SUPPORTING THE TASK-DRIVEN SKILL IDENTIFICATION IN OPEN SOURCE PROJECT ISSUE TRACKING SYSTEMS

By Fabio Santos

A Dissertation

Submitted in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

in Informatics and Computing

Northern Arizona University

May 2023

Approved:

Marco A. Gerosa, Ph.D. Co-Chair

Igor Steinmacher, Ph.D. Co-Chair

Eck Doerry, Ph.D.

Toby Hocking, Ph.D.

Anita Sarma, Ph.D.

ABSTRACT

SUPPORTING THE TASK-DRIVEN SKILL IDENTIFICATION IN OPEN SOURCE PROJECT ISSUE TRACKING SYSTEMS

FABIO SANTOS

Selecting an appropriate task is challenging for contributors to Open Source Software (OSS), mainly for those contributing for the first time. Therefore, researchers and OSS projects have proposed various strategies to aid newcomers, including labeling tasks. In this research, we investigate the automatic labeling of open issues strategy to help the contributors to pick a task to contribute. We label the issues with API domains—categories of APIs parsed from the source code used to solve the issues. We employed mixed methods. We qualitatively analyzed interview transcripts and the survey's open-ended questions to comprehend the strategies communities use to assist in onboarding contributors. We applied quantitative studies to analyze the relevance of the API-domain labels in a user experiment and compared the strategies' relative importance for diverse contributor roles. We mined project and issue data from OSS repositories to build the ground truth and predictors to infer the API-domain labels and compared precision, recall, and F-measure with state-of-the-art. Additionally, inspired by previous research, we advocate that label prediction might benefit from leveraging metrics derived from communication data and social network analysis (SNA) for issues in which social interaction occurs. Thus, we study how these "social metrics" can improve the automatic labeling of open issues with API domains. We mined conversation data from OSS projects' repositories and organized it in periods to reflect the contributors' project participation seasonality. We replicated social metrics from previous work and added them to the corpus to predict API-domain labels. Social metrics improved the performance of the classifiers compared to using only the issue description text in terms of precision (0.922), recall (0.978), and F-measure (0.942). These results indicate that social metrics can help capture the patterns of social interactions in a software project and improve the labeling of issues in an issue tracker.

The results showed that assigning labels to issues is an essential strategy for diverse roles in OSS communities. The API-domain labels are relevant, mainly for experienced practitioners. Labeling the issues with the API-domain labels indicates the skills involved in an issue. The labels represent possible libraries (aggregated into domains) used in the source code related to an issue. We also implemented a free, open-source software demonstration tool to assist newcomers in finding a task to start based on a set of skills elected in a user interface. By investigating this research topic, we expect to assist OSS communities to attract and onboard new contributors.

ACKNOWLEDGEMENTS

Firstly, I would like to express my gratitude to my family for their unwavering encouragement, understanding, and love throughout this journey. My parents, Jairo and Célia, always emphasized the importance of education as the key to success. My dear children, Felipe and Mariana, are my motivation and the driving force behind my endeavors. I must also thank my courageous and supportive wife, Bianca, who stood by my side despite facing her own personal struggles.

Undertaking a doctoral program is a challenging and time-consuming process that requires tremendous effort. I am grateful for the support and guidance of the academic community that helped shape me into the person I am today. I want to extend my appreciation to my advisors, Dr. Marco A. Gerosa and Igor Steinmacher, for their invaluable teachings and precise direction in my research. Their expertise and insights were instrumental in guiding me toward my intended purpose. I also want to express my gratitude to Dr. Igor Wiese, who assisted me during the initial stages of my research, and Dr. João Felipe Pimentel.

I am indebted to Dr. Viacheslav "Slava" Fofanov who welcomed me as a Graduate student, and my Capstone supervisor, Dr. Eck Doerry, who integrated me as a Capstone Mentor and provided me with numerous learning opportunities. I am immensely proud of the hard work we put in and the articles we published together, along with Joseph Vargovich and Jacob Penney - the best team ever!

I would also like to acknowledge my former advisors, Dr. Kate Revoredo, Dr. Fernanda Baião, and Dr. Carlos Eduardo Mello, for their valuable time, attention, and advice during my time at UniRio - Brazil, despite their busy schedules.

I learned a great deal from the professors at SICCS through their lectures, meetings, and conversations. I am grateful to the members of my committee, Dr. Anita Sarma, Dr.

Eck Doerry, and Dr. Toby Hocking, for their observations, feedback, and suggestions that enabled me to refine and enhance my work. I would also like to express my appreciation to the SICCS secretariat for their tireless support of students.

Last but not least, I am grateful to my colleagues on this journey, notably Ivan Gonzalez, Ana Chaves, Vinod, Mahsa, Dina, Italo, Isaac, and the CHURRAS research team in Brazil, for their camaraderie and support throughout this demanding process.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Context and Problem

The first steps toward contributing to OSS are challenging for some developers [2]. Developers are often required to pick up a task from a list of open issues with varying complexity levels and requiring different skills to be completed. Based on the available task data, the skills required for a task are hard to presume since OSS tasks involve diverse concepts and technologies and the descriptions provide insufficient information for newcomers without the background knowledge about the project [3–5].

Adding labels to the issues (a.k.a tasks, bug reports) helps contributors when they are choosing their tasks [6]. However, community managers find manually labeling issues a challenging and time-consuming activity [7]. Many recent studies point to manual or automatic methods to classify issues, but the classification is restricted to the nature of the issue—bug/non-bug, questions, documentation, etc. [8–16].

We extend the existing research by presenting a method to identify skills required to work on an issue. We predict labels based on categories of APIs, called API-domain labels ("UI", "Cloud", "Error Handling," etc.). APIs usually encapsulate modules with specific purposes (e.g., cryptography, database access, logging, etc.), abstracting the underlying implementation. If the contributors know which categories of APIs are required to work on each issue, they could choose tasks that better match their skills or involve skills they want to learn.

We mined text (title, description, and comments) and source code from Issue Tracking Systems (ITS), such as GitHub and Jira, to identify the API domains and create a ground truth to supervised machine learning algorithms used for the predictions. In addition to the text data we mined from the OSS repositories, we also mined conversion

data since ITSs provide an environment that enables collaboration among developers. The sociotechnical essence of software engineering [17] suggests the conversation data may be related to the domain of the project [18] and, therefore, the discussions in issues may indicate the domain of the task solution. Previous studies leverage "social metrics" calculated from data mined from the developers' interactions to compose successful predictive models in diverse domains: co-changes [19], defects [20] and failures [21]. Wiese et al. [22] studied which metrics were used as predictors. Our research explores whether such metrics can improve the API-domain label predictive models.

Our research focused on examining the methods adopted by new contributors to identify relevant issues based on their skillset, as well as the techniques employed by communities to support the onboarding process of these newcomers. Through our investigation, we discovered disparities in the priority assigned to these strategies by various stakeholders, including new contributors, frequent contributors, and maintainers. Using our labeling system, we evaluated the accuracy and progress of the contributions made by these individuals. Ultimately, we developed an open-source software tool to implement our findings. Uncovering good predictors can also help hypothesize conceptual relationships between social aspects of software development and skill identification.

Our work has the following contributions:

- definition of categories or API-domain labels for software projects.
- an approach to predict labels using the information related to the issue and conversations in issue track systems [23–25].
- an investigation of the usefulness of the API-domain labels [23, 25].
- an evaluation of the labeling skills as a strategy with diverse stakeholders [26].

- an OSS implementation of the proposed approach [27]. *GiveMeLabeledIssues* is structured in two layers: the frontend web interface[1] and the backend REST API[2].

To the best of our knowledge, no previous work provides ways to label automatically API domains in OSS issues using data from issue text and conversation data. No previous work has also investigated the relevance of such labels.

1.2 Research Goals and Questions

We propose the following research questions:

RQ1. To what extent can we predict the domain of APIs used in the code that fixes a software issue?

To answer this question, we conducted a case study on JabRef, a reference manager written in Java, with the aim of developing a prediction model. We then expanded our approach to encompass five projects, including two from the industry and incorporated two additional programming languages, C++ and C#, as well as a new Natural Language Processing (NLP) technique, BERT. Our analysis also incorporated metrics from social network analysis within communication networks as features.

RQ2. How does the labeling strategy impact the issue choice and the contribution?

RQ2 aims to investigate the label's strategy and the API-domain labels from the point of view of communities and contributors. In pursuit of this objective, we first conducted an empirical experiment with newcomers to pick issues to a possible contribution and report the information that was relevant to the decision. Next, we interviewed 17 maintainers from 28 projects to identify the strategies utilized by newcomers to locate issues and the

[1]https://github.com/JoeyV55/GiveMeLabeledIssuesUI
[2]https://github.com/JoeyV55/GiveMeLabeledIssuesAPI

strategies employed by communities to facilitate the identification of suitable tasks for these individuals. We also conducted a survey involving 64 stakeholders, requesting that they rank the identified strategies with the goal of identifying any points of agreement or disagreement with respect to these strategies. Finally, we ran an experiment to measure the participants' progress and the contributions' correctness.

Creating a method to identify and label issues with API-domain labels as a proxy for skills can aid in assisting both new and experienced contributors in selecting issues on ITS. By examining the strategies employed by communities to aid contributors, researchers can develop a better understanding of this approach. This, in turn, can assist newcomers in selecting appropriate tasks and streamline the onboarding process, which is essential for maintaining the sustainability of OSS communities.

1.3 Research Method

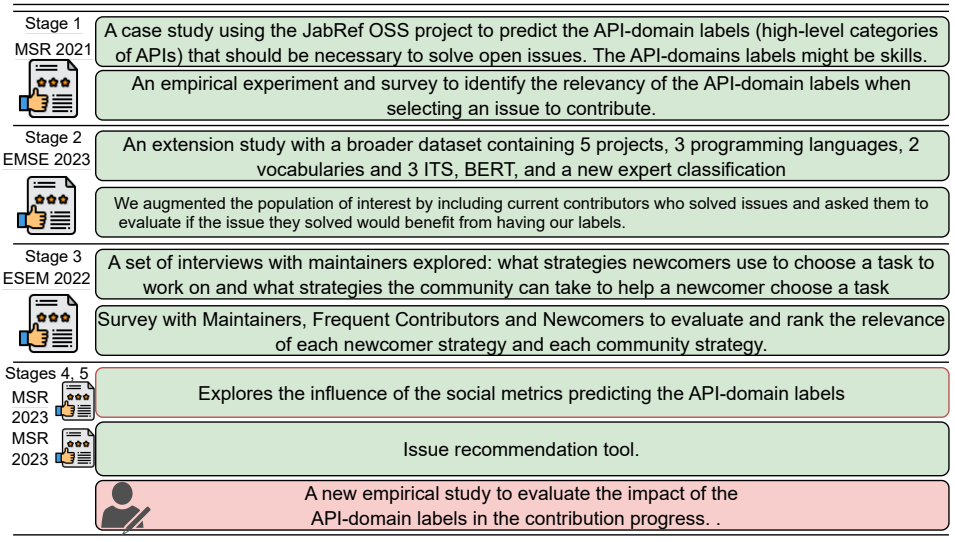

FIGURE 1.1: Research Method Overview. Green = published, Red = not yet published.

The study started with a case study (JabRef project), which identified skills based on API domains mined from GitHub issue texts. The API labels were predicted with precision,

recall, and F-measure of 0.755, 0.747, and 0.751, respectively, using the Random Forest algorithm [23] (Figure 1.1 - Stage 1, Chapter 4).

Next, we conducted a user experiment with 74 students and practitioners to assess the relevancy of the labels. The participants picked a task on a mocked page in which we inserted API-domain labels. They also reported which regions of the open issue page they perceived as relevant. Finally, they explained why the issue regions were relevant and what type of labels they would like to see in an ITS [23] (Figure 1.1 - Stage 1, Chapter 4). The participants found the labels information important to decide which issue to contribute and the API-domains labels more relevant than the components labels that are present in the project.

We extended the tooling to identify skills employed in the case to use a model trained with BERT, three different ITS (GitHub, Gerrit, and Jira), three programming languages (Java, C++, and C#), and five projects (JabRef, audacity, PowerToys, RTTS, and Cronos), including two from the industry. Extending to industry projects aims to verify whether stakeholders can apply the research to industry software. The improvements included a new semi-automatic API classification model carried out manually in the first experiment (Figure 1.1 - Stage 2, Chapter 5). We ran an experiment in a global organization with project developers to give us feedback about the labels.

Then, we interviewed 17 maintainers of the projects to learn *"how do newcomers choose an issue, and how can the community help?"*. (Figure 1.1 - Stage 3, Chapter 6). We also investigated mismatches of perceptions from diverse OSS community stakeholders regarding how to help new contributors find a task. The investigation included a survey with 64 participants who evaluated the newcomers' and maintainers' strategies proposed by the 17 interviewees. The study revealed differences in perceptions about the relative importance of onboarding strategies, including labeling issues [26].

We used the results from social network analysis to predict the API-domain labels [24]. We investigated the performance of the social metrics to improve the previously proposed API-domain labels. We found a set of predictors able to enhance the model's performance. Our results suggest that the conversations on issues convene discussants interested in or experts on those subjects. We substantially improve the API-domain label predictions up to 0.922 (precision), 0.978 (recall), and 0.942 (F-measure) using the conversation data from the projects (Figure 1.1 - Stage 4, Chapter 7).

A demonstration tool was implemented to suggest issues to contribute based on the contributor's skills provided in a web form [27] (Figure 1.1 - Stage 5, Chapter 9).

The API-domain labels were evaluated in a user study to assess the impact of the labels on the contribution process. Users picked an issue, and the researchers tracked the time, progress, and correctness of contributions by comparing two groups: users with access to the labels and users without access to the labels (Figure 1.1 - Stage 5, Chapter 8).

1.4 Publications

The research conducted as part of this dissertation resulted in the publications:

Santos, F., Vargovich, J., Trinkenreich, B., Santos, I., Penney, J., Britto, R., Pimentel, J.F., Wiese, I., Steinmacher, I., Sarma, A. and Gerosa, M.A., 2023, Apr. Tag that issue: Applying API-domain labels in issue tracking systems. In: *2023 Empirical Software Engineering (EMSE).*

Santos, F., Penney, J., Pimentel, J.F., Wiese, I., Steinmacher, I. and Gerosa, M.A., 2023, May. Tell Me Who Are You Talking to and I Will Tell You What Issues Need Your Skills. In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR).* IEEE.

Vargovich, J., Santos, F., Penney, J., Gerosa, M.A. and Steinmacher, I., 2023, May. GiveMeLabeledIssues: An Open Source Issue Recommendation System. In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR Data and Tool Showcase)*. IEEE.

Santos, F., Trinkenreich, B., Nicolati Pimentel, J.F., Wiese, I., Steinmacher, I., Sarma, A. and Gerosa, M.A., 2022, June. How to choose a task? Mismatches in perspectives of newcomers and existing contributors. In: *2022 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* . ACM/IEEE.

Santos, F., Trinkenreich, B., Wiese, I., Steinmacher, I., Sarma, A. and Gerosa, M.A., 2021, May. Can I Solve It? Identifying APIs Required to Complete OSS Tasks. In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (pp. 346-257)*. IEEE.

I also participated in the CHASE 2021 conference as WebChair, the Northern Arizona University three-minute poster competition (2021), EMSE 2022 doctoral symposium, and the ICSE 2023 Student competition.

CHAPTER 2: BACKGROUND

This chapter presents an overview of the related work, including the definition and representation of skills, which is the theoretical foundation for proposing API-domain labels.

2.1 Skills

When children are born, they have little to no knowledge of how to interact with their surroundings. As they grow older, they acquire the necessary skills to survive in an environment that was once unfamiliar and perilous. The environment plays a significant role in shaping individuals. Identifying the skills that aid in their growth and survival is essential for all individuals. However, there is no consensus regarding the factors contributing to skills' evolution. While researchers generally agree that both the individual and the environment impact the acquisition of skills, their relative importance and roles vary in the literature. Skinner [28] and Bandura and Walters [29] privileged the environment as the driver of skill development, while Piaget [30], Flavell [31], and Beilin [32] prioritized the organism (body, individual).

Skill theory is a field of study that examines the ordered and hierarchical development of skills [33]. According to Merriam-Webster, skills are "the ability to use one's knowledge effectively and readily in execution or performance." Skills theory models the progression, stages, and connections between developmental moments, behavior, and cognition concepts [33]. Hence, the theory provides an abstract representation of the structures of skills and their properties. In addition, a set of transformation rules associates the skills' structures [33]. Piaget, for example, identified eight stages in the learning sequence and created a formal model depicting them. Using this model, he established relationships such as equivalence and sequence, which can be applied to various skill categories [34].

According to Piaget's [33] skill theory, skills are organized into ten levels that increase in difficulty and three types: motor (sensory), representational, and abstract. Developing a new level of a particular skill involves making small advances through transformational procedures. Individuals combine skills from different levels, and their environment encourages them to progress until they reach their limit. However, as individuals develop, their limits may become overwhelming. The developmental process starts with motor skills and progresses through representational skills until the individual reaches the highest level of abstraction [33]. This level of abstraction encompasses all areas and skills, requiring tasks or individuals to be classified accordingly.

In our study, we are particularly interested in mapping the structure of individual skills to tasks or issues to aid OSS contributors in identifying an appropriate starting point. By identifying and organizing the skills needed for a given task, it becomes easier to match tasks with individuals possessing the necessary skills.

2.1.1 Skill and Barriers

According to [35], OSS contributors experience difficulty in finding a task with which to start. As identified by [35], the root sources of this difficulty include a lack of confidence in choosing a task, issues with the information provided in the task descriptions, outdated task descriptions, and a lack of information about required skills and difficulty level. Additionally, some contributors may choose tasks that are too difficult.A lack of documentation and outdated tutorials may also present as barriers. This suggests a shared responsibility between contributors and the tasks page. The tasks page may not provide enough information to clarify the skills needed to solve a task. A potential solution to supporting newcomers in finding a suitable task involves enhancing the task page to provide more information about the task, enabling newcomers to evaluate whether it aligns with their skill set. Given the diverse concepts involved in skills, it would be challenging

to organize and exhibit the skills to newcomers. Mining the repositories, we may find several properties and relationships that may define the skills. Furthermore, skills can be combined and have varying degrees of importance for different tasks.

2.2 Collaboration in Software Engineering

Collaboration plays a crucial role in software development, especially when practitioners work in teams or projects [36]. With the emergence of open-source software and its communities, collaboration has transcended organizational structures, hierarchies, and formal processes [37]. Although this new way of organizing and conducting software engineering projects has presented challenges, it has also provided opportunities. Initially, version control systems were primarily used to sustain updates, but they have since evolved to include integrated electronic communication features, such as chats and messaging tools [38]. This phenomenon has facilitated the collection of communication data, which can be analyzed to provide general feedback about project issues [19], human resources participation [39], and possible training models for prediction [22]. Therefore, contributors' participation can reveal software aspects that are often hidden from view.

2.3 Data Mining

The process of data mining typically involves several steps: data cleaning, data integration, data reduction, data transformation, data mining, pattern evaluation, and knowledge representation [40, 41]. According to [41], data cleaning is the elimination of noisy and irrelevant data from the dataset. The combination of heterogeneous data from multiple sources is called data integration. After cleaning and integrating, data reduction retrieves relevant data for analysis, which may involve machine learning techniques. Sometimes a data transformation phase is needed to convert data into the appropriate format,

and a data mining phase applies intelligent techniques to extract patterns. Finally, pattern evaluation identifies patterns that potentially represent knowledge [41].

In our research context, we are interested in mining the software repositories content (text and conversations) to indicate skills needed to solve open issues in ITS used by OSS communities (our environment). We aim to extract skills from the repositories by analyzing data from five primary sources: issues, commits/pull requests, past contributors, discussions, and source codes. To ensure accuracy, we cleaned the data and only kept consistent information. Combining the data from these sources through integration helps establish meaningful connections. After selecting the most relevant data, we transformed its format to improve computation and reveal patterns. Identifying these patterns confirms that the mined data represents a skill.

2.4 Related Work

Studies that share similarities to our work can be classified into four main categories: support task selection (API related); support task selection (understanding); automatic issue labeling; and expert identification (developer-based).

2.4.1 Support Task Selection (API related)

APIs can provide relevant information for selecting tasks. Recent work on APIs focuses on understanding the crowd's opinion about API usage [42], understanding and categorizing the API discussion [43], creating tutorial sections that explain a given API type [44], generating/understanding API documentation [45, 46], providing API recommendations [47–49], offering API tips to developers [50], and defining the skill space for APIs, developers, and projects [51]. Table 2.1 provides an overview of these works.

TABLE 2.1: Articles from the Literature Review - API

Article	Summary	Our Work
[47]	Identify API methods to solve tasks	Identify API domains to solve issues
[49]	Identify errors in Java documentation based on parameters and exceptions and propose fixes	Identify API domains to solve issues
[50]	Extract tips from answers to the questions relevant to 48 frequently used PHP API methods	Extract API from source code to define and predict API domains
[43]	Categorize discussions involving APIs	Tag API domains in issues
[44]	Expand API documentation	Mine APIs in source code
[45]	Mine Stack Overflow entries to add tips about API documentation	Mine ITS to label issues with API domains
[42]	Identify and categorize opinions about APIs in Java	Use APIs to categorize into domains.
[48]	Categorize use of APIs	Identify API domains to solve issues

The focus of these studies is to support the developers' decisions to adopt a new API. In this work, we have a complementary goal. Given that projects already have APIs in different domains, our goal is to enable developers to find tasks that include APIs with which they are more familiar—we focus on predicting the domain of the API used in the code that fixes an issue.

2.4.2 Support Task Selection (understanding)

Previous work that focused on supporting task selection identify essential data to start a contribution. They also propose or create collaborative environments or queries to find information to increase newcomers' awareness. [52] query bug selections. [53] provide information about project history, contributors, and packages [53]. [54] classified data from the README files and used eight categories of information that should be present to help new contributors to the project. [55] studied contribution guidelines to evaluate the corresponding process used in GitHub contributions. Finally, episodic contributions were studied in [56] to propose ways to manage and retain episodic volunteers (Table 2.2). While these approaches indicate ways to assist new contributors, they are as limited in the outcome as the approaches that only offer additional information about the contribution process, files, and assets instead of assisting newcomers with where to start. In contrast, by labeling issues with domains of the APIs, our approach can support new contributors

TABLE 2.2: Articles from the Literature Review - Understanding the Issue

Article	Summary	Our Work
[57]	Enable users to search for a bug or feature and identify related bugs, relevant resources	Tag issues with API domains should be in a possible solution
[58]	Investigate the information needs of newcomers and benefits of information visualization in supporting newcomers	Investigate the information should be present in issues using framework 5W2H and evaluate regions of issues pages newcomers prefer to decide what issue to contribute
[54]	Parse readme files and tag with categories	Use code files to parse APIs used to define and predict the API domains
[55]	Use contribution.md files to evaluate the contribution process	Use code files to parse APIs used to define and predict the API domains
[59]	Identified 65 practices to manage Epicotic Volunteers and 16 concerns from maintainers	Identified 7 groups of communities strategies and 5 groups of newcomers strategies to find a task to start contribute

of different skills to find a task with which to work. Table 2.2 presents an overview of these related works.

2.4.3 Expert Identification

Studies related to expert identification aim to explore developer's interests and technical skills by mining data activity from Stack Overflow and GitHub (Q&A). The goals vary from measuring individuals' performance on a set of programming tasks or extracting expertise using traces of contributions data (from source code, bug report data, merged pull requests).

To identify experts, the literature presents techniques based on general metrics, such as developer interests (e.g., what they program in their free time) and technical skills (e.g., programming languages they are using) [60], developer activity from Stack Overflow and GitHub [61], and individual performance on programming tasks [62]. Some techniques use traces from source code [63, 64], bug report data [65], and merging pull requests data [66] to rank developers who are the most appropriate to review code, merge a pull request, or comment on an issue. Table 2.3 presents an overview of these works. While some studies use pieces of the source code and historical contributions to associate experts with activities, they do not aim to match newcomers to tasks when onboarding

TABLE 2.3: Articles from the Literature Review - Expert Recommendation

Article	Summary	Our Work
[60]	Identify developers' interest by mining historical contributions	Mine issues, pull requests and discussions to predict API domains
[61]	Developer's activity from Stack Overflow and GitHub to evaluate expertise	Mine developers' discussions to predict API domains
[62]	Developer's performance on programming tasks to derive skills	Developer's discussions to predict API domains
[67]	Identify experts by mining APIs in source code and frameworks	Identify issues API domains mining source code
[64]	Set levels for recommenders to indicate them to future issues	Automatically predict API domains
[65]	Recommend developers to fix issues, interested in the issues and components related to the issues using the issue Title and Description with normalized frequencies of the words	We label issues with API-domains to assit newcomers to find an issue using Title, Body, Comments TF-IDF, Doc2Vec and Bert
[68]	Rank developers who are the most appropriate to review code, merge a pull request, comment or an issue	Rank strategies OSS communities use to onboard newcomers and newcomers use to pick an issue

open-source projects and they are not intended to recommend issues at the API domain granularity level.

2.4.4 Automatic Issue Labeling

New contributors need specific guidance on what to contribute [6, 53]. In particular, finding a relevant issue can be daunting and discourage contributors [35]. Social coding platforms like GitHub[1] and other communities (e.g. LibreOffice,[2] KDE,[3] and Mozilla[4]) encourage projects to label issues that are easy for new contributors. However, community managers argue that manually labeling issues is difficult and time-consuming [7].

Several studies have proposed automatic ways to label issues as bug/non-bug, combining text mining techniques with classification to mitigate this problem. For example, Antoniol et al. [8] compared text-based mining with naive Bayes (NB), logistic regression (LR), and decision trees (DT) to process data from titles, descriptions, and discussions

[1] `http://bit.ly/NewToOSS`http://bit.ly/NewToOSS
[2] `https://wiki.documentfoundation.org/Development/EasyHacks`
[3] `https://community.kde.org/KDE/Junior_Jobs`
[4] `https://wiki.mozilla.org/Good_first_bug`

and achieved a recall up to 82%. Pingclasai et al. [13] used the same techniques to compare a topic versus a word-based approach, finding F-measures from 0.68 to 0.81 using the topic-based approach. More recently, Zhou et al. [15] used two-stage processing, introducing structured information from the issue tracker and improving the recall obtained by Antoniol et al. [8]. Kallis et al. [10] simplified the data mining step to produce a tool to classify issues on demand. They used the title and body to create a bag of words to classify issues as "bug report," "enhancement," or "question." El Zanaty et al. [69] applied type detection on issues and attempted to transfer learning to other projects using the same training data. The best results had F-measures around 0.64 - 0.75. Finally, Xia et al. [14] employed a multi-label classification using text data from Stack Overflow questions, obtaining recall from 0.59 to 0.77. Table 2.4 shows an overview of these related works.

Despite being able to classify tasks into distinct labels, previous work only use pre-existing labels related to the types of issues. Instead of using predefined labels extracted from the existing issues or provided by default on GitHub, our approach defines labels based on API domains. This kind of labeling can guide new contributors toward what to contribute, which can be a daunting task without guidance [35].

Table 2.4: Articles from the Literature Review - Automatic Labeling

Article	Summary	Our Work
[8]	Mine 3 projects using TF-IDF to tag as bug/non-bug and 3 classifiers	Mine 5 projects to tag with API domains using TF-IDF, Doc2Vec and BERT and 5 different classifiers
[13]	Compared topics and NLP approaches using 3 classifiers to tag with bug/non-bug	Use 5 classifiers tag with API-domain labels comparing TF-IDF, Doc2Vec and Bert
[15]	Two-stage classification tagging with bug/non-bug using naive Bayes	One classifier tags the API-domain labels
[14]	Multilabel classification using text data from software information sites and Stack Overflow questions: bug/feature/question	Multilabel classification using API-domain labels
[69]	Categorize issues into bug/non-bug using TF-IDF and 4 classifiers	Categorize issues into 31 API domains using TF-IDF, Doc2Vec and Bert
[10]	Tag bug/non-bug/feature using Title and Body with fastText algorithm	Use Title, Body, Comments and TF-IDF, Doc2Vec, Bert to tag API domains
[70]	Collect data from issues labeled with: bug, duplicate, enhancement, help wanted, invalid, question, wont fix regarding the label influence and usage	Labels issues with 32 different API domains
[71]	Recommend similar issues regarding 15 possible labels. Uses Word2Vec and Neighborhood-Based Node Embeddings	Tag issues with API-domain labels using TF-IDF, Doc2Vec or Bert
[47]	Identify API methods to solve tasks	Identify API domains to solve issues
[49]	Identify errors in Java documentation based on parameters, exceptions and propose fixes	Identify API domains to solve issues
[50]	Extract tips from answers to questions about frequently used API methods	Extract API from source code to define and predict API domains
[43]	Categorize discussions involving APIs	Tag API domains in issues
[44]	Expand API documentation	Mine APIs in source code
[45]	Mine Stack Overflow entries to add tips about API documentation	Mine ITS to label issues with API domains
[46]	Two-stage classification tagging with bug/non-bug using naive Bayes	One classifier tags the API-domain labels
[51]	Define skills space to match projects, API and developers	Define API domains to help newcomers find a task

TABLE 2.5: Articles from the Literature Review - Labeling with Machine Learning

Article	**Algorithms**	**Metrics**	**NLP**	**#Projects**	**Dataset Rows**	**Predictions**	**Features**	**Cleaning**	**Baseline**
[8]	DT, NB, LR	P, R	Raw frequency	3	>1,800	Bug/non-bug	Title, description, discussion	Punctuation, stemming	Grep
[13]	ADTree, NB, LR	F	Topic modeling	3	>5,000	Bug/non-bug	Title, description, discussion	HTML tags, tokenization, stemming	WordVector
[15]	MNB, Bayesian Net	P, R, F	Likelihood	5	3,200	Bug/non-bug	Textual summary, severity, priority, component, assignee	Tokenization, stop word stemming	Multinomial NB, previous studies
[14]	Multinomial Naive Bayes	R5, R10	TF-IDF	2 ITS	>7,000	Existing tags	Bag of words	Tokenization, filtering threshold, stop word, stemming	Previous study
[69]	KNN, MNB, SVC, MLP	P, R, F, AUC	TF-IDF	3	<1,000	Bug/non-bug	Title, body	Stop words, stemming	Random
[10]	fastText	P, R, F	Bag of words	12,112	30,000	Bug, question, enhacement	Title, body	Tokenization	NA
[71]	Similarity function	ND	NBNE, Word2Vec	50	628,298	Similar labels	Labels	NA	Existing labels
[23] [25] [24]	LR, DT, MLP, MlKNN, RF, Bert	P, R, F, Hla	TF-IDF, Bert, Doc2Vec, social metrics, APIs	5	22,231	31 API domains	Title, description, comments social metrics	Punctuation Templates, URLs, HTML tags, XML tags, stop words, stemming	Dummy previous studies

Table 2.5 presents the machine learning techniques used to label issues. Most articles evaluate the approaches using precision (P), recall (R), and F-measure (F). We also found studies employing the area under the curve (AUC) and hamming loss (Hla). The baselines used are mainly previous studies or another elected technique. The majority use the title and body as a corpus. The number of projects, dataset size, ML algorithms (DT - Decision Tree, ADTree - Advanced DT, NB - Naive Bayes, RF - Random Forest, MlKNN - Multilabel K Nearest Neighbors, MLP - Multilayer Perceptron, MNB - Multinomial Naive Bayes, SVC - Support Vector Machine, LR - Logistic Regression), and cleaning procedures vary. The baselines encompass previous studies, random, dummy classifiers, and other techniques like grep unix command, Multinomial NB, Word2Vec, or similarity with existing labels.

