe how we collected such a dataset from the *MVNRepository* website, before training several candidate classifiers and comparing their performance.

#### Collecting a Corpus of Tagged Libraries

To collect a dataset of tagged libraries, we implemented one crawler to obtain their binaries and one crawler to obtain their tags.

The first crawler operates on the Maven package repository.<sup>11</sup> From this source we gathered the `groupId - artifactId - version` triplet for each library and its associated bytecode. We only consider the latest version of each library in this study. In total, we collected 235,011 Java and Scala libraries in this way.

For the remainder of the evaluation, we extracted a statistically significant sample from this data. A confidence level of 99% and a confidence interval of  $\pm 2$  were used as parameters for the online Sample Size Calculator tool.<sup>12</sup> The result is a recommended sample size of 4,088 libraries.

The second crawler operates on the *MVNRepository* indexing platform<sup>7</sup>. Besides tags, categories and usage statistics are collected on this platform. Our

<sup>11</sup> <https://repo1.maven.org/maven2/>

<sup>12</sup> <https://www.surveysystem.com/sscalc.htm>

crawler retrieves the tags for each of the 4,088 libraries in our random sample. As expected, not all sampled libraries are tagged. Figure 5.3 depicts the distribution of the number of tags in our library sample. Only 3,137 libraries have been tagged with one or more tags, representing 77% of the sample. In the remainder of the evaluation, we will restrict the data to libraries carrying two or more tags, corresponding to the 59% of libraries with at least one tag. As we will illustrate below, we do not deem a single tag sufficient to facilitate searching through an ecosystem nor for training an automated classifier.

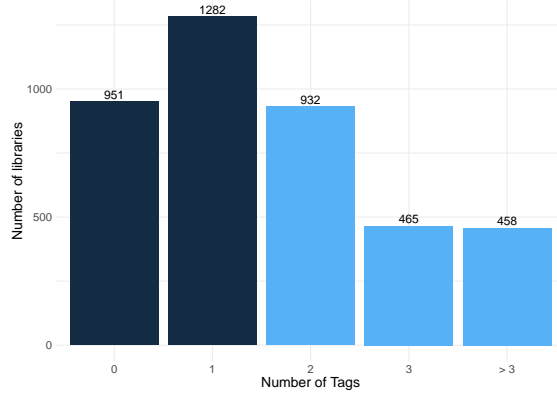


FIGURE 5.3: Distribution of the number of tags for the 4,088 sampled libraries.

### Instantiating MUTAMA with Multi-label Classifiers

We use the open source MEKA<sup>13</sup> tool to instantiate our MUTAMA approach with a multi-label classifier and its base classifier.

Classifier	2 Tags				3 Tags				>3 Tags			
	A	F1-M	F1-m	HL	A	F1-M	F1-m	HL	A	F1-M	F1-m	HL
BR [129]	0.01	0.01	0.03	<b>0.05</b>	0.07	0.08	0.16	<b>0.08</b>	0.19	0.21	0.42	<b>0.06</b>
CC [103]	0.02	0.03	0.06	<b>0.05</b>	0.1	0.11	0.18	0.09	0.21	0.23	0.45	<b>0.06</b>
RaKel [130]	0.01	0.01	0.03	<b>0.05</b>	0.08	0.1	0.16	0.09	0.15	0.18	0.35	0.07
EML(BR) [102]	0.04	0.05	0.08	<b>0.05</b>	0.12	0.13	0.24	0.09	0.24	0.24	<b>0.47</b>	0.07
EML(CC) [102]	<b>0.24</b>	<b>0.29</b>	<b>0.32</b>	0.09	<b>0.17</b>	<b>0.21</b>	<b>0.29</b>	0.12	<b>0.27</b>	<b>0.31</b>	0.46	0.09

TABLE 5.2: Performance metrics of MUTAMA instantiated with different multi-label classifiers. The best results for each metric are highlighted in bold.

Table 5.2 depicts the results for the different instantiations of MUTAMA on the collected dataset. Results of the multi-label classifiers (cf. Section 2.4.2) were obtained using 10-fold cross-validation, where nine folds are considered for training and one for evaluation. The metrics for each of the folds are later averaged to get the numbers depicted in Table 5.2. The table is divided into three regions for the analysis of libraries that carry two, three, and more than three tags. As can be seen from the table, the results differ between the regions.

<sup>13</sup><https://sourceforge.net/projects/meka/>

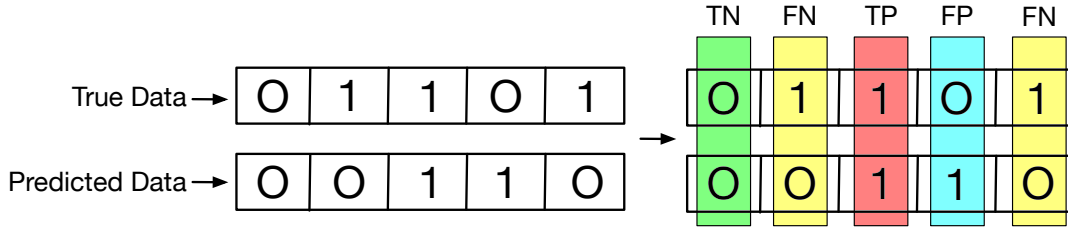


FIGURE 5.4: A binary array example, its predictions and posterior classifications into *TP*, *TN*, *FP*, *FN*.

The metrics considered in the evaluation are *Accuracy* (*A*), *F1 macro score* (*F1-M*), *F1 micro score* (*F1-m*) and *Hamming loss* (*HL*). Accuracy is described by Equation (5.1) and describes how the classifiers are generally performing for all tags.

$$A = \frac{TP + TN}{FP + FN + TP + TN} \quad (5.1)$$

Figure 5.4 provides an explanation for true positives (*TP*), false positives (*FP*), true negatives (*TN*) and false negatives (*FN*) in the context of multi-label tag classification. The binary array on the top left is the true data marking three tags to be predicted by the classifiers (i.e., elements that are “1” in the array) out of five tags (i.e., number of elements in the array). The bottom left binary array is the predicted data for a classifier, e.g., only two tags out of five are predicted in this case. The right side of the figure shows the same arrays but highlighting the specific classifications per element. For example, the first element is *TN* as the prediction is the same as the true data marking that this tag should not be selected. The third element also has similar true and prediction data, however, they both indicate the presence of a tag in the array and a *TP* prediction. Dissimilar data can be observed for the second, fourth and fifth elements. The second and fifth elements in the array are equally indicating an *FN* as the predicted value shows an absence of a tag, whereas in reality such a tag should be present. Lastly, the fourth element in the binary array is *FP* because the prediction marked this tag as present where in reality is not.

*F1 macro score* refers to the average of the *F1* scores per predicted tag, that is, it reflects the quality of the individual predicted tags. *F1 micro score* is computed by considering the prediction of all tags in the dataset instead of predictions per tag as in the *F1 macro*. *F1 macro* and *micro* are computed with the same formula as in Equation (4.3) but considering macro or micro precision and recalls. Indeed, *F1 macro* uses macro precision and recall in Equation (5.2) and Equation (5.3), respectively. On the other hand, *F1 micro* is computed by considering Equation (5.4) and Equation (5.5), respectively.

$$\text{Macro-Precision} = \frac{\sum_{i=1}^{|N|} P_i}{|N|} \quad (5.2)$$

$$\text{Macro-Recall} = \frac{\sum_{i=1}^{|N|} R_i}{|N|} \quad (5.3)$$

$$\text{Micro-Precision} = \frac{\sum_{i=1}^{|N|} TP_i}{\sum_{i=1}^{|N|} TP_i + FP_i} \quad (5.4)$$

$$\text{Micro-Recall} = \frac{\sum_{i=1}^{|N|} TP_i}{\sum_{i=1}^{|N|} TP_i + FN_i} \quad (5.5)$$

$|N|$  refers to the number of classes in the dataset, whereas  $P_i$  and  $R_i$  refer to the precision and recall respectively for a specific class  $i$ .  $TP_i$ ,  $FP_i$  and  $FN_i$  are the  $TP$ ,  $FP$  and  $FN$  for class  $i$ , respectively.

Hamming Loss is the ratio of incorrectly predicted tags to the total number of tags. This loss function described by Equation (5.6) should be optimised by the considered classifiers, zero being its optimal value.  $|D|$  refers to the number of instances in the data while  $y$  is true binary array and  $\hat{y}$  is the predicted array in Figure 5.4. Elements in the arrays are indicated by the  $i, j$  indexing (e.g.,  $y_{i,j}$ ).

$$HL = \frac{1}{|D| \cdot |N|} \sum_{i=1}^{|D|} \sum_{j=1}^{|N|} XOR(y_{i,j}, \hat{y}_{i,j}) \quad (5.6)$$

The results for libraries with only two tags are low for all classifiers except EML in combination with the base classifier CC (i.e., EML(CC)). There is, a noticeable gap between the F1 micro score for the best (0.32) and the second-best classifier (0.08). The same can be said about the results for the F1 macro metric. This reflects the effectiveness of ensemble algorithms in general, and the classifier chain ones in particular. Surprisingly, the classifier scoring best on accuracy, F1 micro, and F1 macro scores the lowest on the Hamming loss metric. However, the difference of 0.04 can be considered negligible.



Library	True Tags	Predicted Tags
Akka HTTP Backend	distributed, actor, akka, concurrency, client, http	distributed, actor, akka, concurrency, http
AWS SDK Scala.js Facade QuickSight	scala, aws, amazon, scalajs, sdk	scala, aws, amazon, scalajs, sdk
TestNG Interface	io, testing	testing
Camel Labs IoT Components Device IO	github, io	-
Gradle Code Quality Tools Plugin	tools, build, build-system, plugin, groovy, gradle	-

TABLE 5.3: Multi-tag predictions made by the best trained multi-label model.

The results for the libraries that carry exactly three tags are similar. The ensemble classifier EML(CC) once more obtains the best scores for the accuracy, F1 micro, and F1 macro metrics but the worst score for the Hamming loss metric. The gap between this classifier and the others is again noticeable. The other ensemble classifier EML(BR) consistently ranks as the second best.

The results for libraries that carry more than three tags in the ground truth are considerably interesting. We had expected the classifiers to score lower than before as more tags need to be predicted. Their performance is better than before. The two mentioned ensemble classifiers once again achieve the best scores.

## 5.4.2 Discussion

The differences in performance of the classifiers on libraries with two, three, or more than three tags could stem from differences in their respective distribution in the dataset. These differences are apparent from Figure 5.3, which depicts the number of libraries in each group in a lighter shade of blue. More importantly, each group is imbalanced as some tags are simply more prevalent on the *MVNRepository* indexing platform. The use of traditional cross-validation in our evaluation does not help in this case, as it does not consider data distributions in its assignment of instances to folds. Although some stratification techniques have been proposed to address imbalanced datasets for multi-label problems [30], they are not yet available in the MEKA tool suite which we used to instantiate MUTAMA.

For five randomly selected libraries, Table 5.3 depicts the ground truth and the predictions made by the best performing EML(CC) classifier. The prediction for library *Akka HTTP Backend* in the first row is only missing the *client* tag. As this library focuses on the backend, a *server* tag might have been more appropriate both in the ground truth and in the prediction. In the second row, all tags in the ground truth for library *AWS SDK Scala.js* are predicted.



For the third row, the classifier failed to predict that library *TestNG* is somehow linked to *io* operations. Again, this failure might as well be attributed to mistakes or unexpected tags in the data on the *MVNRepository* indexing platform. The tags on the platform are, after all, maintained by volunteers who might wrongly assign a tag or miss appropriate ones. Finally, for the libraries on the last two rows, the classifier was unable to predict a single tag.

We believe that such failures could be caused by several reasons. First, the number of libraries for each combination of tags is small. Machine learning classifiers train better with a higher number of instances from which they can extract patterns. Relevant patterns between library APIs and combinations of tags may therefore remain unlearned. Second, the API information extracted from the binaries might not be representative enough. Information relevant for tagging purposes might be hiding in the implementation of the API. Finally, word embeddings of class and method names might not sufficiently capture the features of a library. Alternatives such as the ASTs of their source code, traces of their static symbolic execution, or execution logs on Travis CI<sup>14</sup> could be explored as complementary sources of information.

### 5.4.3 Limitations of Tag-based Approaches for Features Discovery

A set of tags does indeed form a more descriptive kind of documentation of the features provided by a library. As shown in Table 5.3, features can be effectively captured using tags which may facilitate discovering libraries in a vast ecosystem. However, on platforms such as *MVNRepository* or Stack Overflow, the number of tags given to a library is usually limited to a few (e.g., most of the time to five). This limits the library feature information one can obtain from those websites. More importantly, until this point, the automatically computed feature descriptions have not included examples of code snippets that illustrate how to use a feature. **Both the category-based and tag-based approaches result in feature descriptions that are too coarse-grained and lack information about how the API of the library should be used for each feature.** Therefore, we decided to consider Stack Overflow posts as a source of library information, which provides API-level granularity and natural language text surrounding library usages. We refer the reader to Section 2.1 for an overview of Stack Overflow and the information its posts provide.

## 5.5 LiFUSO: An Approach to Discover Features from API Usages on Stack Overflow

Figure 5.5 depicts an overview of our approach LiFUSO and the sequencing of its computational steps to uncover features from SO library usages. The

<sup>14</sup> <https://travis-ci.org/>

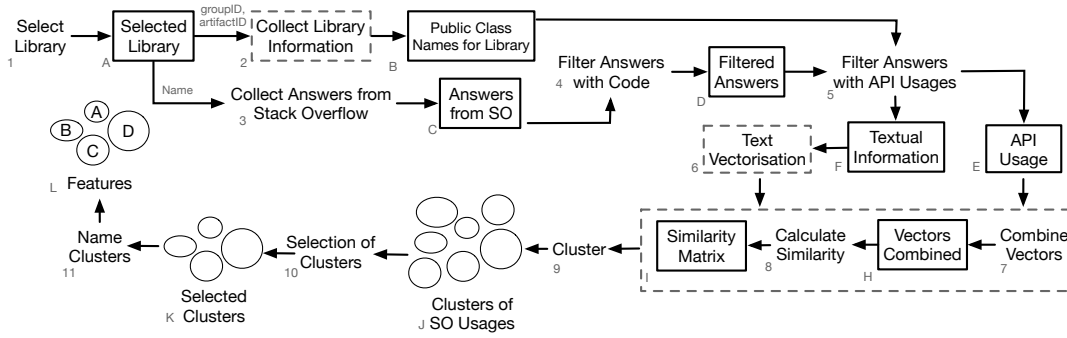


FIGURE 5.5: Overview of the approach to extract features from SO posts.

steps are denoted by a number (e.g., 1, 2), while letters indicate the data serving as input to or resulting from a step (e.g., A, B). Figures 5.6 and 5.8 zoom in on the internals of composite steps. The remainder of this section details each of the steps.

### 5.5.1 Data Collection (Steps 1-3)

**The first step of the approach corresponds to the selection of a target library for which we uncover the API features.** Our prototype implementation requires that the `groupId` and `artifactId` (e.g., `com.google.guava` and `guava` respectively) of at least one version of the library is published in the Maven central repository. Once the user has selected a target library, the approach automatically collects information about its public API from its published JAR files (steps 2.1 and 2.2 in Figure 5.6) and example usages of this API from SO snippets (step 3 in Figure 5.5).

To collect information about the public API of a library (cf. Figure 5.6), our approach considers all versions of the library published on Maven central and thus downloads their JAR files (data 2.A). Next, step 2.2 extracts the names of the public classes in these JAR files using the *Apache BCEL*<sup>8</sup> library. *Apache BCEL* allows extracting information from Java Bytecode files (e.g., `*.class`) such as those in JAR artifacts. Although the public API of a library might evolve over time, we collect all class names that were once considered part of it. We opt for this strategy because API usage examples on SO rarely mention the exact library version that they exemplify. Moreover, the first automated approach [153] to determining the compatible library versions for the API usage within a given code snippet still produces version ranges.

Step 3 in Figure 4.2 collects SO answers from the SOTorrent dataset [12]<sup>15</sup> that are likely to contain API usages of the selected library. Our approach uses a heuristic that considers all answers of which the question has been tagged with the name of the library (e.g., `guava`, `pdfbox`). More relaxed heuristics could also be used, such as requiring the library name to appear in the title

<sup>15</sup> November 16th, 2020 version from <https://zenodo.org/record/4287411>

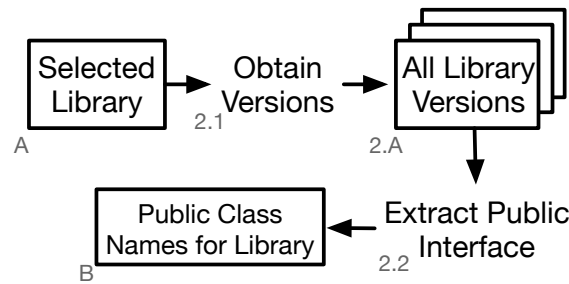


FIGURE 5.6: Collecting the public interface of a library for all its versions.

or in the body of the question. We opted for the tags heuristic as it is computationally inexpensive and because its strictness minimises false positives. In Chapter 6 we will revisit this heuristic by considering the API types predicted by RESICO (cf. Chapter 4) for an SO snippet.

In addition to the body of the answer, step 3 also collects the title, tags, and the body of the question for further analysis.

### 5.5.2 Data Processing (Steps 4-5)

Selecting SO answers based on the tags of their question ensures that the question concerns the target library. **Unfortunately, not all answers associated with a question contain code snippets from which API usage examples can be extracted.** For the Weka library<sup>16</sup>, for instance, we found that around 57% of the answers about questions tagged with `weka` do not contain any code. Weka can be used as a library as well as an independent application for machine learning. Step 4 in Figure 5.5 therefore filters out all answers without code.

At this point in the approach, the class names within the public API of the selected library (data B) and the answers from SO with code snippets in which the API of the library is likely to be used (data C) have been collected. Step 5 identifies answers with code snippets that use the public API of the selected library. To this end, it relies on a robust parser generated by a custom-built island grammar. Parsers generated by an island grammar [88] focus on some constructs of interest (i.e., islands) and consider the remainder of the text to parse as irrelevant (i.e., water). They have been shown well-suited to parsing and lightweight analysis of code that is grammatically incomplete (e.g., a statement without a surrounding method) or that contains syntax errors (e.g., three dots instead of an expression) such as the snippets on SO. Ponzanelli et al. [97, 98], for instance, have used an island parser to recognise code snippets within the natural language of SO posts.

As our data processing focuses on the public API of a library, our own island parser focuses on the syntactic constructs in which method invocations can occur. Figure 5.7 depicts an example of its output for the code snippet shown

<sup>16</sup> <https://www.cs.waikato.ac.nz/ml/weka/>



FIGURE 5.7: The API usages extracted by our island parser and its lightweight analysis for the code snippet on the left.

on the left. For a single method invocation within a variable declaration or expression statement, it produces the name of the statically declared type of the receiver expression followed by the name of the invoked method. For a chain of successive method invocations, common for libraries with a fluent API, it produces the statically declared type of the first receiver expression followed by the names of the successively invoked methods. Note that it is not possible to resolve the receiver types of an invocation within the chain without resorting to more heavyweight program analyses.

We implemented our island parser, and its lightweight analysis that extracts API usage information using the *parboiled*<sup>17</sup> framework. The parser pushes constructs of interest to an internal stack whenever islands are encountered in the sea of water. For the encountered variable declarations and parameters, it pushes the variable type and identifier. For the encountered method invocations, it pushes the method name and the receiver expression (instance method) or type name (static method). Upon the completion of parsing, the lightweight analysis inspects the lexical scopes on the stack for potential matches between the identifiers in receiver expressions and those in variable and parameter declarations. The receiver expressions for which there is a match, are replaced by the type of their corresponding variable or parameter declaration. As a result, the analysis retrieves the simple name of the receiver type of both instance and static method invocations.

With the API usage within each code snippet extracted by the island parser, step 5 merely must match the type and method names against those of the public API of the library (i.e., data B). SO answers without any match are discarded. For the remaining answers, the extracted API usage information is kept (i.e., data E) along with the natural language in their body as well as the title and the natural language in the body of the question to which the answer belongs (i.e., data F).

### 5.5.3 Data Transformation (Steps 6-8)

The next step in our pipeline is to transform the collected data. The transformation (i.e., box with dashed lines containing steps 7 and 8) independently process two inputs (e.g., text and code) and produces a similarity matrix per input.

The transformation process is twofold where I) the text data is pre-processed and a TF-IDF model [147] is trained and II) both code and text data are

<sup>17</sup> <https://github.com/sirthias/parboiled/wiki>

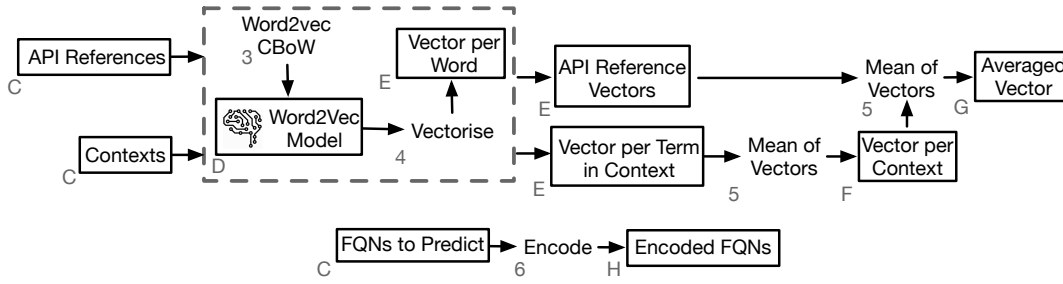


FIGURE 5.8: Vectorisation of the textual information.

**transformed into vectors.** Figure 5.8 details how the vectorisation proceeds for the textual information (i.e., data F). Step 6.1 removes all stop words, the name of the programming language (i.e., Java), and the name of the library. Symbols such as commas, question marks, and dots are also removed in this pre-processing step.

Next, step 6.2 applies a TF-IDF vectoriser to the pre-processed data. Once its model has been trained (data 6.B), the vectors for 3 text-related attributes are calculated, namely: the body of the question, the body of the answer, and the title. Therefore, three text-based vectors will be produced for each of the considered SO answers (data G). In the case of code data (data E), no pre-processing or model training is needed. The following code-related attributes are sent to step 7: the original method names, the method names after splitting camel case, and the API usage itself as extracted by the island parser.

Step 7 combines a preconfigured selection of attributes. Text-based vectors are averaged, and code-based attributes are concatenated. This is to support our empirical study into whether a single vector, a combination of some vectors, or all vectors produce more cohesive clusters. Regardless of the selected attribute to combine, the outcome of this step will be a single vector of numbers or code (e.g., classes and methods) per SO answer for text or code inputs, respectively. A similarity matrix resulting from computing a similarity metric between all vectors of all different answers, is finally calculated in step 8. We use cosine similarity to compute the distance between text-based vectors and Jaccard similarity for code-based vectors. Cosine similarity is a well-known metric to compute distances in text-related data, while Jaccard is more suitable for precise information such as code [55]. The generated matrices are the input to the next step.