As opposed to these related works—which primarily focus on classifying the type of issue (e.g., bug/non-bug)—our work focuses on identifying the domain of APIs used in the implementation code, using issue descriptions and conversation from the ITS, which might reflect skills needed to complete a task. We also evaluate the relevance of the proposed approach with students, industry practitioners, and developers from OSS projects and industry.

2.4.5 Using social metrics in prediction models

Social metrics have been the object of software engineering research. In previous work, researchers mined data from discussions in issue trackers to predict co-changes [19], categorized studies on how social networks impact software quality [72], and explained software characteristics based on organizational structure [73]. For example, Wiese et al. [19] mined social data to build a model to predict co-changes of source code files, obtaining low rates of false negatives and false positives and accuracy from 0.89 to 1.00. The work of Kikas et al. [74] predicted issue lifetime based on social metrics, such as the number of

comments and actors. They mined data from 4000 projects and found that social metrics complement the textual description of titles and bodies. Differently from these studies, we propose to enrich the set of predictors used in Santos et al.'s [23] study to improve label prediction.

CHAPTER 3: RESEARCH DESIGN OVERVIEW

This chapter presents this dissertation's research plan overview as shown in Figure 3.1. Methodology and results are detailed in each study chapter.

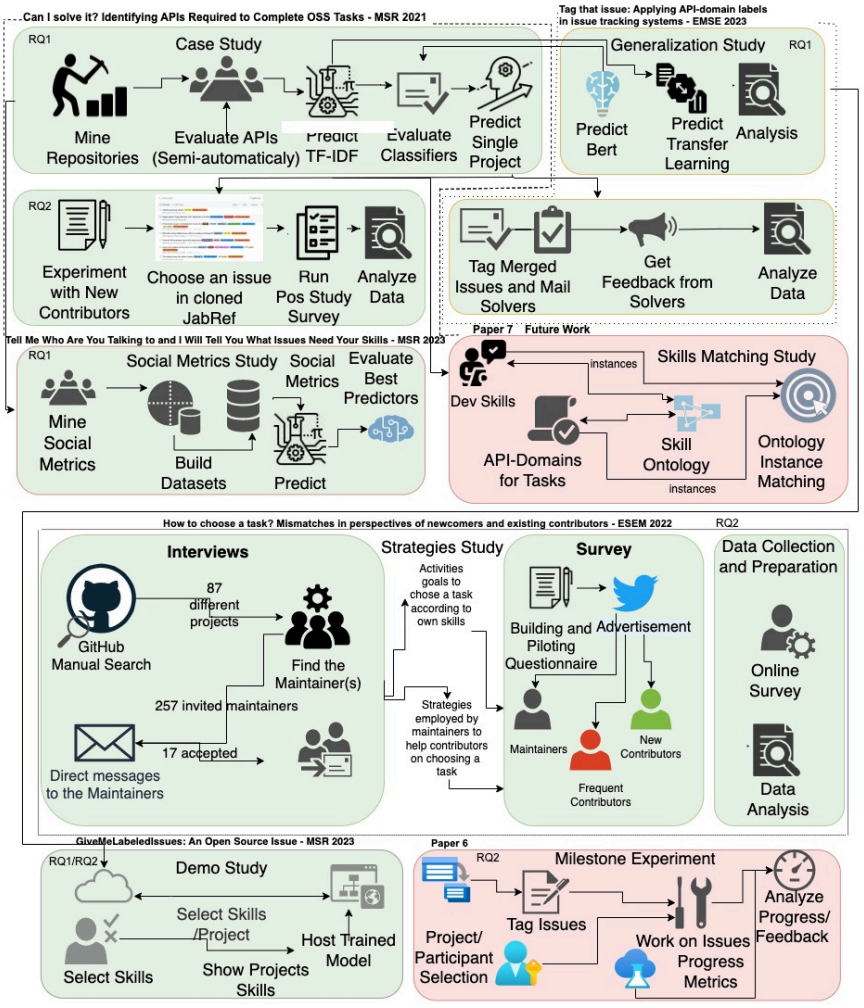

FIGURE 3.1: The Research Design Overview

Stage 1 comprised an exploratory study about labeling APIs in the JabRef project, verifying its relevancy. Stage 2 embraced the extension of the JabRef case study to generalize the research. More projects with different programming languages, natural languages,

and issue-tracking systems were added to the machine-learning model. Next, stage 3, is a study to identify mismatches between the maintainers' and contributors' views about strategies adopted in projects and adopted by contributors to raise awareness of the skills needed to complete an OSS task. Stage 4 explores using features derived from conversations in the issues track systems as predictors to the APIs. Finally, stage 5 presents the evaluation of the defined labels, in a timed empirical experiment with contributors. We also present a demonstration tool that labels the issues in real-time. Future work proposes matching contributors' and authors' skills.

CHAPTER 4: CASE STUDY

Finding tasks to contribute to in Open Source Software (OSS) projects is challenging [2, 35, 57, 75, 76]. Open tasks vary in complexity and required skills, which can be difficult to determine by reading the task descriptions alone, especially for new contributors [3–5]. Adding labels to the issues (a.k.a tasks, bug reports) helps new contributors choose tasks [6]. However, community managers find issue labeling challenging and time-consuming [7] because projects require skills in different languages, frameworks, databases, and Application Programming Interfaces (APIs).

APIs usually encapsulate modules with specific purposes (e.g., cryptography, database access, logging, etc.), abstracting the underlying implementation. If contributors know which types of APIs will be required to work on each issue, they can choose tasks that better match their skills or involve skills they want to learn.

To facilitate contributors' task selection, we investigate the feasibility of automatically labeling issues with domains of APIs. Since an issue may require knowledge in multiple APIs, we applied a multi-label classification approach, which has been used in software engineering for classifying questions in Stack Overflow (e.g., [14]) and detecting types of failures (e.g., [77]) and code smells (e.g., [78]).

Employing an exploratory case study and a user study, we aimed to answer the following research questions:

RQ1: To what extent can we predict the domain of APIs used in the code that fixes a software issue? To answer RQ1, we employed a multi-label classification approach to predict the API-domain labels. We also explored the influence of task elements (i.e., title, body, and comments) and machine learning setup (i.e., n-grams and different algorithms) on the prediction model. Overall, we found that pre-processing the issue body using unigram and the Random Forest algorithm can predict the API-domain

labels with up to 82% precision and up to 97.8% recall. This configuration outperformed recent approaches reported in the literature [69].

RQ2. How relevant are the API-domain labels to new contributors? To answer RQ2, we conducted a study with 74 participants from both academia and industry. After asking participants to select and rank real issues to which they would like to contribute, we provided a follow-up survey to determine what information was relevant to make the decision. We compared answers from the treatment group (with access to the API-domain labels) to the control group (who used only the pre-existing project labels). The participants considered API-domain labels more relevant than the project labels—with a large effect size.

4.1 Method

The JabRef case study had two phases. In phase 1, we mined the JabRef repository, categorized the APIs, built the corpus, and ran the classifiers to predict the API-domain labels (Figure 4.1).

In phase 2, we ran an empirical experiment to evaluate the API-domain labels. We collected the experiment data by surveying the participants. Finally, we quantitatively analyzed the data to evaluate the labels' relevancy.

4.1.1 Mining Software Repository

We used the GitHub API to collect data from JabRef. The data was collected in April 2020. Next, we filtered out open issues and pull requests not explicitly linked to issues. Our final dataset comprises 705 issues and their corresponding pull requests.

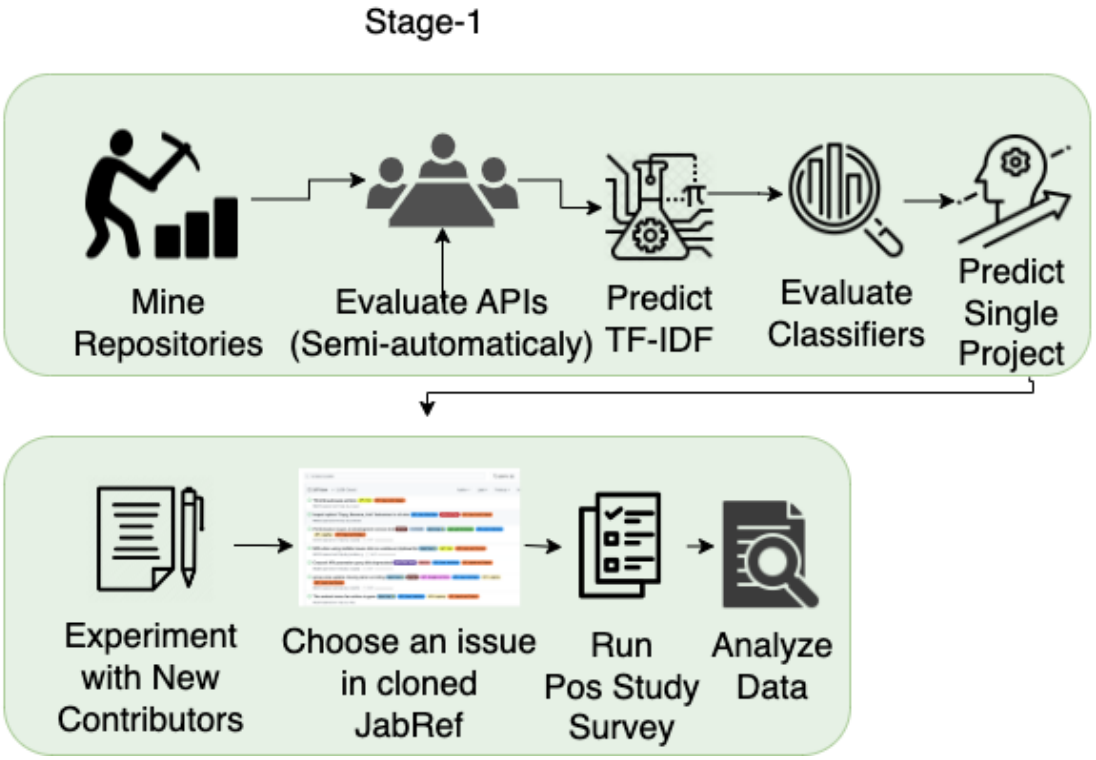

FIGURE 4.1: The Research Method - Stage 1 - Case Study

We also wrote a parser to process all Java files from the project. In total, 1,692 import statements from 1,472 java sources were mapped to 796 distinct APIs.

4.1.2 API's Categorization

We employed a card-sorting approach to manually classify the imported APIs into higher-level categories based on the API's domain. For instance, we classified "java.nio.x" as "IO', "java.sql.x" as "Database", and "java.util.x" as "Utils"'. A three-member team performed this classification (first, second, and fourth authors), one of whom is a contributor of JabRef, and another who is an expert Java developer.

4.1.3 Corpus Construction

To build our classification models, we created a corpus comprising issue titles, body, and comments. We converted each word to lowercase and removed URLs, source code, numbers, and punctuation. After that, we removed stop-words and stemmed the words.

Next, similar to other studies [79–81], we applied TF-IDF, which is a technique for quantifying word importance in documents by assigning a weight to each word.

We split the data into training and test sets using the ShuffleSplit method [82], which is a model selection technique that emulates cross-validation for multi-label classifiers.

4.1.4 Classifiers

To create the classification models, we chose five classifiers that work with the multi-label approach and implement different strategies to create learning models: Decision Tree, Random Forest (ensemble classifier), MLPC Classifier (neural network multilayer perceptron), MLkNN (multi-label lazy learning approach based on the traditional K-nearest neighbor algorithm) [82, 83], and Logistic Regression.

We chose these five algorithms because they support multi-label classification and implement different strategies to create learning models. For instance, MLKNN is a KNN variation of the multi-label problem, MPLC is based on a neural network, and Random Forest is an ensemble classifier. The work from [84] summarized the main algorithms to multi-label classification. We tested different classifiers used in many software engineering articles [77, 78, 85], embracing a representative set of them using a grid search to cover their hyperparameters. Per [86], the RF algorithm outperformed the SVM and we

considered using the MLPC instead of the SVM. To simplify the comparison, we adopted the sklearn library using the algorithms available there [1].

4.1.5 Data Analysis

To conduct the data analysis, we used precision, recall, f-measure, and hamming loss as metrics, and logged the confusion matrix after each model's execution.

We tested different inputs and compared them to TITLE only; all alternative settings provided better results. Next, we investigated the use of bi-grams, tri-grams, and four-grams, comparing the results to the use of unigrams.

Finally, to investigate the influence of the machine learning (ML) classifier, we compared several options using the title with unigrams as a corpus. The options included: random forest (RF), neural network multilayer perceptron (MLPC), decision tree (DT), LR, MlKNN, and a dummy classifier with the strategy "most_frequent". Dummy or random classifiers are often used as a baseline [87, 88].

4.1.6 Empirical Experiment

To evaluate the relevancy of the API-domain labels from a new contributor's perspective, we conducted an empirical experiment with 74 participants. We created two versions of the JabRef issues page (with and without our labels) and divided our participants into two groups (between-subjects design). We asked participants to choose and rank three issues to which they would like to contribute and answer a follow-up survey about what information supported their decision. The artifacts used in this phase are also part of the replication package.

[1] `http://scikit.ml/index.html`

Table 4.1: User Experiment and Survey Demographics

Role	#	%	**Affinity with JabRef technology**	#	%
Practitioner	41	55.5	High Affinity	31	42
Student	33	44.5	Low Affinity	43	58
Experience as Developer	#	%	**Experience as OSS Contributor**	#	%
Experienced	28	38	Experienced	10	13.5
Novice	46	62	Novice	64	86.5

4.1.7 Participants

We recruited participants from both industry and academia. We reached out to our students, in addition to instructors and IT managers of our personal and professional networks, and asked them to help in inviting participants. From industry, we recruit participants from one medium-sized IT startup hosted in Brazil and the IT department of a large global company. Students included undergraduate and graduate computer science students from one university in the US and two in Brazil. We also recruited graduate data science students from a university in Brazil, since they are also potential contributors to the JabRef project. We present the demographics of the participants in Table 4.1. We offered an Amazon Gift card to incentivize participation.

We categorized the participants' development tenure into novice and experienced coders, splitting our sample in half—below and above the average "years as professional developer" (4). We also segmented the participants between industry practitioners and students. Participants are identified by the letter "P," followed by a sequential number and a character representing the location where they were recruited (university: U & industry: I).

Table 4.2: Demographics Subgroups for the Experiment's Participants

Population	**Quantity**	**Percentage**	**Tenure**	**Quantity**	**Percentage**
Industry	41	55.5	Expert	19	25.7
Student	33	44.5	Novice	55	74.3

The participants were randomly split into two groups: control and treatment ("C" and "T"). From the 120 participants that started the survey, 74 (61.7%) finished all the steps, and we only considered these participants in the analysis. We ended up with 33 and 41 participants in the Control and Treatment groups, respectively (Table 4.2).

4.1.8 Experiment Planning

We selected 22 existing JabRef issues and built mock GitHub pages for control and treatment groups. The issues were selected from the most recent ones, aiming to maintain similar distributions of the number of API-domain labels predicted per issue and the counts of predicted API-domain labels. The control group mocked page had only the original labels from the JabRef issues, and the treatment group mocked page presented the original labels in addition to the API-domain labels.

4.1.9 Survey Data Collection

The survey included the following questions/instructions:

- Select the three issues that you would like to work on.
- Select the information (region) from the issue page that helped you decide which issues to select (Fig: 4.2).
- Why is the information you selected relevant? (open-ended question)
- Select the labels you considered relevant for choosing the three issues

The survey also asked about participants' experience level, experience as an OSS contributor, and expertise level in the technologies used in JabRef.

FIGURE 4.2: Survey question about the region's relevance

Fig. 4.2 shows an example of an issue details page and an issue entry on an issue list page. After selecting the issues to which to contribute, the participant was presented with this page to select what information (region) was relevant to the previous issue selection.

4.1.10 Data Analysis

Next, to understand participants' perceptions about what information (regions) they considered important and the relevancy of the API-domain labels, we first compared treatment and control groups' results. We used violin plots to visually compare the distributions and measured the effect size using the Cliff's Delta test.

Then, we analyzed the data, aggregating participants according to their demographic information. We calculated the odds ratio to check how likely it would be to elicit similar responses from both groups.

4.2 Results

We report the results grouped by research question.

4.2.1 RQ1. To what extent can we predict the domain of APIs used in the code that fixes a software issue?

To predict the API domains used in the files changed in an issue (RQ1), we tested a simple configuration used as a baseline. For this baseline model, we used only the issue TITLE as input and the Random Forest (RF) algorithm, since is insensitive to parameter settings [89] and has shown to yield good prediction results in software engineering studies [90–93]. Then, we evaluated the corpus configuration alternatives, varying the input

information: only TITLE (T), only BODY (B), TITLE and BODY (T, B), and TITLE, BODY, and COMMENTS (T, B, Comments). To compare the different models, we selected the best Random Forest configuration and used the Mann-Whitney U test with the Cliff's-delta effect size.

We also tested alternative configurations using n-grams. For each step, the best configuration was kept. Then, we used different machine learning algorithms compared with a dummy (random) classifier.

TABLE 4.3: overall metrics (section III-B-4) from models created to evaluate the corpus. Hla - Hamming Loss

Model	**Precision**	**Recall**	**F-measure**	**Hla**
Title (T)	0.717	0.701	0.709	0.161
Body (B)	0.752	0.742	0.747	0.143
T, B	0.751	0.738	0.744	0.145
T, B, Comments	0.755	0.747	0.751	0.142

As Table 4.3 shows, when we tested different inputs and compared them to TITLE only, all alternative settings provided better results. We could observe improvements in terms of precision, recall, and F-measure. When using TITLE, BODY, and COMMENTS, we reached a precision of 75.5%, recall of 74.7%, and F-measure of 75.1%.

TABLE 4.4: Cliff's Delta for F-measure and Precision: comparison of corpus models alternatives - Section III-B-1. Title(T), Body(B) and Comments (C).

Corpus Comparison	**Cliff's delta**			
	F-measure		**Precision**	
T versus B	-0.86	large***	-0.92	large***
T versus T+B	-0.8	large**	-0.88	large***
T versus T+B+C	-0.88	large**	-0.88	large***
B versus T+B	0.04	negligible	0.04	negligible
B versus T+B+C	-0.24	small	-0.12	negligible
T+B versus T+B+C	-0.3	small	-0.08	negligible

** $p \leq 0.05$; ** $p \leq 0.01$; *** $p \leq 0.001$*

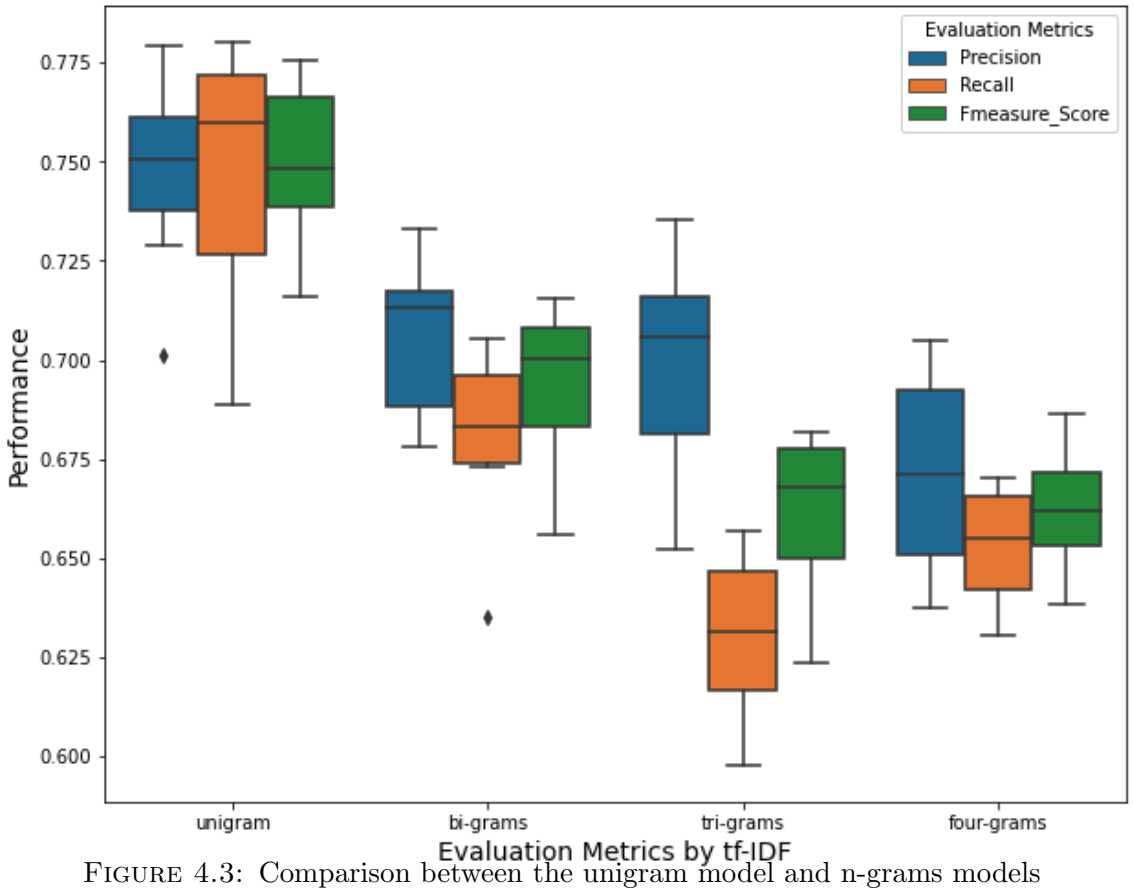

FIGURE 4.3: Comparison between the unigram model and n-grams models

We found statistical differences comparing the results using TITLE only and all three other corpus configurations for both F-measure (p-value ≤ 0.01 for all cases, Mann-Whitney U test) and precision (p-value ≤ 0.001 for all cases, Mann-Whitney U test) with large effect size. TITLE+BODY+COMMENTS performed better than all others in terms of precision, recall, and F-measure. However, the results suggest that using only the BODY would provide good enough outcomes, since there was no statistically significant difference compared to the other two configurations (using TITLE and/or COMMENTS in addition to the BODY), and it achieved similar results with less effort. The model built using only BODY presented only 14.3% incorrect predictions (hamming loss metric) for all 12 labels. Table 4.4 shows the Cliff's-delta comparison between each pair of corpus configurations.

Next, we investigated using bi-grams, tri-grams, and four-grams, comparing the results to unigrams. We used the corpus with only issue BODY for this analysis, since this

configuration performed well in the last step. Fig. 4.3 and Table 4.5 present how the Random Forest model performs for each n-gram configuration. The unigram configuration outperformed the others with a large effect size.

TABLE 4.5: Cliff's Delta for F-measure and Precision: Comparison between n-grams models - Section III-B-5

n-Grams Comparison	**Cliff's delta**			
	F-measure		**Precision**	
1 versus 2	1.0	large***	0.86	large***
1 versus 3	1.0	large***	0.84	large***
1 versus 4	1.0	large***	0.96	large***
2 versus 3	0.8	large**	0.18	small
2 versus 4	0.78	large**	0.72	large**
3 versus 4	0.12	negligible	0.62	large*

** $p \leq 0.05$; ** $p \leq 0.01$; *** $p \leq 0.001$*

Finally, to investigate the influence of the machine learning (ML) classifier, we compared several options using the title with unigrams as a corpus. The options included: Random Forest (RF), Neural Network Multilayer Perceptron (MLPC), Decision Tree (DT), LR, MlKNN, and a dummy classifier with strategy "most_frequent." Dummy or random classifiers are often used as a baseline [87, 88]. We used the implementation from the Python package scikit-learn [2].Fig. 4.4 shows the comparison among the algorithms, and Table 4.6 presents the pair-wise statistical results comparing F-measure and precision using Cliff's delta.

Random Forest (RF) and Neural Network Multilayer Perceptron (MLPC) were the best models when compared to Decision Tree (DT), Logistic Regression (LR), MlKNN, and Dummy algorithms. Random Forest outperformed these four algorithms with large effect sizes considering F-measure and precision.

[2] `http://scikit.ml/index.html`

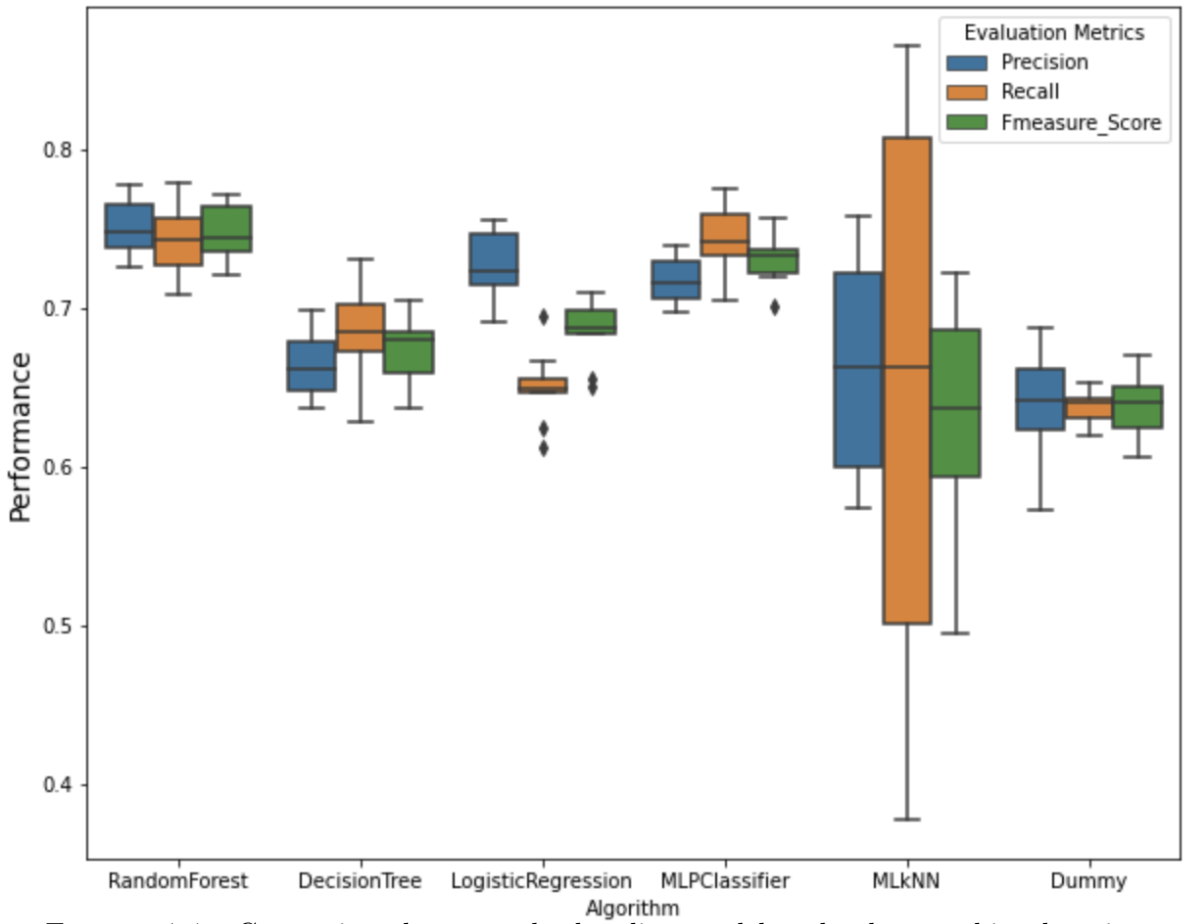

FIGURE 4.4: Comparison between the baseline model and other machine learning algorithms

RQ1 Summary. It is possible to predict the API-domain labels with a precision of 0.755, recall of 0.747, F-measure of 0.751, and 0.142 of Hamming loss using the Random Forest algorithm, TITLE, BODY and COMMENTS as the corpus, and unigrams.