#### 5.5.4 Clustering, Selecting and Naming (Steps 9-11)

Step 9 applies hierarchical clustering (cf. Section 2.6) to the similarity matrix of either API usages, or the textual information extracted for each SO answer. **We use a bottom-up dynamic cutting approach to determine the clusters of elements based on their API usages or their textual information.** The resulting clusters (i.e., data J) are groups of which the elements are close to one another yet far from the elements in other clusters.

The ideal cluster would be that one in which the feature is clearly depicted. However, this might not always be the case. We use the local outlier factor (LOF) [25] (step 10) to check the most common elements within a cluster. In those cases where LOF does not determine a frequent element (or outlier), the cluster is discarded.

To facilitate interpreting the results, step 11 provides a name for the features by computing the most frequent terms that appear within each selected cluster. From the textual information (i.e., title, question, and answer bodies) of each SO answer, we compute the semantic tree of each sentence [84] (cf. Section 2.5). For each noun and verb, we analyse the direct typed dependencies on another verb or noun to extract pairs of the form noun-verb or verb-noun.

Frequencies are calculated for each pair. Finally, the most frequent pairs are obtained through LOF as previously used for the most relevant code elements. Data L will ultimately contain a feature with related natural language terms corresponding to its name (e.g., fig. 5.9) and API usages.

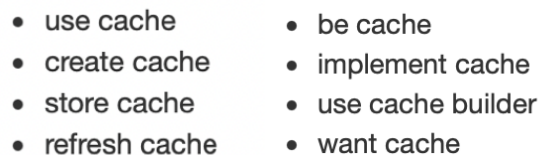
- 
- use cache
  - create cache
  - store cache
  - refresh cache
  - be cache
  - implement cache
  - use cache builder
  - want cache

FIGURE 5.9: A name for an uncovered feature suggesting operations on cache.

## 5.6 Instantiation of the LiFUSO Approach

### 5.6.1 Implementation

Features can now be extracted from library usages according to our previously described approach (cf. Section 5.5). **However, an instantiated version of our approach still needs a way to represent features to the developer.** Figure 5.10 presents how to instantiate a tool from the approach in Section 5.5. Depicted within the red square are the steps we chose to follow when instantiating our approach into the LiFUSO tool.

In step *Representing Features*, the tool selects the top-5 most frequent verb-noun pairs as the representation of the name of the feature. We found that selecting more frequent pairs (e.g., top-10, top-20) hinders discerning interesting features among several displayed next to one another and encumbers understanding individual features displayed in isolation. As the representation for the API references in the clustered snippets, the LiFUSO tool displays only those extracted through the LOF technique in a previous step. As such, clusters with the most frequent name pairs and API references are outputted.

The actual visualisation is produced in the *Creating Visual Interface* step. Depending on the view selected by the developer, the features common to a set



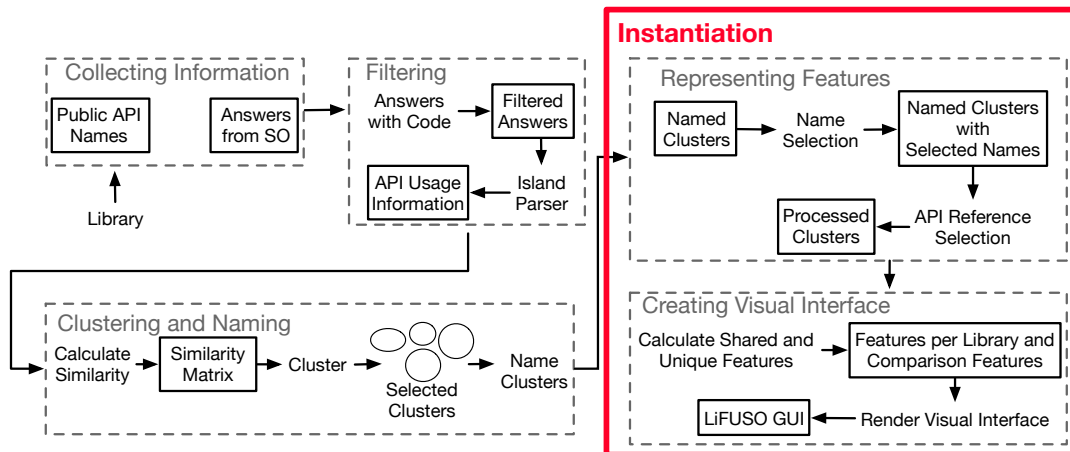


FIGURE 5.10: Described approach implemented by the LiFUSO tool. The instantiation steps are within the red square.

of libraries as well as features unique among a set of libraries are computed and displayed. Individual features can be inspected in detail too.

The LiFUSO tool implements the approach in Figure 5.10 using a series of components, each responsible for a separate data processing step. All steps up to and including the computation of the shared and unique features are pre-computed and their results are persisted. The tool’s graphical user interface (GUI) uses the pre-computed information, so navigation is swift. The following choices were made in the implementation of each component:

**Collecting Public API Names** Implemented in Scala, this component uses the *requests* library<sup>18</sup> to query Maven Central for all versions of a library. Once their JAR files have been downloaded, the names of the public API elements are extracted by processing their bytecode using the *Apache BCEL*<sup>8</sup> library.

**Collecting SO Answers with Code** The Scala implementation of this component uses the *Jsoup*<sup>19</sup> library to process the HTML of each SO answer that has the name of the library among its tags. Answers without a code block are discarded.

**Extracting Natural Language Terms** The *Java* implementation of this component uses the *Stanford NLP Toolkit*<sup>20</sup> to extract the natural language terms from the title of and the text surrounding the remaining SO answers.

**Extracting API Usage Information** We constructed an island parser with the help of the *parboiled*<sup>21</sup> library. Using the API usage information extracted by the parser for each SO answer, we used the Jaccard similarity to construct a similarity matrix.

<sup>18</sup> <https://github.com/com-lihaoyi/requests-scala>

<sup>19</sup> <https://jsoup.org/>

<sup>20</sup> <https://nlp.stanford.edu/software/>

<sup>21</sup> <https://github.com/sirthias/parboiled>

**Clustering** To compute the clusters given the similarity matrix, we used the `hclust` function in the *stats* package from the *R* standard library and relied on the dynamic tree cut feature of the third-party *dynamicTreeCut*<sup>22</sup> library.

**Cluster Selection and Naming** To discard low-quality clusters without frequent API references, we used the LOF implementation provided by the *scikit-learn*<sup>23</sup> library of *Python*. We rely on *reticulate*<sup>24</sup> for the interoperability between *R* and *Python*. All other data between components is exchanged through CSV and TXT files. The previously extracted natural language terms are now used to name each remaining cluster.

**Extracting Feature Representations** As mentioned, the top-5 most frequent verb-noun pairs are extracted as the name representation whereas API references filtered by LOF are selected for the code part of the feature. The usage frequency of the API references in all clustered code snippets is also calculated and included in the feature information. We implemented these frequency calculations using the *R* programming language.

**Calculating Common and Unique Features** Every pair of two libraries in the ecosystem is considered and their features are compared. The comparison focuses on the names produced for their features: if two features from different libraries have a common name (i.e., a verb-noun pair), then these features are included in those considered shared by the two libraries. Conversely, if no other library has a feature with the same name, then that feature is considered unique to the library under analysis. We implemented this comparison in the *Julia* programming language.

**Graphical User Interface** We used the *Nuxt*<sup>25</sup> framework for *JavaScript* to implement the tool's graphical user interface. Through this interface, users can explore features present in an ecosystem and compare libraries to one another. Currently, the interface only supports selecting two libraries for comparison. Future improvements will support comparing a selection of multiple libraries.

The polyglot nature of our implementation is due to our decision to select a programming language and the third-party libraries for each component separately.

## 5.6.2 Graphical User Interface

Figure 5.11 depicts the main interface through which users can explore the features of a library. The tab bar surrounded by the rectangle labelled *A*, enables switching between the feature *Search* and the library *Compare* features

<sup>22</sup> <https://cran.r-project.org/web/packages/dynamicTreeCut/>

<sup>23</sup> <https://scikit-learn.org/stable/index.html>

<sup>24</sup> <https://rstudio.github.io/reticulate/>

<sup>25</sup> <https://v3.nuxtjs.org>



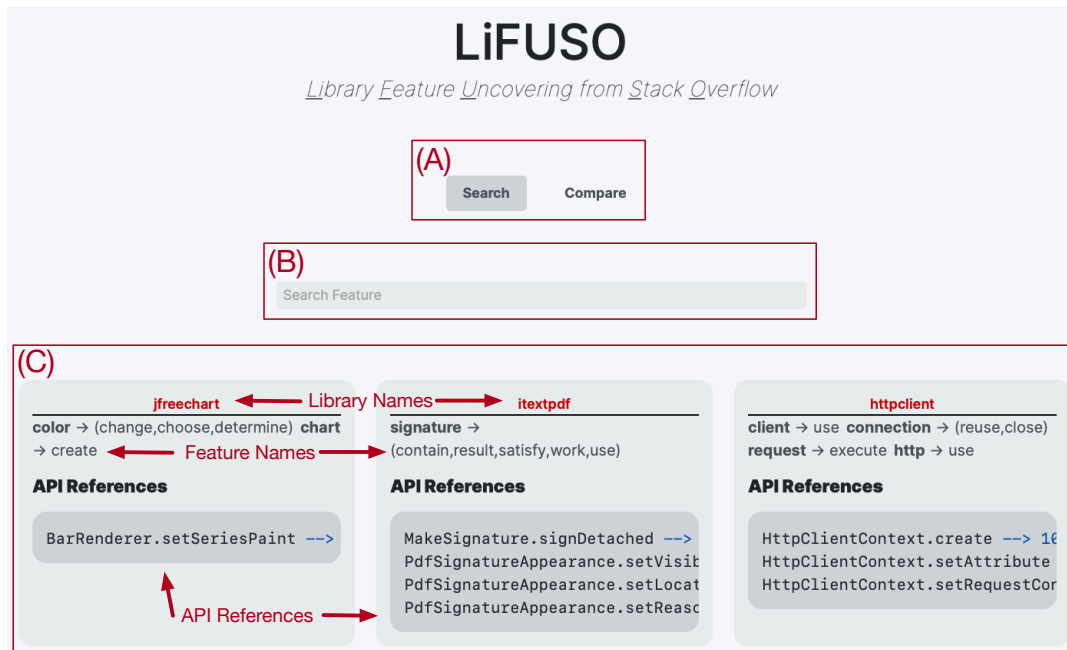


FIGURE 5.11: Front page of the LiFUSO tool with the Search feature tab activated.

of the tool. The *Search* tab is active by default and contains the elements with labels B and C. The search box with label B enables searching through the features uncovered for the ecosystem using natural language queries, which are matched against the feature names. Example queries include “collection”, “filter collection”, “create chart”, or “extract image”.

Features that match the query are displayed in a grid of feature views labelled C. Each individual feature view depicts the following information:

**Library Name** Centered in red at the top of the box (e.g., *jfreechart*).

**Feature Name** Collection of frequent verb-noun pairs describing the feature. Each noun is rendered in a bolder font, and followed by the verbs that are commonly used together with that noun.

**API References** The most frequent API references from SO posts to that feature are shown in a darker colour at the bottom of the box. The percentage shown next to each reference corresponds to the frequency at which the reference occurs within the SO posts clustered together for the feature.

Figure 5.12 shows the *Compare* tab activated and its widgets. The two combo boxes in the rectangle labelled D are populated with the libraries in the ecosystem and enable selecting the two libraries of which the features need to be compared. Section 5.8.4 will demonstrate more of the LiFUSO user interface in a case study.

## 5.7 Evaluation

Our LiFUSO approach is assessed by measuring the cohesiveness of the generated clusters, by comparing the obtained features with ground truth code in cookbooks or tutorials, and lastly by matching non-covered features in sampled GitHub projects. The evaluation of our approach focuses on the API calls of the generated features.

Specifically, our evaluation aims to answer the following research questions:

- RQ1.** Which combination of SO answer attributes produces the most cohesive clusters?
- RQ2.** How similar are the automatically uncovered features to documented tutorial features?
- RQ3.** To what extent do the uncovered features that do not match documented tutorial features correspond to actual API usage in client projects?

### 5.7.1 Selection of Libraries

As candidate subjects for the evaluation of our approach, we consider any Java library from the Maven software ecosystem. We restrict candidate libraries to those that are used in SO snippets, so our approach can extract features from this usage. Moreover, candidate libraries need to have publicly available cookbooks or tutorials that illustrate the proper usage of the library. This is to enable comparing the features extracted by our approach with those documented for the library that will serve as ground truth.

We selected the 50 most popular Maven libraries, measured in terms of GitHub repositories that use them according to the Libraries.io dataset.<sup>26</sup> From the initial population, we were able to find usage examples of features in tutorials or cookbooks in seven libraries: *Guava*, *HttpClient*, *JFreeChart*, *JSoup*, *PDFBox*, *Apache POI-OOXML* and *Quartz*. These libraries form the subject systems for our evaluation and cover diverse application domains.

*Guava* is a multi-purpose library collecting various auxiliaries related to data structures, I/O, caching, hashing, etc. *HttpClient* enables sending HTTP requests and process their responses. *JFreeChart* provides an API for creating and exporting graphics. *JSoup* facilitates manipulating elements present in web pages (e.g., links, divs). Similarly, *Apache POI* facilitates processing documents created with the Microsoft Office suite. We selected the sub-library dedicated to Excel worksheets (i.e., *Apache POI-OOXML*). Finally, *Quartz* allows launching and scheduling jobs programmatically.

Table 5.4 depicts the number of SO code snippets from where API usages are extracted for each library.

---

<sup>26</sup> <https://libraries.io/api>

	Guava	HttpClient	JFreeChart	JSoup	PDFBox	POI	Quartz
Snippets	1,522	628	882	3,465	945	2,010	511

TABLE 5.4: SO code snippets making use of the libraries.

### 5.7.2 Features Terminology

For disambiguation purposes, we consider it necessary to define some terminology before starting with the evaluation. We consider three types of constructs referred to as features or possibly containing a feature. **First, we refer to the features generated by our approach as “uncovered features”, i.e., clusters that contain related API usages. Second, we refer to the features which we extracted manually from cookbooks as “tutorial features”.** We use these tutorial features as a first ground truth to compare our uncovered features with. **Third, the “GitHub API usages” correspond to usages of a library at the method level in a GitHub client project depending on the library.** The GitHub API usages form our second ground truth in the evaluation below.

### 5.7.3 RQ1. Which combination of SO answer attributes produces the most cohesive clusters?

#### Design

This research question investigates which of the attributes related to API usage examples on SO produce more cohesive clusters and therefore more defined features (i.e., features with API calls present solely in one cluster).

To this end, we recall the attributes extracted from a SO post: the title of the question, the body of the question, the body of the answer, method names, method names splitted by CamelCase (CC), and complete API calls. Section 5.5 categorised these attributes into a text attribute category and a code attribute category.

We check the cohesiveness of the clusters resulting from clustering solely on each attribute, as well as from clustering on combinations of attributes (e.g., question bodies combined with titles). Each vector in an attribute matrix represents the combination corresponding to one code snippet for that attribute (or the combination of several attributes). Once our approach obtains a matrix for each attribute, it proceeds to generate combinations with other matrices. The number of possible combinations equals 14 when considering combinations of size 1 to 3 for the attributes in each category; hence each category will produce 7 matrix combinations. For the 8 cases where a combination involves more than one attribute, a mean matrix (between the matrix attributes’ vectors) results from the average of matrices in the combination. Therefore, 14 matrices, one per combination, will be analysed below.

We apply hierarchical clustering to each computed matrix and compute the quality of the resulting clusters. We determine the cohesiveness of the clustering of each combination using the Silhouette score [104]:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \quad (5.7)$$

where  $i$  represents one vector in the matrix,  $a(i)$  calculates the distance between a vector  $i$  and its  $j$  cluster neighbours,

$$a(i) = \frac{1}{|C_i| - 1} \sum_{j \in C_i, i \neq j} d(i, j) \quad (5.8)$$

whereas  $b(i)$  measures the distance between the vector  $i$  and other vectors  $j$  that are part of a different cluster,

$$b(i) = \min_{k \neq i} \frac{1}{|C_k|} \sum_{j \in C_k} d(i, j) \quad (5.9)$$

The Silhouette score is in the range  $[-1, 1]$ ; values closer to 1 imply more cohesive clusters, on the other hand, values closer to -1 imply clusters with incorrectly grouped elements. In other words, more cohesive clusters are groups of which the elements are close to each other and at the same time further away from the elements of other clusters.

## Results

Table 5.5 shows the Top- $K$  Silhouette scores for all combinations with  $K$  ranging from 1 to 3. The best attribute combinations are very similar for most of the libraries. *Methods* CC appear in the Top-3 for all libraries. Other combinations are frequent throughout libraries such as *Methods* and *Methods* + *Methods* CC. *Methods*, *Methods* CC and *Methods* + *Methods* CC are the most frequent in the Top-1 with 3 appearances each (notice the same score for Top-2 and Top-3 in some cases).

The mean of the Top-1 scores is 0.48, indicating that clusters are sufficiently cohesive and differentiated. Their quality, on average, is on the borderline of what Kaufman and Rousseeuw [66] term “reasonable structure”. A vast majority of the combinations are vectors formed by single attributes (e.g., *Methods*) denoting lower scores for combinations of more attributes. Finally, all combinations in Table 5.5 refer to code attributes instead of textual attributes. Code information is more concise and less ambiguous than textual information, as a feature can be expressed in many ways using natural language text. Moreover, the text information present in each SO answer is rather specific to the solved task and might not be related to other answers.

Library	K=1	K=2	K=3
Guava	Methods (0.6)	Methods CC (0.6)	Methods + Methods CC (0.59)
HttpClient	API Calls (0.46)	Methods (0.45)	Methods CC (0.45)
JFreeChart	Methods (0.38)	Methods CC (0.38)	Methods + Methods CC (0.38)
JSoup	API Calls (0.63)	Methods CC (0.51)	Methods (0.51)
PDFBox	Methods + Methods CC (0.37)	Methods CC (0.36)	Methods (0.36)
Apache-POI	Methods (0.52)	Methods CC (0.52)	Methods + Methods CC (0.51)
Quartz	Methods + Methods CC (0.45)	Methods CC (0.42)	API Calls (0.42)

TABLE 5.5: Top-K scores for all combinations of library attributes. Between parentheses is the obtained Silhouette score.

Among the 14 considered attribute combinations, the singleton attribute combination *Methods CC* is the most frequent among those resulting in the Top-3 most cohesive clusterings. The Top-1 attribute combinations achieve a mean score of 0.48, and the formed clusters exhibit reasonable structure on average.

#### 5.7.4 RQ2. How similar are the automatically uncovered features to documented tutorial features?

##### Design

For each of the 7 selected libraries, we manually extracted the usages from their tutorial features, similar to the outcome of the island parser in Figure 5.7.

Tutorial features are compared one by one with all uncovered clusters. A tutorial feature is matched to an uncovered feature based on the class and method names within each other. When matches occur, we store the uncovered feature identifier (i.e., a number) and the tutorial feature that was matched. Classes and methods in both types of features (i.e., tutorial and uncovered) could be fully or partially matched. For example, for *HttpClient* chained API calls are common in its SO answers. A tutorial may split these chains into separate invocations over several lines that produce the same outcome. Conversely, a single tutorial feature could illustrate the usage of more than one of our uncovered library features.

As API calls of a tutorial feature could be underrepresented in the uncovered feature's population, i.e., other classes or methods might appear in the latter

which are not part of the former, we assess our approach using a relevance score:

$$relevance = avg(rel_c + rel_m) \quad (5.10)$$

where *relevance* is the average between  $rel_c$  and  $rel_m$  that measure the relevance of the uncovered feature classes and methods, respectively. In turn,  $rel_c$  and  $rel_m$  are defined as:

$$rel_c = \frac{\sum_{i=1}^{NCF} \frac{NC_i}{CA}}{NCF}, rel_m = \frac{\sum_{j=1}^{NMF} \frac{NM_j}{CA}}{NMF}$$

where  $NC_i$  is defined as the number of SO answers in the uncovered feature that contains the tutorial class  $i$ , while  $CA$  denotes the total number of SO answers in the uncovered feature. Similarly,  $NM_j$  denotes the number of SO answers in the uncovered feature that contains the tutorial method  $j$ . Finally,  $NCF$  and  $NMF$  represent the number of classes and methods respectively that comprise a tutorial feature.

$rel_c$  measures the average percentage of SO answers in the uncovered feature that contain each tutorial feature class, and accordingly  $rel_m$  measures the percentage for methods. Both  $rel_c$  and  $rel_m$  are within the range  $[0, 1]$  thus bounding the relevance score in Equation (5.10) within the same range. Values closer to 0 or 1 indicate weak or strong relevance of the uncovered features with respect to the tutorial features, respectively. We aggregate relevance values for uncovered features of a library by averaging the relevance scores of each feature.

In a similar way to the relevance score, we compute an overflow metric that calculates the number of classes and methods, within an uncovered feature, that differ from the tutorial features. The overflow metric is defined as:

$$overflow = avg(over_c + over_m) \quad (5.11)$$

$$over_c = \frac{CF \setminus CC}{NCF} - \frac{CC \setminus CF}{NCC}, over_m = \frac{MF \setminus MC}{NMF} - \frac{MC \setminus MF}{NMC}$$

where  $NCF$  and  $NMF$  are defined in Equation (5.10),  $NCC$  and  $NMC$  are defined as the number of classes and methods within an uncovered feature, respectively.  $CC$  and  $CF$  are the classes of uncovered and tutorial features respectively; equivalently  $MC$  and  $MF$  represent the methods also in uncovered and tutorial features, correspondingly. The overflow ranges from  $[-1, 1]$  with a score closer to -1 meaning an abundance of classes and methods in uncovered features not being present in tutorial features. In contrast, values closer to 1 allow concluding that the tutorials contain several elements from the uncovered features. An overflow metric close to 0 means that a tutorial and an uncovered feature are at an average distance of zero from each other. A value of 0 can arise in two possible scenarios: I) there is a similar disparity between  $over_c$  and  $over_m$  but with different signs (e.g., -0.5 and 0.5), or II)

there is a similar set of classes and methods between the tutorial and the uncovered feature. Scenario II is desirable because it will mean that the ground truth and the uncovered features are closer to each other.

Finally, we measure the accuracy of the matches, i.e., how many uncovered features match tutorial ones. Other metrics considering the total number of features (e.g., recall) are not realistic since those found in tutorials might only represent a fragment of the features of a library.

## Results

Table 5.6 presents the results for the matches between uncovered and tutorial features. The second column of the table shows the number of tutorial features for each of the libraries, while the third column displays the number of matches between the uncovered features and the tutorial features. Also, we report the number of tutorial features that are not found in our data (fourth column), and the scores of accuracy, relevance and overflow in the last four columns. Note that a tutorial feature might be present in several uncovered features with different relevance scores, however, we select the cluster with the highest relevance. A higher relevance implies similar classes and methods between the features in comparison. The averages of highest relevances and overflows per library are shown in Table 5.6.

Library	No. Feat.	Match.	Not Found	Acc.	R-Acc.	Relv.	Over.
Guava	30	22	7	0.73	0.97	0.71	-0.28
HttpClient	12	9	3	0.75	1.0	0.71	-0.23
JFreeChart	9	8	0	0.89	0.89	0.73	-0.26
JSoup	15	11	0	0.73	0.73	0.95	-0.51
PDFBox	15	8	4	0.53	0.73	0.82	-0.30
Apache-POI	20	10	3	0.50	0.59	0.75	-0.39
Quartz	22	12	5	0.55	0.71	0.65	-0.58

TABLE 5.6: Analysis of the matched features per library.

A total of 123 features (sum of *No. Feat.*) were collected. Our approach achieves high accuracy for the *JFreeChart* library with 89% and at least 50% for all libraries in the analysis. It achieves more modest accuracy scores on *Apache-POI*, *PDFBox* and *Quartz* with 50%, 53% and 55% respectively.

Interestingly, the features that our approach did not recover comprise API calls that were not found in SO. For *Guava*, 7 out of 30 tutorial features were missing from the SO data, representing around 23% of *Guava*'s features. The same goes for *HttpClient*, for which 3 out of 12 (25%) of the tutorial features did not have SO snippets involving their API calls. Column *R-Acc.* therefore, depicts the accuracy of uncovered features, but this time only considering the tutorial features of which the API usage also appears in SO snippets. Here, our approach improves its accuracy by 13% with respect to the *Acc.* column.



The relevance results (*Relv.* column in Table 5.6) shows good performance generally. The average relevance achieved for the libraries is 76% with a standard deviation of  $\pm 9$ , with the lowest score being achieved for *Quartz*. The highest relevance score, 95%, was achieved on the *JSoup* library. This library does not have a great variety of classes and methods; hence, the generated clusters cover the majority of classes and methods referenced in tutorials.

The overflow metric (*Over.* column in Table 5.6) shows negative scores on average for all libraries indicating the presence of classes and methods in the uncovered features not being in the tutorial features. The overflow results combined with the relevance scores suggest that tutorial features are in their majority covered, but there might be additional classes or methods in the uncovered feature that are frequently used.

Uncovered features have an average accuracy of 67%, which increases to 80% when comparing only to tutorial features with calls that appear in SO code snippets. High relevance scores indicate that uncovered features are highly similar to tutorial features. Our uncovered features are very likely to contain additional classes or methods, as indicated by the scores on the overflow metric.

### 5.7.5 RQ3. To what extent do the uncovered features that do not match documented tutorial features correspond to actual API usage in client projects?

#### Design

Tutorial features might cover but a fragment of the features of a library. As the complete set of features is not well-defined by developers nor by the community, tutorials do not allow us to fully evaluate the features uncovered by our approach. We therefore investigate alternative sources to verify the unmatched uncovered clusters from the previous research question (cf. Section 5.7.4). We select GitHub as such an alternative source. Numerous GitHub projects can depend on a particular library, each using a diverse set of library features.

To obtain the API usage from GitHub clients for a particular library, we first query the API of Libraries.io to retrieve candidate client projects with a declared dependency on the library under analysis. Note that these client projects declare a dependency to libraries in their configuration file, but do not necessarily contain actual API usage of the library in their source code [114]. We therefore discard GitHub client projects without actual API usage of the library under analysis. We continue this filtering process until we have collected, for each library in our evaluation, a statistically significant sample of GitHub client projects from Libraries.io with a confidence level of 95% and a confidence interval of 5%.<sup>12</sup>

We clone the sampled GitHub repositories, extract the bodies of their method declarations, and obtain the API usages within each method using our island



parser. We only extract and keep the API usage that belongs to the library under analysis.

We then compare the resulting API usages to the non-matched clusters from RQ2. In this experimental setting, the relevance (Equation (5.10)) and overflow (Equation (5.11)) scores are also computed to compare GitHub API usages with the uncovered features produced by our approach.

## Results

Library	Clients	Sample	No. Methods
Guava	103,158	383	7,198
HttpClient	71,540	382	1,672
JFreeChart	2,830	338	3,829
JSoup	33,203	380	4,575
PDFBox	3,703	348	2,017
Apache-POI	35,517	380	1,769
Quartz	17,460	376	1,737

TABLE 5.7: Client projects information from GitHub.

Table 5.7 presents information about the GitHub projects that use each library. The number of client projects (second column in Table 5.7) ranges from approximately 3K to more than 100K. This is expected since a multi-purpose library such as *Guava* is useful across application domains.

Library	No. Feat.	M-RQ2	U-RQ2	M-RQ3	%	Relv.	Over.
Guava	110	22	14	91	95	0.45	0.00
HttpClient	38	9	7	24	77	0.50	0.05
JFreeChart	70	8	5	55	85	0.35	0.02
JSoup	81	11	11	52	74	0.60	-0.21
PDFBox	44	8	7	32	86	0.42	0.01
Apache-POI	81	10	9	53	74	0.50	-0.10
Quartz	31	12	5	19	73	0.48	0.00

TABLE 5.8: Newly matched features from GitHub client projects.

Table 5.8 presents the information with respect to matching the uncovered features with GitHub API usages. Note that we split our results since we I) perform a second evaluation for the uncovered features that were matched to tutorials in RQ2 (cf. Section 5.7.4) and II) evaluate the remaining clusters, i.e., those features that were not matched against tutorials in RQ2. Columns *No. Feat.* and *M-RQ2* refer to the number of uncovered features extracted by our approach and the number of matched tutorial features from the cookbooks (cfr. RQ2) respectively. The features matched in RQ2 (column *M-RQ2*) were inspected to check unique matches in column *U-RQ2*.

Features in *U-RQ2* were removed from the uncovered features (column *No. Feat.*) to compute the new matches. Values in the column *M-RQ3* show the number of new matched features found in GitHub API usages and their coverage percentages (column %).

GitHub API usages cover the unmatched uncovered features to a high degree, with the highest coverage scores being realised for *Guava*, *PDFBox* and *JFreeChart* with 95%, 86%, and 85% respectively. This finding reveals that many tutorials paint an incomplete picture of a library's features. Moreover, some of the unmatched ones might still be used in a project outside of our sample or they might represent rare features.

We measure once again the relevance (column *Relv.*) of the uncovered features with respect to the GitHub usages to quantify the similarity of the two types of data. Relevance in Table 5.8 is on average lower in this case. Lower relevance is related to the overflow metrics (column *Over.*) which have positive values for most of the libraries, in contrast to Table 5.6. Although overflow scores are positive, they are still close to zero; however, a closer inspection reveals that the overflow of classes remains mostly negative whereas methods are in their majority shifted to positive values, hence the mean seems steady towards zero. This finding (reflected in Table 5.9) suggests that the GitHub methods from which we extracted the API usage either tend to use several uncovered features together in their body, or that the way developers group API calls together into client methods does not align with the boundaries of our grouping into uncovered features.

	Guava	HttpClient	JFreeChart	JSoup	PDFBox	POI	Quartz
Classes	-0.18	0.03	-0.21	-0.27	-0.15	-0.30	-0.24
Methods	0.18	0.06	0.26	-0.15	0.17	0.10	0.24

TABLE 5.9: Detailed overflow scores per library.

The majority of the uncovered features are found in sampled GitHub client projects. On average, the uncovered features are considered relevant with respect to the API usage within GitHub client methods. GitHub usages might be less focused or encompass more than one of our features.

## 5.8 Discussion

This section discusses the results, limitations, and potential impact of the approach to discover features from API usages in SO posts.

### 5.8.1 Clusters as Features

Our LiFUSO approach uses a set of attributes about SO posts to obtain a coherent clustering of API usages. The evaluation shows that the best clusters

```
1 // draw line, use chart, add jpanel, ...
2 XYSeries.add(...);
3 ChartFactory.createXYLineChart(...);
4 // show value, show percentage, change size, ...
5 DefaultPieDataset.setValue(...); // or
6 DefaultCategoryDataset.setValue(...)
7 ChartFactory.createPiechart(...);
```

LISTING 5: Examples of features uncovered for JFreeChart.

```
1 // convert image, convert pdf
2 PDDocument.load(...);
3 PDDocument.getPage(...);
4 PDPageContentStream.drawImage(...);
5 PDDocument.save(...);
6 // merge file, reuse PDFMergerUtility, ...
7 PDFMergerUtility.mergeDocuments(...);
```

LISTING 6: Examples of features uncovered for PDFBox.

are formed under attribute configurations that include the methods that are involved in an API usage (see Table 5.5). The combination *Methods + Methods* CC might initially introduce a repetition of the information, especially in methods with no CamelCase style. However, many of the libraries (6 out of 7) in Table 5.5 present a different score for the mentioned combination and their constituents, except for *JFreeChart*. Although small, these differences indicate some degree of CamelCase usage and therefore, different information to be analysed.

Some of the resulting clusters even contain calls to API methods that belong to different classes. For example, one of the clusters obtained for *Guava* has the transform API method as the most frequent element. This feature is intended for transforming one collection data type into another. As a result, several classes implement it such as Lists and Maps.

Features can also include several API method calls that have to be used together. Listings 5 and 6 depicts examples from *JFreeChart* and *PDFBox* respectively.

However, LiFUSO also produces some clusters that are difficult to interpret as features of a library. Listing 7 depicts clusters that consist of a single API call, which should not be used on its own but together with other calls. Line 2 could be used as a closing statement for an HTTP response, line 4 as an additional attribute to the axis configuration of charts, and lines 6 and 8 as predicates within a conditional. Such clusters are likely to result from calls that appear in isolation without further context within the text of an answer.

```
1 // Feature from HttpClient
2 CloseableHttpResponse.close();
3 // Feature from JFreeChart
4 CategoryPlot.setAxisOffset(...);
5 // Feature from Apache-POI
6 POIXMLDocument.hasOOXMLHeader(...);
7 // Feature from Quartz
8 JobDetail.equals(...);
```

LISTING 7: Examples of non-feature clusters uncovered.

### 5.8.2 Limitations

The lack of previous work on automatically uncovering features for libraries represents an obstacle to the evaluation of our approach. There is no ground truth of features for a library. The feature set uncovered by our LiFUSO approach may not be complete itself. The results in Table 5.8 already indicated that some API usage in GitHub client projects does not match any of the features uncovered from SO data. This might be due to our approach producing an incorrectly formed cluster that does not correspond to a feature, or due to the API usage in the client project corresponding to a rarely used feature for which little SO data is available. As such cases are rare, however, the evaluation strengthens our belief that crowd knowledge (i.e., SO data) covers many of library features. The fact that most uncovered features was found in client projects strengthens our confidence in their correctness.

Another limitation of our approach is that it relies on SO posts being tagged with the name of the library for which features need to be uncovered. In other words, in the description of our approach in Section 5.5, we mention the selection of those SO posts having the name of the selected libraries as part of their tags. More specifically, step 3 checks the library's name in the tags and excludes posts not fulfilling the requirement. This heuristic might exclude posts that have not been tagged, but that might make use of a library. Therefore, Chapter 6 will propose another post selection technique that does not rely on tags, thereby possibly improving the results achieved by LiFUSO.

### 5.8.3 Potential Impact

As revealed by our evaluation, a substantial number of library features is not documented in any tutorial for their library. Our approach can help users to understand the features offered by a library, and library maintainers to document these features. Once applied to all libraries within a software ecosystem, our approach will enable comparing competing libraries in terms of their feature sets. This might help developers in selecting the most appropriate library for the task at hand, and library maintainers to assess the ease of use of the APIs of different libraries for the same feature. This could stimulate cross-pollination between libraries and increase the dissemination of novel ideas within a library domain, thereby improving the health

of the ecosystem. Lastly, developers could take advantage of the tool instantiation of LiFUSO (cf. Section 5.6) to compute a set of uncovered features for a given library and explore them. This exploration could enhance the process of library selection which tends to be biased towards older libraries, and the process of library comparison by founding it on uncovered feature sets.

### 5.8.4 Case Study

We now report on a small case study in which we use the tool for a feature-based comparison of the *ITextPDF*<sup>27</sup> and *PDFBox*<sup>28</sup> libraries. This small study will demonstrate the current capabilities of the proposed approach. While the considered libraries differ in their actual API, we expect them to offer similar features as both are intended for manipulating PDF files. We should, therefore, be able to spot shared natural language terms for the names of the features computed by LiFUSO, as well as some that are unique to each library.

We configured the tool with the appropriate groupId and artifactId for the two libraries. The number of initial SO answers with at least one tag with the name of the libraries is 8,511 and 2,643 for *ITextPDF* and *PDFBox* respectively. The remaining number of answers after the processing is 3,004 and 945 respectively.

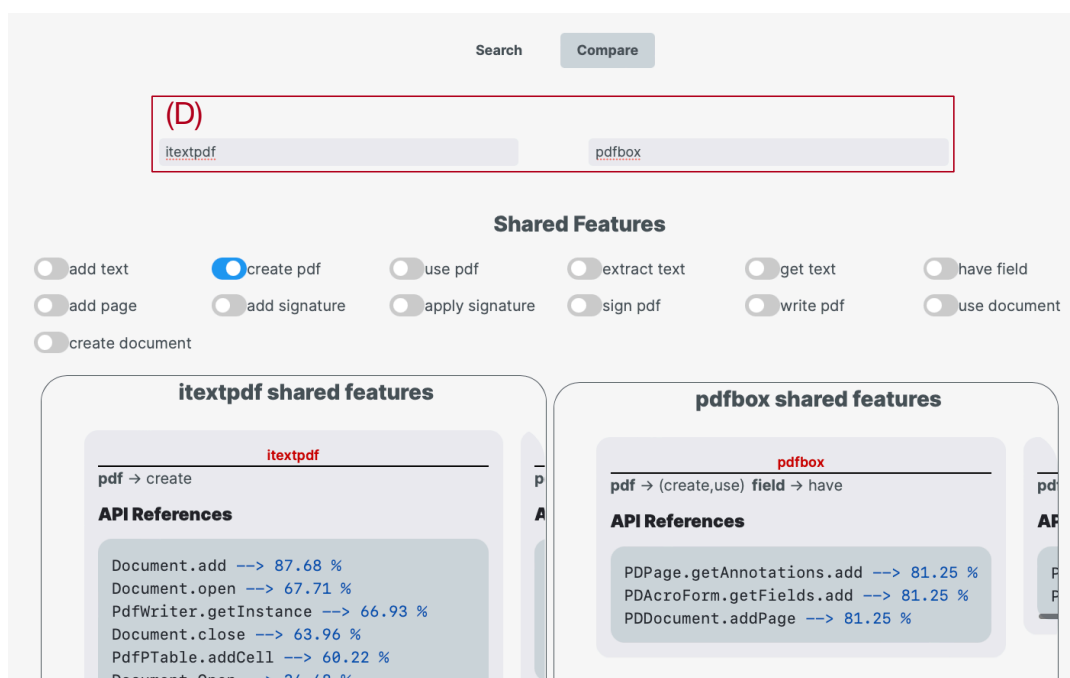


FIGURE 5.12: Shared features for the studied libraries.

The tool was able to compute 70 features for *ITextPDF* and 36 features for *PDFBox*. We used the *Compare* tab of the GUI to select these two libraries. Figure 5.12 depicts features in common to each of the libraries.

<sup>27</sup> <https://itextpdf.com>

<sup>28</sup> <https://pdfbox.apache.org>

First, a list of checkboxes is displayed, each of which corresponds to a shared feature. The number of features being shared is 13. Upon any checkbox activation, the corresponding features in each library with the same name are shown in the view below the checkboxes. The left and right sides of the view depict the shared features of the library selected in the left and right combo box respectively. This view enables seeing how two competing libraries offer the same feature in different ways, and to inspect their API differences. In the case of Figure 5.12, the feature `create pdf` is shared by both libraries and is therefore displayed on both sides of the view. *ITextPDF* has four features that include `create pdf` as name, whereas *PDFBox* has two.

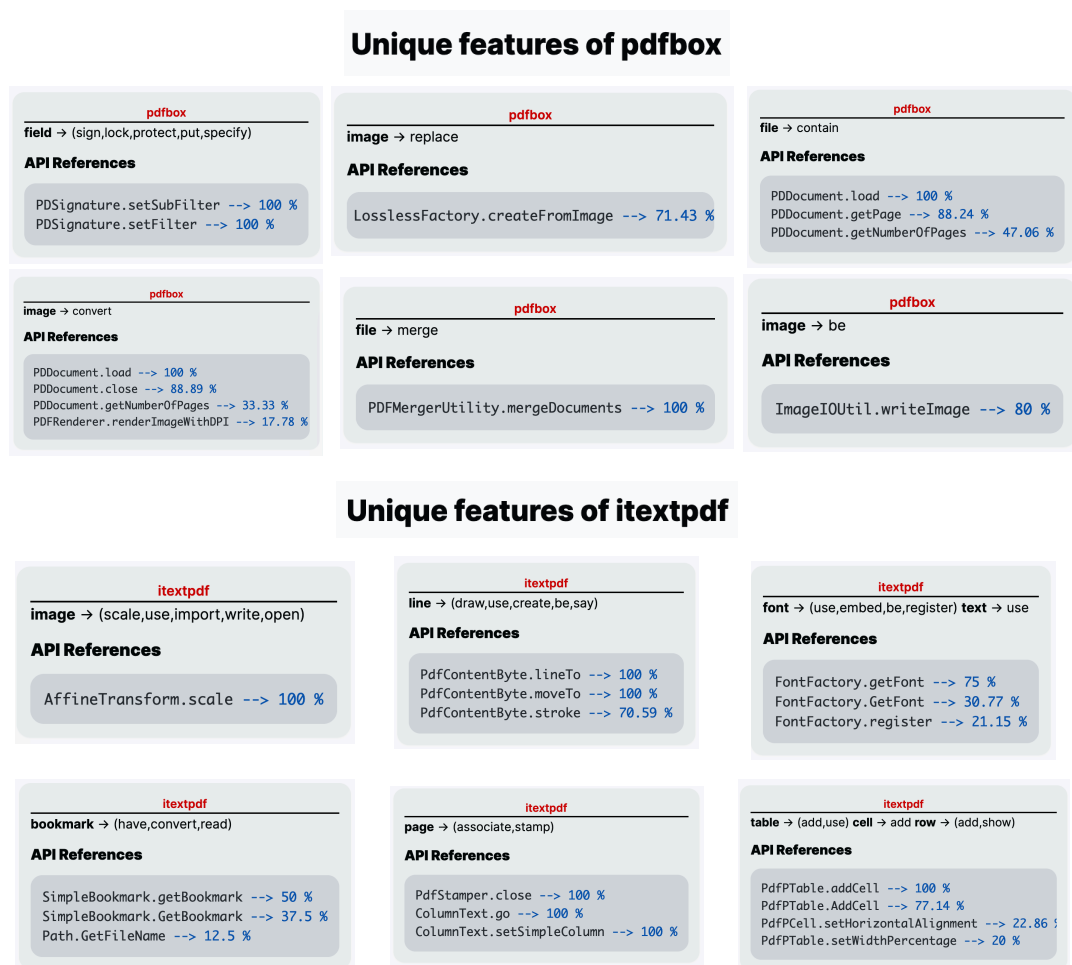


FIGURE 5.13: Unique features for the studied libraries.

Several features are unique to *ITextPDF* and *PDFBox*, as depicted in Figure 5.13. In particular, the features about scale image, draw line and use font seem to be unique to the *ITextPDF* library. Alternatively, *PDFBox* appears to have some unique features about converting or rendering an image (e.g., convert image) and merging two PDF documents (e.g., merge file, etc.). Other interesting unique features of *ITextPDF* are related to tables and bookmarks such as add table, add cell and read bookmark. Conversely, replace image, lock field and contain field seem unique to *PDFBox*.

To verify whether the unique features of a library are not implemented in the library being compared to, we might need to consult the developers of *both* libraries or to manually inspect their documentation. We leave this for our immediate future work. Note, however, that we have already compared the features computed for *PDFBox* against online tutorials for the library during the actual quantitative evaluation of the approach underlying the tool (cf. Section 5.5).

## 5.9 Threats to Validity

### 5.9.1 Threats to Construct Validity

Our approach considers an SO post to be related to a library if the library's name appears among the tags of the post. This heuristic minimises false positives, but more relaxed ones could be used (e.g., library name in question body). Nonetheless, our final contribution in Chapter 6 adopts a learning-based approach to determine whether a SO post without the name of a library in its tags, is related to the library.

There is also a minor risk that some GitHub projects considered for RQ3 (cf. Section 5.7.5) might have copied their code from SO. However, we believe that the impact of this threat is very low because the results of RQ2 (cf. Section 5.7.4) align with the results of RQ1 (cf. Section 5.7.3).

### 5.9.2 Threats to Internal Validity

We chose TF-IDF for the vectorisation of text attributes, even though more semantic approaches such as Word2Vec have recently been proposed. An initial experiment with Word2Vec resulted in less than five clusters per library where one cluster grouped most of the elements, but other vectorisation algorithms might still outperform TF-IDF. For the clustering algorithm, we selected hierarchical clustering based on its speed, effectiveness, and prior successes in API classification [55]. Other clustering algorithms such as e.g., K-Means or DBSCAN might produce different results.

### 5.9.3 Threats to External Validity

The success of our approach cannot be generalised readily to other Maven libraries, as it depends on the extent that their features are used in SO answers. Our approach might fail to uncover features for less popular libraries for which there is little usage in SO answers (cf. Table 5.4). Nonetheless, many of the designed steps in the LiFUSO approach could be adopted in techniques exploring features of libraries from other sources. For example, the clustering and NLP processing might also be adopted by methods relying on unit test cases and their text descriptions as alternative to SO API usages. In this way, this approach could possibly be expanded to more cases since most libraries include a test suite.



### 5.9.4 Threats to Conclusion Validity

Our selection of the best-performing attribute combinations relies on the Silhouette score. The score is a proven technique for analysing the quality of clusterings [66, 104, 112]. Other metrics proposed in Section 5.7.4 compute the relation of the uncovered features to tutorial and GitHub API usages. When analysed in depth, the metrics are designed to calculate such similarities without introducing bias towards the results. Moreover, as Table 5.6 and Table 5.8 indicate, there is no indication that the equations are inclined towards any library in the study.

## 5.10 Conclusion

This chapter proposed the LiFUSO data-driven approach to uncovering library features from API usage in Stack Overflow answers. We use a clustering technique (i.e., hierarchical clustering with dynamic tree cutting) to group the answers based on vectorised attributes. Our approach extracts the API references from possibly incomplete code snippets by means of a tailor-made island parser. Similarities between the API references within the snippets are calculated to finally return features as clusters, named according to the most frequent noun-verb pairs in the surrounding sentences. Natural language processing techniques are also used to name the clusters. A LOF technique is used to select clusters that correspond to features.

We evaluated our approach on seven popular libraries with cookbook-style documentation of their features. We achieve good performance for our uncovered features since accuracies and relevances are 67% and 76% on average, respectively. Uncovered features are highly covered by GitHub API usages (81% on average), however they might be part of larger usages as our results indicate.