4.2.2 RQ2. How relevant are the API-domain labels to new contributors?

To answer this research question, we conducted an empirical experiment with 74 participants and analyzed their responses to a survey.

TABLE 4.6: Cliff's Delta for F-measure and Precision: Comparison between machine learning algorithms - Section III-B-5

Algorithms Comparison	**Cliff's delta**			
	F-measure		**Precision**	
RF versus LR	1.0	large***	0.62	large*
RF versus MLPC	0.54	large*	0.88	large***
RF versus DT	1.0	large***	1.0	large***
RF versus MlkNN	0.98	large***	0.78	large***
LR versus MLPC	-0.96	large***	0.24	small
LR versus DT	0.4	medium	0.94	large***
LR versus MlkNN	0.5	large*	0.48	large*
MPLC versus DT	0.98	large***	0.98	large***
MPLC vs. MlkNN	0.94	large***	0.32	small
MlkNN versus DT	-0.28	small	0.0	negligible
RF versus Dummy	1.0	large***	1.0	large***

** $p \leq 0.05$; ** $p \leq 0.01$; *** $p \leq 0.001$*

What information is used when selecting a task? Understanding the type of information that participants use to select an issue on which to work can help projects better organize such information on their issue pages. Fig. 4.5 shows the different regions that participants found useful. In the control group, the top two regions of interest included the body of the issue (75.7%) and the title (78.7%), followed by the labels (54.5%) and then the code itself (54.5%). This suggests that the labels generated by the project were only marginally useful, and participants also had to review the code. In contrast, in the treatment group, the top four regions of interest by priority were: title, label, body, and then code (97.5%, 82.9%, 70.7%, 56.1%, respectively). This shows that participants in the treatment group found the labels more valuable than the participants in the control group: 82.9% usage in the treatment group as compared to 54.5% in the control group. Comparing the body and the label regions in both groups, we found that participants from the treatment group selected 1.6x more labels than the control group (p=0.05). The odds ratio analysis suggests labels were more relevant in the treatment group.

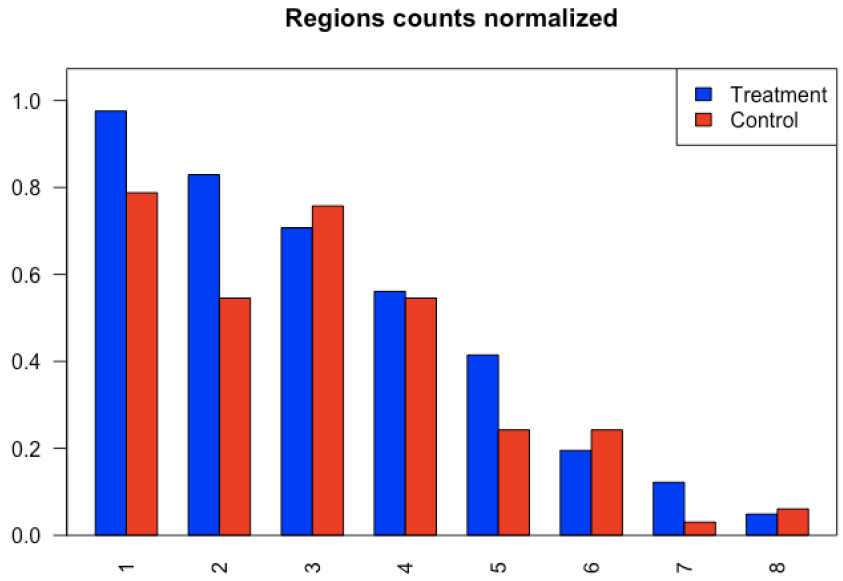

FIGURE 4.5: The region counts (normalized) of the issue's information page selected as most relevant by participants from treatment and control groups. 1-Title,2-Label,3-Body,4-Code,5-Comments,6-Author,7-Linked issues,8-Participants.

Qualitative analysis of the reason for the participants' choice of regions on the issue page in the treatment group reveals that the title and the labels together provided a comprehensive view of the issue. For instance, P4IT mentioned: *"labels were useful to know the problem area and after reading the title of the issues, it was the first thing taken into consideration, even before opening to check the details"*. Participants found the labels helpful in pointing out the specific topic of the issue, as P14IT stated: *"[labels are] hints about what areas have a connection with the problem occurring"*.

What is the role of labels? We also investigated which type of labels helped the participants in their decision-making. We divided the labels available to our participants into three groups based on the type of information they imparted.

- Issue type (already existing in the project): This included information about the type of task: bug, enhancement, feature, good first issue, and GSoC.

- Code component (already existing in the project): This included information about the specific code components of JabRef: Entry, Groups, External.Files, Main Table, Fetcher, Entry.Editor, Preferences, Import, Keywords.
- API-domain (new labels): the labels generated by our classifier (IO, UI, network, security, etc.). These labels were available only to the treatment group.

TABLE 4.7: label distributions among the control and treatment groups

Type of Label	Control	C %	Treatment	T %
Issue Type	145	56.4	168	36.8
Components	112	43.6	94	20.6
API Domain	-	-	195	42.7

Table 4.7 compares the labels that participants considered relevant (section III-C-3) across the treatment and control groups distributed across these label types. In the control group, a majority of selected labels (56.4%) relate to the type of issue (e.g., Bug or Enhancement). In the treatment group, however, this number drops down to 36.8%, with API-domain labels being the majority (42.7%), followed by code component labels (20.6%). This difference in distribution alludes to the usefulness of the API-domain labels.

To better understand the usefulness of the API-domain labels as compared to the other types of labels, we further investigated the label choices among the Treatment group participants. Figure 4.6 presents two violin plots comparing: (a) API-domain labels versus code component labels; and (b) API-domain labels versus issue type. Wider sections of the violin plot represent a higher probability of observations taking a given value. The thinner sections correspond to a lower probability. The plots show that API-domain labels are more frequently chosen (median is 5 labels) as compared to code component labels (median is 2 labels), with a large effect size ($|d| = 0.52$). However, the distribution of the issue type and API-domain labels are similar, as confirmed by negligible effect size ($|d| = 0.1$). These results indicate that while the type of issue (bug fix, enhancement, suitable for a newcomer) is important, understanding the technical

(API) requirements of solving the task is equally important in developers deciding which task to select.

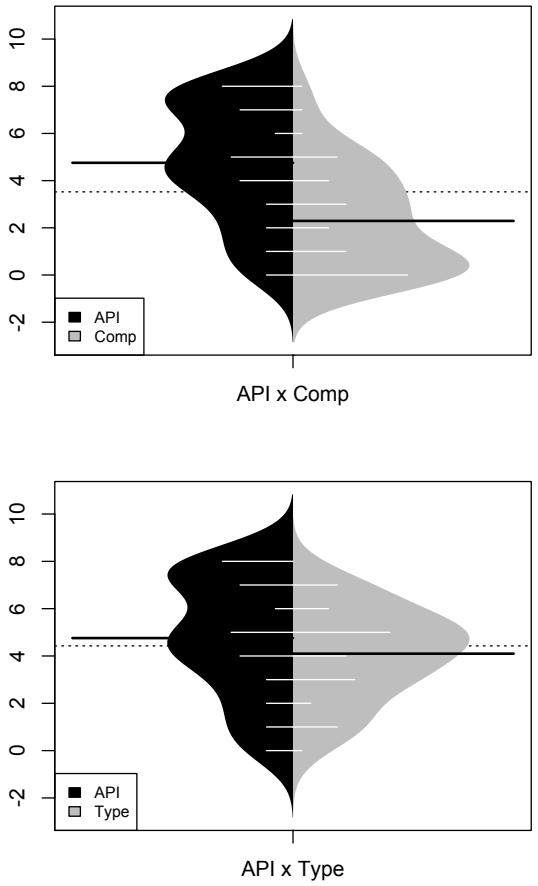

FIGURE 4.6: Density Probability Labels (Y-Axis): API-domain x Components x Types.

Finally, we analyzed whether the demographic subgroups had different perceptions about the API-domain labels (Table 4.8). When comparing industry versus students, we found participants from industry selected 1.9x (p-value=0.001) more API-domain labels than students when we control by component labels. We found the same odds when we control by issue type (p-value=0.0007). When we compared experienced vs. novice coders, we did not find statistical significance (p=0.11) when controlling by components labels. However, we found that experienced coders selected 1.7x more API-domain labels than novice coders (p-value=0.01) when we control by type labels.

The odds ratio analysis suggests that API-domain labels are more likely to be perceived as relevant by practitioners and experienced developers than by students and novice coders.

TABLE 4.8: Answers from different demographic subgroups regarding the API labels (API/Component/Issue Type)

Subgroup	Comparison	API %	Comp or Type %
Industry	API/Comp	**56.0**	44.0
Students	API/Comp	40.0	**60.0**
Exp. Coders	API/Comp	**50.9**	49.1
Novice Coders	API/Comp	41.5	**58.5**
Industry	API/issue Type	45.5	**55.5**
Students	API/issue Type	30.6	**69.4**
Exp. Coders	API/issue Type	43.5	**56.5**
Novice Coders	API/issue Type	30.9	**69.1**

RQ2 Summary. Our findings suggest that labels are relevant for selecting an issue on which to work. API-domain labels increased the perception of the labels' relevancy, and are especially relevant for industry and experienced coders.

4.3 Discussion

Are API-domain labels relevant? Finding an appropriate issue on which to work involves multiple aspects, one of which is knowing the APIs required for a task, which is what we investigate here. Our findings show that participants considered API-domain labels relevant in selecting issues. API-domain labels were considered more relevant as compared to the component type and slightly more favored than the type of issue. This suggests that a higher-level understanding of the API-domain is more relevant than deeper information about the specific component in the project. Therefore, our automated labeling approach can be of service to open source projects.

When controlling for issue type and component, API-domain labels were considered more relevant for experienced coders than novices (or students). This might suggest that

novices may need more help than "just" the technology for which they need skills. For example, novices could be helped if the issues provide additional details about the complexity levels, how much knowledge about the particular APIs is needed, the required/recommended academic courses needed for the skill level, estimated time to completion, contact for help, etc. Further research is needed to provide effective ways to help novice contributors in onboarding.

What are the effects of the characteristics of the data corpus? Observing the results reported for different corpora used as input, we noticed that the baseline model created using only the issue body had similar performance to the models using issue title, body, and comments, or better performance than the model using only title. By inspecting the results, we noticed that by adding more words to create the model, the matrix of features becomes sparse and does not improve the classifier performance.

We also found co-occurrence among labels. For instance, "UI", "Logging", and "IO" appeared together more often than the other labels. This is due to the strong relationship found in the source files. By searching the references for these API-domain categories in the source code, we found that "UI" was in 366 source code files, while "IO" and "Logging" was in 377 and 200, respectively. We also found that "UI" and "IO" co-occurred in 85 source files, while "UI" and "Logging" and "IO" and "Logging" co-occurred in 74 and 127 files, respectively. On the other hand, the API-domain labels for "Latex" and "Open Office Document" appeared only in five java files, while "Security" appears in only six files. Future research can investigate co-occurrence prediction techniques (e.g., [19]) in this context.

We suspect that the high occurrence of "UI", "Logging", and "IO" labels ($>$ 400 issues) compared with the smallest occurrence of "Security," "Open Office Documents," "Database," "PDF," and "Latex" ($<$ 32 issues) may influence the precision and F-measure values. We tested the classifier with only the top 5 most prevalent API-domain

labels and did not observe statistically significant differences. One possible explanation is that the transformation method used to create the classifier was binary relevance, which creates a single classifier for each label, overlooking possible co-occurrence of labels.

What are the difficulties in labeling? Despite the lack of accuracy to predict the rare labels, we were able to predict those with more than 50 occurrences with reasonable precision. We argue that JabRef's nature contributes to the number of issues related to "UI" and "IO." "Logging" occurs in all files and therefore explains its high occurrence. On the other hand, some specific API domains that are supposedly highly relevant to JabRef—such as "Latex," "PDF," and "Open Office Documents"—are not well represented in the predictions.

Looking at Table 4.9 and comparing it with the aforementioned co-occurrence data, we can determine some expectations and induce some predictions. For example, the "database" label occurred with more frequency when we had "UI" and "IO". So, it is possible that when an issue has both labels, we likely can suggest a "database "label, even when the machine learning algorithm could not predict it. The same can happen with the "Latex" label, which co-occurred with "IO" and "Network". A possible future work can combine the machine learning algorithm proposed in this work with frequent itemset mining techniques, such as apriori [94]. Thus, we can find an association rule between the previously classified labels.

4.4 Final Considerations

The case study aimed to perform exploratory research to evaluate the method and refine the instruments we built to obtain data to examine. In the next chapter, we move toward method generalization, stretching the limits of the case study.

TABLE 4.9: overall metrics from the selected model

API-Domain	TN	FP	FN	TP	Precision	Recall
Google Commons	107	15	27	30	66.6%	52.6%
Test	112	18	29	20	52.6%	40.8%
OS	152	8	8	11	57.8%	57.8%
IO	9	30	3	137	82.0%	97.8%
UI	30	26	10	113	81.2%	91.8%
Network	107	10	30	32	76.1%	51.6%
Security	167	6	2	4	40.0%	66.6%
OpenOffice	165	6	3	5	45.4%	62.5%
Database	154	3	6	16	84.2%	72.7%
PDF	164	5	4	6	54.5%	60.0%
Logging	19	32	18	110	77.4%	85.9%
Latex	170	1	1	7	87.5%	87.5%

CHAPTER 5: GENERALIZATION STUDY

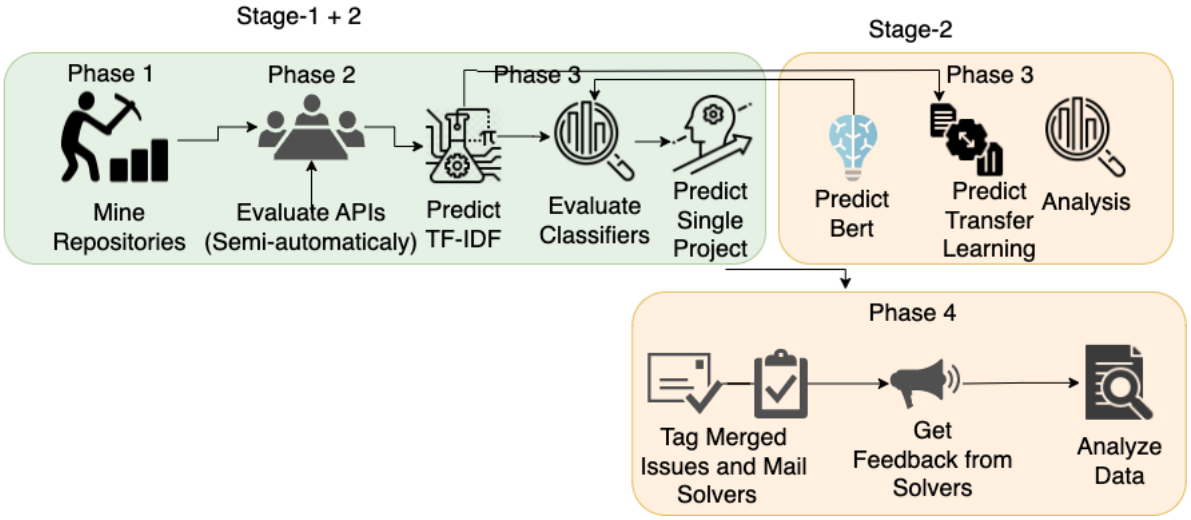

FIGURE 5.1: The Research Method - Stage 2 - Generalization Study

In our preliminary work [23], we conducted an exploratory experiment on a single project (JabRef) (Figure 5.1 - Stage 1). The generalization study includes four new projects. We selected projects to increase the diversity of domains, programming languages, and human languages (vocabularies). We particularly sought a mix of popular open-source (OSS) and closed-source currently active projects with a large number of issues and pull requests. As we aimed to run surveys within the project communities, we previously contacted maintainers/managers of candidate projects, explained our goals, and looked for support to reach contributors for the user studies. The selected projects and their characteristics are in Table 5.1.

We also conducted experiments using another modern text classification algorithm, BERT. We compared BERT to the previous TF-IDF classification pipeline within the ITS issue labeling problem context. BERT determines the meaning of words in a corpus based on their context within a sentence. The surrounding words around a target word create a context that can be used to determine the meaning of each word and sentence (Figure 5.1 - Stage 2).

TABLE 5.1: Projects Details

Project	**Releases**	**contributors**	**Stars/ Forks**	**Closed Pulls/ Issues**	**Domain**	**OSS**
JabRef	42	337	15.7K 1.8K	2.7K 4.1K	Articles manager	Y
Audacity	25	154	6.9K 17.1K	1.1K 0.6K	Audio editor	Y
PowerToys	50	262	65.5K 3.7K	3.3K 9.9K	Utilities for Windows	Y
RTTS	121	40	N.A. N.A.	N.A. N.A.	Telecommunication product	N
Cronos	123	–	N.A. N.A.	N.A. N.A.	Time Tracker	N

We also analyzed (Section 5.4.4) open-ended questions from the previous study [23].

The generalization study (we called "Tag my issue" method) was divided into four phases. Phase 1 mines software from diverse projects encompassing several programming languages and repositories, reusing the mining pipeline used in stage 1. Phase 2 presents an improved semi-automatically API categorization that enables a decrease in the effort done by experts. Phase 3 contains steps from Stage 1 (JabRef Case Study). Phase 3 runs the classifiers, including the BERT. Finally, in phase 4, we extended our empirical study by augmenting the population of interest by including current contributors (developers) who solved issues and asked them to evaluate if the issue they solved would benefit from having our labels.

This study aimed to address the research questions:

RQ.1: *How relevant are the API-domain labels to new contributors?* This RQ evaluates the manually curated labels with potential new contributors. We divided the participants into two groups. After mimicking the project's issues pages for 22 issues, we added API-domain labels to the issues for the treatment group and kept the page as-is for the control

group. We asked the participants to select issues to which to contribute and fill out a survey about their selection process.

RQ.2: *To what extent can we automatically attribute API-domain labels to issues?* In this RQ , we investigate the feasibility of predicting API-domain labels. We mined software repositories to collect issues, their associated pull requests, and the APIs used in the source code. Subsequently, we manually classified the APIs into API domains to build machine learning classifiers. To answer the sub-questions, we predicted the API-domain labels using each project dataset separately (RQ.2.1), a dataset with all projects merged (RQ.2.2), and different source and target datasets (RQ.2.3).

RQ.3: *How well do the API-domain labels match the skills needed to solve the issue?* Finally, In this RQ, we asked contributors to provide feedback on the usefulness of the labels that we predicted in identifying skills needed to complete the issue.

5.1 Method - Mining Repositories

We started by gathering data from the repositories to train a machine-learning model to predict the API labels. To achieve this goal, we mined closed issues and merged pull requests. Table 5.2 summarizes the projects' characteristics and demographics. From August to November 2021, we collected a total of 22,231 issues and 4674 pull requests (PR). For the OSS projects, we used the GitHub REST API v3 to collect data such as title, body, comments, and closure date. We also collected the name of the files changed in the PR and the commit message associated with each commit. The industry projects used Gerrit (RTTS) and Jira + MTT (Cronos). From RTTS, we extracted two CSVs files: one containing the "issues" (troubles in RTTS) and the second containing the commits. The Cronos project uses a combination of Jira to track the open issues and the MTT

TABLE 5.2: Projects Mined and Issue Tracker Systems

Project	**Prog Lang**	**Issue Tracker/ Vocab-ulary**	**Extraction Method**	**Issues/ PR**	**Linked Issues & PR**	**Source Code Files**	**Distinct APIs**
JabRef	Java	GitHub EN	GitHub API V3	4,471 1,966	1914	1,690	1,944
Audacity	C++	GitHub EN	GitHub API V3	1,440 310	341	624	1,478
PowerToys	C#	GitHub EN	GitHub API V3	12,571 853	1011	794	264
RTTS	Java	Gerrit EN	Export CSV	2,836 470	470	9,779	8,645
Cronos	Java	Jira/MTT BR	Export CSV/TXT	913 1075	206	220	441
Total				22,231 4,674	3942	13,107	12,772

(Minds At Work Time Tracker) software, an in-house solution, to manage the revisions and allocation time. We extracted a CSV file from Jira and a TXT from MTT.

We partially repeated some of the previous case study mining steps but with the novelty of addressing new ITSs. To permit easier reading and flow, we are explaining all the steps, even when the step was used in the previous study.

We kept only the data from issues linked with merged and closed pull requests to train the model since we needed to map issue data to source code APIs. To find the links between pull requests and issues in open source projects, we searched for the symbol #issue_number in the pull request title and body and checked the URL associated with each link. We also filtered out issues linked to pull requests without at least one source code file (e.g., those associated only with documentation files) since they do not provide the model with content related to any API. Similarly, we linked projects hosted by Gerrit and Jira/MTT, using the trouble id and key fields (Gerrit), and for the project managed with Jira/MTT, we linked using the change id and revision fields. The TXT file from MTT must be parsed to look for the revision field. We discarded entries without source code or linked data. In total, 734 entries were discarded.

5.2 Method - API Classification

Phase 2 encompasses API extraction and expert classification.

API extraction. To identify the APIs used in the source code affected by each pull request, we updated the parser, used in the case study, to process all source files from the projects. 12,772 library declaration statements from 13,107 source files were mapped to 185,159 possible relationships between files and APIs. The parser looked for specific commands, i.e., import (Java), using (C#), and include (C++). The parser identified all classes, including the complete namespace from each import/using/include statement. We considered only the most frequent language per project, even if the project has been built using more than one programming language.

Then, we filtered out APIs not found in the latest version of the source code (JabRef 5.3, audacity 3.1.0, and PowerToys 0.49.1; RTTS and Cronos are industry projects, and we used the last provided version) to avoid recommending APIs in source code that was no longer used in the project.

Our final dataset comprises 22,231 issues, 4,674 pull requests, 13,107 files, and 12,772 distinct APIs (Table 5.2).

Expert Classification. Three software engineering experts, including one of the authors of this article, proposed domains (UI, IO, Cloud, Error handling, etc.) to classify the APIs found by the parser. Domains were considered generic enough to suit a wide range of projects. After four rounds of discussions, the experts reached a consensus, and 31 API domains were defined. (Table 5.3).

After defining the 31 API domains, we started to classify the APIs semi-automatically (Figure 5.2). The intuition behind the API classification method is that libraries' namespaces often reveal architectural information and, consequently, their categories or API

TABLE 5.3: Labels Definition

Label Generated	Definition
Application (App)	Third-party apps or plugins for specific use attached to the System
Application Performance Manager (APM)	Monitors performance or benchmark
Big Data	APIs that deal with storing large amount of data, with variety of formats
Cloud	APIs for software and services that run on the Internet
Computer Graphics (CG)	Manipulating visual content
Data Structure	Data structures patterns (e.g., collections, lists, trees)
Databases (DB)	Databases or metadata
Software Development and IT Operations (DevOps)	Libraries for version control, continuous integration and continuous delivery
Error Handling	Response and recovery procedures from error conditions
Event Handling	Answers to events like listeners
Geographic Information System (GIS)	Geographically referenced information
Input-Output (IO)	Read, write data
Interpreter	Compiler or interpreter features
Internationalization (i18n)	Integrate and infuse international, intercultural, and global dimensions
Logic	Frameworks, Patterns like Commands, Controls or architecture-oriented classes
Language (Lang)	Internal language features and conversions
Logging	Log registry for the app
Machine Learning (ML)	ML support like build a model based on training data
Microservices/Services	Independently deployable smaller services. Interface between two different applications so that they can communicate with each other
Multimedia	Representation of information with text, audio, video
Multi-Thread (Thread)	Support for concurrent execution
Natural Language Processing (NLP)	Process and analyze natural language data.
Network	Web protocols, sockets, RMI APIs
Operating System (OS)	APIs to access and manage a computer's resources
Parser	Breaks down data into recognized pieces for further analysis.
Search	API for web searching
Security	Crypto and secure protocols
Setup	Internal app configurations
User Interface (UI)	Defines forms, screens, visual controls
Utility (Util)	Third-party libraries for general use
Test	Test automation

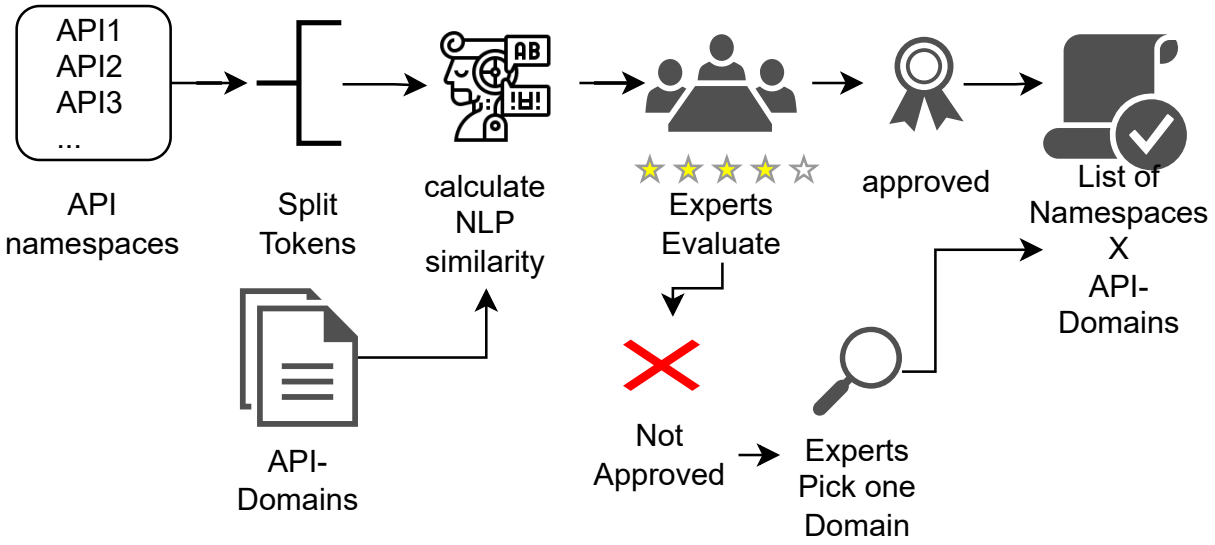

FIGURE 5.2: API expert evaluation

domains [95, 96]. To identify the possible API domains for each API, we split all the API namespaces into tokens. For instance the API "com.oracle.xml.util.XMLUtil" was split in "com", "oracle", "xml", "util", and "XMLUtil". Next, we eliminated the business domain name extensions (e.g., "org", "com"), country code top-level domain ("au", "uk", etc.), the project and company names ("microsoft", "google", "facebook", etc.). In the example, we kept the first token "xml", second token "util", and full namespace "com.oracle.xml.util.XMLUtil".

For each token, we identified how similar it is to the 31 proposed API domains using an NLP similarity function. The idea is to suggest possible fits for the APIs to the experts. We used the NLP Python package spacy. Spacy is a multi-use NLP package and can retrieve the semantic similarity of words using word2vec. We set up the spacy package with the largest trained model available (large full vector package, en_core_web_lg, which includes 685k unique vectors).

To assist the expert evaluation in reducing the search scope, we aggregated the tokens by the namespaces. For instance, to evaluate the APIs for the Cronos project, the experts received a list with 32 "first tokens" + a list with 73 "second tokens" identified and their respective suggestions ordered by similarity metric. Therefore, instead of classifying the 441 APIs found in the source code, they checked the NLP suggestions in the list of first and second tokens. (Table 5.4).

Then, three experts (one author and two senior developers) employed a card-sorting approach to manually accept or reject the suggestions for each token in the list. Each expert picked up one of the suggestions or chose a better API domain based on their experience. The experts could also check the list of full namespaces if they did not agree with the NLP suggestions. For example, considering the namespace "com.oracle.xml.util.XMLUtil": for the first token, "xml", it suggested (Input and Output: 0.7, Error Handling: 0.69, Parser: 0.57). For the token "util", it suggested (Utility: 0.9, Data Structure: 0.49). Therefore,

Table 5.4: API classification

Project	Total APIs	APIs submited to experts 1st token	APIs submited to experts 2nd token	APIs % 1st round	APIs % 1st + 2nd round
Audacity	1,478	562	106	38.0%	45.1%
PowerToys	264	37	20	14.0%	21.6%
Rmca	8,645	10	95	1.7%	2.3%
Cronos	441	32	73	7.2%	23.8%
JabRef	1,692	137	45	8.0%	10.8%
Total / avg	12,520	869	339	6.9%	9.64%

the namespace "com.oracle.xml.util.XMLUtil" was classified as "Utility". The majority of the APIs were classified using the first or second token. In only a few cases ($< 10\%$), the experts had to classify the entire namespace. After classifying all the tokens, the experts conducted a second round to achieve a consensus ($\sim$16 hours for all projects).

We used these 31 categories (API-domains labels) for the 22,231 issues previously collected based on the presence of the corresponding APIs in the changed files. We used this annotated set to build our training and test sets for the multi-label classification models.

5.3 Method - Building the Classifiers

Since solving an issue may require multiple types of APIs, we applied a multi-label classification approach, which has been used in software engineering for purposes such as classifying questions in Stack Overflow (e.g., [14]) and detecting types of failures (e.g., [77]) and code smells (e.g., [78]). To build the classifiers, we first needed to prepare or build the corpus and then run and evaluate the classifiers.

Corpus construction. The corpus construction comprised pre-processing, cleaning, diagnostics, and splitting into training and test datasets.

Pre-processing: We built two distinct models—one that uses TF-IDF [79] and another that uses BERT [97]. These corpora include the issue title, body, and comment texts of the selected issues.

Next, similar to the previous case study and many other studies [79–81], we applied TF-IDF, which is a technique for quantifying word importance in documents by assigning a weight to each word. After applying TF-IDF, we obtained a vector of TF-IDF scores for each issue's word. The vector length is the number of terms used to calculate the TF-IDF, and each term received the TF-IDF score. These TF-IDF scores are then passed to one of the selected classifiers (e.g., RandomForest) to then label each issue. Each label receives a binary value (0 or 1), indicating whether the corresponding API domain is present in the issue.

For BERT, we created two separate CSV files: an input binary with expert API-domain labels paired with the issue corpus, as well as a list of the possible labels for the specific project. BERT directly labels the issue with the corpus text and labels list without the need for an additional classifier.

We also evaluated the classifier's performance by combining in one dataset all the projects using English vocabulary. Therefore, we also had to build a new composed ID (ID + project name) for all projects to guarantee uniqueness. For this experiment, after we created the new IDs, we merged the binaries of the project, including the classes missing for each project (RTTS does not have a Computer Graphics label, for example). We compared various algorithms to identify the best setup.

Cleaning: To build our classification models using TF-IDF, we repeated the cleaning procedures from the previous study. First, we converted each word in the corpus to lowercase and removed URLs, source code, numbers, and punctuation. We also removed stop-words and stemmed the words using the Python nltk package. We filtered out the

issue and pull request templates[1] since their repetitive structure introduced noise and was not consistently used among the issues.

We follow the work from Izadi et al. [98] to process data for BERT. This work demonstrated that an unclean input corpus best maintained the context of words needed for BERT to determine their meaning and significance. As a result, the uncleaned corpus increased the performance of BERT and the accuracy of predictions.

Diagnostics: Multi-label datasets are usually described by label cardinality and label density [82]. Label cardinality is the average number of labels per sample. Label density is the number of labels per sample divided by the total number of labels averaged over the samples. For our dataset, the label cardinality is 8.19. The density is 0.26. These values consider the 22,231 distinct issues and API-domain labels obtained after the previous section's pre-processing steps. Since our density can be considered high, the multi-label learning process or inference ability is not compromised [99].

Training/Test Sets: We reproduced the training procedure from the case study. We split the data into training and test sets using the ShuffleSplit method [82], which is a model selection technique that emulates cross-validation for multi-label classifiers. For example, in the JabRef project, we had 1,914 linked data, and since one PR could be linked with more than one issue, we kept 1,648 entries which we randomly split into a training set with 80% (1,318), 70% (1,154), and 60% (989) of the issues and a test set with the remaining 20% (330 issues), 30% (494) and 40% (659). To avoid over-fitting, we ran each experiment ten times, using ten different training and test sets to match 10-fold cross-validation. To improve the balance of the data set, we ran the SMOTE algorithm for the multi-label approach [100].

[1] `http://bit.ly/NewToOSS`

Classifiers. To create the classification models, we kept the classifiers used in the case study and added one more: BERT. All are suitable to work with the multi-label approach and implemented different strategies to create learning models: Decision Tree, Random Forest (ensemble classifier), MLPC Classifier (neural network multilayer perceptron), MLkNN (multi-label lazy learning approach based on the traditional K-nearest neighbor algorithm) [82, 83], Logistic Regression, and BERT. We ran the classifiers using the Python sklearn package and tested several parameters doing a grid search. For the RandomForestClassifier, the best classifier, we kept the following parameters: $criterion$ = *'entropy'*, $max\_depth = 50$, $min\_samples\_leaf = 1$, $min\_samples\_split = 3$, $n\_estimators = 50$.

The BERT model was built using the open source python package, Fast-Bert [2] , which builds on the Transformers [3] library for Pytorch. Before training the model, the optimal learning rate was computed using a lamb optimizer [101]. Finally, the model fit over 11 epochs and validated every epoch. This training and validation occurred for every fold in the ShuffleSplit 10-fold cross-validation. The BERT model was trained on an NVIDIA Tesla V100 GPU within a computing cluster. The CPU used was the Intel(R) Xeon(R) Gold 6132 CPU. The choice of hardware is not critical so long as the target GPU has sufficient VRAM to train the BERT model.

Classifiers Evaluation: To compare the results with the case study and evaluate the classifiers, we employed the same metrics we used in our case study (also calculated using the scikit-learn package):

- **Hamming loss** measures the fraction of the wrong labels to the total number of labels.

[2] `https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html`

[3] `https://huggingface.co/docs/transformers/index`

- **Precision** measures the proportion between the number of correctly predicted labels and the total number of predicted labels.
- **Recall** corresponds to the percentage of correctly predicted labels among all truly relevant labels.
- **F-measure** calculates the harmonic mean of precision and recall. F-measure is a weighted measure of how many relevant labels are predicted and how many of the predicted labels are relevant.

Transfer Learning. Next, we investigate the behavior of the metrics when we use different sets to train and test the model. We combined four projects using English vocabulary using three projects for training and one for testing. For instance, we trained a dataset with JabRef, PowerToys, and Audacity to test using the RTTS project. Next, we substituted the test dataset with one in the training set until we finished all possible combinations.

Data Analysis. We used the aforementioned evaluation metrics and the confusion matrix logged after each model's execution to evaluate the classifiers. We used the Mann-Whitney U test to compare the classifier metrics, followed by Cliff's delta effect size test. The Cliff's delta magnitude was assessed using the thresholds provided by Romano et al. [102], i.e., $|d|<0.147$ "negligible", $|d|<0.33$ "small", $|d|<0.474$ "medium", otherwise "large". We considered p-value < 0.05 as the limit to determine a statistical difference.

For the remainder of our analysis, we filtered out the API labels with no occurrence. "Cloud" and "Machine Learning" did not appear in any issues/PR mined and, therefore, had no predictions. We also filtered out labels that appeared in more than 90% of rows when running models for each project. Those could bias our predictions since the classifier could always suggest them. For PowerToys, for example, the labels "NLP", "Network", "DB", "Error Handling", "Language", "DevOps", "IO", "ML", "Security",

"Cloud", "Event Handling", "CG", "Multimedia", "Thread", "Big Data" and "GIS" had no occurrences and, therefore were removed. The label "Util" was also removed because it surpassed the labels threshold (presented more than 90% of the rows in the dataset). The "Util" label was the most present with 699 occurrences, followed by 501 occurrences of "App" and 498 of "UI". The less representative set had "Test" (6), "Logging" (6) and "i18n" (4). PowerToys is a set of enhanced utility tools for Microsoft Windows. The high frequency of "Util" labels is not a surprise.

Considering the predictions using the dataset with all projects, our distribution of labels changed a lot. The most frequent Labels were "UI" with 762 occurrences, followed by "Util" with 726 and "Logic" with 575. The less frequent labels were: "NLP" (45), "CG" (16), and "GIS" (10). Despite some labels being popular and having been used for tagging many APIs by the experts, the lack of pull requests submitted that touched source codes with those APIs may cause their rareness. The lack of linked issues and pull requests that mention those labels can also cause the absence in the dataset. Finally, training all the datasets together helped to spread the labels' frequency, for instance: "Util" and "Logic" labels were dropped when training the JabRef project because they reached the threshold of 90% of label predictions. When training using the dataset with all projects combined, those labels prevailed, being below the 90% threshold, and were used to tag the issues (Figure 5.3).

Finally, we checked the distribution of the number of labels per issue (Figure 5.4). We found 110 issues with six labels, 106 issues with three labels, 104 issues with seven labels, and 102 issues with eight labels. Only 4.1% (=40) of issues have one label, which confirms a multi-label classification problem (Figure 5.4).

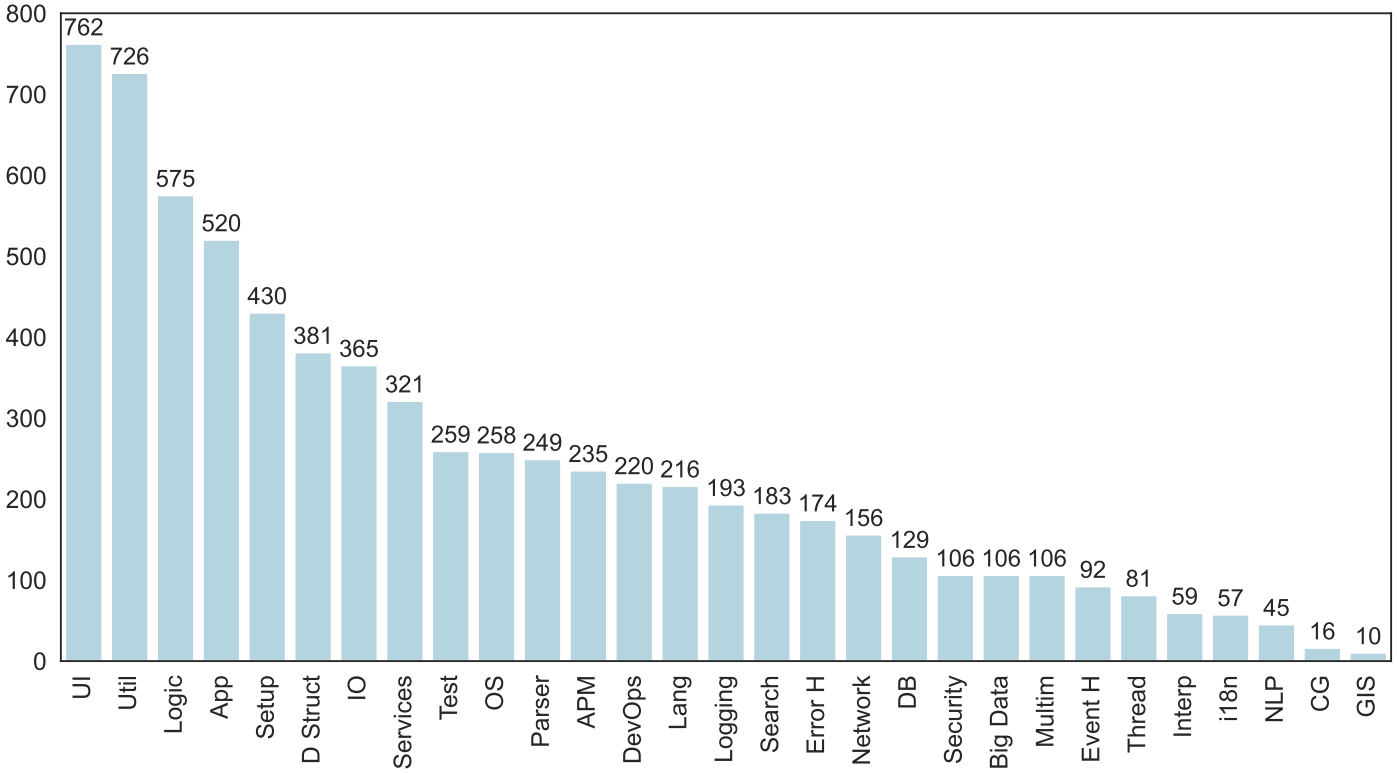

FIGURE 5.3: Number of labels per type

5.4 Method - Developers' Evaluation

To answer RQ3, we use the Random Forest algorithm, issue description BODY as the corpus, and unigrams (the best configuration we found in RQ.2) to generate labels for the issues.

5.4.1 Labels Generation

We predicted labels for 91 issues in OSS projects (PowerToys = 21, audacity = 18, Cronos = 24, and RTTS = 28). The predictions covered all the 29 proposed API-domain labels (Cloud and ML do not have samples in our projects).

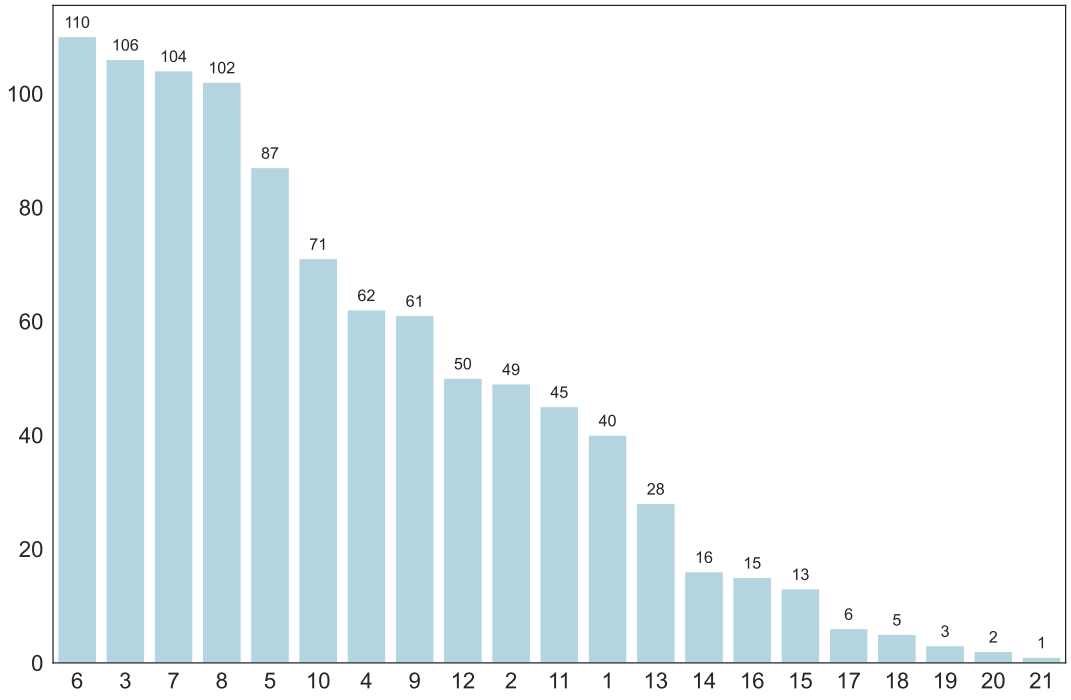

FIGURE 5.4: Number of labels per issue

5.4.2 Contributors Assessment

In this step, we recruited 20 participants (PowerToys (1), Cronos (13), and RTTS (6)). To recruit participants from those projects we sent emails to maintainers from PowerToys and Audacity and contacted development managers from Cronos and RTTS projects. We asked participants from those projects to evaluate if the labels represent the skills needed to solve the issues and could be applied to the issues to help newcomers or others experienced developers that want to start contributing and work on those issues. All of the participants were experts in their project and received a gift card as a token of appreciation for their participation.

We asked the following questions:

- How important do you consider having these labels on the issue to help new contributors identify the skills needed to solve them? (Evaluate each label) (Likert: Very Important, Important, Moderately Important, Slightly Important, Not Important)
- Why?
- What labels are missing?

5.4.3 Qualitative and Quantitative Analysis

Based on the data gathered in the contributors' assessment, we performed qualitative analysis to asses the labels generated. To analyze the open questions in which the contributors could explain their opinions about the labels generated, we employed open coding and axial coding procedures [103].

5.4.4 Open Questions Analysis.

To understand the rationale behind the label choices, we qualitatively analyzed the answers to the open questions ("Why was the information you selected relevant?" and "What kind of label would you like to see in the issues?"). We selected representative quotes to illustrate the participants' perceptions of the labels' relevancy.

We qualitatively analyzed the answers by inductively applying open coding in groups, where we identified the participant's reason for considering the provided information as relevant and what information the participant would like to be provided. We built post-formed codes as the analysis progressed and associated them with respective parts of the transcribed text to code the information relevance according to the participants' perspectives.

Researchers met weekly to discuss the coding. We discussed the codes and categorization until reaching a consensus about the meaning of and relationships among the codes. The outcome was a set of high-level categories as cataloged in our codebook[4].

5.5 Results

In this section, we present the results of our ongoing generalization investigation grouped by research question.

5.5.1 RQ.1: How relevant are the API-domain labels to new contributors?

The way contributors analyzed the issues. We used the questionnaire's open-ended question from Santos et al. [23] to evaluate how subjects used the information to decide whether the task was appropriate for them (Section 4.1.9).

Our qualitative analysis revealed a set of categories of information reported as relevant by contributors when they decide on a task to which to contribute. We organized the 22 categories of information based on an existing model from literature, the 5W2H framework, as we explain below and illustrate in Figure 5.5. The 5W2H framework (5-Wh and 2-How questions) is often used for clarifying a problem, issue, error, or nonconformity, or to facilitate implementing effective actions. The framework was initially applied to the automotive and other manufacturing industries [104] and later to quality management [105] and software engineering [1].

Who will solve the issue? This category contains information about the forces influencing people to choose to work on an issue. Contributors mentioned what can influence one's

[4] `https://doi.org/10.5281/zenodo.6869246`

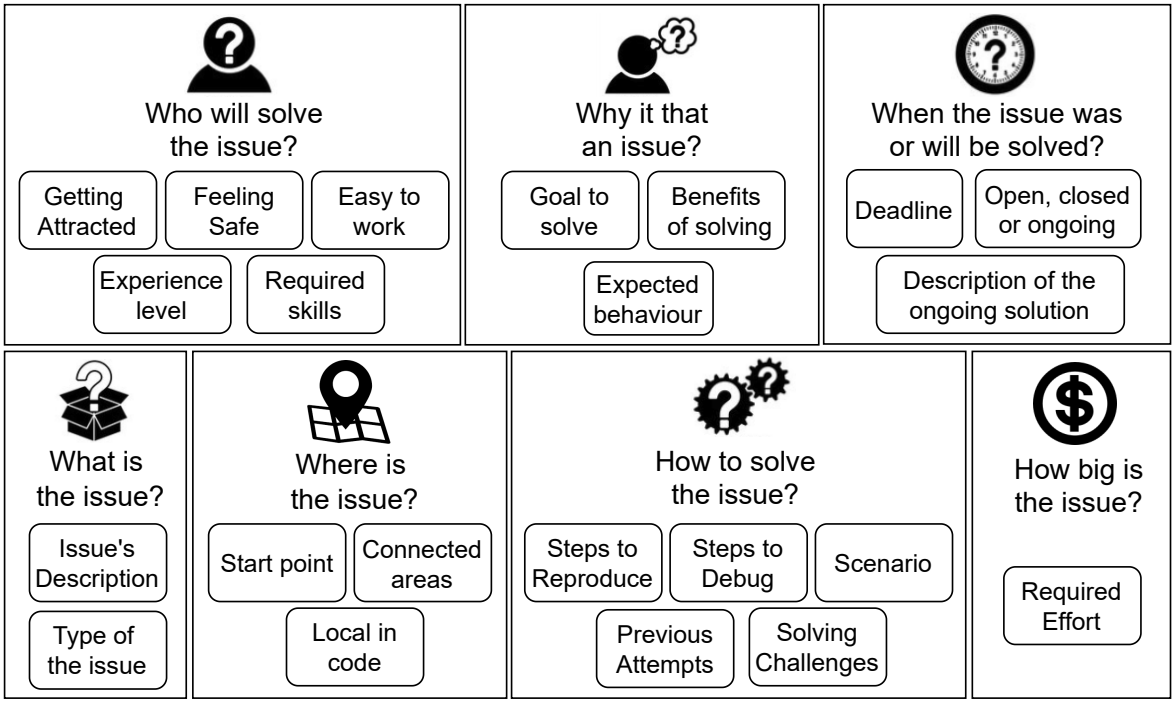

FIGURE 5.5: The information reported by contributors as relevant to choosing a task. We mapped the categories of our participants' definitions (rounded squares) to the 5W2H framework [1], which organizes information for decision-making across seven questions.

decision to select the issue. A newcomer can BECOME ATTRACTED to select the issue when *"filtering labels to search issues that [they] would like to contribute the most"* (P34) and reading the title (P18) to see if it *"includes something that is not too wordy and if it uses words [they]could easily understand"* (P21). When opening the issue, participants also reported the body and the comments were relevant to *"gain interest on the issue"* (P4) , as a *"detailed body and helpful comments from experienced people in the project is extremely helpful to make the newcomer* FEELING SAFE *to try the issue"* (P6). The contributors' confidence to decide about an issue can increase when they match their EXPERIENCE LEVEL with the indication of difficulty to solve the issue (P8, P4) which could be shown in a label such as "easy, medium, hard" (P4) , "good first issue" (P6), or "good challenging issue" (P7). Besides their experience, contributors can use the REQUIRED SKILLS to work on the issue to judge *"if they have the [necessary] skill to help"* (P31) or *"whether or not [are] capable of finding a solution"* (P32). The required

technical skills mentioned by participants included the programming language of the code (P21, P27, P33), the architecture layer - front-end, back-end, interface (P27, P3, P2), APIs (P41, P42), database (P33), frameworks, and libraries (P20).

Why is that an issue? is a category that justifies the issue as an issue. Participants mentioned the reasoning for an issue to exist could raise interest in new contributors, such as knowing the GOAL TO SOLVE and *"what is the purpose of the issue"* (P44), and the BENEFITS OF SOLVING or *"why solving it will help users"* (P45). Additionally, the EXPECTED BEHAVIOR of the software can help to clarify why the reported issue is an issue in comparison to the normal behavior of the software. Hence, the expected behavior represents a *"critical information to decide what is happening in the system and what is expected"* (P61). Indeed, one participant reported: *"I would only contribute something that I know how it works"* (P22).

When was the issue solved, or when will it be solved? introduces time-related information and constraints regarding the issue. Participants reported they would like to know the DEADLINE TO SOLVE the issue or the *"urgency"* (P13). Participants suggested that the priority appear in a label (P17) and be defined according to the impact that the issue has on businesses or users (P15). Another issue related to time is the *"status to check the issue's state"* (P33), which can be OPEN, CLOSED, OR ONGOING, allowing contributors to use a filter in the issues' page. Since they "don't look at closed issues much, [...] the open flag grabs [their] attention" (P43). When a contributor is currently working on a solution, they should have their names assigned to the issue and include a comment with the DESCRIPTION OF AN ONGOING SOLUTION that should "demonstrate the issue's status" (P35).

What is the issue? relates to the description of the issue itself. Participants raised the importance of clear ISSUES' DESCRIPTION, including both a summarized *"idea of what the issue is about"* (P28) and a comprehensive explanation *"to help understand what is the*

problem" (P45) about. When an issue provides both levels of details, it *"tells about the problem, first in a general term and later giving [them] details about it"* (P12). The issue's TYPE in labels *"demonstrate [...] how [the issue] is classified"* (P35). The participants suggested the issue should have *"labels that inform precisely which type of issue is"* (P40): bug (P41), a new feature (P42), performance (P42), enhancement (P42), and security. One participant (P43) emphasized that *"'all issues should have a type so [they] can see if [their] skill set is useful"* (P43).

Where is the issue? references the localization of the issue in the code or project, guiding contributors to a START POINT or *"where to start looking at in the code/library to investigate the problem"* (P4). The LOCAL IN CODE or the code block, method, or class which is causing the issue, and CONNECTED AREAS. This information would *"give some hints about what areas have a connection with the problem occurring"* (P4) and *"code snippet to provide context for wherein the program this issue was happening"* (P18).

How to solve the issue? brings practical directions to guide solving the issue. Awareness of "PREVIOUS ATTEMPTS *to solve [an issue]"* (P30) helps contributors with *"valuable information about what has already been done and properly documented"* (P42). Contributors who are deciding about an issue can read "[SOLVING] CHALLENGES" (P35) to avoid wasting time on previous attempts and focus their effort on new paths to achieve the solution. When working on the issue, having STEPS TO REPRODUCE the error (P45) on a controlled environment also help to solve the issue. Participants also mentioned they would like to see *"linked issues and comments to help understand the* SCENARIO*"* (P33), and STEPS TO DEBUG to *"decipher what the problems really is"* (P41).

How big is the issue? is information that can provide visibility of the REQUIRED EFFORT for *"[a contributor] to work on alone until [they] solve it"* (P7). If the issue does not have this information, the developer tries to *"grasp what's the idea of the issue, to better measure how long it would take to solve it"* (P19).

Finally, the question "What region has this information?" identifies the regions where the participants found the information in this study. The title appeared 7 times, body 8, comments 13, labels 5, status 2, code snippet 3, and linked issue 2.

5W2H outcomes: the analysis confirmed the relevance of the title, body, comments, and labels and helped to create a taxonomy of what contributors analyze when deciding whether they want to contribute to an issue. The qualitative code we built for this open-ended question may be explored in future work to create ways to show the contributor such information using templates, labels, bots, or other UI objects.

Preferred types of labels. Towards the evaluation of the labels contributors want to see on issues pages, 42 participants (out of 74) answered the open question Q2 ("What kind of label do you want to see in the issues?"). The TYPE, PRIORITY to solve, and PROGRAMMING LANGUAGE *"in which the code was written in"* (P21) were the three most mentioned, followed by DIFFICULTY LEVEL, TECHNOLOGY, and API. Some participants suggested different semantics for the label TYPE: bug (P3, P41), improvement (P3), performance (P35, P42), new feature (P42), or security (P36). Other participants also suggested different semantics for DIFFICULTY LEVEL: *"good first issue"* (P6), *"good challenging issue"* (P7), or *"easy, medium, hard"* (P4). The semantics for each label can be explored in future work. We present the 11 categories of suggested labels that we qualitatively coded from the participants' answers in Table 5.5.

Participants prefer to see labels on priority, type, programming language, complexity, technology, and APIs more than architecture, status, GitHub info, database, and framework. GitHub info is general information about the project repository (e.g., "ranking about the most commented" (P10P0) and "branches" (P22I0)).

RQ.1 Summary. API is one of the labels users want to see in labels. 5W2H analysis has confirmed the relevance of labels and can guide contributors on how to write an issue.

TABLE 5.5: Labels desired by participants to select the issue

Group	Desired Labels	Participants who Mentioned
Management	Type	P41, P35, P42, P3, P36, P37, P38, P39, P43, P40
Management	Priority	P12, P13, P19, P20, P14, P15, P16, P29, P17, P18
Technical	Programming language	P34, P33, P21, P22, P27, P23, P24, P22, P26
Management	Difficulty level	P4, P5, P6, P7, P9, P8
Technical	Technology	P30, P34, P33, P32, P20, P31
Technical	API	P41, P42, P3, P1, P20
Technical	Architecture layer	P2, P3, P27, P18
Management	Status	P28, P9, P29
Management	GitHub info	P10, P19
Technical	Database	P33
Technical	Framework	P20

5.5.2 RQ.2.1: To what extent can we automatically attribute API-domain labels to issues using data from the project?

To predict the API-domains labels, we started by testing a simple corpus: only the issue TITLE as input and the Random Forest (RF) algorithm, since it is insensitive to parameter settings [89] and has shown to yield good prediction results in software engineering studies [90–93]. Then, we evaluated the corpus configuration alternatives, varying the input information: only TITLE (T), only BODY (B), TITLE and BODY (T+B), and TITLE, BODY, and COMMENTS (T+B+C) comparing the average of all projects. To compare the different corpus configurations, we kept the Random Forest algorithm and used the Mann-Whitney U test with the Cliff's-delta effect size.

We also tested alternative configurations using n-grams. For each step, the best configuration was kept. Then, we used different machine learning algorithms and compared them to a dummy (random) classifier.

As Figure 5.6 and Table A.1 (Appendix A) show, when we tested different inputs and compared them to TITLE only, all alternative settings provided better results with TF-IDF. We observed improvements in terms of precision, recall, and F-measure from the previous study [23]. When using BODY, we reached a precision of 84%, recall of 78.6%, and F-measure of 81.1%. In contrast, while BERT had worse results, the model with the TITLE outperformed the other BERT models with 61.6% precision.

TABLE 5.6: Cliff's Delta for F-measure and Precision: comparison of corpus model alternatives for TF-IDF and BERT. Title(T), Body(B) and Comments (C).