We also instantiated LiFUSO into a data-driven tool for enumerating the features offered by the libraries in the Maven software ecosystem. The tool instantiation is completely based on the previously described approach. We have described the engineering aspects of the tool and reported on a case study in which we conducted a feature-based comparison of two libraries in the same domain. The tool is publicly available and can be used and extended to other popular libraries in the Maven software ecosystem.

The proposed approach and its tool instantiation represent an alternative to the current library selection process. The LiFUSO tool puts the focus on the features of libraries instead of *popularity* metrics. A case study using the tool demonstrated its potential for boosting the exploration and comparison of libraries in the Maven software ecosystem by providing feature descriptions with examples of the required API usage.



## Chapter 6

# Uncovering Library Features based on Resolved Code Snippets

### 6.1 Introduction

The approaches and tools discussed in previous chapters might be seen as isolated solutions to different problems. However, there is a connecting thread that enables their fusion into a single and cohesive approach. On the one hand, RESICO (cf. Chapter 4) provides API type resolution for incomplete code snippets such as those found in Stack Overflow posts. On the other hand, LiFUSO (cf. Chapter 5) discovers library features from API usages in Stack Overflow, taking advantage of their surrounding natural language descriptions.

The fusion point can be found in one of the steps of the LiFUSO approach in Figure 5.5. More precisely, step 3 is related to collecting answers from Stack Overflow. This step checks that the library name is among the tags in the post tag list. Posts without such a tag are discarded. This heuristic permits LiFUSO to be confident about the SO data in the analysis since posts with a library name in their tag list most likely contain API usages of the referenced library. Although the heuristic is precise, it might omit content posted by users that do not comply with the tagging policy of Stack Overflow.<sup>1</sup> Therefore, there might still exist a group of questions and answers in Stack Overflow that are related to libraries but which are not correctly tagged. RESICO, as an API resolution approach, could assist in solving the above problem.

The resolution capabilities of RESICO might help LiFUSO move beyond the tag heuristic. Our API resolution approach has demonstrated correct classifications for incomplete code snippets (cf. Section 4.4). RESICO can resolve the FQN of an API reference given previous training on the context where the reference is used without requiring syntactically correct and complete code. As such, RESICO may be able to recognise snippets on Stack Overflow that are likely using a particular library, thereby providing LiFUSO with more data from which to extract library features. However, the evaluation of RESICO was conducted in a controlled environment, its robustness in the wild is yet

<sup>1</sup> <https://stackoverflow.com/help/tagging>

to be shown. This chapter provides material to judge whether RESICO can perform effectively in the wild and, additionally, how well it serves as a supporting technique for LiFUSO.

LiFUSO is based on the tags heuristic to extract SO posts related to the usage of a particular library. In this chapter we complement the heuristic with an alternative technique. The information we extract solely from SO posts that satisfy the tags heuristic is precise but, as we hypothesise, incomplete. Considering more posts for extraction might offer the following benefits:

**Discovery of additional features** Considering more data to mine likely enables the discovery more features. Although this depends on the number of newly considered posts, we might expect new features to be discovered.

**Enrichment of already mined features** It might also be that new data does not provide new features but enhances already mined ones. For instance, considering more natural language information might refine the feature description.

**Inclusion of previously discarded features** The LiFUSO extraction process discards library usage patterns that do not occur with sufficient frequency. With the included data, some of these discarded patterns might gain sufficient support to be reconsidered as features.

**In the following sections, we will detail the design and implementation of strategies to expand the analysis of SO posts related to the LiFUSO libraries.** This expansion is based on the re-training of RESICO on a new GitHub corpus and its application to the whole SO dataset. Additionally, we present the results of our strategies through various evaluation methods. As part of the evaluation, we investigate the effects of the newly mined data on the LiFUSO extraction process. The discussion section dives into the results obtained in the evaluation and discusses the limitations of the new strategies. We list some threats to the validity of the evaluation and lastly conclude the chapter.

## 6.2 GitHub API Usages as RESICO Training Data

The RESICO approach produces a trained classification model resulting from a learning process on complete and syntactically correct code. Recall that Chapter 4 describes how the RESICO approach is trained on a corpus of 50K compilable Java projects [85]. The extracted information consists of API usages, their contexts and FQNs, the last ones serving as prediction for the multiclass classification model. However, the corpus of 50K projects is not focused on any particular usage of a library. Rather, the requirement for a Java project to be included in the repository is that it can be successfully compiled. Some libraries might not be used at all in the corpus. **Therefore, we analysed the usage of the libraries considered in the LiFUSO approach**

(cf. Chapter 5) to investigate whether to keep the RESICO model trained on the 50K-C dataset or to re-train it on new data.

An alternative source for API usages is GitHub and its recently included dependency graph network.<sup>2</sup> This network allows the exploration of dependencies and dependents for a GitHub repository. The latter is useful in our case because we look for API usages of libraries, and since many Maven libraries are also on GitHub, an exploration of the library repository dependents might yield many usage examples. Dependents are all projects in any programming language that include a library in their configuration files.

We used a tool<sup>3</sup> to obtain the dependents of each LiFUSO library. Each dependent includes at least the library under analysis in its build file (e.g., `pom.xml`). The presence of a library in the build file does not necessarily mean usage in the project code. To further explore the library usage, we relied on an API usage extractor that analyses the code in all Java files and extracts usage from the libraries. We built a custom API extractor inspired on a previous tool based on Eclipse JDT parser (cf. Chapter 4). New functionalities are added in this extractor such as the extraction of commit hash numbers, file location, among others (cf. Table 6.1).

Lastly, we measured the API coverage of the LiFUSO libraries across all 50K compilable Java projects. API coverage refers to the percentage of API method calls in a library used at least once by a project or GitHub repository. Similarly, we performed the same measurement for 48,312 dependent GitHub repositories and compared for which dataset of projects (i.e., the 50K-C or the GitHub dependents) we obtain more API coverage. More API usage implies that a RESICO model will be trained on more simple names of API types and will, therefore, have better classification capabilities.

Figure 6.1 shows the coverage of unique API elements (i.e., public types, fields and methods) of the *Quartz* library in the GitHub-dependent repositories. As depicted, the API coverage increases as more GitHub projects are explored. However, this increase is not steady. On the contrary, once the most popular parts of an API have been used by the first few repositories, coverage of the remaining uncovered API elements increases slowly by newly-considered repositories. This observation agrees with the findings of Harrand et al. [53] who discovered that just a fraction of a library's API is used by its clients. Indeed, as seen in Figure 6.1 the repositories explored last do not result in an increase in the usage of the API. The API element coverage graphs for the remaining LiFUSO libraries are similar.<sup>4</sup>

Figure 6.2 depicts the API element coverage for the dataset RESICO was trained on (i.e., 50K-C) and the extracted data from GitHub dependents. The data in the figure shows unequivocally a higher coverage of the API elements in the GitHub dependents than in the 50K-C dataset. For libraries such as

<sup>2</sup> <https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph>

<sup>3</sup> <https://github.com/github-tooling/ghtopdep>

<sup>4</sup> The API element coverage figures other LiFUSO libraries can be found on the Appendix A.

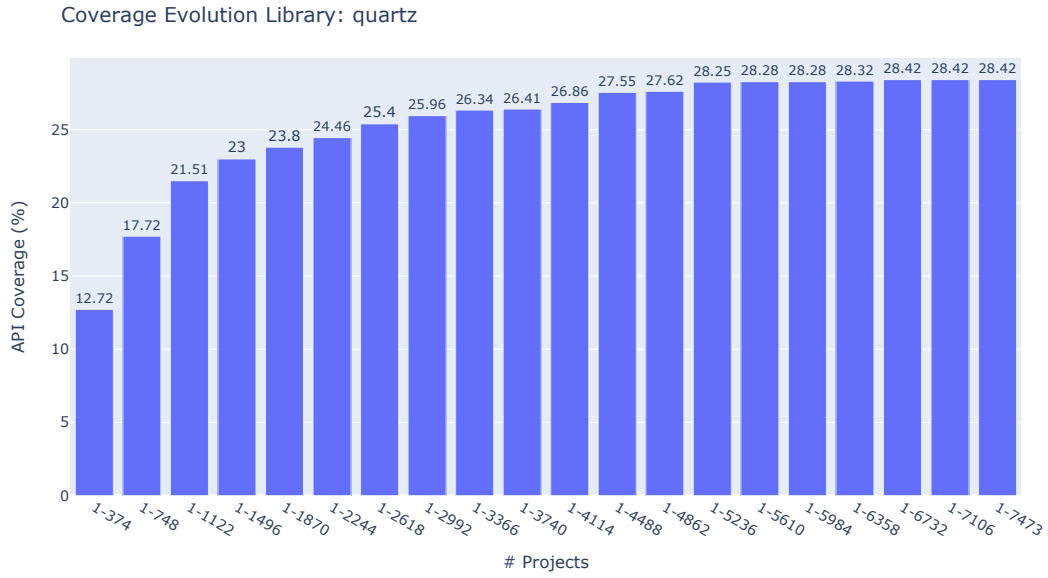


FIGURE 6.1: API element coverage of the LiFUSO library *Quartz* from its dependent GitHub repositories.

*PDFBox*, *POI-OOXML* and *Quartz*, coverage in the 50K-C dataset was only around 3%. Their coverage is much higher in their GitHub dependents. Indeed, these results confirm that one would find more usages in the repositories that use a library as a dependency than in a dataset with compilable Java projects without any particular library as a target. Based on the results of Figure 6.2, we opted for the data from GitHub-dependent repositories since they cover more library API elements.

### 6.3 Extending the LiFUSO Dataset with Additional SO Posts

This section proposes a new technique to find and consider additional posts related to the libraries under analysis. The technique complements the tag-based heuristic previously used by LiFUSO and is based on the API-type resolutions from RESICO. In addition to the RESICO model, the technique incorporates learnt rules. Rules are learnt from a SO post dataset constructed through an ambiguity analysis and manual inspection of SO post characteristics. The trained rules take the form of if-then clauses such as: if `#classes > X` then `library_post` else `not_library_post`, where `X` is also learnt.

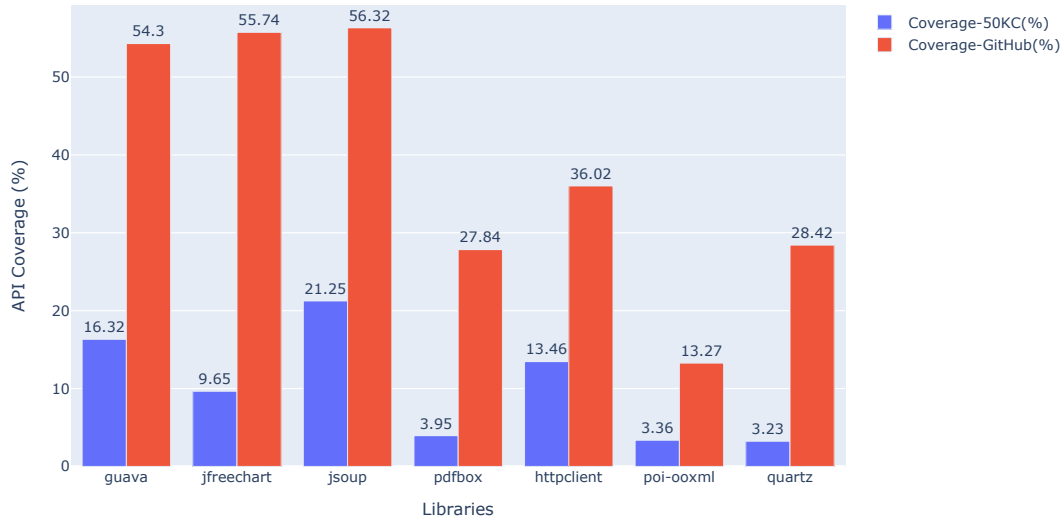


FIGURE 6.2: Comparison of API element coverage for each LiFUSO library between the 50K-C and GitHub-dependent datasets.

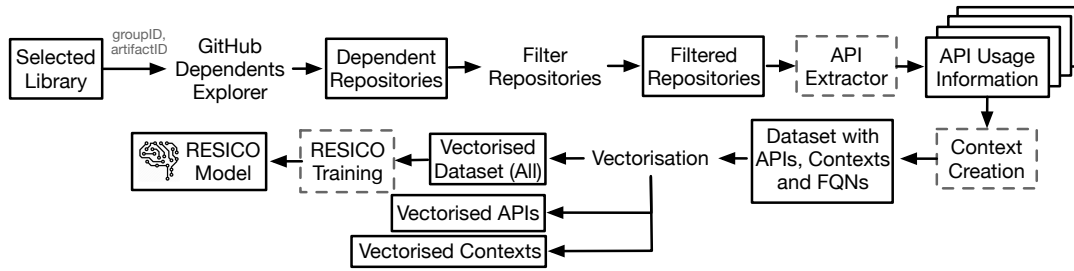


FIGURE 6.3: Approach to recognising posts related to a library - Part I.

### 6.3.1 Part I: Extraction of GitHub Data and Re-Training of RESICO

#### Repository Mining and Filtering

Figure 6.3 presents the first part of the designed approach that goes beyond the tags heuristic to recognise SO posts related to a library of interest. The new approach starts similarly to LiFUSO (cf. Figure 5.5) with the selected library from which features are to be extracted. This library is described by its `groupId` and `artifactId`. The next step extracts the GitHub-dependent repositories from the GitHub location of the library. Section 6.2 previously detailed how such an extraction is performed and the convenience of adopting this source instead of the former 50K-C dataset. We first extract all dependent repositories as a list of GitHub repository links to explore. These repositories, however, might contain forks, which may change only a small part of the original repository they were forked from. Therefore, the approach removes forks, thus only retaining unique repositories.

```
1 mvn dependency:copy-dependencies -DoutputDirectory=JARS_FOLDER/
```

LISTING 8: Command to obtain all library dependencies from a Java repository into a specific folder.

Filtered repositories form the input for the next step about extraction of API usages. Several steps are involved in the API extraction, which have been detailed in Figure 6.4.

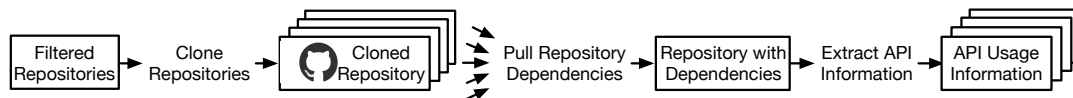


FIGURE 6.4: The extraction of API usages from GitHub repositories.

The first step is to clone the cleaned repositories from GitHub. For each locally cloned repository, the Maven build tool `mvn` gets the library dependencies declared in the `pom.xml` file. To obtain all dependencies in the form of JARs we rely on the command depicted in Listing 8.

As the code in Listing 8 indicates, all library dependencies are copied to the specified folder `JARS_FOLDER`. This folder is used in the next step to extract API usages and their corresponding FQNs. It is relevant to mention that the command in Listing 8 does not succeed for all repositories. Failures in the execution of the command might be caused due to several reasons: local libraries not being included in the project, errors in the compilation of the project, etc. Regardless of the execution state, the repository is explored with all possible JAR files the command could download. The next step is to obtain API usage information from the cloned GitHub repositories about the library considered by LiFUSO.

### API Extraction and Context Creation

The API extraction information step is based on the Eclipse JDT library previously used in Chapter 4 to create a dataset from the 50K-C corpus. Eclipse JDT requires, among other things, the location of the necessary JARs to extract complete information about all elements in the analysed projects. If not all required JARs are provided, only the information the Eclipse JDT can extract can be further analysed. Here, the `JARS_FOLDER` directory in Listing 8 is used to inform the Eclipse JDT processing. The extracted API information from the GitHub-dependent repositories includes the fields described in Table 6.1.

Each saved record is precise enough in case a future re-visit to the repository is required. The gathered information is employed in subsequent steps to form contexts surrounding the API usage.

Field Name	Description
Project Name	The name of the GitHub-dependent repository.
Commit Hash Number	The specific commit hash number in case the information is accessed at a different commit in the future.
Date of Access	The date when the repository was accessed.
File Location	The exact path of the Java file in the project being analysed.
Method Name	The name of the method in the GitHub-dependent project being analysed.
Method Start Line	The line number in the file where the method in the GitHub-dependent project starts.
Method End Line	The line number in the file where the method in the GitHub-dependent project ends.
API Usage	The code fragment where the API is used, either for a method call or a variable declaration.
FQN	The fully qualified name of the previous API usage.
API Line Number	The line number in the file where is the API usage.
Type	The type of API usage, either a variable declaration or a method invocation.

TABLE 6.1: Extracted fields from the GitHub-dependent repositories and their descriptions.

Once the information is collected from all GitHub-dependent repositories, the next step in the pipeline creates contexts around API usage. Recall that our RESICO approach requires surrounding contexts at the method level to learn the Word2Vec vector representations. The contexts' creation is also depicted as a composite step in Figure 6.3. The details about this process can be observed in Figure 6.5.



FIGURE 6.5: The process to create contexts from the previously extracted API usage information.

The first step is discarding usages of API types with FQNs that might conflict with those of the LiFUSO libraries. Such cases of FQNs might be found in repositories that have adopted the *clone-and-own* strategy [41] and include a variant of the original library into their code. According to Dubinsky et al. [41], a common reason for clone-and-own includes: “the freedom to change the original library code”. We check whether the API types are defined in the binary JAR file of the library or defined, and possibly modified, in the source code of the repository under analysis. Usages of the latter API types are discarded.



The creation of contexts follows the above cleaning of the collected data. This step first groups all API usages extracted in the same method since RESICO's scope is at that level of granularity. Each group is then analysed by considering usages of the library's API as input and the other API usages in the method as context. This second step is done similarly as described in Section 5.5. As a result of the processing, a dataset is constructed where the columns are the API usage, the context of this usage, and the FQN of the used API element. The collected dataset contains all the necessary information for the following vectorisation step.

### Vectorisation and Training

The gathered API usage information is vectorised similarly to RESICO (cf. Section 4.3.3). In other words, we used Word2Vec to transform textual information into vector representations. As contexts are composed of multiple words, we used the averaged vector of all vectors in a context. This step vectorises API usages and contexts to later average them into a single vector per record with an associated numbered FQN.<sup>5</sup>

The final step in the first part of the new approach are related to re-training the RESICO classification model. Training a new model also involves more than one internal step as described in Figure 6.6.

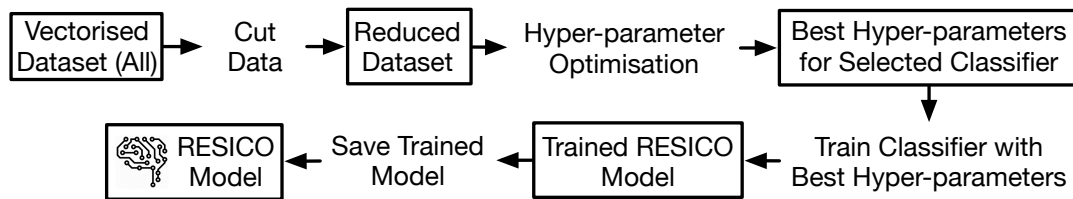


FIGURE 6.6: Steps to (re)train the RESICO classification with the previously GitHub-dependent dataset.

A pre-processing of the vectorised data is first performed in which the number of occurrences per FQN is analysed. The dataset must be filtered as some instances might occur only once or twice. The cutting threshold to exclude unpopular instances should not be as high as in the RESICO evaluation (cf. Section 4.4, where SO was used) since there is no other approach with established thresholds to base oneself on (i.e., Section 4.4 borrowed the threshold from COSTER [106]). The higher the threshold, the more data will be excluded from the training process. However, we should also consider that API usages with FQNs with low occurrences (e.g., less than 10) might be problematic in a 10-fold cross-validation approach. Therefore, we opted for putting the threshold to 10 and avoid future problems in the pipeline.

The reduced dataset is used in the next step to learn the best hyperparameters for a KNN classification algorithm. KNN was selected because it was deemed the best ML classifier in the RESICO evaluation in Section 4.4. Since the dataset changed from the 50K-C to GitHub dependents, hyperparameters

<sup>5</sup> Same as in Section 4.3.3

must be re-learned. As before, we used the Optuna optimisation framework to select the best parameter combination for the selected classifier. The best combination is used to train the classifier and finally save a local copy of the resulting model. The trained model is further employed to classify the FQN of an API usage given its surrounding context. All trained models, including the new RESICO model and the API and contexts vectors, will be used in the next part of the pipeline. This part is responsible for the library usage determination and the relatedness rule learning.

### 6.3.2 Part II: Library Usage Determination and Relatedness Rule Learning

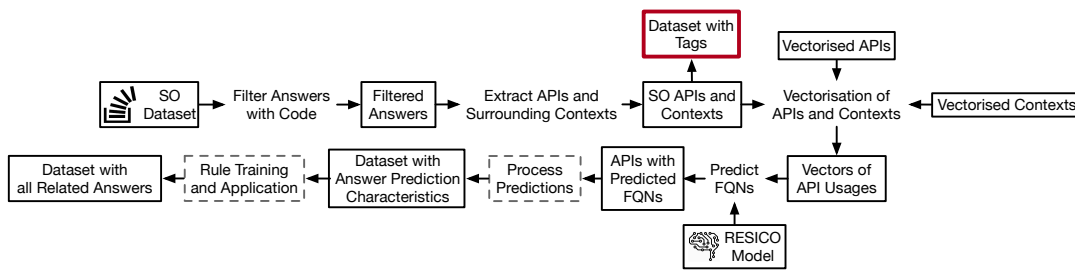


FIGURE 6.7: Approach to classify posts related to a library - Part II.

#### SO Extraction, Vectorisation and Model Classification

The second part of the approach to extract related information from Stack Overflow is depicted in Figure 6.7. The input to this second part is the SO dataset with all Java posts and answers. A necessary step to follow is to consider answers only containing some form of code since we focus on the API usage of a library. To filter answers without code fragments, we use the same approach as in LiFUSO (cf. Section 5.5), which considers only those with the HTML tags `<pre>` and `<code>`. If an answer does not contain such tags, it is considered codeless and is therefore discarded.

Filtered answers with code are processed by our island parser which extracts API usages and their contexts. Recall that many of the code snippets in this step are incomplete and syntactically incorrect, thus, the usage of a regular parser such as the Eclipse JDT is unfeasible. Our island parser, on the other hand, is flexible enough to recognise variable declarations and method invocations in otherwise syntactically incorrect snippets. The outcome of the extraction step is the referenced (i.e., by the answer id) API usages and their corresponding contexts. A local copy of the SO answers having the name of the library in their tag list is stored for later usage. This copy is highlighted in red in Figure 6.7.

Extracted API usages and their contexts from all SO Java posts data need to be vectorised to further serve as input for the RESICO model. The vectorisation step employs previously trained Word2Vec vectorised API usage and

contexts from the GitHub-dependent repositories. Each API usage and each word in its context are mapped to a Word2Vec vector in the stored vector files. As with RESICO, if an API usage is not in the trained data, the analysis is aborted for such a record. However, if a context word was not considered in the training, the analysis continues with the rest of contextual words. A surrounding context is expected to be more variable than the API usage. As before, all successfully mapped context vectors are averaged to calculate the mean between such averaged context vector and the API usage vector resulting in a single vector per API usage. The computed vector encapsulates the API usage and context in a suitable form to be the input of the formerly trained RESICO model.