TF-IDF/BERT	**Corpus Comparison**	**Cliff's delta**			
		F-measure		**Precision**	
TF-IDF	T versus B	-0.005	negligible	-0.15	small***
TF-IDF	T versus T+B	-0.10	negligible***	-0.12	negligible***
TF-IDF	T versus T+B+C	-0.03	negligible***	-0.01	negligible
TF-IDF	B versus T+B	0.10	negligible***	0.02	negligible
TF-IDF	B versus T+B+C	-0.02	negligible	0.14	negligible***
TF-IDF	T+B versus T+B+C	0.07	negligible***	0.11	negligible***
BERT	T versus B	0.07	negligible	0.11	negligible
BERT	T versus T+B	0.13	negligible	0.03	negligible
BERT	T versus T+B+C	0.03	negligible	0.09	negligible
BERT	B versus T+B	0.10	negligible	-0.04	negligible
BERT	B versus T+B+C	-0.006	negligible	0.08	negligible
BERT	T+B versus T+B+C	-0.09	negligible	-0.01	negligible

** $p \leq 0.05$; ** $p \leq 0.01$; *** $p \leq 0.001$*

For TF-IDF, we found statistical differences comparing the results using TITLE only and all the three other corpus configurations: F-measure (p-value ≤ 0.001 when comparing with TITLE+BODY or TITLE+BODY+COMMENTS, Mann-Whitney U test) and precision (p-value ≤ 0.001 when comparing with BODY or TITLE+BODY, Mann-Whitney U test), both with negligible effect size when comparing the precision from TITLE and BODY. The corpus configured with BODY performed better than all others in terms of precision, followed closer by the one set up with TITLE+BODY, which performed better in recall and F-measure. However, the results suggest that using only the BODY would provide good enough outcomes since there was a negligible effect size compared to the other

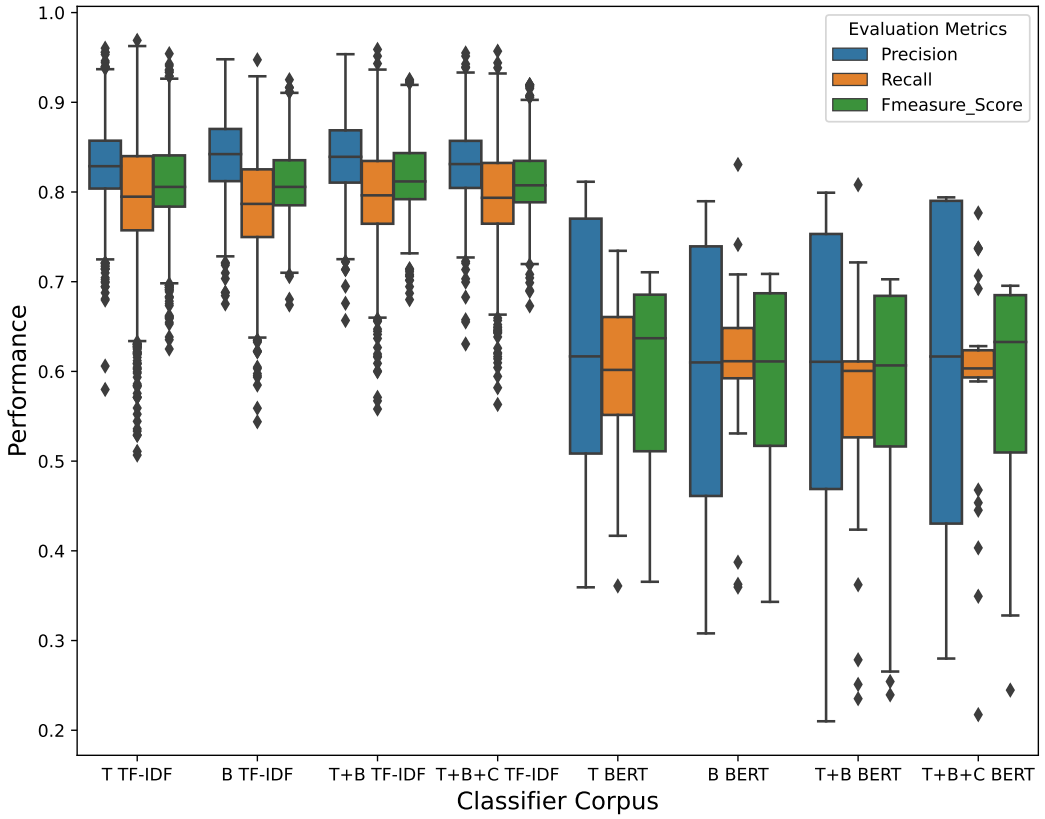

FIGURE 5.6: Comparison between the corpus models inputted to TF-IDF and BERT. T=Title, B=Body, C=Comments

two configurations—using TITLE and/or COMMENTS in addition to the BODY—achieving similar results with less effort. Table 5.6 shows the Cliff's-delta comparison between each pair of corpus configurations, and Figure 5.6 shows the box plots confirming the similar results carried out by the three diverse setups. For BERT, all the models had the same distribution in precision and F-measure.

Next, we investigated the use of bigrams, trigrams, and quadrigrams, comparing the results to those using unigrams. We used the corpus with only the issue BODY for this analysis since this configuration was chosen in the previous step. Table A.2 (Appendix A) and Table 5.7 present how the algorithms perform for each n-gram configuration. While the unigram configuration has a slightly better F-measure, the quadrigram has slightly

better precision. However, their differences in precision have a negligible effect size, and their differences in F-measure have a small effect size. Additionally, the unigram uses less computational effort and memory [106]. Hence, we kept the unigram as the best option.

TABLE 5.7: Cliff's Delta for F-measure and precision: Comparison between n-grams models

n-Grams Comparison	**Cliff's delta**			
	F-measure		**Precision**	
1 versus 2	0.09	negligible***	-0.02	negligible**
1 versus 3	0.11	negligible***	-0.01	negligible
1 versus 4	0.15	small***	-0.06	negligible
2 versus 3	0.02	negligible	0.01	negligible***
2 versus 4	0.06	negligible***	-0.04	negligible**
3 versus 4	0.04	negligible**	-0.05	negligible***

** $p \leq 0.05$; ** $p \leq 0.01$; *** $p \leq 0.001$*

TABLE 5.8: Cliff's Delta for F-measure and precision: Comparison between machine learning algorithms

Algorithms Comparison	**Cliff's delta**			
	F-measure		**Precision**	
RF versus LR	0.27	small***	0.06	negligible*
RF versus MLPC	0.02	negligible	0.009	negligible
RF versus DT	0.06	negligible*	0.09	negligible***
RF versus MlkNN	0.28	small***	0.13	negligible***
RF versus BERT	1.0	large***	1.0	large***
LR versus MLPC	-0.21	small***	-0.07	negligible*
LR versus DT	-0.15	small***	-0.15	small***
LR versus MlkNN	0.07	negligible*	0.08	negligible*
LR versus BERT	1.0	large***	1.0	large***
MPLC versus DT	0.03	negligible	-0.08	negligible***
MPLC vs. MlkNN	0.24	small***	0.13	negligible***
MLPC versus BERT	1.0	large***	1.0	large***
MlkNN versus DT	-0.19	small***	-0.20	small***
MlkNN versus BERT***	1.0	large	1.0	large***
DT versus BERT***	1.0	large	1.0	large***
RF versus Dummy	1.0	large***	0.50	large***

** $p \leq 0.05$; ** $p \leq 0.01$; *** $p \leq 0.001$*

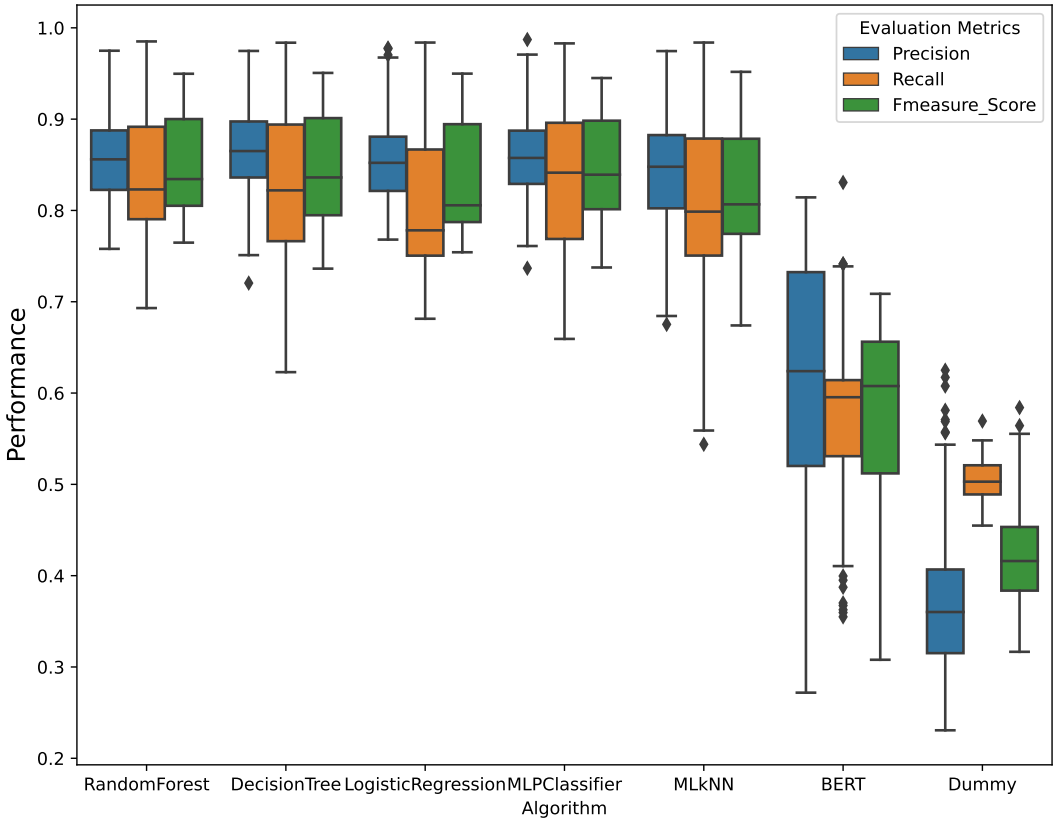

FIGURE 5.7: Performance comparison between the machine learning algorithms

To investigate the influence of the machine learning (ML) classifier, we compared several options using the BODY with unigrams as a corpus. The options included: Random Forest (RF), Neural Network Multilayer Perceptron (MLPC), Decision Tree (DT), LR, MlKNN, BERT, and a Dummy Classifier with strategy "uniform." Dummy or random classifiers are often used as a baseline [87, 88]. We used the implementation from the Python package scikit-learn [5]. Figure 5.7 and Table A.3 (Appendix A) show the comparison among the algorithms, and Table 5.8 presents the pair-wise statistical results comparing F-measure and precision using Cliff's delta.

Random Forest (RF) was the best model when compared to Decision Tree (DT), Logistic

[5] `https://scikit-learn.org/`

Regression (LR), Neural Network Multilayer Perceptron (MLPC), MlKNN algorithms, and BERT. Random Forest outperformed these five algorithms with negligible/small effect sizes considering F-measure and precision. Compared to BERT and the Dummy Classifier, the effect size was large. The observed difference among some algorithms are fairly small and therefore might vary according to project corpus properties.

Finally, we compared the predictions obtained in projects using English and Portuguese natural languages (Figure 5.8).

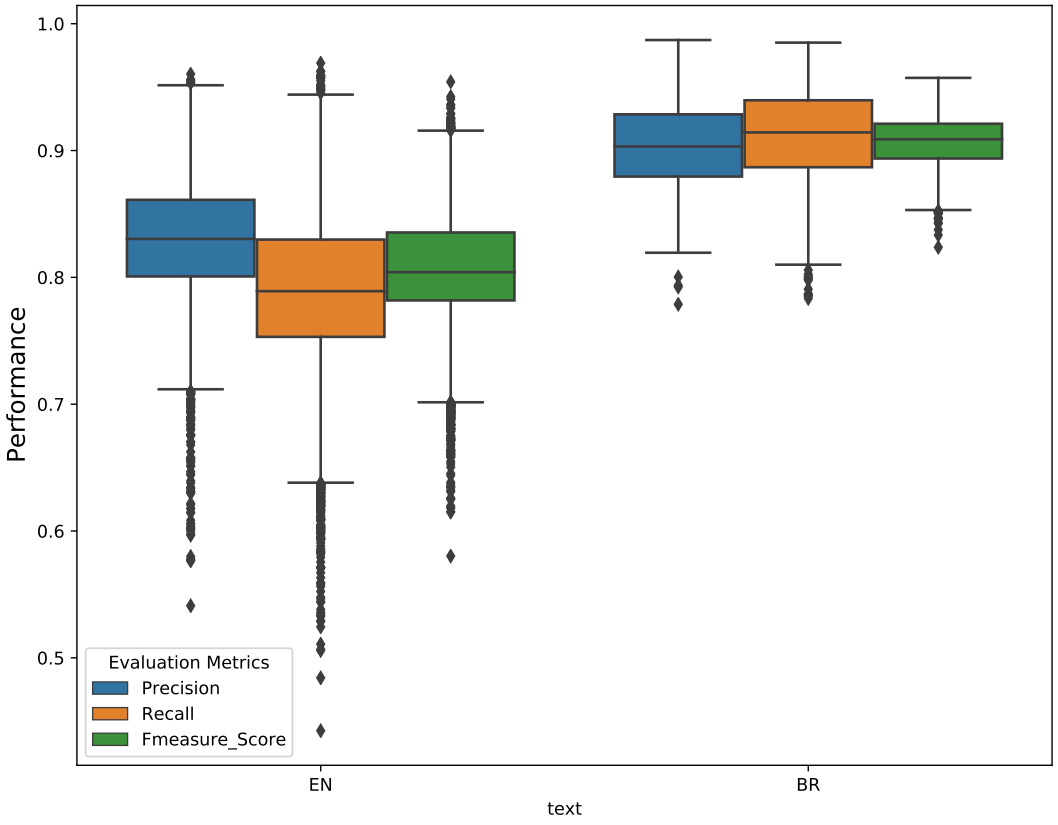

FIGURE 5.8: Performance comparison between the natural languages

The project written with issues text in Portuguese (PT-BR) overcomes the projects with issues text in English when comparing the precision, recall, and the F-measure with large effect size (Cliff's delta = 0.81, 0.90, and 0.95 and all p-values < 0.001).

The results showed the classifier is suitable for predicting labels in projects written in different programming languages (C++, C#, and Java), with issues with vocabulary in English and Portuguese.

> ***RQ.2.1 Summary.*** It is possible to individually predict the API-domain labels for each project with a precision of 0.864, recall of 0.786, and F-measure of 0.811 using the Random Forest algorithm, BODY as the corpus, and unigrams.

5.5.3 RQ.2.2: To what extent can we automatically attribute API-domain labels to issues using data from other projects?

Next, we merged the datasets that use English vocabulary (RTTS, JabRef, Audacity, and PowerToys), predicting the API-domain labels for all the projects. Removing the project with Portuguese vocabulary was necessary since the BERT model was trained with English vocabulary. The predictions were carried out with BODY as the corpus (and unigrams for the TF-IDF). Figure 5.9 shows the performance obtained with various algorithms. RF still had the best precision while the MLPC had the best F-measure; BERT had better precision than MLkNN and recall than Logistic Regression. BERT was less impacted by the loss of metrics when predicting the API-domain labels with the all-projects combined dataset (Table A.4 - Appendix A).

> ***RQ.2.2 Summary.*** Predicting using a dataset with all English-language projects combined decreased the precision by 9.15% using Random Forest and increased the precision using BERT by 20.63%.

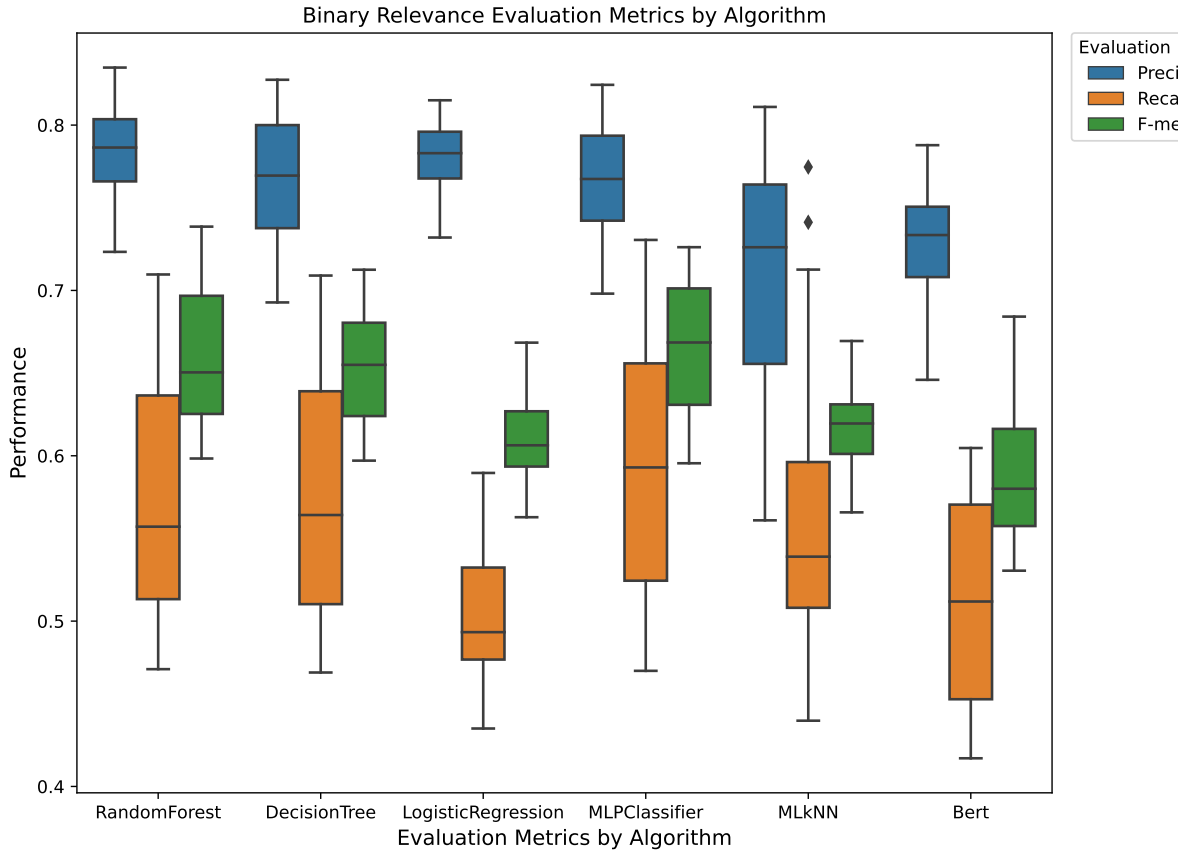

FIGURE 5.9: Performance comparison between machine learning algorithms using the dataset with all projects - Vocabulary: EN

5.5.4 RQ.2.3: To what extent can we automatically attribute API-domain labels to issues using transfer learning?

TABLE 5.9: Overall performance from models created to evaluate the transfer learning.

Training	Test	P	R	F
RTTS, Audacity, PowerToys	JabRef	0.305	0.294	0.299
JabRef, Audacity, PowerToys	RTTS	0.713	0.175	0.281
JabRef, RTTS, PowerToys	Audacity	0.688	0.284	0.402
JabRef, RTTS, Audacity	PowerToys	0.296	0.525	0.379
JabRef, Audacity, PowerToys	RTTS*	0.718	0.272	0.394

** RTTS - labels most important to the users (Section 5.5.5)*

Finally, Table 5.9 shows the results for all combinations tested with transfer learning. The results had a significant range in precision and recall varying from 0.713 to 0.296 in precision predicting RTTS and PowerToys, respectively, and from 0.525 to 0.175 in recall predicting Audacity and RTTS, respectively.

Additionally, we ran a transfer learning experiment targeting the RTTS project labels evaluated by developers (Section 5.5.5). We dropped all labels with fewer than three evaluations and up to 50% of "Not Important" evaluations (see Figure 5.10). Therefore, in the RTTS project, the labels that persisted are: "Network", "Logging", "Setup", "Micro/services", and "UI". Since Audacity, JabRef, and PowerToys projects were not evaluated by developers (Section 5.4), they were not included in this experiment. We observed a small increase in precision (0.713 to 0.718) and a significant increase in recall (9.7% - 0.175 to 0.272) and F-measure (11.3% - 0.281 to 0.394) - Table 5.9.

RQ.2.3 Summary. Transferring learning with diverse configurations considering source and target projects decreased the metrics from 15.12% to 64.74% and the recall from 33.21% to 77.74%, depending on the sources and target project. Evaluating the transfer learning concerning only the API-domain labels evaluated as important by the developer who solved the issues improved the recall by 9.7% and F-measure by 11.3%.

5.5.5 RQ3. How well do the API-domain labels match the skills needed to solve an issue?

From the 91 issues predicted, we received 29.67% feedback (26 issues and 16 different API-domain labels). We did not receive feedback from the Audacity contributors. PowerToys had only one contributor who evaluated only three issues encompassing only four labels. Due to insufficient data, we removed this project from the results.

Cronos. A total of 13 contributors assessed the generated labels. Based on the results (Figure 5.10), the contributors described 5 labels (i.e., DevOps, UI, DB, Lang, and Security) as very important or important and APM, Setup, NLP, and IO labels are unimportant. DevOps, UI, DB, and Lang were highly rated as important, with many "Very important" and "Important" evaluations. Not all the participants justified their response, but among the reasons those contributors mentioned that "*It was a simple UI issue.*" (P15) indicating the success of the "UI" prediction. Another developer mentioned "*This issue also required database, logic, and lang skills.*" (P6). This issue was tagged with "UI" and "DevOps" (evaluated as "Very important") but the developer missed some skills. Related to the missing labels, some contributors mentioned that "*The issue is related to a restriction. It requires UI skills and DevOps skills (not included in the predictions. [...]But it also requires other skills.*" (P18). Another contributor also missed some labels and mentioned "*This issue also required database, logic, and language skills.*" (P06).

RTTS. Concerning the RTTS project, five contributors assessed the labels generated for the issues they had solved; in this scenario, we have contributors evaluating from 1 to 3 issues each. Our findings (Figure 5.10) highlight that the following labels were classified by contributors as important to very important: Services, UI, Logging, Setup, Network, and Data Structure. Among the reasons contributors highlighted, we can observe the positive feedback as mentioned in: "*I can totally agree with the labels for this, as to find the problem and apply the solution all the skills are necessary.*" (P20). Moreover, contributors classified the following labels as not important or slightly important: Lang, Parser, DevOps, UI, and Data Structure. Some contributors mentioned that: "*I partially agree, some of the labels could give an initial point of view to the reported issue, but some are not related, like Language, Data structure and Setup*" (P2). Some contributors reported missing some labels according to what was mentioned: "*Logging skills would be necessary to troubleshoot the issue and get the relevant information from the application,*

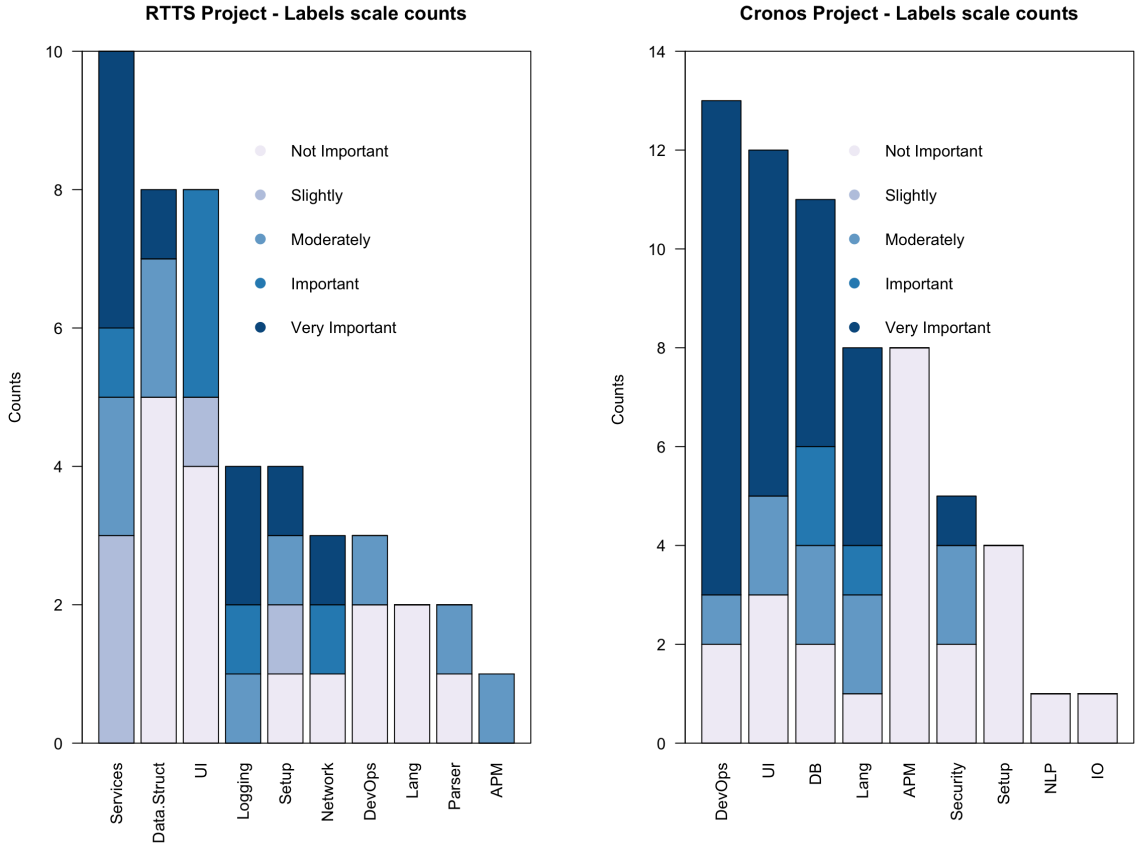

FIGURE 5.10: Labels assessment by project. Cronos - PT-BR and RTTS - EN

while service skills (knowledge about how service discovery and the service registry in the system works) would be necessary to find that the service version of the requested service didn't match what was registered. As for Network, it could have been useful to be able to determine that this issue was not caused by some error/faulty response from the requested service, but in this case, the log stated explicitly that the requested service did not exist. I don't find that the other labels/skills apply to this issue." (P04).

RQ.3. Summary. Our findings suggest the labels would be useful to help identify the skills needed to solve them. The efficiency of labels generated differs by project, for Cronos we had 61.9% of the labels evaluated in the range from slightly to Very important and 64.4% in the RTTS project in the same range.

5.6 Discussion

Do developers have a well-defined preference for labels? The feedback shared by study participants in Section 5.5.1 showed us the importance of the different types of labels to ease the issue selection process. However, the developers expressed preferences about different types of labels, and some preferences are ambiguous. For instance, P32 indicated "The technology used" when we asked what kind of labels they want to see in the issues. The technology could refer to a "programming language," or an "API." While both classifications could be used, we would prefer to define the technology as API because it is more specific than a programming language or even a framework that can encompass many libraries. A similar situation emerged with "priority" (P12). The "priority" could be restricted only to "low" or "high," or could it include other aspects like the "impact on operations", as suggested by P15.

We can group the kind of labels in technology (technology, API, programming language, database, framework, and architecture layer) and management (type, priority, status, difficulty level, GitHub info). Management labels are more often used in issue trackers. In this work, we propose to add a kind of technological label to the issues, the API-domain labels, which we claim are a proxy for the skills needed to solve an issue. Nonetheless, one should avoid overloading the issue trackers with too many labels. Future research can investigate the right balance of offering labels without creating a visual overhead for the contributor.

Are API-domain labels relevant? Our findings show that participants considered API-domain labels relevant in selecting issues. More specifically, newcomers to the projects considered API-domain labels more relevant than other general labels that describe the components and slightly more favored than management labels describing the

type of issue. This suggests that a higher-level understanding of the API domain is more relevant than deeper information about the specific component in the project.

When controlling for issue type and component, API-domain labels were considered more relevant for experienced coders than novices (or students). This suggests that novices may need more help than "just" the technology for which they need skills. Our results also show that novices could be helped if the issues provide additional details about the complexity levels, how much knowledge about the particular APIs is needed, the required/recommended academic courses needed for the skill level, estimated time to completion, contact for help, etc.

Although each contributor is a newcomer when they move to a new project, previous experience counts when the new project shares technology with the previous projects. As opposed to experienced newcomers, who may transfer knowledge from previous projects and jump directly to the issue solution, novice newcomers spend more time understanding the project structure, the underlying technology, and how to set up the environment [107], which might suggest why practitioners from the industry and experienced participants selected more API-domain labels than students and novices. Perhaps the API granularity is deeper than what the novices are looking for. Future research may consider the appropriate technical information to assist novice newcomers.

In addition to API-domain labels, what issue characteristics are relevant to identify skills in issues? In addition to labels, new contributors mentioned the TITLE, BODY, and COMMENTS as sources of information to identify the necessary skills to work on the issues. Such elements can be structured with issue templates or written ad-hoc. Santos et al. [107] asked maintainers to suggest community strategies to help newcomers find a suitable issue. Among the identified strategies, maintainers suggested 15 diverse ways of labeling the issues (e.g., labeling with skills, knowledge area, programming languages, libraries, and others) and several ways of organizing the issues, which include

creating templates.

While these other issue elements may indicate the skills and other characteristics of the issues that are not on the labels, some issues – and existing templates – are incomplete, lacking important information for contributors. The 5W2H analysis we applied in this paper can help us to holistically understand what should be written in issues by covering the seven dimensions of information - who, why, when, what, where, how to solve, and how big the issue is. Future work can use the 5W2H questions to inspect the completeness of existing templates in terms of covering the seven dimensions of information.

Despite the importance of issue templates, we removed template sentences in an effort to clean repeated text to be ingested by the data processing pipeline. For example, one template sentence is "Steps to reproduce." Since this fixed text appears in many issues (regardless of their categories) and the templates changed over time, we removed it before processing the issue corpus. This removal only affects the trained model, and we still should use the results of the 5W2H analysis to create a human-oriented template able to point new contributors to information relevant to them.

What are the effects of the corpus characteristics on the labels' classifications? Observing the reported results (TF-IDF) for different corpora used as input, we noticed that the model created using only the issue body performed similarly to the models using the issue title, body, and comments, and better than the model using only the title. By inspecting the results, we noticed that by adding more words to create the model, the matrix of features becomes sparse and does not improve the classifier's performance.

We also found co-occurrence among labels. For instance, "Test", "Logging", and "i18n" appeared often together (Figure 5.11). This is due to the strong relationship found in the source files. By searching the references for these API-domain categories in the source code, we found "Test" in 4,579 source code files, compared to "Logging" in 903. The label "i18n" appeared in only 73 files. On the other hand, the API-domain labels for

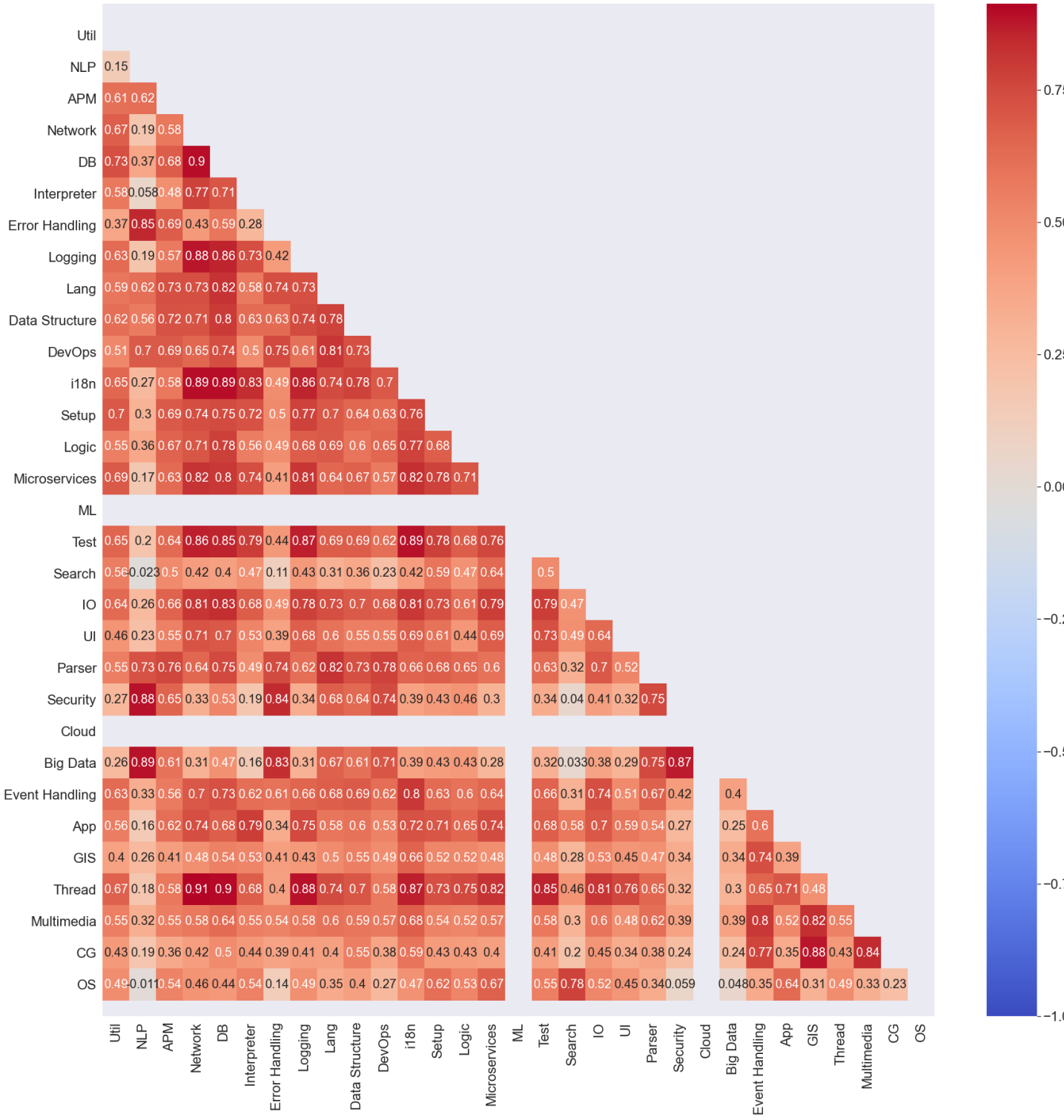

FIGURE 5.11: Heat Map - Label correlation in the dataset with all projects combined. The darker, the more correlation exists between the labels.

"CG" and "Security" usually do not co-occur. "CG" only appeared in five java files, while "Security" appeared in only 47 files. Future research can investigate co-occurrence techniques to predict co-changes in software artifacts (e.g., [19]) in this context.

Figure 5.11 exhibits the labels' co-occurrence for the dataset containing all the projects. A co-occurrence matrix presents the number of times each label appears in the same context as each possible other label. Examining the aforementioned co-occurrence data, we can determine some expectations and induce some predictions. For example, the "DB" label (Database) occurred with more frequency alongside "Network" and "Thread." So, it is possible to guess when an issue has both labels, and we likely can suggest a "Database" label, even when the machine learning algorithm could not predict it. A possible future work can combine the machine learning algorithm proposed in this work with frequent itemset mining techniques, such as apriori [108].

What are the difficulties in labeling accurately? We suspect that the high occurrence of "UI", "Util", and "Logic" labels (> 500 issues) compared with the low occurrence of "i18n", "Interpreter", "GIS", and "NLP" (< 57 issues) may influence the precision and F-measure values. We tested the classifier with only the top 5 most prevalent API-domain labels and observed no statistically significant differences. One possible explanation is that the transformation method used to create the classifier was Binary Relevance, which creates a single classifier for each label and overlooks possible co-occurrence.

The dataset is unbalanced due to the characteristics of the projects. Since JabRef, for instance, is a desktop application, the API-domain label "UI" appears more frequently. Table 5.10 shows the confusion matrix for the dataset containing all projects (for individual projects, see the appendix). This impacts the prediction of the minor labels even with the SMOTE algorithm, which improves the occurrences of rare labels. Some labels only appear in a few projects. Therefore, even when they are common in a specific project when training and testing with all projects, they may become rare. The recommendation

of labels with poor results should be avoided because of the risk of indicating a wrong skill to the contributor.

TABLE 5.10: Confusion matrix data and performance from the selected model with all projects

API-domain	TN	FP	FN	TP	Precision	Recall
APM	125	4	44	8	0.66	0.15
App	80	22	19	60	0.73	0.75
Big Data	152	0	29	0	0	0
Data Structure	78	24	6	73	0.75	0.92
DB	163	2	11	5	0.71	0.31
DevOps	113	26	0	42	0.61	1
Error Handling	97	32	5	47	0.59	0.90
Event Handling	162	1	8	10	0.9	0.55
GIS	178	2	0	1	0.33	1
Interpreter	173	2	3	3	0.6	0.5
IO	141	8	5	27	0.77	0.84
i18n	166	7	5	3	0.3	0.375
Lang	112	36	0	33	0.47	1
Logging	174	1	4	2	0.66	0.33
Logic	68	9	2	102	0.91	0.98
Micro/services	151	1	23	6	0.85	0.2
Network	175	0	6	0	0	0
NLP	164	0	17	0	0	0
OS	119	9	8	45	0.83	0.84
Parser	101	28	3	49	0.63	0.94
Search	134	9	15	23	0.71	0.6
Security	151	0	30	0	0	0
Setup	38	56	9	78	0.58	0.89
Test	166	0	15	0	0	0
UI	10	33	3	135	0.8	0.97
Util	84	4	16	77	0.95	0.82
Total	3275	316	286	829		

Despite the lack of accuracy in predicting the rare labels, we were able to predict those with more than 200 occurrences (all projects together) with reasonable precision (0.84) and/or recall (0.78). We argue the project's nature contributes to the number of issues related to their domain. For example, since the Audacity project is an audio editor and recorder, a high occurrence of "UI", "IO", and "Multimedia" labels is expected. We argue that Audacity's nature contributes to the number of issues related to the labels above. Labels with few samples suffered from low or unstable metrics. "DB", for example, varied from 0.09 to 0.9 in recall on predictions depending on the text/train split.

Improving the performance of BERT. We conjectured that these cleaning steps would increase the performance of the BERT pipeline. However, this was not the case. Common preprocessing steps, such as the removal of stemming and stopwords from an

Table 5.11: BERT - Corpus Tuning

Configuration Name	JabRef		Audacity		Powertoys	
	H	F	H	F	H	F
Unfiltered						
UNCLEAN	0.193	**0.718**	0.310	0.504	**0.187**	0.570
CLEAN	0.184	0.708	0.305	0.512	0.188	0.585
Filtered						
UNCLEAN	**0.168**	0.710	**0.259**	**0.609**	0.189	**0.618**
CLEAN	0.197	0.608	0.338	0.535	0.202	0.590

input corpus, significantly lowered the evaluation metrics for the BERT classifier within the context of our experiments. This starkly contrasts the TF-IDF pipeline, which benefits from very clean data. BERT benefits from a lightly cleaned or completely unclean corpus, as the empirical results demonstrated higher metrics without any cleaning steps applied to the input corpus. This is due to the nature of BERT, as it relies heavily upon the context of a sentence to leverage the meaning of each word. This necessitates little to no cleaning within the data used to train and test a BERT classifier. This light-cleaning approach is common within BERT models and publications. For example, [109] performs several cleaning steps, such as removing stop words and converting words to lowercase, when analyzing and training BERT on GitHub repository data such as the repository description and README [109]. However, we concluded that no cleaning was the most optimal data format for our Fast-Bert-based implementation's performance. Table 5.11 shows the F-measure results and Hamming loss results for different configurations of the corpus. The dataset filtered and uncleaned slightly overcame the cleaned and filtered option. Filtered removed only the templates imposed by maintainers to organize the body text in issues. The dataset was stemmed, and had the stop words, punctuation, special characters, and URLs removed.

In addition to the number of occurrences of a label, the BERT metrics can be improved by increasing the training set size. Wang et al. [110] and their exploration of several

trained deep learning models for GitHub labeling provide important insights into potential performance increases with BERT. The authors showed that the BERT model performed better than the other language models for large datasets with at least 5,000 issues, achieving the highest accuracy, precision, recall, and F-measure scores. However, for small datasets with less than 5,000 issues, CNN outperformed BERT as the best model overall. This suggests that BERT depends on the size of the training set of corpus data. Therefore, the performance of BERT when labeling GitHub issues will improve with an increased dataset size for the targeted open-source project. When the project datasets were merged (Table 5.9), the BERT metrics decreased the difference from about 26% to 6% in precision compared to the other classifiers.

What is the impact of the expert classification? Experts can also help increase the classification metrics for all models. We could observe the C++ project achieved the best F-measure compared with the Java and C# projects (0.84, 0.82, and 0.80, respectively, with small to large effect sizes). Although we evaluated only one C++ project, the results might suggest after examining Table 5.4 that the number of APIs evaluated by the experts impacts the metrics we will obtain. On the other hand, manual evaluation of a high number of APIs may make generalization unfeasible. The classification carried out by the experts in the C++ project comprised a higher percentage of APIs analyzed. This might be caused by the language characteristics: the libraries' names parsed from the C++ source code had limited information about their use. Thus, classification was more time-consuming. Indeed, the C++ project demanded more effort from the experts to classify it. Ultimately, it became a more detailed classification with better prediction metrics.

While experts' analyses are time-consuming, some outlier projects require much less effort than others. For instance, experts analyzed fewer than 3% of the APIs in RTTS. Since this project imports popular libraries, reuses many libraries across the entire source code and is modular, the expert's work was easier. A possible relationship between popular

APIs, modularization, and expert evaluation should be explored in future work. Another possible future work should identify what programming language characteristics impact the expert classification.

The performance of issue predictions was better in Portuguese (PT-BR) than in English, as shown in Figure 5.8. However, we suspect the experts' evaluation may have been biased towards Portuguese projects, as it was their native language. It is possible that internal libraries with Portuguese names used in the projects may have made the evaluation process easier for the expert team. This hypothesis requires further investigation. Additionally, since the project sample size was small, it could have influenced the classification process. In future work, we suggest exploring the impact of project size on classification and analyzing issues written in a variety of natural languages.

In contrast, developers' evaluations pointed to a 61.9% of labels being rated as "Very Important" to "Slightly Important." for the Portuguese project (Cronos) while 64.4% of labels being rated as "Very Important" to "Slightly Important", in the English project (RTTS) (Figure 5.10). The results are similar, and only one project for each language was evaluated by the developers. Thus, these differences should be better investigated in future work.

To what extent does the proposed method generalize? The semi-automatic classification process decreased the effort carried out by the experts to define the expertise of the APIs. Despite there being considerable effort remaining, as the dataset increases, the rate of new APIs to classify should decrease since projects reuse an average of 35-53% of core APIs. Third-party libraries account for 8-32% and 45% on average (Core + third-party). The use of popular open-source APIs could lead to an impressive 85% of shared APIs between projects [111]. Farther, the project sizes grow much more quickly than the size of uniquely-used API entities [111].

Thus, the demand for expert evaluation should decrease significantly when the number of mined libraries reaches a critical mass (for each programming language), and even new projects may use previous expert evaluations. This might impact the method when applied to industry projects, which may use a variety of unique non-free APIs. However, API sharing may happen inside companies or business units, repeating the phenomenon of the libraries' critical mass. Nevertheless, we did not observe this effect, and we could predict labels for an OSS project using data from an industry project.

To what extent does the model perform transfer learning? Transfer learning is crucial when projects lack data for training (cold start) or the time or infrastructure to develop their own models. This can be particularly problematic in the industry since the data can have restricted access due to security precautions or to comply with procedures or laws. In this situation, the ability to use a pre-trained model is necessary. Using pre-trained models to predict from new data is also desirable because it is faster and cheaper than retraining a model every time a new source project is added to the dataset [112, 113]. The projects may also benefit from the complementary data from another project when the project dataset is too small for training a predictive model.

The transfer learning experiments found a decrease in precision and recall. The metrics definition: $Precision = \frac{TP}{TP+FP}$ and $Recall = \frac{TP}{TP+FN}$ indicates the number of False Negatives (FN) and False Positives (FP) that should impact the results. For example, in training and testing individual projects, the RTTS project had a small number of PFs and FNs compared to the transfer learning experiment when RTTS was a target project (Tables 5.12 and 5.13). When targeting the RTTS project, the high number of FNs significantly decreased the recall metric. On the other hand, targeting PowerToys, the number of FPs negatively impacted the precision (Tables 5.14, A.6, and 5.9). The projects only shared a small number of labels (5 in 31) and are imbalanced among the datasets. For example, "Setup" is popular in the RTTS project and rare in JabRef, suggesting the conditional probability distribution of the sources and targets differ. These characteristics

might determine which projects match and, therefore, be used to decide the transfer learning source or target. Future work should investigate whether the domain, platform (Web, Desktop, Mobile), architecture, or other project property derives a good match. Furthermore, investigating proxy techniques, such as the one proposed by Nam et al. [112], to minimize the data distribution difference between target and source projects to predict software engineering defects can be applied to predictions of domain labels of API. Results for the JabRef (Table A.9) and Audacity (Table A.10) projects using transfer learning are available in Appendix A. We can observe the high number of FP and FN comparing the Audacity transfer learning results in Table A.10 and the results of training and testing the Audacity dataset alone (Table A.7). Similarly, we can observe the same pattern in the JabRef results in Tables A.5 and A.9.

Table 5.12: Confusion matrix and performance. Project RTTS trained/tested alone

API-domain	TN	FP	FN	TP	Precision	Recall	F-measure
APM	112	4	2	24	0.85	0.92	0.88
Big Data	134	1	2	5	0.83	0.71	0.76
Data Structure	25	36	3	78	0.68	0.96	0.80
DB	84	4	19	35	0.89	0.64	0.75
DevOps	60	17	13	52	0.75	0.80	0.77
Error Handling	126	5	4	7	0.58	0.63	0.60
Event Handling	129	0	2	11	1	0.84	0.91
i18n	121	6	7	8	0.57	0.53	0.55
Lang	28	29	11	74	0.71	0.87	0.78
Logging	47	24	18	53	0.68	0.74	0.71
Microservices	1	12	0	129	0.91	1	0.95
Network	65	20	21	36	0.64	0.63	0.63
Parser	69	17	14	42	0.71	0.75	0.73
Security	129	0	8	5	1	0.38	0.55
Setup	66	10	33	33	0.76	0.50	0.60
UI	4	15	0	123	0.89	1	0.94
Total	1200	200	157	715			

How did the contributors rate the labels generated for the issues they solved?

TABLE 5.13: Confusion matrix and performance: Project RTTS - transfer learning.

API-domain	TN	FP	FN	TP	Precision	Recall	F-measure
APM	197	0	38	0	0	0	0
Data Structure	100	0	135	0	0	0	0
DB	145	0	90	0	0	0	0
Error Handling	223	0	12	0	0	0	0
Event Handling	212	0	23	0	0	0	0
Lang	114	0	121	0	0	0	0
IO	44	21	131	39	0.65	0.22	0.33
i18n	214	0	21	0	0	0	0
Logging	102	22	91	20	0.47	0.18	0.26
Logic	13	17	146	59	0.77	0.28	0.41
Microservices	28	0	206	1	1	0.004	0.009
Network	129	0	106	0	0	0	0
Parser	156	0	79	0	0	0	0
Setup	115	14	98	8	0.36	0.07	0.12
Thread	175	0	60	0	0	0	0
UI	3	38	26	168	0.81	0.86	0.84
Total	1970	112	1383	295			

TABLE 5.14: Confusion matrix and performance: Project PowerToys - transfer learning.

API-domain	TN	FP	FN	TP	Precision	Recall	F-measure
APM	264	1	88	0	0	0	0
App	315	9	28	1	0.66	0.76	0.71
Data Structure	344	3	6	0	0.03	0.40	0.06
i18n	209	134	4	6	0	0	0
Interpreter	342	1	10	0	0	0	0
Logging	153	196	0	4	0.007	0.25	0.01
Logic	173	27	100	53	0.01	1	0.02
Microservices	0	348	0	5	0.25	0.49	0.33
Parser	6	105	10	232	0.07	0.13	0.09
Setup	351	0	2	0	0.50	0.07	0.13
Test	174	82	56	41	0.006	0.66	0.01
UI	105	245	1	2	0.68	0.92	0.78

Overall, participants evaluated the generated labels with positive feedback. The labels classified as important or very important across all the projects were: DevOps (10), DB (5), Services (4), UI (14), Lang (5), Security (1), and Logging (3). Moreover, the labels

that were classified as not important were: APM (8), Setup (7), Data Structure (6) and UI (9), and Security (2).

In the RTTS project, all four best-evaluated labels (Services, Logging, Setup, and Network) had precision above 0.75, and two of the four worst-evaluated ones (Data Structure, UI, DevOps, and Lang) had precision ≤ 0.7. A threshold could determine whether a label must be reported.

Participants from Cronos projects mentioned they would like to see the label "Data Structure" for the evaluated issues. This occurred because we removed the label Data Structure once it was generated for 90% of the issues selected in the Cronos project. One possibility for that case would be to include in the description of the project that it is strongly based on data structures and that the reported issues likely would involve this knowledge.

In addition, participants reported some labels could provide a clue for looking for the bug's root cause or determining the work needed to address a new feature request. For Example: *"...some of the labels could give an initial point of view to the reported issue"* (P2) or *"Network: While network tag wasn't that necessary for this particular case, the issue could have been caused by a communication error between the services in which case they would have been"* (P4). On the other hand, some participants preferred not to see more general labels, like Data Structure or Logging, since they are present in many issues: *"Data Structure is literally everywhere, there wouldn't be any program without them"* (P1), while others missed the Data Structure label (not present in the predicted list because it reached the 90% threshold) and suggested including it (P14, P16, and P17). Future work can determine how to address developer preferences regarding the inclusion of general labels.

The generalization of the method proposed in this paper assisted us in embracing more projects. Nevertheless, it also brought problems. We proposed generic labels able to

fit a wide range of project types. This might explain the comments about the generic labels. *"...It was a backward compatibility issue with user-defined configuration data, so with a generous interpretation Setup was accurate, but I would have preferred Information Model or Domain Model had it existed"* (P1). Analyzing the participant's suggestion for a "Validation" label, we recollect to the point where the NLP similarity suggested possible API domains for the library related to the issue and the experts' choice. We found the selected API domain was "Logic" since no "Validation" API-domain label was available. If the experts came from the project, perhaps the API-domain label "Validation" could be present and thus meet the participant's needs.

Future work can explore more API-domain labels to expand and propose more options to fit additional projects. Customizing labels for the project may generate more precise directions about the skills needed but will require more expert work time. On the other hand, generalization expands the method to a huge range of projects and can decrease the meaning level of the API domains.

5.7 Final Considerations

Once the approach has been generalized we would like to explore alternative strategies that can be adopted to address the "find an issue to start" problem. The next chapter aims to shed light on how the labeling strategy fits in the options available in the literature and how the stakeholders think about the strategies.

CHAPTER 6: STRATEGIES IDENTIFICATION

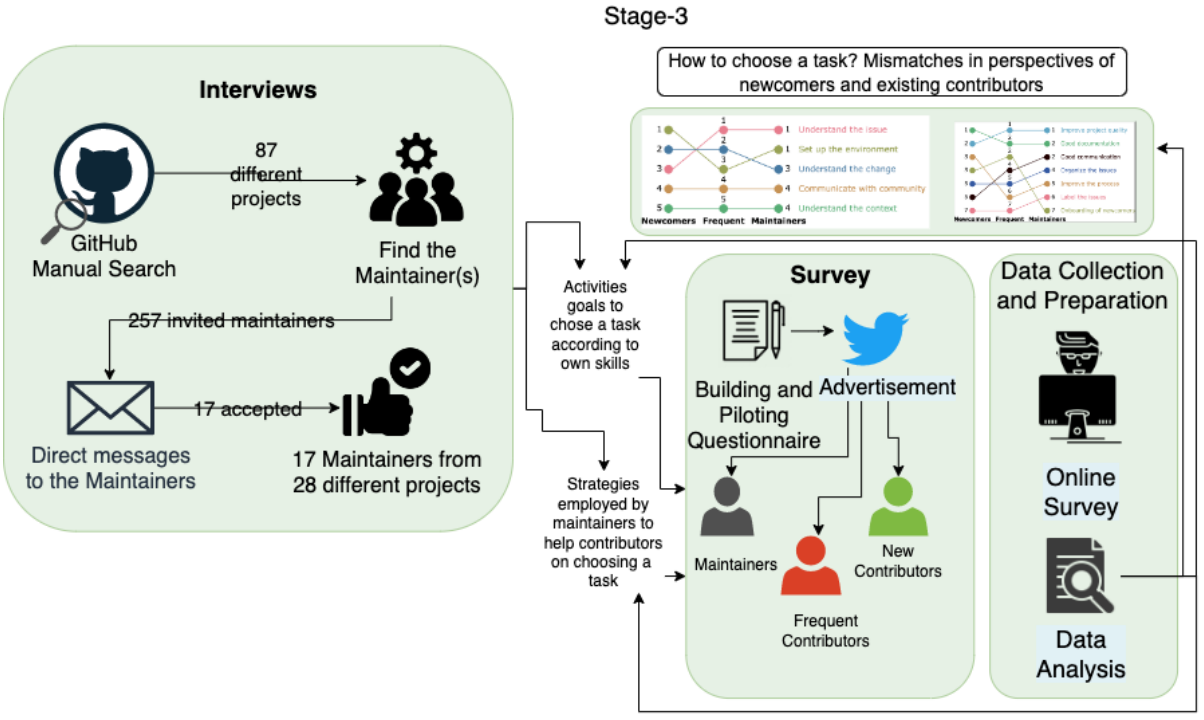

FIGURE 6.1: The Research Method - Stage 3 - Strategies Study

Given the results from the empirical experiment with possible contributors carried out in stage 1 (case study), we investigate what strategies communities adopt to help onboard new contributors and strategies contributors adopt to find a task to start work with.

Since we employed labels to assist contributors to pick issues in stage 1, we must dig deeper to verify how useful is that strategy from the point of view of communities and contributors.

Projects employ various strategies to assist newcomers in finding starter tasks. Existing work presents a compendium of such strategies [23, 114, 115]. For example, research has proposed labeling issues that signal newcomer friendliness (e.g., starter task, newcomer task, good first issue) [116] as an important strategy to aid newcomers in identifying tasks they can undertake [117]. Others have proposed mechanisms that aid newcomers in

understanding the issue to be solved [118]. Most of these strategies are based on research on newcomer barriers [2, 114] and contributors' recommendations for overcoming them.

A missing piece in our understanding of what newcomers need is how newcomers' perspectives fit in with those of existing contributors. A discrepancy between these two perspectives—newcomers and existing contributors—can create a gulf in expectations. Such a gulf, in turn, means that the projects' strategies are less likely to succeed, and newcomers continue to struggle.

Our goal was to investigate the difference in perspectives of newcomers and existing contributors in (i) the strategies newcomers use to choose a task and (ii) the strategies communities need to employ to support newcomers. Exploring the different perspectives can help OSS communities devise tailored strategies that match newcomers' needs.

We aim to answer the following research questions:

RQ1. What strategies help newcomers choose a task in OSS?

RQ2. How do newcomers and existing contributors differ in their opinions of which strategies are important for newcomers?

6.1 Method

The strategy investigation was carried out through interviews with maintainers and surveys with stakeholders (maintainers, frequent contributors, and new contributors).

6.1.1 Interviews - Building the strategies models

Our research goal was to understand the maintainers' perspective on (i) what strategies newcomers use to choose a task to work on; and (ii) what strategies the community can

take to help a newcomer choose a task. Due to the complexity of the phenomenon under study, we employed in-depth interviews (Figure 6.1 - Interviews).

6.1.2 Interviews Planning

We aimed to recruit project maintainers to talk about task selection while they browsed through their issue trackers. Therefore, we looked for active projects on GitHub and with recent pull requests and issues. We went through the list of popular projects and analyzed the projects' metadata, including the description, number of stars, number of forks, closed issues, pull requests closed, number of contributors, number of commits, and programming languages. Our goal was to select a set of diverse projects in terms of languages, sizes, and domains. We selected 87 projects.

Then, we identified the maintainers of these projects by observing the behavior of approving or rejecting pull requests, the comments, and the auto-denominated role in their profiles or the project repository. We selected 257 maintainers who had public attract usernames or email addresses publicly available on their profiles. We invited and interviewed them in random order until we could not find any new strategy related to our goals for three consecutive interviews. We offered interviewees a 25-dollar gift card as a token of appreciation. A consent letter was sent in advance along with a questionnaire to collect project and interviewee demographic information.

We conducted two pilot interviews to validate the script and time-box the interviews, ensuring that their duration would be about 60 minutes. Two researchers evaluated the responses and made minor adjustments to the instrument. The pilot interviews were discarded.

Our final sample comprised 17 maintainers in OSS, responsible for validating changes and performing merges in 26 different projects. This number is in line with what is foreseen

in the literature as a valid number to unveil the characteristics of a study domain [119]. Table 6.1 presents their demographics.

6.1.3 Data Collection

We collected the data using semi-structured interviews [120]. Three researchers experienced in qualitative studies conducted the interviews using a videoconferencing tool (15) or textual chat (2). We used a script (see Table 6.2) to guide the different areas of inquiry, while also listening for unanticipated information during the flow of the conversation.

The interviews revolved around the central question of *"how do newcomers choose an issue, and how can the community help?"* We approached this topic after establishing rapport with the interviewee by asking about their contributions. Then, the researcher asked the interviewees to open issues from their project and show how they could be analyzed and what newcomers could do to choose a task. We used the think-aloud technique while the interviewee navigated the issue track system. Interviewees reported

TABLE 6.1: Interview demographics (n=17) P* Prefer not to say

Participant ID	Years of Experience	Gender	1st contribution	Team member since
P1	20	M	2004	2007
P2	8	M	2014	2014
P3	15	M	2016	2017
P4	8	W	2014	2014
P5	8	M	2014	2016
P6	15	M	2006	2018
P7	3	M	2018	2019
P8	10	M	2018	2018
P9	7	M	2016	2016
P10	10	P*	2015	2015
P11	3	M	2018	2020
P12	2	M	2018	2020
P13	7	M	2017	2020
P14	15	M	2005	2017
P15	4	M	2017	2017
P16	20	M	2008	2019
P17	8	W	2016	2020

TABLE 6.2: Interview Script (excluding demographic questions)

[**RAPPORT**]
Q1 - What are the last projects that you contributed/maintained?
Q2 - What are your areas of expertise?
Q3 - In these projects, how did you choose tasks that fit your expertise?
[**SKILL TYPES**]
Q4 - Look at the issue #XX. Suppose you are a newcomer: How can you choose a task to contribute? Can you, please, think aloud so that we know what you are thinking

what strategies contributors should use to choose a task and how to prepare a project (usually using their project as an example) to help newcomers. Despite the pre-planned script, the interviewers took advantage of the opportunities that emerged during their conduction, using the principle of flexibility to obtain extra data [120]. A concluding part of the interview sought to obtain additional information and pointers to other potential respondents (snowballing). Two respondents were recruited using these leads.

With participant consent, we recorded all interviews. The first author of this paper transcribed the interviews, which lasted between 45 and 65 minutes. We used `Otter.ai` and listened to each recording, adjusting the corresponding transcriptions, mainly regarding technical terms and project names.

Our sample comprised maintainers across 26 OSS projects, including Spark, Apache Cordova, Brain.js, Microsoft PowerToys, Prisma, Azure Data Studio, ggplot2, Presto, bookdown, Godot Engine, Oppia, Jina.ai, Turn, and CASA. Some interviewees maintain more than one project. These projects vary in terms of the number of contributors (30 to 1,791 contributors), product domains (including infrastructure and user-application projects), and types (backed by foundations, communities, and companies). Table 6.1 presents the demographics of our sample. Because of the terms of consent, we cannot link each participant to their projects.

6.1.4 Data Analysis

We qualitatively analyzed the transcripts of the interviews by inductively applying open coding in groups. We built post-formed codes as the analysis progressed and associated them with respective parts of the transcribed text. The codes revealed strategies according to the participants' perspectives, who were identified as P1 to P17.