The previously described transformation results in vectors for all extracted API usages and their contexts. The references to the SO answer and post IDs are also kept for future inspection. In addition, the code from the snippets from where API usages and contexts are extracted is also conserved for an upcoming analysis. This analysis employs the trained RESICO model to classify FQNs by providing it with the API usage vectors as input. As one can imagine, for every vector fed as input to the model, there is a corresponding outcome. The model responds with the numbered FQN assigned to the most similar vector in its training dataset. For many cases, however, the outcome does not match the simple name of the used API element in the code snippet. Therefore, we check that the FQN classified by the model ends with the simple name of the API. If this matches, the classification is likely to be correct. Otherwise, representing a failure from the model, we treat the record as unclassified.

### Processing RESICO's Classifications

Only records with a classification that satisfies our sanity check proceed to the next. Although records have been classified at this point in the pipeline, there may still be misclassifications by the model. To detect such cases and to improve RESICO's classifications, we gathered the characteristics of SO answers related to ambiguous cases. A box with dashed grey lines in Figure 6.7 indicates that the processing classification step includes an ambiguity analysis depicted in Figure 6.8.

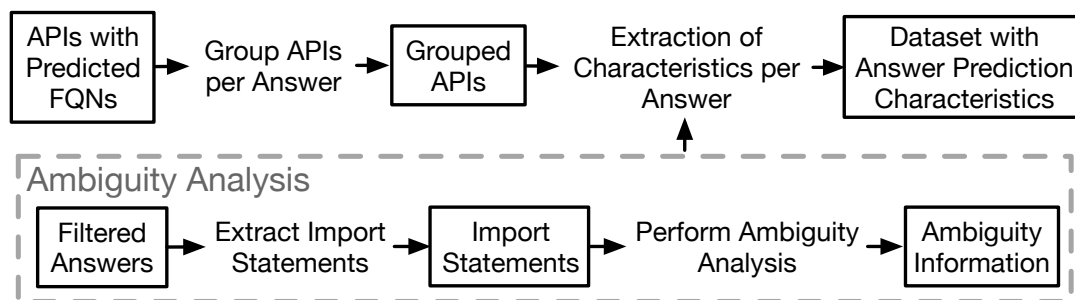


FIGURE 6.8: Steps to extract characteristics from answers with classifications.

Current classifications are based on a single API usage and its context in an SO answer. However, the purpose of our new approach is to detect whether answers are related to a library of interest. Therefore, the analysis continues by grouping all API usages by their SO answer in the dataset. The grouping is based on the AnswerIDs each API usage record has. Once all API usages have been grouped by the answers they are part of, the ambiguity analysis is conducted.

The FQNs classified by RESICO can be erroneous due to ambiguous simple names. One disadvantage of the GitHub dependents dataset is that the focus has been placed on a handful of libraries and their usages, thereby restricting in this way the vocabulary for the Word2Vec models. A more limited vocabulary gives rise to more ambiguous cases among the FQNs since just a few libraries are covered. In Section 4.4.6, we performed an ambiguity analysis of the misclassifications by RESICO on the external datasets considered for the evaluation. Similarly, we investigated the ambiguity of simple names in SO Java posts as shown in Figure 6.8.

The analysis comprises multiple steps that start with extracting all import statements from code snippets. These snippets are used in the first steps of the pipeline in Figure 6.7. Each import statement is parsed, and its simple name (if it exists) is extracted for comparison purposes. Import statements that include all elements of an API (i.e., ending with the star (\*) character) are discarded since it is impossible to extract simple names from them. After the extraction, all simple names are compared by computing the unique FQNs from where they were extracted. A JSON file, where the keys are simple names and the values are arrays of FQNs for the name in the reference, is locally stored for later consultation. These characteristics will improve the classifications by RESICO and are described in Table 6.2.

Characteristic	Description
AnswerID	The ID of the answer in the analysis.
Classes	The number of API classes in an answer.
Methods	The number of API methods in an answer.
Ambiguity	The number of API classes in an answer that are ambiguous simple names.
Imports	The import statements in the analysed answer.
Classified Library	The library classified by the re-trained RESICO model.
Confirmed Library	Whether the library in the analysis is confirmed to be used in the answer.

TABLE 6.2: Group of answer characteristics extracted to improve RESICO's classifications.

The information extracted from each SO answer is *AnswerID*, *Classes*, *Methods* and *Import* statements. The *Ambiguity* of classes is computed by checking the existence of their simple names in the previously extracted JSON file. A classification of the RESICO model is also kept in the dataset. The last column in the dataset indicates whether it has been confirmed that the classified library

is used in the answer. Such a confirmation is only possible at this stage of the pipeline by checking whether the classified library is part of the tag list in the post where the answer appears. For cases where it is not possible to confirm a usage, an undetermined mark (e.g., a dash (-)) is used instead.

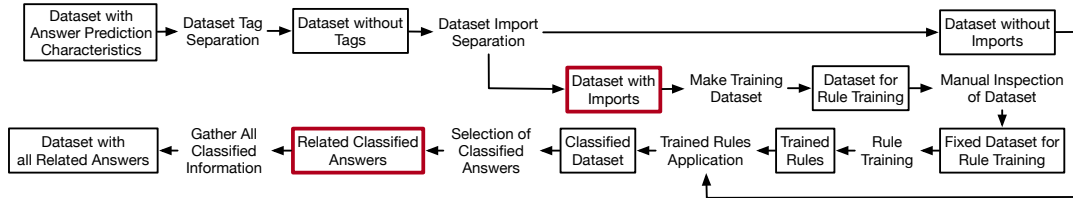


FIGURE 6.9: Steps to train and apply a rule-based model to improve RESICO classifications. Highlighted in red are the data elements related to the usage of LiFUSO libraries.

### Dataset Creation for Rule Training

Once the dataset with the described characteristics per answer has been extracted, the approach starts by processing it to assess the RESICO classifications. The steps in Figure 6.9 provide details about the final stages of the approach. The initial steps in the figure focus on separating the collected dataset into sub-datasets. First, instances in the dataset with a library usage confirmation are separated. The confirmation, as previously described, is determined by the library name in the tag list and placed in the *Library Used* column of the dataset. Cases where two or more tags correspond to LiFUSO libraries are discarded as our goal is to create a library-isolated dataset of SO posts. Confirmed instances were previously stored in a separate dataset (i.e., *Dataset with Tags* highlighted in red in Figure 6.7).

The remaining instances, i.e., those not confirmed to be using library, go through another split step. This time the separation is based on the presence of import statements since they represent another form of library usage confirmation. Therefore, the second split of the dataset results in one dataset with import statements, and another one without. The dataset with no import statements will remain untouched until a future step in the pipeline. For now, we focus on the dataset containing imports.

The next step confirms whether the library classified by RESICO is among instances with import statements. This can be quickly done by checking if any import statement in the answer starts with the import pattern of the classified library. For example, the *Guava* library has `com.google.common` as an import package structure. Instances with a confirmed usage according to the check mentioned above are labelled as *used* in the *Confirmed Library* column, otherwise, the label indicates that there is *no usage*. Confirmed classified instances are also stored in a separate dataset (e.g., *Dataset with Imports* highlighted in red), which will be joined with the previously saved tag-based dataset in the final step of the approach.

The classified dataset with import statements is useful for classifying the remaining instances without imports. Indeed, this dataset is valuable as we can automatically determine whether instances are classified correctly or incorrectly by the RESICO model. An incorrect classification is not possible in the previous tag-based dataset since all instances comply with the tag policy. The presence of binary labels (i.e., correct and incorrect classifications), in addition to already computed attributes, makes the task of determining the usability of a library suitable for a learning-based technique. Therefore, a logical following step is the creation of a training dataset for a new model.

We selected three attributes: *Classes*, *Methods* and *Ambiguity* to classify the target *Confirmed Library*. As we wanted the dataset to be balanced according to its two classes, we selected all instances of the minority class and a random sample of the majority class with the same number of instances as the former. Once the training dataset is formed, we shuffled all its instances to avoid bias in the learning process. Our decision process for classifying an instance as likely to be using a library is as follows:

- If possible, it has a low number of ambiguous API class references. The lower the ambiguity, the simpler it is to identify usages belonging to a library.
- If possible, uses more than one method of the library to reinforce the usage of classes.
- If possible, it uses a high number of classes of the library. The more classes are used, the more likely a classified library is employed.

The more an instance adheres to these conditions, the higher the likelihood of a veridical classification. We inspected all instances and re-label those that do not satisfy these conditions.

The manual work is not performed per instance. Since we deal with vectors of numbers corresponding to attributes, we can rely on the cosine similarity<sup>6</sup>, and assign the same label to similar instances in the dataset. We implemented a simple yet effective labelling mechanism that checks bulks of similar vectors for their assigned label. The algorithm allowed us to label multiple instances at the same time. We also made it flexible enough to only label certain elements instead of a group. The re-labelled dataset represents the input for the next step in which we train an automated classifier.

### Classifier Training and Gathering of a New LiFUSO Input Data

Provided with the re-labelled dataset, any learning technique can perform optimally. However, because of several reasons, we selected a technique that produces rules from the instances. First, generated rules are based on the distribution of the attribute values on the dataset, i.e., the outcome is if-then

<sup>6</sup> <https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.cosine.html>



clauses. Second, since these techniques are based on the values of the attributes, we can interpret learnt rules and see whether they agree with our previously defined goals.

To proceed with the training, we selected a decision tree classifier named J48 that generates rules from a dataset. The algorithm is easily accessible from the Weka tool [145], from where it is also possible to configure the evaluation strategy. Weka additionally saves all output of the training process including learnt rules and scores achieved in the evaluation method. The resulting rules are transformed into code as executable if-then clauses.

Rules are applied to all instances that do not contain any import statement or tags that may hint the usage of a library. Once the rules have been applied, we select instances that are deemed to use a library. Those instances carry a confirmation by the RESICO model and by the learnt rules. The last step in the pipeline gathers all usages of the library under analysis by concatenating all highlighted rectangles in Figure 6.7 and Figure 6.9 into a single dataset. The resulting dataset contains all API usages related to a library that can be extracted from the SO data. Figure 6.10 depicts its integration with LiFUSO (cf. Figure 5.5).



FIGURE 6.10: Integration of the new approaches into the LiFUSO pipeline.

As observed, the approaches described in this section are highlighted in dashed red lines. The input of the *Library Usage Post Assessment (I)* (cf. Section 6.3.1) is the same as the input to LiFUSO. The output of the *Library Usage Post Assessment (II)* (cf. Section 6.3.2) seamlessly provides the necessary information to continue with the remaining steps in LiFUSO. The steps described in this section therefore integrate the RESICO (cf. Chapter 4) and LiFUSO (cf. Chapter 5) approaches into one coherent approach.

## 6.4 Evaluation

In this section, we describe the design and the results of the evaluation conducted on the incorporation of RESICO into LiFUSO. The evaluation assesses the performance of the approach and the impact of a more extensive dataset on LiFUSO.

Our evaluation aims to answer the following research questions:

- RQ<sub>1</sub>** How well does the rule-based classifier perform on the manually labelled dataset?
- RQ<sub>2</sub>** What is the impact of the new SO answer dataset on the features uncovered by LiFUSO?



### 6.4.1 Datasets Collection

In this subsection, we report on the datasets collected through the different steps of our approach. **We describe the source and the characteristics of the dataset, i.e., the number of records, their attributes, etc.**

The dependent GitHub repositories dataset (described in Section 6.2) consists of GitHub repositories that include one of the LiFUSO libraries in their build files. As previously explained, we obtain the dependents of a library using the `ghtopdep` tool. The number of dependent repositories for each LiFUSO library is depicted in Table 6.3.

	Guava	HttpClient	JFreeChart	JSoup	PDFBox	POI	Quartz
Dependents	14,147	75,069	8,416	83,851	14,626	4,289	53,542
Cleaned dependents	12,569	58,069	7,358	64,422	12,899	3,910	36,349

TABLE 6.3: Dependent GitHub repositories for each LiFUSO library before and after the cleaning.

Table 6.3 also contains a second row about the number of client projects remaining after the cleaning described in Section 6.2. Each library has fewer unique dependent repositories afterwards, which is mainly due to the existence of repository forks. API usage information is extracted from the remaining repositories.

The reader might notice that the coverage figures in Figure 6.1 and in Appendix A do not add up the number of repositories reported in Table 6.3. For some libraries where the number of repositories is considerably large (e.g., *JSoup* and *HttpClient*), we aborted the analysis early. For many large repositories, the API usage extraction is computationally expensive since it includes the download of all dependencies, a compilation of the project, and the actual extraction. When analysing more projects resulted in less than 1% additional API usage being discovered, we halted the extraction and moved forward with the information collected thus far.

Once the API extractor has processed the cleaned repositories, we proceed with a filtering step. As described in Section 6.3.1, we discard each API usage record of which the FQN occurs fewer than 10 times to facilitate later cross-validation. The number of discarded records is 1,713 out of 3,152,837 (only 0.05%), representing a minor reduction of our dataset.

Next, we vectorise the API usages and their contexts which are the training data for RESICO. The size of the vocabulary for the collected API usages is 14,500. The size of the vocabulary for the contexts is 63,372 words. Once RESICO is trained, these API and context vectors are used to classify FQNs from incomplete SO code snippets.

Therefore, another fundamental data source is the SO Java dataset from where LiFUSO features can be extracted. The initial number of Java answers from SO is 3,017,471; all posterior datasets are derived from this set. From that initial group, 2,198,731 answers contain at least one code fragment. The API

usage extractor described in the approach can successfully filter 1,591,763 answers with at least one surrounding context. Cases where this is not possible, include, for example, a single word of code surrounded by natural language text.

The vectorisation of contexts and API usage (cf. Section 6.3.1) from SO code snippets depends on whether they are present in the RESICO training dataset. This means that words which are not in the training data will not have a learnt vector to be mapped, and therefore, no transformation will occur. We excluded answers where the mapping to API elements is unsuccessful, or no context word is transformed. The result of the transformation and filtering process is 75,365 SO answers.

Once the API usages and their contexts have been vectorised for each SO answer, a classification is made. The trained RESICO model takes as input a vector and classifies the most likely FQN for it. This vector is the average between the API usage vector and the resulting vector from averaging all context vectors. Each answer can contain more than one API usage to resolve the FQN of. Therefore, we might have multiple API classifications, each keeping their AnswerID reference. As previously described in Section 6.3.2, a sanity check verifies the correctness of each classification with the simple name of the API usage. We obtained 100,363 classified API elements within 45,787 answers for the LiFUSO libraries.

The ambiguity analysis described in Section 6.3.2 is based on the import statements in the 45,787 answers. There are 26,730 unique import statements. The number of ambiguous simple names is 213.

As shown in Figure 6.9, the current dataset of 45,787 answers is divided and each part analysed. The first portion consists of 10,495 answers with the name of the classified library in their tag list. These are many of the answers considered in Chapter 5 to uncover library features.<sup>7</sup> The other 35,292 answers are further divided according to whether they have import statements. This is the case for 4,829 answers. The 30,463 answers without import statement are left to be checked by the learnt rules.

RESICO classifies the FQNs from the API references in an SO answer. As such classifications are limited to the libraries considered by LiFUSO, only FQNs from them are recommended. All classifications in an answer are, in most cases, about one of the LiFUSO libraries. Therefore, we consider the library with more classifications as the most relevant library in the SO answer. The few cases where we obtain an equal number of classifications for more than one library are manually inspected since the process cannot automatically determine the most important one.

Confirmed answers containing a match between classification and imports are 1,068. The remaining 3,761 answers have been misclassified by the RESICO model. As we can see, relying only on the classifications by the RESICO

<sup>7</sup> The increment of 532 (i.e., Table 5.4 previously reported 9,963 instances) may be because we consider [a more recent SO snapshot](#) than in Chapter 5.

model leads to false positives. Therefore, we selected all positively confirmed answers and a random sample of the negatively verified records to train a rule-based model. We constructed a binary balanced dataset of 2,136 instances: 1,068 are labelled with one (i.e., confirmed answers with a match) and the other half with zero (i.e., confirmed answers with no match).

### 6.4.2 RQ1. How well does the rule-based classifier perform on the manually labelled dataset?

#### Design

This research question investigates the performance of the rule-based classifier on the manually-labelled dataset of SO answer characteristics.

To achieve this goal, we trained and evaluated a J48 decision tree using 10-fold cross-validation as previously considered for RESICO (cf. Section 4.4.3). We used stratified cross-fold validation: each fold contains approximately the same distribution of labels to classify. The implementation of stratified cross-validation and of the J48 decision tree itself are provided by Weka.

Once the model's classifications have been gathered for each fold, we compute Precision, Recall and F1-Score. We use the standard definition of these metrics. In this binary setting, precision is defined as the number of true positives out of the number of true positives and false positives (cf. Equation (6.1)). The recall is the number of true positives out of the true positives and false negatives (cf. Equation (6.2)). The harmonic mean of Precision and Recall (i.e., F1-Score) follows the definition by Equation (4.3).

$$Precision = \frac{TP}{TP + FP} \quad (6.1)$$

$$Recall = \frac{TP}{TP + FN} \quad (6.2)$$

In Equation (6.1) and Equation (6.2) True Positives (TP) refer to those SO answers labelled as using one of the LiFUSO-related libraries and classified as such by the model. False Negatives (FN) are SO answers labelled as using a LiFUSO library but falsely classified as not so. False Positives (FP) are SO answers that do not use a LiFUSO library, yet were classified as doing so by the rule-based model. Finally, we average the metrics for all folds and report them for each binary class.

#### Results

We report on the results obtained by the rule-based trained classifier.

Listing 9 shows the decision tree produced by the J48 classifier trained on the manually-validated dataset. The decision tree is represented as if-then clauses by the Weka framework. This representation has advantages such as

```
1 if ambiguity == 0:
2     if methods == 0:
3         return 'N'
4     return 'Y'
5 else:
6     if classes <= 4:
7         if methods == 0:
8             return 'N'
9         else:
10            if ambiguity <= 1:
11                if classes <= 2:
12                    return 'N'
13                    return 'Y'
14                return 'N'
15        else:
16            if ambiguity <= 2:
17                return 'Y'
18            else:
19                if classes <= 8:
20                    if ambiguity <= 3:
21                        if classes <= 6:
22                            return 'N'
23                            return 'Y'
24                        return 'N'
25                else:
26                    if ambiguity <= 7:
27                        if ambiguity <= 5:
28                            return 'Y'
29                        else:
30                            if classes <= 13:
31                                return 'N'
32                                return 'Y'
33                    else:
34                        if classes <= 18:
35                            return 'N'
36                        else:
37                            if methods <= 19:
38                                return 'Y'
39                                return 'N'
```

LISTING 9: Trained rules of the J48 decision tree based on the manually labelled dataset.

being transparent about the learnt knowledge and that it can be translated to any programming language.

Some insights can be gained from the rules in Listing 9. For example, the presence of ambiguous class references is a decisive factor in the classification since SO answers without such ambiguity are likely to be using the library (cf. Lines 1-4). The combination of ambiguity, a low number of class references and no method calls indicate that the library might not be used in the answer (cf. Lines 5-8). Finally, the order of the generated rules is also relevant as many instances may satisfy the initial but not the last checks.

Class	Precision	Recall	F1-Score	Pred.		
				Act.	N	Y
N	95.9	99.8	97.8	N	1,630	4
Y	99.8	86.3	92.2	Y	69	433

TABLE 6.4: Performance metrics (Left) and confusion matrix (Right) of the trained rules on the manually labelled dataset.

As depicted in Table 6.4, the performance metrics of the trained rules are outstanding. Precision is excellent for both classes in the dataset, surpassing 95% effectiveness in each class. The recall metric is still exceptional for the ‘N’ class, and relatively good for ‘Y’ class with a score above 85%.

Table 6.4 also includes a confusion matrix on the right. As expected from the previous metrics, most instances are located on the matrix’s main diagonal, indicating correct classifications. Another aspect of the confusion matrix is also noticeable: class imbalance is present in the manually-labelled dataset. We therefore, re-trained and measured the performance of the rule-based model on a new balanced version of the manual dataset. However, the metrics report an F1-Score of 92.9% and 93.5% for classes ‘Y’ and ‘N’, respectively, which when averaged (93.2%) are worse than the averaged metrics for the imbalanced dataset (95%). This observation agrees with previous research on interpretable rule-based classifiers and class imbalance. For example, Tantithamthavorn et al. [121] and Gao et al. [43] have explored the impact of balancing techniques on the performance of interpretable classifiers. They found that re-sampling might not be beneficial for some classifiers, especially rule-based ones for which re-balancing may change the distribution of the training data.

The interpretable rules produced by the J48 decision tree classifier to determine whether an SO answer is likely to use a LiFUSO-supported library, have an excellent performance according to the precision, recall and F1 metrics. A re-balance of the manually-labelled dataset does not improve the performance metrics, in accordance with previous research in the domain.

### 6.4.3 Rule-based Model Application on the Manually-labelled Dataset

The manually-labelled dataset is used to train and evaluate the rule-based model as described in the previous section. The obtained model will classify whether the unclassified 30,463 SO answers remaining in the dataset are related to the LiFUSO libraries. The number of SO answers positively classified as related by the rule model is 2,702, whilst the other 27,761 SO answers are classified as likely unrelated. The final dataset of LiFUSO-related SO answers comprises the previous answers that have the name of a library in their tag list (10,495), the answers that import library types through import statements (1,068) and answers of which the library usage has been confirmed by the rule-based model (2,702) and counts 14,265 SO answers in total.

	Guava	HttpClient	JFreeChart	JSoup	PDFBox	POI	Quartz
Old Tagged Answers	1,522	628	882	3,465	945	2,010	511
<b>New Tagged Answers</b>	1,473	733	869	3,552	1,034	2,317	517
Untagged library-using Answers	1,373	1,564	49	189	101	437	57
<b>Total New Answers</b>	2,846	2,297	918	3,741	1,135	2,754	574

TABLE 6.5: Previous and newly collected SO answers that use a LiFUSO library.

Table 6.5 details the composition of the newly collected dataset. The number of answers deemed to be using a LiFUSO-related library has increased for all libraries. The growth is the highest for the *Guava* and *HttpClient* libraries, whereas the *JFreeChart* library saw the smallest growth. The new dataset of SO answers will serve as input to the LiFUSO approach in the next research question.

### 6.4.4 RQ2. What is the impact of the new SO answer dataset on the features uncovered by LiFUSO?

#### Design

This research question analyses the impact of the newly constructed dataset of SO answers on the features uncovered by LiFUSO.

To this end, we compare LiFUSO's outcome on the new dataset to its outcome on the dataset with only tagged answers. However, before comparison, we have split the dataset into seven smaller ones each dedicated to a single LiFUSO library. As explained in Section 5.5 and also depicted in Figure 6.10, information about a single library is required each time features are computed. In other words, a dataset containing mixed posts about multiple libraries is not desired. A few answers use two or more LiFUSO libraries (e.g., *JFreeChart* and *PDFBox*) at the same time. We manually inspected each of those answers to select the most appropriate library.

Once each library has its own two datasets of tagged and tagged + untagged found SO answers, we applied LiFUSO to each. LiFUSO applies its island parser to each answer to extract API usages (cf. Section 5.5.2), and then

constructs a similarity matrix with all the collected usages. The similarity matrix reflects closeness between all usages using the Jaccard metric (cf. Section 5.5.3). The matrix, along with answer information such as natural language terms and links to SO, is passed to the LiFUSO clustering method (cf. Section 5.5.4). The LiFUSO clustering method is executed, and a book of feature descriptions is obtained as *HTML* files. These description files contain all the information about the features of the LiFUSO libraries. For example, they are labelled with a number (e.g., Feature 10), described in natural language terms (e.g., “*save image*”), and their usage is illustrated using example code (cf. Listing 6).

The generated documentation files enable us to automatically compare old and new features. We use three comparison methods on the features uncovered for each library. The first comparison approach counts how many features are obtained for each dataset. The second comparison measures the impact on the obtained features by calculating the average of API calls per code snippets comprising each feature. This comparison investigates how different in size, on average, the code snippets that comprise the features are. Lastly, the third comparison investigates the number of API calls on average for all code snippets of all features. This last comparison investigates the changes in method call usages and their effect on the features.

## Results

Library	#F	#NF	$\Delta F$	AA	NAA	$\Delta AA$	AM	NAM	$\Delta AM$
Guava	114	174	+60	1.83 ( $\pm 4.79$ )	1.94 ( $\pm 3.13$ )	-0.11	1.79 ( $\pm 4.28$ )	1.80 ( $\pm 3.24$ )	-0.01
HttpClient	41	80	+39	3.78 ( $\pm 9.28$ )	6.01 ( $\pm 11.09$ )	-2.23	4.61 ( $\pm 10.57$ )	6.33 ( $\pm 10.46$ )	-1.72
JFreeChart	85	85	0	1.66 ( $\pm 1.35$ )	1.66 ( $\pm 1.35$ )	0.00	1.38 ( $\pm 0.87$ )	1.38 ( $\pm 0.87$ )	0.00
JSoup	91	92	+11	7.25 ( $\pm 19.94$ )	6.75 ( $\pm 17.28$ )	+0.50	4.57 ( $\pm 7.64$ )	4.35 ( $\pm 6.86$ )	+0.25
PDFBox	51	57	+6	2.45 ( $\pm 2.67$ )	2.28 ( $\pm 2.33$ )	+0.17	2.02 ( $\pm 1.83$ )	1.89 ( $\pm 1.94$ )	-0.13
POI	99	99	0	2.86 ( $\pm 5.49$ )	4.25 ( $\pm 13.70$ )	-1.39	2.26 ( $\pm 2.91$ )	3.51 ( $\pm 9.90$ )	-1.25
Quartz	34	32	-2	2.94 ( $\pm 5.66$ )	3.22 ( $\pm 6.55$ )	-0.28	3.12 ( $\pm 4.78$ )	3.31 ( $\pm 5.06$ )	-0.19

TABLE 6.6: Comparison between old and new features.

Table 6.6 reports on the comparison of features from tagged SO answers and the newly obtained features from the combined tagged and untagged information. Column names and their descriptions are as follows:

**#F** The number of features extracted from the usages of each library in the tagged dataset.

**#NF** The number of features extracted from the combined dataset.

**$\Delta F$**  The difference between the features from the tagged and combined datasets.

**AA** The averaged number of API calls in the snippets of the tagged dataset. Between parentheses is the standard deviation of the averaged value. The same applies to the remaining columns.



**NAA** The averaged number of API calls in the snippets of the combined dataset.

**$\Delta$ AA** The delta value between NAA and AA. This number could be either positive or negative.

**AM** The averaged number of method calls in the snippets of the tagged dataset.

**NAM** The averaged number of method calls in the snippets of the combined dataset.

**$\Delta$ AM** The delta between NAA and AA. Both positive and negative values appear here.

As shown in Table 6.6 the new additions of untagged answers enable extracting more features for most libraries. This is reflected in column  $\Delta F$ , where all deltas are positive, except for the *Quartz* library with two features less and the *JFreeChart* and *Apache-POI* libraries with no new features extracted. *Guava* is the library for which the most additional features have been uncovered (i.e., 60).

On average, the size of the snippets comprising features differs for all libraries except for *JFreeChart* with no changes. The most significant differences are found in the libraries *JSoup* with an increment of +0.50 and *HttpClient* with a decrement of -2.23. This finding indicates that in the newly constructed dataset, there are on average almost two less API calls for *HttpClient* features. The result also reflects a significant decrease in API calls in the *Apache-POI* features.

The differences in the number of method calls are, in contrast, minimal (cf. column  $\Delta AM$ ). All library features report changes that do not exceed one unit of difference, except for *HttpClient* (with -1.72) and *Apache-POI* (with +0.25). *Guava* merely reports a delta of -0.01, meaning that almost the same number of methods are used in the tagged and combined datasets. This agrees with the characteristics of this library, e.g., most of its features contain few method calls.

More features were discovered in the newly constructed dataset of SO answers, with *Guava* having the highest growth and *Quartz* with a slight decrement. The size of the snippets integrating the features changed significantly, whereas, on average, the number of used methods remained similar.

## 6.5 Discussion

This section discusses the results obtained above and describes the limitations that remain in combining RESICO and LiFUSO.

### 6.5.1 Newly Discovered Features

As shown in Table 6.6, new features are discovered for all LiFUSO libraries. These new features result from newfound API usage patterns in the newly populated dataset.

```
1 // Library Guava
2 // compare char, declare range
3 CharMatcher.anyOf(...);
4 CharMatcher.inRange(...).or(...);
5 // describe graph, define graph
6 MutableGraph.putEdge(...);
7 MutableGraph.addNode(...);
8
9 // Library JSoup
10 // get sibling, select sibling
11 Element.siblingElements(...);
```

LISTING 10: Some examples of newly discovered features for the LiFUSO libraries.

The features in Listing 10 are a sample of the newly discovered features for the LiFUSO libraries. As shown in the listing, features are still generated from patterns containing either a single or multiple API calls. Some features that were already present have been improved with respect to their natural language description. In other words, more terms related to the code in the features have been incorporated into their description.

This is illustrated by Listing 11. Some features see new *verb-noun* pairs added to their description. In other cases where the frequency of terms is low, they are replaced by new and more meaningful natural language terminology. As the patterns' frequency increases, a greater number of natural language terms emerges resulting in extracted descriptions that are increasingly coherent.

### 6.5.2 Limitations

This chapter combines the RESICO and LiFUSO approaches proposed in Chapter 4 and Chapter 5, respectively. However, their combination still has some limitations. For example, some manual input is required for parts of the proposed pipeline such as the meta-data of a library (e.g., groupId and artifactId) as well as its corresponding tag name. Additionally, the GitHub repository hosting the library is required as input to the ghtopdep tool, which retrieves the dependent repositories. All this information is required to execute our pipeline.

Another limitation covers the quality of the natural language terms. Some descriptions are not in exact agreement with the code required to use a feature. This could be caused by a mediocre quality of the posts. New language

```
1 // Library JSoup
2 // extract text, parse html Added -> (extract node, parse tag)
3 Element.dataNodes(...);
4 // select parent, select child Added -> (iterate element, get element)
5 Element.parents(...);
6
7 // Library Guava
8 // use api, have method Replaced by -> (use set, get set)
9 Sets.union(...);
10
11 // Library JFreeChart
12 // set break Replaced by -> (calculate histogram, display bar)
13 HistogramDataset.addSeries(...);
14 ChartFactory.createHistogram(...);
```

LISTING 11: Examples of improved features for the LiFUSO libraries.

generation models with capabilities such as code interpretation (e.g., CodeParrot<sup>8</sup> and InCoder<sup>9</sup>) might help improve the quality of the feature descriptions.

## 6.6 Threats to Validity

The threats to validity discussed in this section are similar to those discussed in Chapter 4 and Chapter 5. Many are inherited from RESICO and LiFUSO, but some have been mitigated by their combination.

### 6.6.1 Threats to Construct Validity

The vocabulary of the RESICO model used in this chapter was limited to the usages of the seven LiFUSO-supported libraries (i.e., those used for the evaluation of LiFUSO in Chapter 5). A limited vocabulary may lead to more erroneous API type resolutions due to the presence of ambiguous simple names. We mitigated this risk by training a rule-based model on the characteristics of code snippets that assesses whether a RESICO resolution is likely using a library in a given snippet.

### 6.6.2 Threats to Internal Validity

The internal threats are very much aligned with those in Chapter 5. TF-IDF is selected for the vectorisation of text attributes, even though techniques such as Word2Vec exist. Our experiments with Word2Vec resulted in less than five clusters per library, where one cluster grouped most of the elements. Once again, we selected hierarchical clustering as a clustering algorithm based on

<sup>8</sup> <https://huggingface.co/codeparrot>

<sup>9</sup> <https://huggingface.co/facebook/incoder-6B>

its speed and effectiveness. As previously mentioned, language generation models might improve our feature descriptions. We deem this part as future work to improve the already discovered features.

### 6.6.3 Threats to External Validity

Our approach is currently not ready to be generalised to an important number of Maven libraries as it depends on the extent to which their features are used in SO answers. LiFUSO might fail to uncover features for less popular libraries. LiFUSO has the potential to be adopted in techniques exploring features of libraries from other sources apart from SO. For example, GitHub usages and the natural language elements surrounding them such as method and variable names, docstrings, or messages of the commits where they are introduced. Another alternative source of information are unit test cases and their text descriptions, which may scale to many libraries since most of them include a test suite.

## 6.7 Conclusion

This chapter proposes a combination of the RESICO and LiFUSO approaches introduced in Chapter 4 and Chapter 5, respectively. We used the API type resolution of RESICO to enlarge the dataset of SO answers used by the LiFUSO approach. This improvement started with an analysis of the original training dataset of RESICO with respect to the usage of LiFUSO libraries. We noticed that dependent GitHub repositories cover more of the libraries' API than the original training dataset of 50K compilable Java programs.

We therefore retrained RESICO on the GitHub repository that depend on the LiFUSO libraries, but noticed that the narrow focus on these libraries may limit the vocabulary known to RESICO. A limited vocabulary affects its API type resolutions in the presence of ambiguous simple names. Therefore, we trained a rule-based model to complement the RESICO's classifications. We evaluate the rule-based model on a manually-labelled dataset. The results show that learnt rules are capable of determining with a high F1-Score whether an SO answer uses the library output by RESICO. RESICO and the accompanying rule-based model are used to enlarge the input dataset of SO answers to LiFUSO.

We have also measured the impact of the newly constructed dataset on the features uncovered by LiFUSO. New features are discovered for all considered libraries. Furthermore, the natural language terms of some features are improved as a consequence of the newly gathered data.



## Chapter 7

# Conclusion and Future Work

It is common in contemporary software development to reuse features provided by third-party libraries. The number of libraries in a popular software ecosystem is so abundant that selecting the most suitable library becomes a problem for developers. This selection problem is aggravated due to the lack of tools that help developers by recommending a library with the sought-after features.

This dissertation presents research on the automatic discovering of library features. Features are defined as API usage patterns with a corresponding description in natural language. We first investigated the feasibility of several sources of information about library features. We selected Stack Overflow as our information source due to its popularity, its large community and the possibility to mine API data about popular libraries. However, many Stack Overflow answers lack completeness and syntactical correctness. We therefore proposed the RESICO approach to resolving API references in such answers to fully qualified names. A second approach called LiFUSO, takes advantage of the API usage patterns and the natural language terms within Stack Overflow answers to uncover the features provided by a library. Finally, we have shown how RESICO can help enlarge the dataset of SO answers LiFUSO operates on.

This chapter recapitulates these approaches and the contexts they were introduced in. Section 7.1 provides a summary of the dissertation by highlighting the most important conclusions of each chapter. Section 7.2 restates the contributions made by RESICO, LiFUSO, and the approach combining them. Section 7.3 discusses possible directions for future work, while Section 7.4 concludes the dissertation.

## 7.1 Summary

**Chapter 1** provided an introduction to this dissertation. We defined the concept of library features, and the problems developers face in selecting an appropriate library from a vast software ecosystem. Current approaches to library recommendation and comparison are based on straightforward metrics such as the number of stars or test coverage. Instead, we propose to

base these recommendations and comparisons on automatically uncovered representations of the features provided by a library.

**Chapter 2** explained the techniques, algorithms, and sources of information our approaches are based on. First, the chapter detailed the different sources from where knowledge about a library can be extracted. We analysed these sources considering aspects such as the availability of code snippets and natural language illustrating the API through which library features can be reused. We selected Stack Overflow as our source of information and detailed the SOTorrent dataset. The chapter then moved onto the background on the techniques that we use to process this information, including text to vector transformations such as TF-IDF and Word2Vec. Details about text classification, Phrase-Structure trees and hierarchical clustering are also discussed in the chapter.

The current state of the art on the research topics covered in this dissertation is discussed in **Chapter 3**. The chapter described previous work operating on SO information, analyses of incomplete code snippets, and machine learning embeddings proposed for software engineering tasks. The state of the art on automated approaches to library usage comprehension, recommendation and documentation is also covered. We concluded with the need for our contributions to the field.

**Chapter 4** presented RESICO, the first contribution of this dissertation. RESICO resolves the simple names of API types in incomplete Stack Overflow snippets to their fully qualified names. The absence of import statements that qualify simple names encumbers the use of code snippets by users and tools alike. RESICO leverages a dataset of library API usage within complete and correct code to learn word embeddings and the most likely fully qualified name for a simple name in a specific context. We instantiated the approach with several machine learning algorithms and evaluated the resulting machine learning models against the state-of-the-art approach COSTER. Most models outperform COSTER in all experiments we conducted. Moreover, our best classifier is faster to train.

**Chapter 5** presented LiFUSO, an automated approach to uncovering the features provided by a library from Stack Overflow answers. The approach is motivated by the need for better support for software ecosystem exploration, library selection, and comparison. As stepping stone towards LiFUSO, we presented two approaches that adopted categories and tags as high-level and coarse-grained descriptions of library features. LiFUSO itself uncovers features in the form of named patterns of API usage, which it mines from Stack Overflow answers. The evaluation reveals that the features uncovered by the approach align with those documented in tutorials and cookbooks about the considered libraries. The approach was instantiated into a tool which enabled us to demonstrate its potential.

**Chapter 6** presents a combination of RESICO and LiFUSO. Following an



analysis of the coverage of the libraries' API considered in the LiFUSO evaluation, the RESICO model from Chapter 4 was retrained on a dataset that comprises their dependent GitHub repositories. Next, the API type resolutions of the retrained model were used to determine whether a given SO answer is related to a particular LiFUSO library. However, since retraining RESICO on a more focused dataset reduced its vocabulary, its type resolutions suffer more from the presence of ambiguous names. Therefore, we complemented RESICO with a new rule-based classifier that determines the likelihood that an SO snippet uses the library predicted by RESICO. Through this combination, we are able to enlarge the dataset of Stack Overflow posts analysed by LiFUSO and thus enrich the features it uncovered.

## 7.2 Contributions

This dissertation presented three contributions: the RESICO and LiFUSO approaches and their combination.

### 7.2.1 RESICO: API Resolution for Incomplete Code Snippets

First, we identified the problem that many code snippets of Stack Overflow posts are incomplete and syntactically incorrect (Chapter 1 and Chapter 4). The lack of import statements that qualify the simple names of API types, encumbers the use of SO snippets in automated analyses and tool support. Therefore, we designed and implemented a learning-based approach capable of resolving the simple name of an API type to its likely fully qualified name for a given SO code snippet (Chapter 4). The corresponding contributions are:

- The design and implementation of the RESICO automated approach which extracts the fully qualified names of API types involved in API usage within complete project repositories, learns their surrounding contexts, and resolves simple names for API types within incomplete code snippets to their corresponding fully qualified name within a given context.
- A thorough evaluation on three datasets in which RESICO outperforms the state-of-the-art approach COSTER.
- A study into the behaviour of the approach on ambiguous cases, an investigation into the origin of faulty predictions, and an overview of the technical limitations of our prototype implementation.
- A replication package containing the RESICO implementation, the datasets used in its evaluation, and all scripts required to conduct our experiments.<sup>1</sup>

---

<sup>1</sup> <https://github.com/softwarelanguageslab/resico-paper>

### 7.2.2 LiFUSO: Uncovering Library Features from their Stack Overflow Usage

Second, we found that the current library selection process can benefit from automatically generated enumerations of the features offered by a library (Chapter 1 and Chapter 5). We proposed LiFUSO, an approach that analyses the API usage of a library within Stack Overflow posts to uncover library features in the form of named API usage patterns (Chapter 5). The corresponding contributions are:

- The design and implementation of an automated approach that collects, processes and clusters API usage from Stack Overflow code snippets into named feature descriptions.
- An extensive evaluation comparing features uncovered by the approach to the way they are described in tutorials, cookbooks, and to the way they are used within GitHub repositories.
- An instantiation of the LiFUSO approach into a tool that enables searching for features given a description in natural language and enables comparing different libraries based on their features.
- A case study in which the tool is used to compare two libraries targeting the same application domain based on their shared and unique features.
- A replication package containing the LiFUSO approach implementation and tool instantiation and the datasets used in the evaluation of the approach.<sup>2,3</sup>

### 7.2.3 Feature Uncovering on Resolved Code Snippets

Third, we combined the RESICO and LiFUSO approaches described in Chapter 4 and Chapter 5 into a single approach that leverages the former's capabilities for resolving API types in incomplete code snippets with the latter's ability to enumerate the features provided by a library (Chapter 6). Specifically, we noticed that RESICO can be used to overcome the limitation of LiFUSO that requires Stack Overflow posts to be tagged with the name of the library under analysis. The corresponding contributions are:

- The design and implementation of an approach that combines RESICO and LiFUSO.
- An analysis of extent to which the original dataset on which RESICO has been trained covers the APIs of the libraries for which LiFUSO has been evaluated.
- A new dataset for training RESICO derived from the GitHub repositories that depend on a LiFUSO library.

<sup>2</sup> <https://github.com/softwarelanguageslab/lifuso>

<sup>3</sup> <https://github.com/cvelazquezr/lifuso-ui>

- A rule-based classifier which supports RESICO API-type resolutions based on code snippet characteristics.
- An in-depth evaluation on the effectiveness of the rule-based classifier, and of the impact on the quality of the uncovered library features of using RESICO and the rule-based classifier to enlarge the dataset of Stack Overflow snippets considered by LiFUSO.
- A replication package containing the combined approach implementation and the resulted datasets from our research.<sup>4</sup>

#### 7.2.4 Advantages and Limitations of Data-Driven Approaches

All our previously listed contributions are data-driven. This means that they analyse a considerably large number of instances in a dataset to learn patterns. Our API type resolution approach RESICO learns the contexts surrounding API references to later resolve simple names to their FQN from a large dataset of over 200,000 records. Similarly, LiFUSO, our feature extraction approach, can uncover features offered by a library from a dataset containing more than 10,000 code snippets.

One of the advantages of data-driven tools is that the discovered patterns may be used by most users. The learnt patterns might cover many of the use cases, and may also reflect what is considered a “*standard*” by users. Another advantage is that they might provide a solution (or an initial approximation) to problems where algorithm-based solutions fall short. For example, algorithmic solutions to the incompleteness problem in SO code snippets are limited to the information provided in the snippet. Data-driven solutions, on the other hand, can take advantage of similar usages and propose a resolution for an incomplete API reference (e.g., RESICO).

Despite the mentioned advantages, data-driven approaches also have limitations. Our two approaches are based on frequent patterns, therefore, a particular use case might not be found or resolved. An infrequent API method might not be resolved by RESICO. Similarly, an infrequent feature might not be extracted by LiFUSO. Another limitation of these approaches is their computationally intensive training. Since the goal is to learn patterns from massive amounts of data, achieving it may be costly. Lastly, obtained models might need to be re-trained whenever new data not initially considered in the original training is ready to be processed. For RESICO and LiFUSO, this might be needed when the API of a library has been updated with new features and has been widely adopted by the developer community. Re-training implies re-running our pipelines from scratch. To facilitate this process, we have provided links to executable scripts. However, these scripts need to be maintained and need to co-evolve with the data source (e.g., Stack Overflow or GitHub APIs).

<sup>4</sup> <https://github.com/cvelazquezr/RESICO-LiFUSO>

## 7.3 Future Work

This dissertation investigates two main lines of research as described in Chapter 4 and Chapter 5. The first research line comprises the analysis of incomplete code snippets in general and API type resolution in particular. The second research line comprises the automated uncovering of library features from their API usage on Stack Overflow, where natural language text accompanies the code. We foresee future research in these two areas and discuss their avenues in detail below.

### 7.3.1 API Type Resolution

Regarding API type resolution for incomplete code snippets, we envision two possible directions for future work:

#### Hybrid API Type Resolution

One of the reasons for selecting Stack Overflow as our library information source is the availability of natural language text related to code snippets (cf. Chapter 2). The text enables us to assign a library feature name to each mined API usage pattern. Additionally, in many cases, the surrounding text provides hints about the API types used within the code snippets. For example, they might indicate links to the API documentation, class or method names within the code, or simply refer to the library name. Although some approaches have been proposed to extract code from text [79, 80], we are not aware of research further relating the extracted code information to API type resolution. Future work could investigate improving API type resolutions by emulating the manual process of users searching for references in the text surrounding code snippets.

#### Other Cases of API Type Resolution

Developers often use code snippets to explain, through code, some concept or feature. However, in many cases, these code snippets might lack references to used fields, methods or classes. API type resolution using our approach might assist in finding missing references. Moreover, the automatic analysis of Git diff files could benefit from our ability to analyse incomplete code snippets. Approaches extracting incomplete code from video tutorials [33, 99] may also benefit from our work. Finally, Discord and Slack bots may also complete code snippets shared during discussions using our work.

### 7.3.2 Uncovering Library Features

In relation to automated library feature uncovering, we foresee five avenues for future work:

#### Analysis of Features in the Library's Architecture

The architecture of each library has been designed and implemented by their maintainers. Features, as mentioned before, are named usage patterns from the library API and are independent of its architecture. An interesting avenue of research concerns the investigation of how features are dispersed across

the library's implementation. In this way, the architecture could be visualised from the features' perspective and depict how different modules relate to each other.

#### **Feature Lag**

Different studies have investigated the technical lag of library version adoption in source code [154, 155]. Library features often consist of multiple API members, which might be adopted by developers at different speeds. Feature lag might therefore, represent an interesting avenue for future work.

#### **Feature Migration**

Extracting named features from library API usage patterns opens up the possibility of comparing libraries, as demonstrated in Chapter 5. Similar features from different libraries targetting the same application domain might be interchangeable in the client's code. Some existing work on library migration [46, 67] might benefit from automatically mined feature descriptions.

#### **Feature Co-Usage Analysis**

Features may be used in client code alongside other features from the same library or with those provided by different libraries. Future work may therefore investigate which libraries or library features are often used together in client code.

#### **Alternative Feature Extraction Sources**

We selected Stack Overflow as the source of information for feature extraction in this dissertation. Future work may investigate alternative sources. For example, library unit test cases or API usage within GitHub repositories. This entails finding an equivalent for the natural language text available on Stack Overflow. For library tests and GitHub repositories, these could be comments, class and method names, the message of the commit that added them, etc. Although not as rich as Stack Overflow's text, these natural language substitutes could also be valuable, especially considering the sometimes low quality of the text in SO posts.