After identifying the strategies, we grouped them into a set of higher-level categories and produced two codebooks, one for newcomers' strategies to choose a task and another for the communities' strategies to help newcomers find a suitable task [1]. Three of the authors met once a week for three weeks to discuss and validate the results. The coding process was conducted using continuous comparison [121] and negotiated agreement [122] as a group. In the negotiated agreement process, the researchers discussed their rationale for categorizing each code until they reached a consensus [122].

6.1.5 Member Checking

After analyzing the strategies reported in the interviews, we conducted member checking to evaluate the validity of our interpretation and collect additional insights. We contacted via email the four participants who had agreed to a follow-up meeting (P2, P4, P14, and P16), sending them an editable visual representation of the description of each strategy. Participants could give feedback by email, annotating the visualization directly, or through an online meeting. Participants P14 and P16 scheduled a virtual meeting, whereas P2 and P4 gave their feedback over email. The virtual meetings lasted about 15 minutes. During the call, we explained the overall definitions of the first level of Fig. 6.3 and 6.4, and asked for suggestions. The email had two questions: What do you think about this model? Did we correctly place your view in the model? The four participants

[1] `https://doi.org/10.5281/zenodo.6508776`

(P2, P4, P14, and P16) verified that the strategies we generated reflected their views. We identified some misunderstandings related to some terms and updated our model to make them clearer.

6.1.6 Survey - Understanding the relative importance of the strategies

We conducted an online survey to obtain the perspectives of a variety of developers on the strategies identified during the interview (Figure: 6.1 - Survey).

6.1.7 Survey Planning

In the survey, we present the newcomers' strategies for choosing an issue and the strategies that communities use to help them. We asked respondents to rank the relevance of each strategy. We also included demographic questions about experience, age, gender identity, and country of residence.

We advertised the survey on social media and community blogs (e.g., Linkedin, Twitter, Facebook, and others). We also sent direct messages to OSS contributors and discussion lists. We offered the participants a chance to enter a raffle for US$25 gift cards to encourage participation.

6.1.8 Data Collection

The survey was available between October 8 and November 2, 2021. We received 209 non-blank responses and filtered out data to consider only valid responses. We analyzed the attention check answers, time to complete the questionnaire, equal/similar e-mail addresses, and inappropriate answers to the open questions (e.g., "XXX," "No," "There

is No," "N"), resulting in 64 valid responses. We present the demographics of the survey participants in Fig. 6.2.

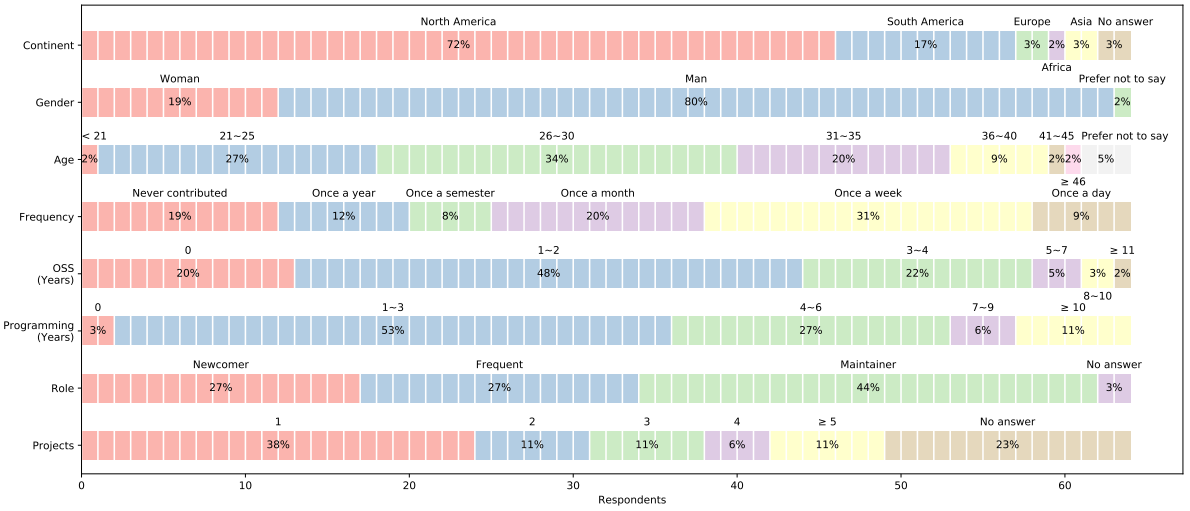

FIGURE 6.2: Personal characteristics of the survey respondents (n=64)

6.1.9 Data Analysis

We used the Schulze method to rank the strategies and their association with groups from the demographic data [123, 124].

6.1.10 Schulze Method

The Schulze method [124] is an election method that computes a single ordered list of preferences (ranking of candidates) from a set of votes, in which each vote represents an ordered list of preferences on its own. That is, each voter selects all the candidates that they prefer in order, ranking them, and the Schulze method aggregates all the rankings into a single winning ranking, or optionally an ordered list with or without ties. This method is considered a Condorcet method. Hence, it prioritizes votes for candidates who win the pairwise comparisons against each candidate in every head-to-head election

scenario possible. This election method has been used for elections and decision-making by the Debian project, Ubuntu, Gentoo, the Wikimedia Foundation, political parties, and others [125]. In our case, we combined the rankings provided by the survey participants to find which strategies have higher relative importance for each group.

6.1.11 Schulze Setup

The Schulze configuration considers the ordered preference of the factors each participant selected that define the relevance of the strategies. In our case, the strategies identified in the previous interview stage are the factors. We created the ballot list by aggregating the number of times each ranking order was chosen. We used the R package "votesys" [126] to compute the list of the most voted strategies using the Schulze method.

See supplemental material [2] for the questionnaire, codebooks, and sample answers.

6.2 Results

In this section, we present the results of our investigation of the strategies grouped by research question.

6.2.1 RQ1. What strategies help newcomers choose a task in OSS?

To answer this question, we interviewed maintainers to understand their perspectives on (i) what strategies a newcomer uses to choose an open issue; and (ii) what strategies the OSS communities can use to help newcomers choose tasks.

[2] `https://doi.org/10.5281/zenodo.6508776`

6.2.1.1 Newcomer strategies to choose a task

From the interviews, we could identify 27 strategies that maintainers expect newcomers to use to choose a task and grouped them into five categories, as presented in Fig. 6.3. In the following, we present more details about our findings, organized by strategy category.

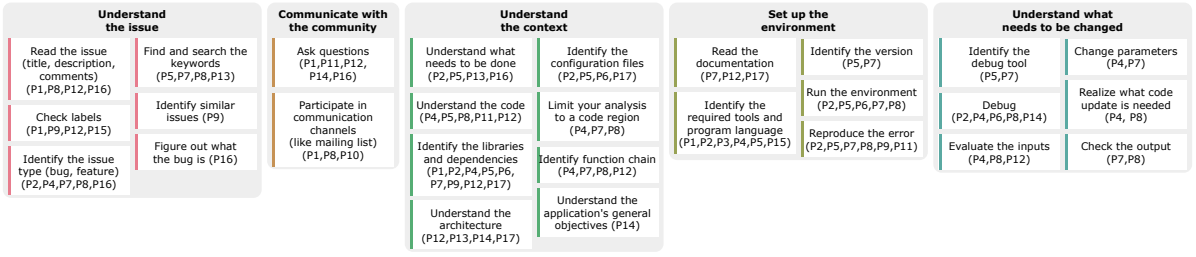

FIGURE 6.3: How newcomers choose their tasks (according to the maintainers).

Understand the issue

According to the maintainers, newcomers should understand the issues beyond their titles. The specific strategies under this category are presented in the first column of Fig. 6.3. The main focus for newcomers is finding signals to help them match their skills with appropriate issues. In this sense, reading through all the issues' information (title, description, comments) and checking issue labels, type (bug/feature), and keywords help newcomers to find meaningful signals relevant to solving the issue (e.g., class names, method names, component, library, etc.). For example, one interviewee mentioned that if the newcomers want to learn if the issue is interesting for them *"that can be concluded from reading the entirety of the proposal and reading the discussion about it. And then the actual code fix is very simple"* (P12).

Communicate with the community Maintainers mentioned the importance of communicating with the community as part of the decision process. A newcomer who does not completely understand an issue should contact to receive support from the community. Specific strategies related to it include (i) posing questions to maintainers or other contributors and (ii) staying in touch with the community to learn about the project

and project roles. As P10 stated: *"usually you can figure out it [...] by talking to other contributors or peers"*.

Understand the context To choose a task, maintainers highlighted that it is important for newcomers to know the context of the problem. Newcomers need to capture details that may help them to define a solution. To do this, newcomers need to read and have a high-level understanding of the codebase and the libraries used. Furthermore, it is suggested that they understand aspects related to the software architecture, like dependencies and configuration files. Maintainers also reported that newcomers should attempt to foresee the scope of the change. If newcomers are not able to understand the context where the issue is, they will probably change more code than necessary to solve the issue, increasing the chance of introducing new bugs: *"The point is, you should know what feature we are working on."*(P13). Fig. 6.3 (third column) lists the strategies related to understanding the context.

Set up the environment

To prepare a solution and submit a pull request, first, the newcomer must identify the appropriate tools to build the software locally. "Set up the environment" is a landscape exploration task since the contribution guidelines documentation is not always up-to-date or does not comprehend all possible operating systems, library versions, and other details. It is also a playground to understand configuration files and the project structure. Contributors need to try to set up the environment to check their skills before proceeding. In addition, reproducing the error is part of the process, as P5 witnessed: *"looking at the debugger [...] we can get clues of what's happening. But for sure, we will want to reproduce it."*(P5). The fourth column of Fig. 6.3 shows the strategies under the Setup environment category.

Understand what needs to be changed Once the newcomer has set up the environment and realized the overall architecture and extent of possible updates that need to

be made, it is time to dig deeper and identify application behavior by changing inputs and verifying how outputs respond to changes. A debugging tool is really useful here, because even having a general idea about the context, the code can often be complicated. When the code is complex, the contributors must analyze the values of the variables and run the code step by step, also changing the values of the parameters of the function. Maintainers claim that once newcomers have a general understanding of the underlying logic, they will be confident about the task. *"... this exception is happening, because somebody added this line. Okay, well, what happens if I remove this line? Does it work? Does something else break? Where is this line used?"* (P8).

6.2.1.2 Community strategies to facilitate task selection

In addition to the strategies that newcomers are expected to take, we found 40 strategies that the communities take to help newcomers choose their tasks. From these strategies, we derived seven categories of strategies, as presented in Fig. 6.4.

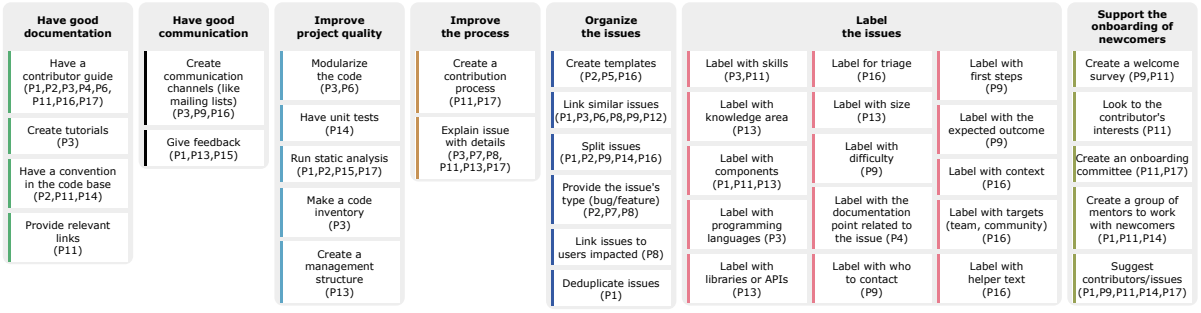

FIGURE 6.4: Community strategies to help newcomers finding a suitable issue.

Have good documentation A way that the community can support newcomers is by providing appropriate documentation. It is important, for example, to arrange guidelines that contain the necessary information to help them understand the contribution process, standards, and how to contact the community. In addition to traditional documentation, providing tutorials to cover crucial aspects related to project architecture and technology is also important. Pointing newcomers to good examples to be followed (issues, commit

messages, etc.) is another point. Participant P2 pointed out that *"the minimum entry-level is just a knowledge of software engineering. Python in this case, and then just following the tutorial, so some patience basically to understand the documentation and so on"*(P2).

Have good communication This strategy is fundamental to making newcomers feel welcome and comfortable discussing problems not covered by the documentation. Having channels specific for onboarding questions and asking for help when starting is something that communities may put in place. One of our interviewees confirmed the importance of the community showing good/appropriate communication skills: *"I see a lot on GitHub, big, big projects with tons of issues. And they take a lot of time to react to comments... And I think that this engages a lot. I think that you have to give time for people to figure it out. But keeping them weeks or months without an answer usually would be too much"*(P13).

Improve project quality Quality improvement aims to find easier ways to fix and evolve the code. One strategy that helps is by having the code organized clearly and explicitly in a modular way. This is mentioned by P3, who said: *"That is yet convenient here: the code of –software name– is structured by module, and each module has a folder."* This facilitates, among other things, locating the pieces of code related to specific issues and features. Keeping the code covered with unit tests and providing static analysis tools make it convenient for newcomers to understand if their code is following the standards.

Improve the process Improving the process was mentioned by participants both in terms of (i) creating a contribution process so newcomers can *"learn where they can contribute"* (P1) by *"going through the official onboarding process"* (P11). At a more granular level, the process of creating a task should guarantee that the proponent is going to (ii) explain the issue with details because *"if the domain knowledge is missing, it is a lot harder for someone to join in"* (P3). Besides explaining the issue, details can

also include previous solution attempts and results, so a contributor is aware of previous strives and avoids rework (P8).

Organize the issues Issue organization benefits the newcomers and the overall team by helping the project prioritize and allocate the right resources to the right issues. Regarding the issue itself, (i) creating a template help to standardize details and guide the author of an issue to fill in the expected data (P16). The template can include (ii) a link to similar issues, which is a detail that can *"inspire the contributor"* (P9) on how to solve the issue. The strategy to (iii) split issues avoids driving newcomers away due to complexity. The issue can have *smaller sub-issues that can be taken by new people, and subsequently added everything so that we can close the issue in the end* (P9). When it comes to the issue tracker, (iv) providing the issue's type that could be used on a filter helps newcomers to reduce the number of choices from a long list if they would prefer to work on a specific type of issue (e.g., feature request or backtrack) (P7). Not only the type but knowing the (v) impacted users can entice newcomers to pick a task by *"knowing how many people are facing this issue"*. Although requiring a manual effort, (vi) *deduplicate issues* (P1) is a strategy to avoid rework by having more than one issue to the same task.

Label the issues Labeling could be part of the issue organization due to its straight relationship. In fact, labels can be a way to provide the issue's type, indicating whether they represent bug reports, feature requests, or other types of tasks. However, due to the great number of ways of labeling identified by the interviewers, we decided to create a category for the labels' strategies. Our participants mentioned they would like to have labels with *"specific skills would be required to solve the issue"* (P11), for example, *"skill: documentation‘ or skill: ruby"* (P3). Regarding technical skills, participants brought out the need to have labels with knowledge area (P13), components (P1, P11, P13), programming language (P3), and libraries or APIs (P13).

The status of the issue can be part of a label that shows if an issue is still in triage or even under ongoing work by another contributor (P16)—in that case, a newcomer can decide to join and collaborate. Both size (P13) and difficulty level (P9) were mentioned in terms of effort and complexity. As for assistance in understanding the issue, P4 recommended having a label to the point of documentation related to the issue, so newcomers can have a better picture of the piece of software they will deal with. In case of questions, when having *"labels regarding who to contact if you need help with that issue?"* (P9), a newcomer can feel safe having someone to contact with. When coming to action, a label with first steps and expected outcomes (P9) can also be a helping hand to newcomers on the pathway to solving the issue. P16 points to the necessity of labeling the issues with the context of the project and indicating not only the target (*"if you don't have context, you don't need to know what accessibility end is. But my team needs to know what that means."*), but also including helper texts that describe the labels.

Support the onboarding of newcomers Supporting the onboarding can include identifying the newcomers' characteristics and potential. It can be done with a survey identifying their skills and interests. An onboarding committee in charge of the integration may define the policies and goals. Knowing the newcomers' potential and interests, the community can recommend a mentor *"to make it easy for a certain person to contribute more"*(P1). The mentor may indicate some easy tasks as *"first issues which will help [the newcomer] to get familiarized with the code base"*(P11).

6.2.2 RQ2. How do newcomers and existing contributors differ in their opinions of which strategies are important for newcomers?

To compare the relative importance of the strategies from the point of view of different stakeholders, we used the Schulze method [124] to combine the rankings for (i) newcomers

and (ii) community strategies. In the following subsections, we present the rankings and how they compare.

6.2.2.1 Mismatches in newcomers' strategies

Figure 6.5 presents the ranking of the preferences from the three groups: newcomers, frequent contributors, and maintainers. The numbers in the figure represent the position of each strategy in the combined ranking (it is possible to have ties).

Regarding strategies that newcomers are expected to use to choose a task, frequent contributors and maintainers had very similar values. However, while the Schulze method found a clear sequence of importance for frequent contributors, two ties occurred for maintainers: "Set up the environment" and "Understand the issue" tied in the first position; and "Communicate with the community" and "Understand the context" tied in the last position.

Newcomers also had similar rankings. However, they value "Set up the environment" more than "Understand what needs to be changed". The former appears in the first position, while the latter—which appears in the first position for frequent contributors and maintainers—appears in the third position for newcomers. Finally, the "Communicate with the community" was ranked in the penultimate position by all groups.

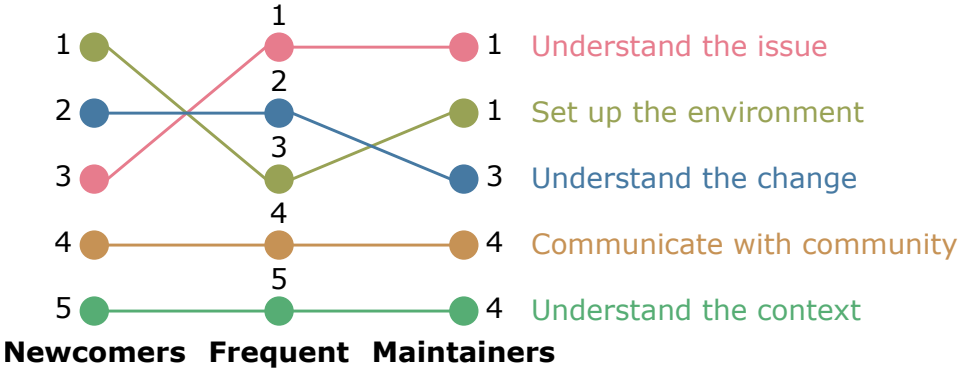

FIGURE 6.5: The relative importance of newcomer strategies

6.2.2.2 Mismatches in maintainers' strategies

Fig. 6.6 presents the combined rankings for maintainers' strategies, according to each stakeholder. Once again, the perspective of frequent contributors and maintainers are similar, with one standout difference: "Support the onboarding of newcomers".

The "Improve project quality" strategy was ranked as the top-ranked strategy for frequent contributors and maintainers. We have almost an agreement since newcomers ranked it in second place. The most important strategy according to the newcomers was "have good documentation", which is also tied as the second most important strategy for frequent contributors and maintainers. Newcomers and frequent contributors agree that "Label the issues" is the least important strategy. Maintainers also agree with its low importance, ranking it in the sixth position.

We also found some mismatches. For newcomers, "good communication" is only the fifth strategy contrasting with the second place for maintainers (tied with "good documentation") and the fourth for frequent contributors. Another mismatch regards the category "improve the process." It was ranked third according to newcomers, but its ranking dropped significantly for frequent contributors and maintainers (sixth and fifth, respectively). Still, for "Support the onboarding of newcomers," while it was ranked last for the maintainers, it was the third for newcomers and second for frequent contributors. This is surprising since the intuition we had was that maintainers should prioritize the onboarding process to count on human resources to work on the issues.

6.3 Discussion

Why does the convergence of relative importance matter? An OSS project is a challenging environment composed of diverse team members with a variety of experience

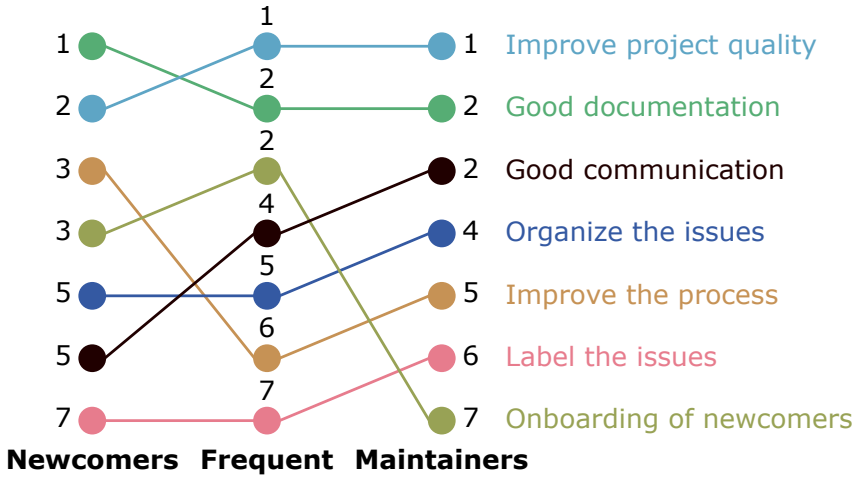

FIGURE 6.6: The relative importance of community strategies

levels and informal relations [127]. Within a dynamic organization, it is difficult to identify the competencies of each member, as people have different styles of development, are physically distant, and lack a structured working relationship [127]. In this environment, knowing the community's interests and concerns and managing to converge them can help better manage the project. For example, Steinmacher et al. [128] report difficulties mentors face in assisting newcomers. Lack of information from maintainers about newcomers denies assistance and makes the prioritization of the onboarding process harder.

In our results, we found a high convergence between frequent contributors and maintainers, both in terms of which strategies newcomers use to choose a task and strategies communities can use to support newcomers in choosing a task. However, newcomers have different interests and concerns. This discrepancy between perspectives might create a gulf of expectations and misunderstandings, making newcomers struggle, and maintainers mismanage the project with ineffective strategies.

Maintainers and contributors fight the same battles from different perspectives. Although contributing to a project is an overarching goal shared by everyone, maintainers and newcomers have different objectives. Maintainers are concerned with keeping the project running smoothly, attending to their customers, and managing the

workload. On the other hand, newcomers may be looking for the benefits of the contribution to their career or the project directly. Thus, easy access to technical tips through documentation and project quality verification plays a special role when looking for a task to start with.

A recent study about the shifts in motivation [129] confirms the diversity of reasons that newcomers join and senior developers keep contributing. The first group wants to learn and aims to improve the career (indeed, the learning process may leverage the career–extrinsic motivation [129]). Therefore they are thirsty for projects with good documentation, whereas the experienced contributors want quality over documentation and good communication channels to ask questions. They aim for altruism or ideology (intrinsic motivations) [129]. Since regular contributors believe the onboarding process is a priority, altruism might direct them to help newcomers.

A study for the Linux Kernel OSS project [130] shows the number of files and commits particularly grows in some modules, while the flow of joiners is stable or even drops. Also, the maintainers' effort increased with author churn [130]. This is particularly observed in many OSS projects. As they cannot count on more newcomers and face team churn, investing time in documentation and quality seems to be aligned with our results.

Maintainers have a deep knowledge of their projects and the ideology they implement. Therefore, the main important task is to improve the project quality and understand the issue's content. As high-ranked officers, they know the battlefield. On the other hand, rookies carefully assess the environment before engaging in a project. Therefore, the ability to set up the environment is crucial to the first contribution.

The work of den Besten et al. [131] shows evidence that open-source project allocation is influenced by code characteristics and complexity. One may be able to assess the skills and the complexity level of a task by looking into the documentation, figuring out how to set up the environment, and identifying the complexity of the change. Newcomers

like to start with a specific kind of problem, involving a less complex, contained, and low workload [116]. Sarma et al. [118] proposed BugExchange: a tool to help newcomers find a task while pointing to related documentation, recommend issues, and communicate with near-peer mentors. The idea behind the tool is to create a learning environment and to aid newcomers to climb the issues' complexity step by step.

As contributors mature and become frequent contributors, they navigate project issues and find the resources they need. In fact, the results showed a decrease in the priority of the "set up the environment" strategy.

Multi-teaming needs documentation and collaboration support. OSS projects usually are multi-teaming (i.e., projects whose members work on multiple projects simultaneously). Therefore, multi-teaming and OSS research have a common ground. Multi-teaming research corroborates the idea brought by OSS communities. The plurality of members may leverage the knowledge inside the project, but, on the other hand, it can hamper coordination and fragment the team's attention [132]. A proposed solution for multi-teaming is the use of information systems to support collaboration and a central repository (i.e., platforms like GitHub) for knowledge modeling or specific tools like the dashboard proposed by Guizani et al. [133]. The information systems may be seen as a document repository and a collaboration platform to assist team members in addressing the shared cognition problem by enabling information flow [132]. Since newcomers seek knowledge and tasks to which to contribute, it meets our findings for good documentation and a quality project. Maintainers, as project managers, must be aware of contributors' needs and prepare the project's repository to meet contributors' expectations. Our findings suggest that maintainers should invest in well-written documentation, a communication channel for the team, and project quality improvement.

Leveraging related research is perhaps a good way to avoid reinventing the wheel. Multi-teaming research may borrow ideas to address the team management encompassing

strategies to integrate newcomers, manage the quality, and prepare the project to use a contribution process suitable to dynamic teams with high churn volume, difficulties of communications, flexible hierarchy, and diverse levels of members commitment [132].

The paradox of choice. This paradox emphasizes the greater the number of options we have, the less satisfaction we will derive from our decisions [134]. When newcomers open an issue tracker list and encounter many open issues, they can struggle to find the most suitable task to contribute to and often give up. When people do not have a strategy to elect viable choices, a decision can become overwhelmed by the options, reducing the likelihood of making a good choice and leading to frustration [134]. Strategic thinking involves planning and thinking. Planning includes analysis and procedures, whereas thinking involves synthesis—encouraging intuitive, innovative, and creative thinking [135]. While the list of issues will continue to exist and, in many cases, as a long list, our results suggest strategic thinking to help a newcomer when choosing a task to contribute. We provide suggestions for both the newcomer (Section 6.2.1.1) and the community (Section 6.2.1.2) to mitigate the paradox of choice in the issue tracker list.

Strategies used in practice or suggested by maintainers are not well-documented in the literature. Despite the recent literature covering several of the strategies that maintainers can use to support newcomers, we are surprised that after many papers about the topic, we still found other strategies. For example, GitHub projects employ many labeling strategies, such as "Label with Components".[3] However, we found other approaches. For example, labels for knowledge area, expected outcome, and context were proposed by our interviewees.

We also found some new strategies to address the organization of the issues. For example, it would be interesting to link the issues with stakeholders who would benefit or be impacted (not developers working on it). This would be valuable since the business unit or customers interested in the solution of the issue would be explicit.

[3] `https://github.com/JabRef/jabref`

The importance of the newcomers' strategy "Setup the environment" is probably due to the increasing complexity of recent applications and the plurality of the configurations. Since OSS projects are not contained in a single company, configuration management (CM) is hard to pursue, creating additional challenges for this strategy [136]. Future work can address specific strategies to handle this complexity, focusing on CM for newcomers.

While the strategies proposed in the literature barely tackle which strategies newcomers should use to communicate with the community, some communities' strategies may help to increase the confidence of newcomers, such as acting with kindness and putting effort to help newcomers feel part of the team and not afraid of the community [6].

6.4 Final Considerations

The strategies are now explored and we have an opportunity to pave a future road map on how to address the "find an issue to start" problem with many options. Next, we want to explore the impact of the collaborative work on the API-domain label predictions with the goal of improving our metrics.

CHAPTER 7: SOCIAL METRICS

Given the sociotechnical nature of software engineering [17], we hypothesize that social metrics capture communication patterns that can help predict API-domain labels. Social metrics have shown promising results in other contexts. For example, [19] evidenced the usefulness of the communication context and communication roles to predict co-changes. [20] predicted defects using developer network metrics. [21] used social network analysis to predict software failures.

The rationale behind using collaboration data available in conversation text on GitHub issues for skills predictions is that the communication context and social network analysis (SNA) encapsulate information about the project. Commenters with specific skills are likelier to participate in discussions where their skills are required. The role of each commenter within a social network constructed from the comments on the issue may reflect how much knowledge is needed to complete a task. Doc2Vec creates different features from the corpus and might improve the metrics compared with TF-IDF.

By understanding how communication context and SNA play a role as predictors, we aim to improve our API-domain label prediction model and increase its precision, recall, and F-measure. More broadly, we also shed light on how social aspects of software development related to the technical knowledge involved in the tasks.

We aim to answer the following research questions:

RQ1. To what extent can social metrics improve the prediction of API-domain labels?

RQ2. To what extent can we transfer learning among projects using social metrics to predict the API-domain labels?

7.1 Definition of Social Metrics

Following Wiese et al.'s [19] work, which leveraged social metrics from systematic literature reviews [22, 137, 138], we organized social metrics into three categories: Communication Context, Developer's Role in Communication, and Network Properties. In this section, we discuss the metrics we used in the paper. We bring more details on how we operationalized these metrics later in Section 7.2.3.