## **7.4 Concluding Remarks**

This dissertation presented approaches for automatically uncovering library features based on their API usage. We believe our work can be used to improve the library selection process for large software ecosystems. LiFUSO, supported by the automatic API type resolution RESICO, realises this goal. Currently, our work supports popular libraries from the Maven ecosystem; however, we may target lesser-known libraries or other software ecosystems in the future. To conclude the dissertation, we summarise our results.

First, we proposed RESICO which resolves missing FQNs for API types in incomplete code snippets such as those found on Stack Overflow. RESICO is essential to automatically analyse posts that lack import statements for API types. Based the context surrounding an API usage, a machine learning classifier proposes the most likely FQN for the API usage. Our results

outperform the current state-of-the-art approach in most of the conducted experiments. The analysis of ambiguous simple names of API types led us to conclude that with sufficient vocabulary, similar usage contexts are the cause of incorrect resolutions. Finally, RESICO supports any library as long as usage examples of this library exist that can serve to train the machine learning classifier, so it can recommend a likely FQN given a similar context.

Second, we proposed LiFUSO which uncovers features from library API usage in Stack Overflow posts. LiFUSO mines library usage from Stack Overflow to gather examples and to extract their surrounding natural language text. An unsupervised machine learning algorithm is used to form clusters of API usage. A filtering process removes clusters without a clearly distinguished API usage pattern. The remaining clusters are then named to form library features, by processing the natural language information in the corresponding SO posts. The uncovered features are compared to those suggested in tutorials and cookbooks of the libraries under analysis. Our results indicate that LiFUSO is able to recommend similar features to those in tutorials and cookbooks. Moreover, we also observed that most uncovered features are found in GitHub client projects. Finally, LiFUSO is library-agnostic as long as there exist API usages and related natural language descriptions.

Third, a combined approach is also proposed in this dissertation. To explore RESICO's ability to assist LiFUSO in its uncovering of library features, we analysed the training dataset of the former approach. Results indicate that an alternative GitHub-based dataset covers more of the APIs of the libraries analysed by LiFUSO, and is therefore worthwhile to retrain RESICO on. However, the newly collected dataset focuses only on the LiFUSO libraries which leads to a more restricted vocabulary. The vocabulary restriction implies that the training process will result in a model that is more vulnerable to ambiguous simple names. Therefore, a rule-based classifier assists RESICO with its resolutions by considering the characteristics of the answers where the API usage occurs. The evaluation of the rule-based classifier shows that it is able to determine the correctness of RESICO's resolutions with a high precision and recall. Supported by the rule-based classifier, RESICO identifies additional Stack Overflow posts to be used as input to LiFUSO. Moreover, the newly gathered posts allow LiFUSO to uncover additional library features and to improve those already suggested.

## Appendix A

### Appendix

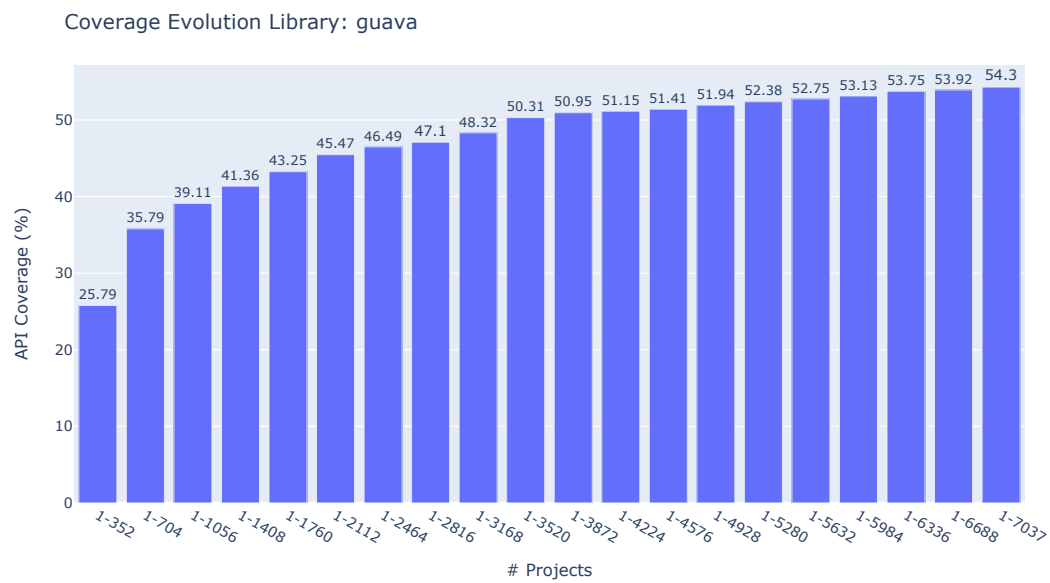


FIGURE A.1: API element coverage of the LiFUSO library *Guava* from its dependent GitHub repositories.



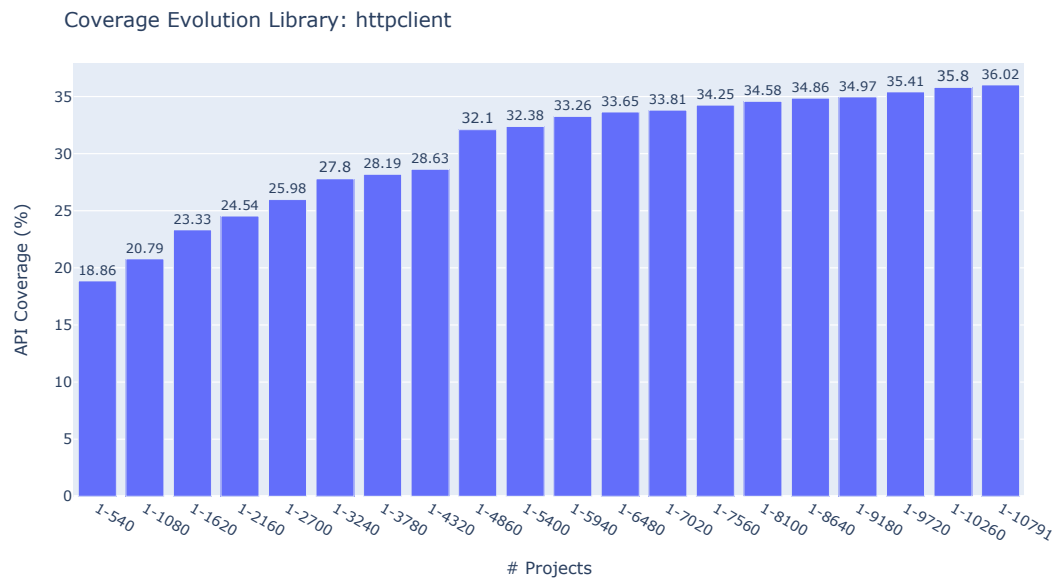


FIGURE A.2: API element coverage of the LiFUSO library *HttpClient* from its dependent GitHub repositories.

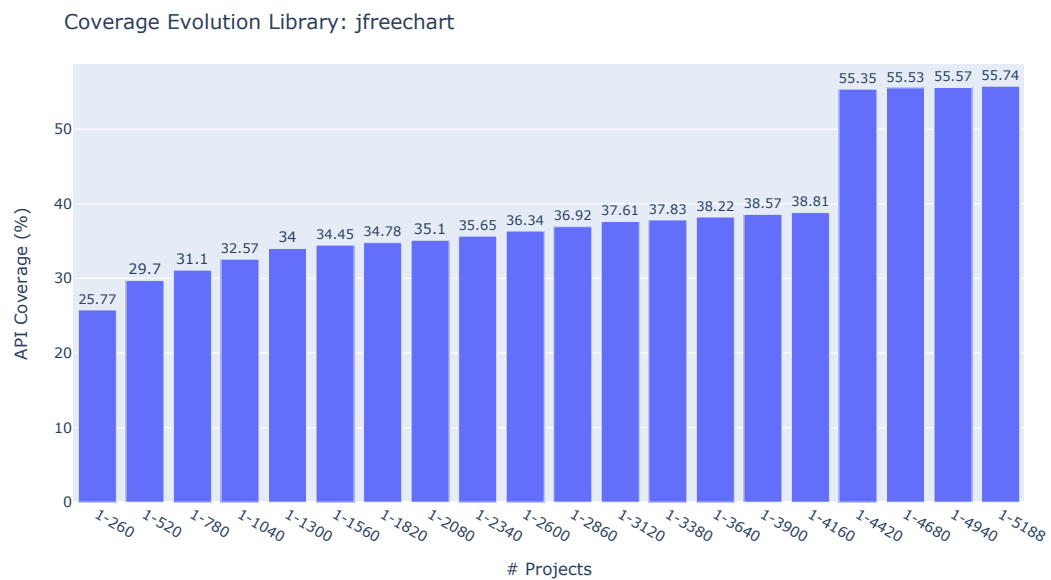


FIGURE A.3: API element coverage of the LiFUSO library *JFreeChart* from its dependent GitHub repositories.

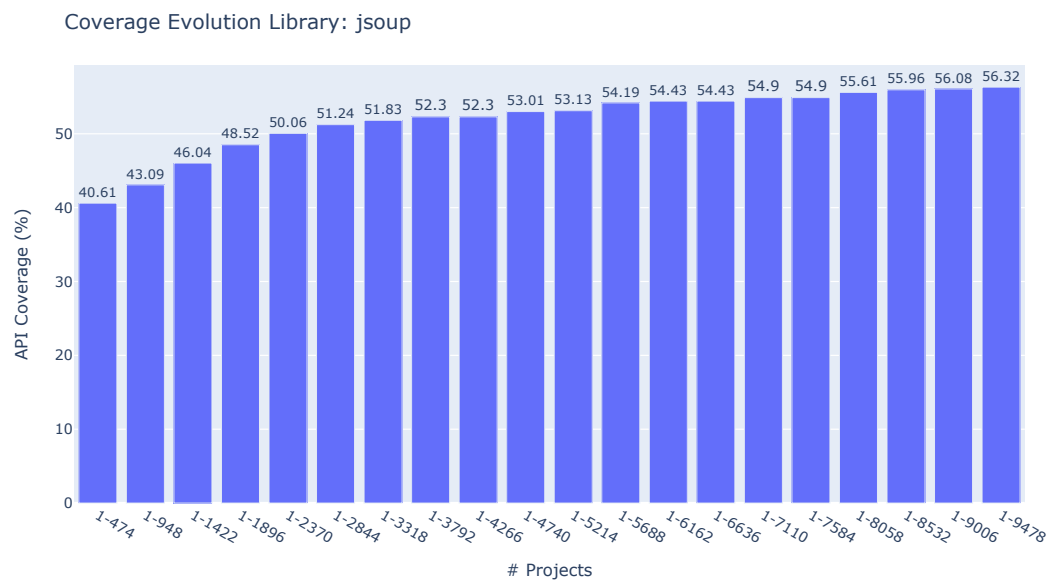


FIGURE A.4: API element coverage of the LiFUSO library *JSoup* from its dependent GitHub repositories.

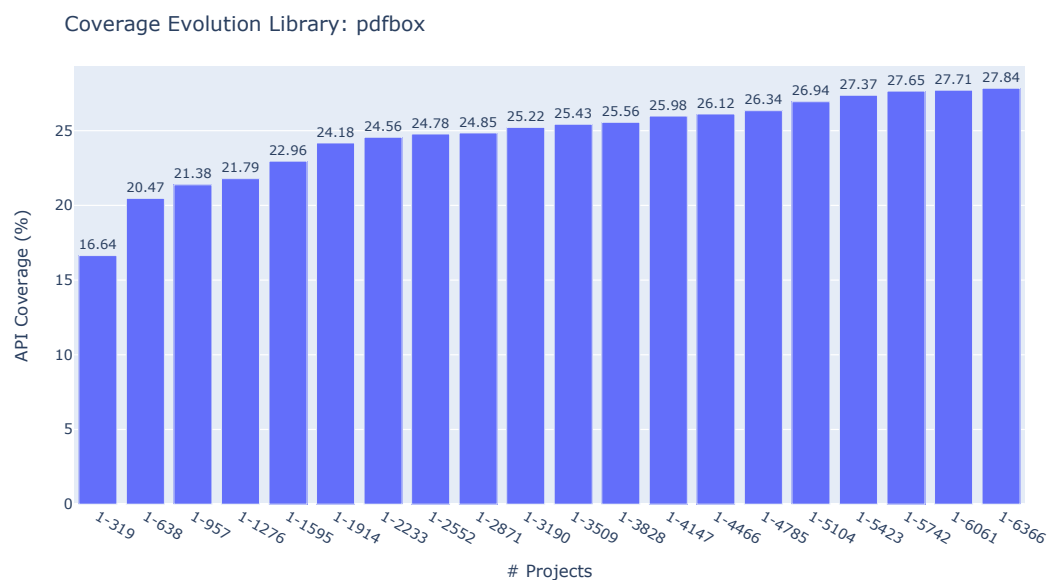


FIGURE A.5: API element coverage of the LiFUSO library *PDF-Box* from its dependent GitHub repositories.

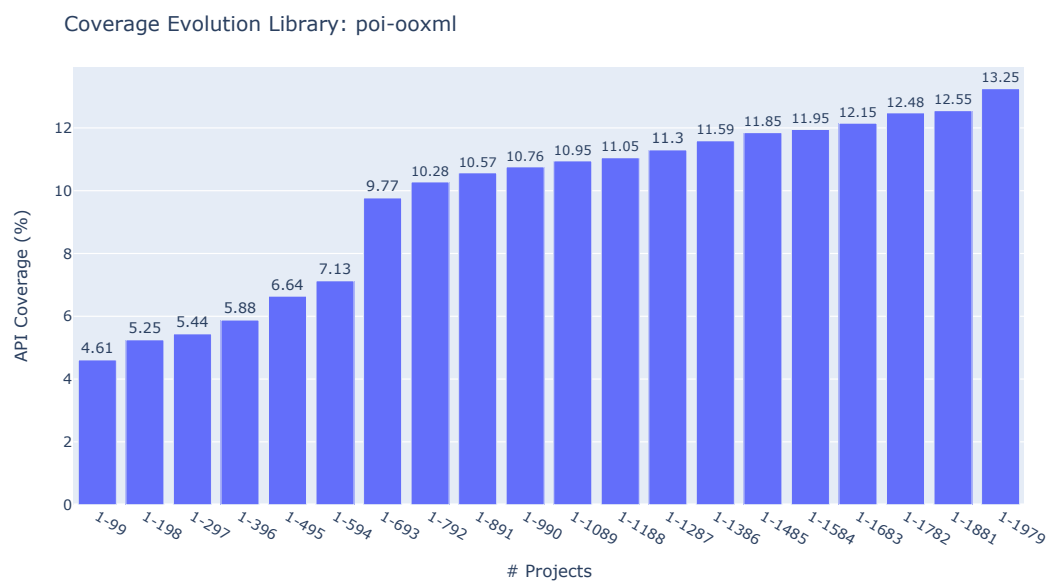


FIGURE A.6: API element coverage of the LiFUSO library *POI-OOXML* from its dependent GitHub repositories.

# Bibliography

- [1] Toufique Ahmed, Premkumar Devanbu, and Vincent J Hellendoorn. "Learning lenient parsing & typing via indirect supervision". In: *Empirical Software Engineering* 26 (2021), pp. 1–31.
- [2] Samuel A. Ajila and Di Wu. "Empirical study of the effects of open source adoption on software development economics". In: *Journal of Systems and Software* 80.9 (2007). Evaluation and Assessment in Software Engineering, pp. 1517–1529. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2007.01.011>. URL: <https://www.sciencedirect.com/science/article/pii/S0164121207000076>.
- [3] A A Al-Subaihin et al. "Clustering Mobile Apps Based on Mined Textual Features". In: *The 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (2016), pp. 1–10. DOI: [10.1145/2961111.2962600](https://doi.org/10.1145/2961111.2962600).
- [4] Miltiadis Allamanis, Hao Peng, and Charles Sutton. "A Convolutional Attention Network for Extreme Summarization of Source Code". In: *International Conference on Machine Learning*. PMLR. 2016, pp. 2091–2100.
- [5] Miltiadis Allamanis et al. "Suggesting Accurate Method and Class Names". In: *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 2015, pp. 38–49.
- [6] Uri Alon et al. "code2seq: Generating Sequences from Structured Representations of Code". In: *arXiv preprint arXiv:1808.01400* (2018).
- [7] Uri Alon et al. "code2vec: Learning Distributed Representations of Code". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), p. 40.
- [8] Sven Amann et al. "A Systematic Evaluation of Static API-Misuse Detectors". In: *IEEE Transactions on Software Engineering* 45.12 (2019), pp. 1170–1188. ISSN: 0098-5589. DOI: [10.1109/tse.2018.2827384](https://doi.org/10.1109/tse.2018.2827384).
- [9] Giuliano Antoniol and Yann-Gaël Guéhéneuc. "Feature Identification: A Novel Approach and a Case Study". In: *The 21st IEEE International Conference on Software Maintenance (ICSM'05)* (2005), pp. 357–366. DOI: [10.1109/icsm.2005.48](https://doi.org/10.1109/icsm.2005.48).
- [10] Maram Assi et al. "FeatCompare: Feature comparison for competing mobile apps leveraging user reviews". In: *Empirical Software Engineering* 26 (2021), pp. 1–38.
- [11] Alberto Bacchelli et al. "Extracting Structured Data from Natural Language Documents with Island Parsing". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE. 2011, pp. 476–479.

- [12] Sebastian Baltes et al. "SOTorrent: Reconstructing and Analyzing the Evolution of Stack Overflow Posts". In: *The 15th International Conference on Mining Software Repositories (MSR 2018)*. 2018, pp. 319–330.
- [13] Sebastian Baltes et al. "SOTorrent: Reconstructing and Analyzing the Evolution of Stack Overflow Posts". In: *Proceedings of the 15th international conference on Mining Software Repositories (MSR)*. 2018, pp. 319–330. ISBN: 9781450357166. DOI: [10.1145/3196398.3196430](https://doi.org/10.1145/3196398.3196430). eprint: [1803.07311](https://arxiv.org/abs/1803.07311).
- [14] Madeleine Bates. "Models of natural language understanding". In: *Proceedings of the National Academy of Sciences* 92.22 (1995), pp. 9977–9982.
- [15] Veronika Bauer, Lars Heinemann, and Florian Deissenboeck. "A Structured Approach to Assess Third-Party Library Usage". In: *28th IEEE International Conference on Software Maintenance (ICSM)*. 2012, pp. 483–492. DOI: [10.1109/ICSM.2012.6405311](https://doi.org/10.1109/ICSM.2012.6405311).
- [16] Amine Benelallam et al. *Maven central dependency graph*. Nov. 2018. DOI: [10.5281/zenodo.1489120](https://doi.org/10.5281/zenodo.1489120). URL: <https://doi.org/10.5281/zenodo.1489120>.
- [17] Amine Benelallam et al. "The Maven Dependency Graph: A Temporal Graph-Based Representation of Maven Central". In: *Proc. of the 16th Int. Conf. on Mining Software Repositories (MSR)*. 2019.
- [18] James Bergstra, Daniel Yamins, and David Cox. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures". In: *International Conference on Machine Learning*. PMLR. 2013, pp. 115–123.
- [19] James Bergstra et al. "Algorithms for Hyper-Parameter Optimization". In: *Advances in neural information processing systems* 24 (2011).
- [20] Vishwanath Bijalwan et al. "KNN based Machine Learning Approach for Text and Document Mining". In: *International Journal of Database Theory and Application* 7.1 (2014), pp. 61–70.
- [21] Christopher M Bishop and Nasser M Nasrabadi. *Pattern Recognition and Machine Learning*. Vol. 4. 4. Springer, 2006.
- [22] Piotr Bojanowski et al. "Enriching Word Vectors with Subword Information". In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 135–146. ISSN: 2307-387X.
- [23] Ameni Bouaziz et al. "Short Text Classification Using Semantic Random Forest". In: *International Conference on Data Warehousing and Knowledge Discovery*. 2014, pp. 288–299.
- [24] Leo Breiman. "Random Forests". In: *Machine learning* 45.1 (2001), pp. 5–32.
- [25] Markus M Breunig et al. "LOF: Identifying Density-Based Local Outliers". In: *The 2000 ACM SIGMOD International Conference on Management of Data*. 2000, pp. 93–104.
- [26] Allan G Bromley. "Charles Babbage's Analytical Engine, 1838". In: *Annals of the History of Computing* 4.3 (1982), pp. 196–217.
- [27] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. "InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees". In: *2021*

- IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 1186–1197.
- [28] Liang Cai et al. “BIKER: A Tool for Bi-information Source Based API Method Recommendation”. In: *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE19)*. 2019, pp. 1075–1079.
- [29] William E Carlson et al. “Introducing Ada”. In: *Proceedings of the ACM 1980 annual conference*. 1980, pp. 263–271.
- [30] Francisco Charte et al. “Addressing imbalance in multilabel classification: Measures and random resampling algorithms”. In: *Neurocomputing* 163 (2015), pp. 3–16.
- [31] Haihua Chen et al. “A Comparative Study of Automated Legal Text Classification using Random Forests and Deep Learning”. In: *Information Processing & Management* 59.2 (2022), p. 102798.
- [32] Noam Chomsky and David W Lightfoot. *Syntactic Structures*. Walter de Gruyter, 2002.
- [33] Nathan Cooper et al. “It Takes Two to Tango: Combining Visual and Textual Information for Detecting Duplicate Video-Based Bug Reports”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 957–969.
- [34] Corinna Cortes and Vladimir Vapnik. “Support-vector Networks”. In: *Machine learning* 20.3 (1995), pp. 273–297.
- [35] Chris Cummins et al. “PROGRAML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations”. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 2244–2253.
- [36] Barthélémy Dagenais and Laurie Hendren. “Enabling Static Analysis for Partial Java Programs”. In: *Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems Languages and Applications*. 2008, pp. 313–328.
- [37] Barthélémy Dagenais and Martin P Robillard. “Recovering Traceability Links between an API and Its Learning Resources”. In: *Proceedings - International Conference on Software Engineering*. 2012, pp. 47–57. ISBN: 9781467310673. DOI: [10.1109/ICSE.2012.6227207](https://doi.org/10.1109/ICSE.2012.6227207).
- [38] Belur V Dasarathy. “Nearest neighbor (NN) norms: NN pattern classification techniques”. In: *IEEE Computer Society Tutorial* (1991).
- [39] Dario Di Nucci et al. “Detecting Code Smells using Machine Learning Techniques: Are We There Yet?” In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 612–621.
- [40] Yiwen Dong et al. “SnR: Constraint-Based Type Inference for Incomplete Java Code Snippets”. In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 1982–1993.
- [41] Yael Dubinsky et al. “An Exploratory Study of Cloning in Industrial Software Product Lines”. In: *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE. 2013, pp. 25–34.

- [42] Rehab El-Hajj and Sarah Nadi. "LibComp: An IntelliJ Plugin for Comparing Java Libraries". In: *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), pp. 1591–1595. DOI: [10.1145/3368089.3417922](https://doi.org/10.1145/3368089.3417922).
- [43] Yuxiang Gao, Yi Zhu, and Yu Zhao. "Dealing with imbalanced data for interpretable defect prediction". In: *Information and Software Technology* 151 (2022), p. 107016.
- [44] Mohammad Ghafari, Konstantin Rubinov, and Mohammad Mehdi Pourhashem K. "Mining unit test cases to synthesize API usage examples". In: *Journal of software: evolution and process* 29.12 (2017), e1841.
- [45] Herman Heine Goldstine, John Von Neumann, and John Von Neumann. "Planning and coding of problems for an electronic computing instrument". In: (1947).
- [46] Haiqiao Gu, Hao He, and Minghui Zhou. "Self-Admitted Library Migrations in Java, JavaScript, and Python Packaging Ecosystems: A Comparative Study". In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2023, pp. 627–638.
- [47] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. "CodeKernel: A Graph Kernel based Approach to the Selection of API Usage Examples". In: *The 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* 00 (2019), pp. 590–601. DOI: [10.1109/ase.2019.00061](https://doi.org/10.1109/ase.2019.00061).
- [48] Xiaodong Gu et al. "Deep API Learning". In: *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 2016, pp. 631–642.
- [49] Yao Guo et al. "What's Inside My App?: Understanding Feature Redundancy in Mobile Apps". In: *The 26th Conference on Program Comprehension* (2018), pp. 266–276. DOI: [10.1145/3196321.3196329](https://doi.org/10.1145/3196321.3196329).
- [50] Piyush Gupta, Nikita Mehrotra, and Rahul Purandare. "JCoffee: Using Compiler Feedback to Make Partial Code Snippets Compilable". In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2020, pp. 810–813.
- [51] Emitza Guzman and Walid Maalej. "How Do Users Like This Feature? A Fine Grained Sentiment Analysis of App Reviews". In: *The 22nd International Requirements Engineering Conference (RE)* (2014), pp. 153–162. DOI: [10.1109/re.2014.6912257](https://doi.org/10.1109/re.2014.6912257).
- [52] Tracy Hall et al. "A systematic literature review on fault prediction performance in software engineering". In: *IEEE Transactions on Software Engineering* 38.6 (2011), pp. 1276–1304.
- [53] Nicolas Harrand et al. "API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client-API usages". In: *Journal of Systems and Software* 184 (2022), p. 111134.
- [54] Zellig S Harris. "Distributional structure". In: *Word* 10.2-3 (1954), pp. 146–162.



- [55] Johannes Härtel, Hakan Aksu, and Ralf Lämmel. "Classification of APIs by Hierarchical Clustering". In: *The 26th International Conference on Program Comprehension (ICPC)*. IEEE. 2018, pp. 233–23310.
- [56] Marti A. Hearst et al. "Support Vector Machines". In: *Intelligent Systems and their applications* 13.4 (1998), pp. 18–28.
- [57] Jordan Henkel et al. "Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces". In: *Proceedings of the 26th Joint Meeting of European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE18)*. 2018, pp. 163–174.
- [58] Tin Kam Ho. "Random Decision Forests". In: *Proceedings of 3rd international conference on document analysis and recognition*. Vol. 1. IEEE. 1995, pp. 278–282.
- [59] Yuan Huang et al. "Towards Exploring the Code Reuse from Stack Overflow during Software Development". In: *30th IEEE/ACM International Conference on Program Comprehension*. 2022, pp. 548–559.
- [60] Srinivasan Iyer et al. "Summarizing Source Code using a Neural Attention Model". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2016, pp. 2073–2083.
- [61] Yuan Jiang et al. "Hierarchical Semantic-Aware Neural Code Representation". In: *Journal of Systems and Software* 191 (2022), p. 111355.
- [62] Thorsten Joachims. *Learning to Classify Text using Support Vector Machines*. Vol. 668. Springer Science & Business Media, 2002.
- [63] George H John and Pat Langley. "Estimating Continuous Distributions in Bayesian Classifiers". In: *Proc. of the 11th Conf. on Uncertainty in Artificial Intelligence (UAI95)*. 1995, pp. 338–345.
- [64] Tetsuya Kanda et al. "Semi-Automatically Extracting Features from Source Code of Android Applications". In: *IEICE Transactions on Information and Systems* E96.D.12 (2013), pp. 2857–2859. ISSN: 0916-8532. DOI: [10.1587/transinf.e96.d.2857](https://doi.org/10.1587/transinf.e96.d.2857).
- [65] Hong Jin Kang and David Lo. "Active Learning of Discriminative Subgraph Patterns for API Misuse Detection". In: *IEEE Transactions on Software Engineering* 48.8 (2021), pp. 2761–2783.
- [66] Leonard Kaufman and Peter J Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Vol. 344. John Wiley & Sons, 1990.
- [67] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. "A Systematic Review of API Evolution Literature". In: *ACM Computing Surveys (CSUR)* 54.8 (2021), pp. 1–36.
- [68] Peter Langfelder, Bin Zhang, and Steve Horvath. "Defining clusters from a hierarchical cluster tree: the Dynamic Tree Cut package for R". In: *Bioinformatics* 24.5 (2008), pp. 719–720.
- [69] Daniel Lehmann and Michael Pradel. "Finding the Dwarf: Recovering Precise Types from WebAssembly Binaries". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 410–425.

- [70] Vladimir I Levenshtein et al. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics doklady*. Vol. 10. 8. Soviet Union. 1966, pp. 707–710.
- [71] Joseph Lilleberg, Yun Zhu, and Yanqing Zhang. "Support Vector Machines and Word2vec for Text Classification with Semantic Features". In: *2015 IEEE 14th International Conference on Cognitive Informatics & Cognitive Computing (ICCI\* CC)*. IEEE. 2015, pp. 136–140.
- [72] Bin Lin et al. "Pattern-based Mining of Opinions in Q&A Websites". In: *Proceedings of the 41st International Conference on Software Engineering (ICSE19)*. 2019, pp. 548–559.
- [73] Chen Lin et al. "Improving Code Summarization with Block-wise Abstract Syntax Tree Splitting". In: *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE. 2021, pp. 184–195.
- [74] Zachary C Lipton, Charles Elkan, and Balakrishnan Naryanaswamy. "Optimal Thresholding of Classifiers to Maximize F1 Measure". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2014, pp. 225–239.
- [75] Linfeng Liu et al. "Universal Representation for Code". In: *Advances in Knowledge Discovery and Data Mining: 25th Pacific-Asia Conference, PAKDD 2021, May 11–14, 2021, Proceedings, Part III*. Springer. 2021, pp. 16–28.
- [76] Wenjian Liu et al. "Identifying change patterns of API misuses from code changes". In: *Science China Information Sciences* 64 (2021), pp. 1–19.
- [77] Adriaan Lotter et al. "Code Reuse in Stack Overflow and Popular Open Source Java Projects". In: *2018 25th Australasian Software Engineering Conference (ASWEC)*. IEEE. 2018, pp. 141–150.
- [78] Kien Luong, Ferdian Thung, and David Lo. "ARSearch: Searching for API Related Resources from Stack Overflow and GitHub". In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 2022, pp. 11–15.
- [79] Kien Luong, Ferdian Thung, and David Lo. "Disambiguating Mentions of API Methods in Stack Overflow via Type Scoping". In: *2021 IEEE International Conference on Software Maintenance and Evolution (IC-SME)*. IEEE. 2021, pp. 679–683.
- [80] Kien Luong et al. "ARSeek: Identifying API Resource using Code and Discussion on Stack Overflow". In: *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 2022, pp. 331–342.
- [81] Minh-Thang Luong and Christopher D Manning. "Achieving Open Vocabulary Neural Machine Translation with Hybrid Word-Character Models". In: *arXiv preprint arXiv:1604.00788* (2016).
- [82] Gjorgji Madjarov et al. "An extensive experimental comparison of methods for multi-label learning". In: *Pattern recognition* 45.9 (2012), pp. 3084–3104.
- [83] Oded Maimon and Lior Rokach. *Data Mining and Knowledge Discovery Handbook. 2nd Edition*. Springer, 2010.

- [84] Christopher D Manning et al. "The Stanford CoreNLP natural language processing toolkit". In: *The 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. 2014, pp. 55–60.
- [85] Pedro Martins, Rohan Achar, and Cristina V Lopes. "50K-C: A dataset of compilable, and compiled, Java projects". In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE. 2018, pp. 1–5.
- [86] Tomas Mikolov et al. "Distributed Representations of Words and Phrases and their Compositionality". In: *Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NIPS13)*. 2013.
- [87] Tomas Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *arXiv preprint arXiv:1301.3781* (2013).
- [88] Leon Moonen. "Generating Robust Parsers using Island Grammars". In: *The 8th Working Conference on Reverse Engineering* (2001), pp. 13–22. DOI: [10.1109/wcre.2001.957806](https://doi.org/10.1109/wcre.2001.957806).
- [89] Kevin P Murphy. *Machine Learning: A Probabilistic Perspective*. MIT press, 2012.
- [90] Tam The Nguyen et al. "Learning API Usages from Bytecode: A Statistical Approach". In: *Proceedings of the 38th International Conference on Software Engineering*. 2016, pp. 416–427.
- [91] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. "Mapping API Elements for Code Migration with Vector Representations". In: *Companion to the Proceedings of the 38th International Conference on Software Engineering (ICSE-C16)*. IEEE. 2016, pp. 756–758.
- [92] Trong Duc Nguyen et al. "Exploring API Embedding for API Usages and Applications". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017, pp. 438–449.
- [93] Tung Thanh Nguyen et al. "Graph-based Mining of Multiple Object Usage Patterns". In: *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*. 2009, pp. 383–392.
- [94] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, 2013, pp. 1–326.
- [95] Hung Phan et al. "Statistical Learning of API Fully Qualified Names in Code Snippets of Online Forums". In: *Proceedings of the 40th International Conference on Software Engineering (ICSE18)*. IEEE. 2018, pp. 632–642.
- [96] John C Platt. "Fast Training of Support Vector Machines using Sequential Minimal Optimization". In: *Support vector learning* (1999), pp. 185–208.
- [97] Luca Ponzanelli. "Holistic Recommender Systems for Software Engineering". PhD thesis. Università della Svizzera italiana, Mar. 2017.
- [98] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. "StORMeD: Stack Overflow Ready Made Data". In: *The 12th International Working Conference on Mining Software Repositories (MSR15)*. 2015.

- [99] Luca Ponzanelli et al. "Automatic Identification and Classification of Software Development Video Tutorial Fragments". In: *IEEE Transactions on Software Engineering* 45.5 (2019), pp. 464–488. ISSN: 19393520. DOI: [10.1109/TSE.2017.2779479](https://doi.org/10.1109/TSE.2017.2779479).
- [100] Luca Ponzanelli et al. "Mining StackOverflow to Turn the IDE into a Self-Confident Programming Prompter". In: *11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings* (2014), pp. 102–111. DOI: [10.1145/2597073.2597077](https://doi.org/10.1145/2597073.2597077).
- [101] William H Press et al. *Numerical Recipes 3rd edition: The Art of Scientific Computing*. Cambridge University Press, 2007.
- [102] Jesse Read. "Scalable Multi-label Classification". PhD thesis. University of Waikato, 2010.
- [103] Jesse Read et al. "Classifier chains for multi-label classification". In: *Machine learning* 85.3 (2011), p. 333.
- [104] Peter J Rousseeuw. "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis". In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65.
- [105] Riccardo Rubei et al. "PostFinder: Mining Stack Overflow posts to support software developers". In: *Information and Software Technology* 127 (2020), p. 106367.
- [106] CM Khaled Saifullah, Muhammad Asaduzzaman, and Chanchal K Roy. "Learning from Examples to Find Fully Qualified Names of API Elements in Code Snippets". In: *Proceedings of the 34th International Conference on Automated Software Engineering (ASE19)*. 2019, pp. 243–254.
- [107] Yasubumi Sakakibara, Kazuo Misue, and Takeshi Koshiba. "Text classification and keyword extraction by learning decision trees". In: *Proceedings of 9th Conference on Artificial Intelligence for Applications (AIAI93)*. 1993, p. 466.
- [108] Federica Sarro et al. "Feature Lifecycles as They Spread, Migrate, Remain, and Die in App Stores". In: *2015 IEEE 23rd International requirements engineering conference (RE)*. IEEE. 2015, pp. 76–85.
- [109] Konstantinos Sechidis, Grigorios Tsoumakas, and Ioannis Vlahavas. "On the Stratification of Multi-label Data". In: *Machine Learning and Knowledge Discovery in Databases* (2011), pp. 145–158.
- [110] Faiz Ali Shah, Yevhenii Sabanin, and Dietmar Pfahl. "Feature-Based Evaluation of Competing Apps". In: *The International Workshop on App Market Analytics* (2016), pp. 15–21. DOI: [10.1145/2993259.2993267](https://doi.org/10.1145/2993259.2993267).
- [111] Kanish Shah et al. "A Comparative Analysis of Logistic Regression, Random Forest and KNN Models for the Text Classification". In: *Augmented Human Research* 5.1 (2020), pp. 1–16.
- [112] Ketan Rajshekhar Shahapure and Charles Nicholas. "Cluster Quality Analysis Using Silhouette Score". In: *The 7th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE. 2020, pp. 747–748.

- [113] Qi Shen et al. "From API to NLI: A new interface for library reuse". In: *Journal of Systems and Software* 169 (2020), p. 110728. ISSN: 0164-1212. DOI: [10.1016/j.jss.2020.110728](https://doi.org/10.1016/j.jss.2020.110728).
- [114] César Soto-Valero et al. "A comprehensive study of bloated dependencies in the Maven ecosystem". In: *Empirical Software Engineering* 26.3 (2021), pp. 1–44.
- [115] Pascal Soucy and Guy W. Mineau. "A Simple KNN Algorithm for Text Categorization". In: *Proceedings - IEEE International Conference on Data Mining, ICDM* (2001), pp. 647–648. ISSN: 15504786. DOI: [10.1109/icdm.2001.989592](https://doi.org/10.1109/icdm.2001.989592).
- [116] Karen Sparck Jones. "A statistical interpretation of term specificity and its application in retrieval". In: *Journal of documentation* 28.1 (1972), pp. 11–21.
- [117] Mervyn Stone. "Cross-validators: Choice and Assessment of Statistical Predictions". In: *Journal of the royal statistical society. Series B (Methodological)* (1974), pp. 111–147.
- [118] Siddharth Subramanian and Reid Holmes. "Making Sense of Online Code Snippets". In: *IEEE International Working Conference on Mining Software Repositories* (2013), pp. 85–88. ISSN: 21601852. DOI: [10.1109/MSR.2013.6624012](https://doi.org/10.1109/MSR.2013.6624012).
- [119] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. "Live API Documentation". In: *Proceedings of the 36th International Conference on Software Engineering (ICSE14)*. 2014, pp. 643–652. DOI: [10.1145/2568225.2568313](https://doi.org/10.1145/2568225.2568313).
- [120] Amann Sven et al. "Investigating Next Steps in Static API-Misuse Detection". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 265–275.
- [121] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. "The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models". In: *IEEE Transactions on Software Engineering* 46.11 (2018), pp. 1200–1219.
- [122] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. "CSnippEx: Automated Synthesis of Compilable Code Snippets from Q&A Sites". In: *Proceedings of the 25th international symposium on software testing and analysis*. 2016, pp. 118–129.
- [123] Bart Theeten, Frederik Van deputte, and Tom Van Cutsem. "Import2vec Learning Embeddings for Software Libraries". In: *Proceedings of the 16th International Conference on Mining Software Repositories (MSR19)*. 2019, pp. 18–28.
- [124] Suresh Thummalapenta and Tao Xie. "PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web". In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated Software Engineering*. 2007, pp. 204–213.
- [125] Fuwei Tian and Christoph Treude. "Adding Context to Source Code Representations for Deep Learning". In: *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2022, pp. 374–378.



- [126] Kristina Toutanova et al. "Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network". In: *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*. 2003, pp. 252–259.
- [127] Kristina Toutanova and Christopher D Manning. "Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger". In: *2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora*. 2000, pp. 63–70.
- [128] Christoph Treude and Martin P. Robillard. "Augmenting API Documentation with Insights from Stack Overflow". In: *Proceedings of the 38th International Conference on Software Engineering (ICSE16)*. 2016, pp. 392–403.
- [129] Grigorios Tsoumakas and Ioannis Katakis. "Multi-Label Classification: An Overview". In: *International Journal of Data Warehousing and Mining (IJDWM)* 3.3 (2007), pp. 1–13.
- [130] Grigorios Tsoumakas and Ioannis Vlahavas. "Random k-Labelsets: An Ensemble Method for Multilabel Classification". In: *European conference on machine learning*. Springer. 2007, pp. 406–417.
- [131] Gias Uddin, Foutse Khomh, and Chanchal K Roy. "Mining API usage scenarios from Stack Overflow". In: *Information and Software Technology* 122 (2020), p. 106277.
- [132] Dheeraj Vagavolu, Karthik Chandra Swarna, and Sridhar Chimalakonda. "A Mocktail of Source Code Representations". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 1296–1300.
- [133] Enrique Larios Vargas et al. "Selecting Third-Party Libraries: The Practitioners' Perspective". In: *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), pp. 245–256. DOI: [10.1145/3368089.3409711](https://doi.org/10.1145/3368089.3409711). eprint: [2005.12574](https://doi.org/10.1145/3368089.3409711).
- [134] Camilo Velázquez-Rodríguez, Eleni Constantinou, and Coen De Roover. "LiFUSO: Uncovering Library Features from API Usage on Stack Overflow". In: *38th IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 03-07, 2022*. IEEE, 2022.
- [135] Camilo Velázquez-Rodríguez, Eleni Constantinou, and Coen De Roover. "Uncovering Library Features from API Usage on Stack Overflow". In: *29th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Hawaii, United States of America, March 15-17, 2022*. IEEE, 2022, pp. 207–217. DOI: [10.1109/SANER53432.2022.00035](https://doi.org/10.1109/SANER53432.2022.00035). URL: <https://doi.org/10.1109/SANER53432.2022.00035>.
- [136] Camilo Velázquez-Rodríguez and Coen De Roover. "MUTAMA: An Automated Multi-label Tagging Approach for Software Libraries on Maven". In: *The 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2020, pp. 254–258.

- [137] Camilo Velázquez-Rodríguez, Dario Di Nucci, and Coen De Roover. "A Text Classification Approach to API Type Resolution for Incomplete Code Snippets". In: *Science of Computer Programming* 227 (2023). DOI: [10.1016/j.scico.2023.102941](https://doi.org/10.1016/j.scico.2023.102941). URL: <https://doi.org/10.1016/j.scico.2023.102941>.
- [138] Camilo Velázquez-Rodríguez and Coen De Roover. "Automatic library categorization". In: *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*. ACM, 2020, pp. 733–734. DOI: [10.1145/3387940.3392186](https://doi.org/10.1145/3387940.3392186). URL: <https://doi.org/10.1145/3387940.3392186>.
- [139] Jue Wang et al. "Mining Succinct and High-Coverage API Usage Patterns from Source Code". In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2013, pp. 319–328.
- [140] Ke Wang and Zhendong Su. "Blended, Precise Semantic Program Embeddings". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 121–134.
- [141] Yawen Wang et al. "Where is Your App Frustrating Users?" In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 2427–2439.
- [142] Richard L Wexelblat. "History of Programming Languages". In: *Proceedings of the ACM SIGPLAN History of Programming Languages Conference*. 1981.
- [143] Maurice V Wilkes et al. *The Preparation of Programs for an Electronic Digital Computer*. 1958.
- [144] L B Wilson and Robert G Clark. *Comparative Programming Languages*. en. International Computer Science Series. Harlow, England: Longman Higher Education, 1988.
- [145] Ian H Witten et al. "The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"". In: *Morgan Kaufmann*. Elsevier Amsterdam, 2016.
- [146] Di Wu et al. "Generating API tags for tutorial fragments from Stack Overflow". In: *Empirical Software Engineering* 26.4 (2021), p. 66.
- [147] Ho Chung Wu et al. "Interpreting TF-IDF Term Weights as Making Relevance Decisions". In: *ACM Transactions on Information Systems (TOIS)* 26.3 (2008), pp. 1–37.
- [148] Tao Xie and Jian Pei. "MAPO: Mining API Usages from Open Source Repositories". In: *Proceedings of the 2006 international workshop on Mining software repositories*. 2006, pp. 54–57.
- [149] Baoxun Xu et al. "An Improved Random Forest Classifier for Text Categorization". In: *JCP* 7.12 (2012), pp. 2913–2920.
- [150] Di Yang, Aftab Hussain, and Cristina Videira Lopes. "From Query to Usable Code: An Analysis of Stack Overflow Code Snippets". In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2016, pp. 391–401.
- [151] Di Yang et al. "Stack Overflow in Github: Any Snippets There?" In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, pp. 280–290.



- [152] Xin Ye et al. "From Word Embeddings To Document Similarities for Improved Information Retrieval in Software Engineering". In: *Proceedings of the 38th International Conference on Software Engineering*. 2016, pp. 404–415.
- [153] Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. "Identifying Versions of Libraries used in Stack Overflow Code Snippets". In: *The 18th International Conference on Mining Software Repositories (MSR)*. IEEE. 2021, pp. 341–345.
- [154] Ahmed Zerouali et al. "A formal framework for measuring technical lag in component repositories—and its application to npm". In: *Journal of Software: Evolution and Process* 31.8 (2019), e2157.
- [155] Ahmed Zerouali et al. "An Empirical Analysis of Technical Lag in npm Package Dependencies". In: *International Conference on Software Reuse*. Springer. 2018, pp. 95–110.
- [156] Ahmed Zerouali et al. "On the Diversity of Software Package Popularity Metrics: An Empirical Study of NPM". In: *The 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2019, pp. 589–593.
- [157] Fengyi Zhang et al. "A hybrid code representation learning approach for predicting method names". In: *Journal of Systems and Software* 180 (2021), p. 111011.
- [158] Jian Zhang et al. "A Novel Neural Source Code Representation Based on Abstract Syntax Tree". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 783–794.
- [159] Tianyi Zhang et al. "Analyzing and Supporting Adaptation of Online Code Examples". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 316–327.
- [160] Tong Zhang and Frank J Oles. "Text Categorization based on Regularized Linear Classification Methods". In: *Information retrieval* 4.1 (2001), pp. 5–31.
- [161] Yingying Zhang and Daqing Hou. "Extracting Problematic API Features from Forum Discussions". In: *The 21st International Conference on Program Comprehension (ICPC)*. IEEE. 2013, pp. 142–151.
- [162] Hao Zhong and Hong Mei. "An Empirical Study on API Usages". In: *IEEE Transactions on Software Engineering* 45.4 (2019), pp. 319–334. ISSN: 0098-5589. DOI: [10.1109/tse.2017.2782280](https://doi.org/10.1109/tse.2017.2782280).
- [163] Hao Zhong and Xiaoyin Wang. "Boosting Complete-Code Tool for Partial Program". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 671–681.