7.1.1 Communication context

The communication context category involves aspects from the discussion around issues. For example, Meneely et al. [21] used metrics from historical communication contexts to predict software failures. In our study, following Wiese et al.'s work [19], the communication context variables include the number of issue comments, the number of commenters, and wordiness. The intuition behind this idea is that comments, participants, and words might be correlated with the nature of the tasks, which can be a proxy for some API domains. For example, issues that are more discussed or involve more people in an approval process might be related to some complex aspect of the software and, therefore, point to some architectural component or domain [18]. The opposite may also be considered: issues with few and short discussions may point to components that use API domains that usually require less explanation and discussion. We collected the communication context by issue (see Section 7.2.3).

7.1.2 Developer's role in communication

The developer's role in communication category includes the betweenness centrality and closeness centrality metrics, which we abbreviate as "betweenness" and "closeness." These

metrics aim to capture the developers' roles in communication [139]. Developers involved in a discussion have different values of betweenness and closeness, and some have a more central role while others have a more peripheral one [140]. The participation of central developers in discussing an issue may also be a proxy for the skill or relevance of the issue and may correlate with certain API domains.

Betweenness, as expounded by sociologist Linton Freeman, is defined as the sum of a point's "partial betweenness values for all unordered pairs of points where $i \neq j \neq k$", where "partial betweenness" is defined as "the probability that point p_k falls on a randomly selected geodesic linking p_i with p_j" [141, 142]. In short, it is the "[n]umber of times that a node acts as a bridge along the shortest path" between two other nodes in a network [19]. Linton provides the formula

$$C_B(p_k) = \sum_{i<j}^{n} \sum^{n} b_{ij}(p_k) \tag{i}$$

where $b_{ij}(p_k)$, the partial betweenness of point p_k between two nodes, is given by the formula

$$b_{ij}(p_k) = \frac{g_{ij}(p_k)}{g_{ij}}$$

where $g_{ij}(p_k)$ is "the number of geodesics linking p_i and p_j that contain p_k" and g_{ij} is "the number of geodesics linking p_i and p_j".

Closeness refers to "the number of steps required to access every other vertex from a given vertex" [143] and is given by the formula

$$C'_c(p_k) = \frac{n-1}{\sum_{i=1}^{n} d(p_i, p_k)} \tag{iii}$$

where $d(p_i, p_k)$ is "the number of edges in the geodesic linking p_i and p_k", and $\sum_{i=1}^{n} d(p_i, p_k)$ is the sum of "the geodesic distances from that point to all other points in the graph" [142].

Obtaining the metrics for developers' roles relies upon a matrix to represent the discussion in periods, as discussed in Subsection 7.2.3.

7.1.3 Communication Network properties

Finally, the communication network properties category comprises communication graph metrics like the number of nodes, edges, diameter, and density obtained upon the communication network. The intuition behind using these metrics is aligned with the Task-Level view of Conway's Law, which conceptualized a unit of work as one that developers are engaged with, such as a task in the issue-tracking system [73]. In this case, the communication structure of an issue may map to the solution.

Mathematician and graph theory scholar Frank Harary [144] that "[t]he diameter $d(G)$ of a connected graph G is the length of any longest geodesic" [144, p. 14]. The formula, given by West [145, p. 71], is

$$max_{u,v \in V(G)} d(u, v) \tag{iv}$$

where $d(u, v)$ is the distance between two vertices in G.

The density of a simple, directed graph G is "calculated as the percentage of the existing connections to all possible connections in the network" [19], a ratio with an antecedent of the number of existing edges and a consequent of the total possible edges. Diestel gives the formula [146, p. 164]:

$$\|G\| / \binom{|G|}{2} \tag{v}$$

where $\|G\|$ and $|G|$ are the number of edges and vertices in the graph, respectively.

We also formulate the following hypotheses:

H_0^1**.** There is no significant difference in adding communication context features to predict API-domain labels.

H_0^2**.** There is no significant difference in adding the Developer's Role in Communication features to predict API-domain labels.

H_0^3**.** There is no significant difference in adding Communication Network Properties to predict API-domain labels.

7.2 Method

To answer the research questions and test the hypotheses, we followed the method below (Figure 7.1):

We predicted the API-domain labels by building the datasets (Section 7.2.5) to assess the defined hypotheses: H_0^1-H_0^3. We included in the dataset the SNA and communication features related to H_0^1 - Communication context; H_0^2 - Developer's Role in Communication; and H_0^3 - Communication Network Properties. Once the dataset is prepared, the classifier will read it and produce the outcome metrics (Section 7.2.8). Next, for each hypothesis, we ran models configured with the Control and Treatment variables (Table A.11 in Appendix A) and compared them using the statistical tests mentioned in Section 7.2.8. To test H_0^n (n = 1..3), we compared the model with the Control independent variables (TF-IDF weights) with a model with Treatment independent variables related to H_0^n (TF-IDF weights + independent variables for H_0^n). For example, to test H_0^2, we compared precision, recall, F-measure, and Hamming Loss from the TF-IDF weights and the H_0^2 (betwenness+closeness)+TF-IDF weights.

We predicted the API-domain labels combining the SNA features to find the best precision, recall, and F-measure combination. Thus, we built datasets with the best SNA predictors found in descending order by the feature importance function list. The results metrics (Section 7.2.8) were compared by projects with the baseline. We also investigated the transfer of learning across projects—we trained all possible models with two projects (source) to predict labels in one (target). The results were compared with the baseline[23].

As we replicated and extended the work from Santos et al. [23], we partially reproduced phases 1, 2, and 3 from that study. We followed the method presented in Figure 7.1. We included Santos et al.'s [23] case-study project (JabRef) in our study and added two other projects with different characteristics (Audacity and PowerToys), as we discuss in Section 7.2.2.

In phases 1 and 2, we collected and filtered closed issues and pull requests from the project repositories. As we needed the source code that solved each issue to identify the APIs and train the classifiers, we filtered out all the issues not linked to pull requests (Section 7.2.3). Then, we parsed the source code changed by each pull request, looking for APIs used in each artifact. We parsed the declarations based on the language of the projects. For JabRef, which was written in Java, we looked for "import" statements; for Audacity, written in C++, we parsed "include" statements; and for PowerToys, written in C#, we looked for "using" statements. Finally, we categorized all the APIs using the process described in Section 7.2.4.

In phase 3, we built the corpus using TF-IDF as our baseline, following the study by Santos et al. [23]. Finally, we split the dataset into training and testing sets, using ten cross-validations. Following Santos et al. [23], we employed MLS SMOTE to improve the rare labels in the imbalanced dataset (Section 7.2.6) and employed the Random Forest classifier to predict the API domains (Section 7.2.7).

In phase 4, we created the dataset with the social metrics. We mined the conversation data from the issues, as defined in Section 7.2.3 and illustrated in Section 7.2.3. Next, we investigated how each metric impacted the predictions and ran the Random Forest classifier with the newly gathered data. Finally, we analyzed (Section 7.2.8) the performance considering groups of predictors divided into hypotheses (Section 7.2).

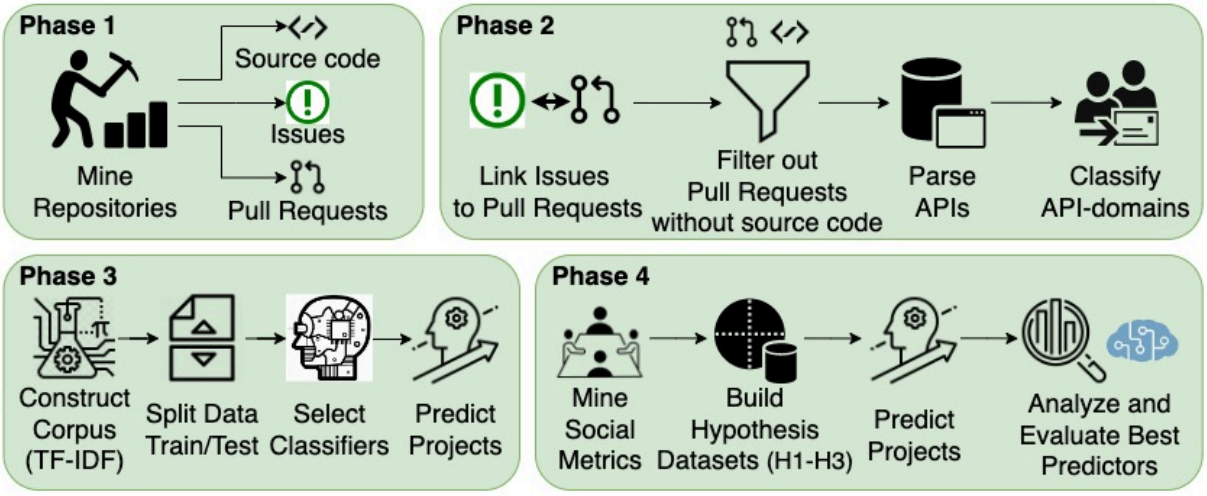

FIGURE 7.1: Method Overview - Social Metrics - Stage 4.

To answer RQ1 (To what extent can social metrics improve the prediction of API-domain labels?), we predict the API-domain labels with and without the social metrics to find the best precision, recall, and F-measure outcomes. We built datasets with the best social predictors ranked by feature importance.

We sought to predict the API-domain labels by building the datasets (Section 7.2.1) to evaluate the defined hypotheses (Section 7.2). Therefore, we included in the dataset the social metrics hypotheses defined above. Finally, we ran models configured with social metrics to compare with [23] features (baseline) and ran the statistical tests mentioned in Section 7.2.8.

To answer RQ2 (To what extent can we transfer learning among projects using the social metrics to predict the API-domain labels?), we ran a transfer learning experiment from

the project that obtained the best results for the others to verify how transferable the predictions are. The projects' metrics (Section 7.2.8) were compared with the baseline.

7.2.1 Mining OSS Repositories

The mining stage was composed of the project selection, mining the issues and pull requests, and mining the social metrics, mimicking Stages 1 (Case Study) and 2 (Generalization Study) (Sections 4.1.1 and 5.1).

Project Selection We selected the OSS projects used in the generalization study. They have different programming languages, more than 6K stars, and diverse goals. We also sought active projects with recently posted issues and solved pull requests. Therefore, besides JabRef (from Santos et al.'s [23] work), we mined two other OSS projects: Audacity and PowerToys. We limited the number of projects to three since our study involved hiring experts to expand the API-domain labels adopted in the previous work [23].

The first project we mined was JabRef [147], an open-source bibliography reference manager. In Jan 2023, the project had 2.9k stars, 3.4k closed issues, 5.7k closed pull requests (PR), 17.7k commits, 37 releases, 2k forks, and 492 contributors.

Next, we mined Audacity and PowerToys. Audacity is an audio editor written in C++, which had 8.7k stars, 1.5k closed issues, 1.6k closed PRs, 16.5K commits, 31 releases, 2K forks, and 166 contributors. PowerToys offers a set of utilities for Windows and is written in C# and had 84.7k stars, 15k closed issues, 4.1k closed PRs, 6.4k commits, 73 releases, 4.9k forks, and 173 contributors.

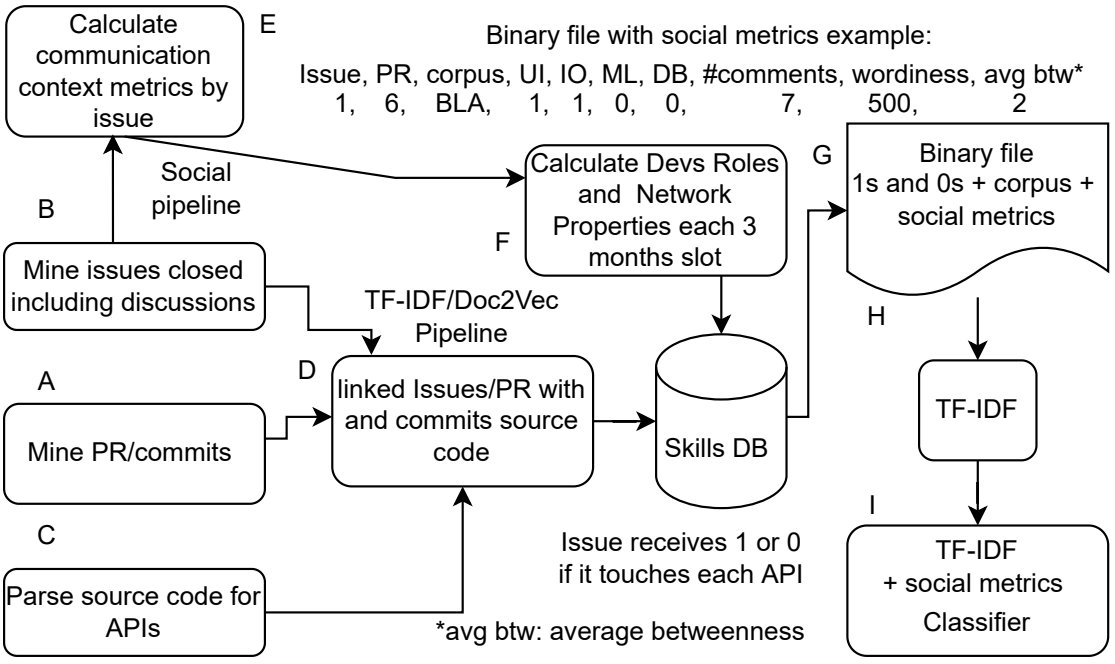

FIGURE 7.2: Processing pipeline.

7.2.2 Mining Issues and Pull Requests

After we collected issues and PRs from the projects using the GitHub API, our dataset included title, description (body), comments, and submission date. We also collected the name of the files changed in each PR and the commit message associated with each commit (Figure 7.2 - A and B). We follow the steps mentioned in Stage 1 and 2 (Sections 4.1.3 and 5.3) but with TF-IDF and Doc2Vec.

To train the model, we kept only the data from issues linked with merged and closed pull requests since we needed to map issue data to source code APIs through the files changed (Figure 7.2 - C and D). We searched for the symbol # followed by an issue number (a set of numeric characters) in the pull request title and body to identify the link between issues and PRs. We also filtered out issues linked to PRs without at least one source code file (e.g., those associated only with documentation or config files) since they do not provide the model with content related to any API.

7.2.3 Mining Social Metrics

For mining necessary communication data, we built an extractor toll[1], which uses the PyGithub[2] library for interacting with the GitHub REST API v3 from Python and mine data from new endpoints. We also used igraph[3] and NetworkX[4] libraries for deriving the various social metrics.

All items in the **communication context** category refer to quantities counted by issues. The number of issue comments was gathered directly from the GitHub REST API. Issue wordiness was measured by finding how many words there are with a length greater than two in the aggregated text of the issue body and all issue comment bodies for an issue, following the example of Wiese et al. [19]. For issue commenters, we are interested in the quantity, which is unique. To find this value, we created a set of all commenter usernames, including the username of the issue author.

Drawing upon previous work [74, 148], we employ temporal periods and dynamic features, such as the evolving number of comments on an issue, to improve prediction models. For that purpose, we created directed, weighted graphs that map issue discussants and their conversations for all issues in a temporal period. A "temporal period" is an interval in which we partitioned the repository's history. For example, suppose the repository has existed for one year, and the chosen interval is three months. In that case, we separate the issues into four distinct temporal periods. The work from Kikas et al. [74] uses three months periods (slots) to measure the number of issues created and closed and commits created by contributors and by projects. In contrast, Wiese et al. [140] preferred six-month periods to predict co-changes. We have chosen three months as the duration for

[1]https://doi.org/10.5281/zenodo.7740450
[2]https://github.com/PyGithub/PyGithub
[3]https://github.com/igraph/python-igraph
[4]https://github.com/networkx/networkx

each temporal window to capture the period similar to the issue life activity (90% of issues get closed in 96.4 days [74]).

For creating the communication network graphs, we used the same rules that Wiese et al. [140] used: nodes are discussants in an issue's conversation, edges are responses from one discussant to another, nodes and edges are added independently, and edges have weights representing the number of responses given. The original poster of the issue is not considered to have responded to anyone by creating the issue, and any response in the lifetime of the issue is considered a response to all prior discussants. We also disallow nodes from having self-edges. Like in previous work, the order or "directions" of discussants in a conversation is crucial because it affects the resulting metrics derived from a graph, such as betweenness and closeness.

We placed each issue in the repository's history into a temporal period. Then, we visited and processed each period of issues into a matrix. In the resulting adjacency matrices, each row represents a unique discussant, and each column represents the number of responses by that discussant to the discussant whose row index corresponds to the column index. To illustrate, let issue 1 be one which discussant A opened and to which discussant B responded once. Let issue 2 be in the same period as issue 1. Assume that B opened the issue, A responded twice to B, C responded once after (meaning that they responded to both A and B), and B responded last (meaning that they responded to both A and C).

$$Issue_1 = \begin{array}{c} \begin{matrix} A & B \end{matrix} \\ \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \end{array} \begin{matrix} A \\ B \end{matrix}$$

$$Issue_2 = \begin{array}{c} \begin{matrix} B & A & C \end{matrix} \\ \begin{bmatrix} 0 & 1 & 1 \\ 2 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \end{array} \begin{matrix} B \\ A \\ C \end{matrix}$$

After all issues in a period have resulting matrices, we used a method adapted from Yu et al. [149] to create a period matrix representing the total communication that transpired in all issues for the period combining all of the period's issue matrices. The resulting period matrix contains all discussants found in the issues for that period and aggregates the number of responses between them. In this scheme, each matrix corresponds to one issue communication model, while the combination matrix summarizes the communication model for the designated period.

To conduct this process, each issue matrix is visited, and the contents are merged with those of the period matrix. Like with the issue matrices, nodes representing discussants are only added if they are not already present. When merging edges from an issue matrix, an edge is added if it is not found. Otherwise, the weight of the edge, a whole number, is incremented by the weight found for that edge in the issue matrix. In the implementation, nodes are added by checking if the period matrix contains the unique identifier for a row/discussant. After all the discussants are integrated, all row contents, i.e., the edge weights indicating the number of responses between two developers, are integrated. Assuming that issues 1 and 2 are the only issues in the period, the resulting period matrix and the directed graph would be:

$$Issue_{1+2} = \begin{array}{c} \begin{matrix} A & B & C \end{matrix} \\ \begin{bmatrix} 0 & 2 & 0 \\ 2 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \end{array} \begin{matrix} A \\ B \\ C \end{matrix}$$

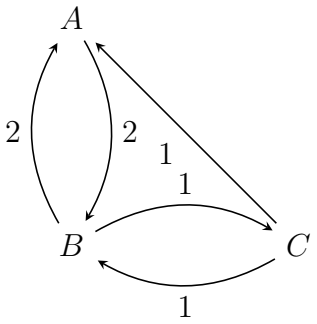

FIGURE 7.3: The communication history example (issue 1+2)

Using igraph and NetworkX libraries, we produced corresponding graphs of these period matrices and retrieved metrics of interest from available function calls (Figure 7.2 - E-F).

The **developers' roles**, which include two centrality measures (betweenness and closeness) were collected by temporal period, and we computed the average, sum, and max value [140]. For each period, once we had the metrics computed, we applied those for each issue within the period, but only averaging, summing, or computing the max value with the individual betweenness and closeness for the discussants present on that issue.

The **communication network properties** were computed directly by counting the number of nodes and edges, in addition to the graph's diameter and density based on the matrix created by period or by issues. The size of a network is the number of nodes (unique discussants) in it. When it is computed by period, nodes are distinct from "issue discussants" communication context metric. The number of edges reflected the number of unique responses in the network for the given time interval—essentially, the sum of all weights in the aggregated adjacency matrix for a period. Thus, it is different from "issue comments". In this study, we opted to model the network properties by issues, and we removed the number of nodes from the model since it has the same value as unique discussants.

7.2.4 Categorization of APIs

API categorization has evolved since the case study. We reproduced the same process used in the generalization study, and we repeat it here for the sake of clarity and flow. We take the opportunity to give more details and use a new example.

Since labeling the issues with dozens of APIs can harm the user interface and cause information overload, we grouped the APIs into high-level categories, described in Stage 2 and reproduced here with the pertinent details for easier comprehension (Section 5.2). The categories provided ground truth to the supervised machine learning model. The categories were related to APIs imported in files that were changed in closed pull requests linked to closed issues. In the source work of our replication, [23] manually categorized and customized the APIs for a single project (JabRef); therefore, it would not necessarily fit other projects. To make the set more generalizable, we recruited for the generalization study three experts—one specialized in each of the three main programming languages adopted in the projects (Expert1: 25 Years of Java; Expert2: 22 Years of Java and 10 of C++; Expert3: 18 years of Java and 12 Years of C#)—offering US$ 25.00 gift cards as a token of appreciation.

The categorization started with proposals from the experts to define generic API domain categories to encompass a broad range of projects (e.g., “UI”, “IO”, “DB”, “ML”, etc.). A card-sorting approach addressed disagreements via discussions until a consensus was reached. After four rounds (8 hours) of discussions, the experts agreed, and 31 API domains were defined.

Next, we parsed the source code files to retrieve the libraries declared in “import” (java), “include” (C++), and “using” (C#) statements (Figure 7.2 - C). The intuition behind the API classification method is that libraries’ namespaces often reveal architectural

information and, consequently, their API domains [95, 96]. To identify the API domains for each library, we split all the API namespaces into packages. For instance, the API "com.oracle.xml .odbc.XMLDatabase" derived "com", "oracle", "xml", "odbc", and "XMLDatabase". Next, we eliminated the business domain name extensions (e.g., "org", "com"), country code top-level domain ("au", "uk", etc.), the project and company names ("microsoft", "google", "facebook", etc.). In the example, we kept the first package, "xml", second package "odbc", and the class name: "XMLDatabase". The class XMLDatabase was split into two words using the Python library `wordninja` [5], which employs word frequency and can identify concatenated words in texts. Therefore "XMLDatabase" was split into "XML" and "Database."

To facilitate the expert's work, for each package and class name, we identified how similar they were to the proposed API domains using an NLP Python package `spacy` [6]. Spacy is a multi-use NLP package that uses a cosine similarity function to compare the average of the word vectors to find similarities between sentences and tokens and can be trained with a software engineering vocabulary.

We used the results of the experts' work for the generalization study. The process was better automated with new scripts to create the lists automatically. The recent update can even ignore the experts' verification by adopting the best NLP suggestion. However, we preferred semi-automatically executing the process below to verify the intermediate results: the experts leveraged the NLP suggestions to classify the APIs. A card-sorting approach addressed any disagreement between the experts. We created lists. The first list contained all the aggregated first packages. The second list contained the aggregated second packages and so on. In addition to the package name, the list showed the NLP suggestions with the confidence levels (i.e., XML: IO=0.85; Lang=0.7; util=0.61). The experts evaluated the lists and accepted one suggestion or rejected all. In case of rejection,

[5]https://pypi.org/project/wordninja/
[6]https://spacy.io/api/doc#similarity

they evaluated the entire namespace of the package/class. By aggregating per package, we reduced the number of libraries evaluated by the experts. In the JabRef project, where the experts had to evaluate 1,692 libraries manually, aggregating it by the first and second packages resulted in only 137 and 45 packages, respectively. Two lists were sufficient to evaluate most libraries. Indeed, the volume of evaluations dropped significantly (to 10.8% in JabRef, 21.6% in PowerToys, and 45.1% in Audacity). In case of evaluation discrepancy between the first and second packages, the second package evaluation prevailed since it is less generic. For example, in the case above, the package "xml" may be evaluated as "IO" and "odbc" as "DB". When the experts accept both, "DB" prevails.

7.2.5 Dataset Setup

Our dataset has one row for each issue, and the columns are composed of "1s" and "0s" for each API-domain column, indicating whether an issue has that API in the source code changed in the associated PR. The dataset also contains the corpus column with the concatenated text from the title, body, and comments on the issue.

The metrics calculated filled the columns of each issue. For example, suppose issue #1 was closed by PR #6, which had seven issue comments and 500 words. The average betweenness of the developers present in the discussant in that period was two. In addition, the files changed by PR #6 included libraries categorized as "UI" and "IO" by the experts, and the issue and PR text mined was "bib file does not load." Suppose the possible API domains are: "UI," "IO," "ML," and "DB." The dataset row contained the following values: issue #: 1, pr_linked number #: 6, corpus: "bib file does not load", #comments: 7, #wordiness: 500, avg betweenness: 2, UI: 1, IO: 1, ML: 0, DB: 0 (Figure 7.2 - G). The same logic applies to the remaining API-domain labels (31 possible), metrics computed by issue, and three-month period metrics (described in Section 7.2.3).

To transform the corpus into a feature set, we followed some studies [79–81] that applied TF-IDF—a technique for quantifying word importance in documents by assigning a weight to each word. To compute the TF-IDF weights, we followed the cleaning process used in the original work [23], which includes stemming and stop word removal. We also converted each word to lowercase and removed URLs, source code, numbers, templates, and punctuation. After applying TF-IDF, we obtained a vector of TF-IDF scores for each issue's word. The vector length is the number of terms used to calculate the TF-IDF, and each term was stored in a column with the TF-IDF weight (Figure 7.2 - H).

Given the recent popularity of different ways to represent documents for prediction models, we decided to evaluate the API-domain label prediction by replacing TF-IDF with Doc2Vec. Doc2Vec is a document representation method that translates text into a vector of features. Like TF-IDF, Doc2Vec was used in many studies where a corpus must be transformed into features such as document classification or sentiment analysis [150, 151]. Using the best setup found for RQ1, Doc2Vec was outperformed by TF-IDF (p=0.03 precision, p=0.00018 recall, and p=0.003 F-measure), and we kept TF-IDF only in the analysis. Despite being a classic NLP technique, TF-IDF usually overcomes newer techniques like Word2vec or Doc2Vec when the corpus is small and unstructured. Doc2Vec cannot identify the semantic and syntactic information of the words [152] in these situations. Many issues have small titles and unstructured or semi-structured bodies and comments. After removing templates, code snippets, and URLs, we may reduce the number of words and lose the semantics and syntax in sentences.

7.2.6 Training and testing sets

We ran each experiment ten times to avoid overfitting, using ten different training and test sets to match 10-fold cross-validation [82]. We used ShuffleSplit provided by `scikit-multilearn`[7],

[7] `http://scikit.ml/index.html`

a model selection technique that performs cross-validation for multi-label classifiers. To improve the dataset's balance, we used the SMOTE algorithm for the multi-label approach [100]. In the baseline study [23], SMOTE improved the F-measure by 6%.

7.2.7 Classifier

An issue may require knowledge of multiple APIs. Thus, we applied a multi-label classification approach, which has been used in software engineering for many purposes, such as classifying questions in Stack Overflow (e.g., [14]) and detecting types of failures (e.g., [77]) and code smells (e.g., [78]).

We used the Random Forest (RF) algorithm (Python `sklearn` package) since it had the best results in the baseline study [23]. RF has been shown to yield good prediction results in software engineering studies [90–93] (Figure 7.2 - I).

7.2.8 Data Analysis

Our analysis started verifying the number of issues each dataset has with num_comments > 3 and num_discussants > 2, since we need some level of social interaction to prospect relevant social metrics. We also tested for different thresholds (as presented in the results section). After computing all the social network metrics, the features' influence was compared to derive the best predictors. We used the feature_importances_ function present in the package `sklearn` to rank the features[8].

To evaluate the classifiers, we employed the same metrics to enable comparison with the previous studies, except the Hamming Loss[9]:

[8] `https://scikit-learn.org/`
[9] `https://scikit-learn.org/`

- **Precision** measures the proportion between the number of correctly predicted labels and the total number of predicted labels.
- **Recall** measures the percentage of correctly predicted labels among all correct labels.
- **F-measure** calculates the harmonic mean of precision and recall. F-measure is a weighted measure of how many correct labels are predicted and how many of the predicted labels are correct.

Since we are computing the metrics for a multi-label problem, we should average the metrics above for the set of predicted labels regarding the ground truth. The metrics for each label can be calculated using different averaging strategies, for instance, the macro or micro [153]. The macro average is the arithmetic mean of all the per-label metrics. In contrast, the micro average, adopted by [23], is the global average metric obtained by summing TP, FN, and FP. We employed the micro to follow the baseline.

We ran statistical tests to verify whether the observed differences between groups of variables were statistically significant. We kept the Mann-Whitney U test to compare the classifier metrics, followed by Cliff's delta effect size test. The Cliff's delta magnitude was assessed using the thresholds provided by [102], i.e., $|d| < 0.147$ "negligible," $|d| < 0.33$ "small," $|d| < 0.474$ "medium," otherwise "large."

7.2.9 Data Availability

The source code and the data generated from our research are publicly available in a repository to help reproducibility[10].

[10]https://doi.org/10.5281/zenodo.7740450

7.3 Results

This section presents the results from the social metrics investigation by research questions.

7.3.1 RQ1. To what extent can social metrics improve the prediction of API-domain labels?

Table 7.1: Comparison - baseline X social metrics

Project/ Metric	**Audacity**		**JabRef**		**PowerToys**	
	Baseline	Social	Baseline	Social	Baseline	Social
Precision	**0.916**	0.897	**0.842**	0.812	**0.796**	0.788
Recall	0.884	**0.906**	**0.835**	0.812	**0.842**	0.833
F-Measure	0.899	**0.901**	**0.838**	0.811	**0.818**	0.809

We started comparing the results presented in Table 7.1, where we compared the baseline (dataset without the social metrics) and the dataset with the social metrics. The first results looked not promising since they did not improve the precision, recall, and F-measure to justify the data mining and processing overhead to use social metrics. Only Audacity overperformed the baseline in recall and F-measure with a small difference. Next, we verified the dataset's average: number of comments, number of discussants, and wordiness per issue among others (Table 7.2). Indeed, filtering Audacity and JabRef by num_comments > 3 (using as a base the average of the PowerToys project) resulted in a dataset with only 16 and 20 rows, respectively, while in PowerToys resulted in 207. The same impact occurred when filtering by wordiness > 300 and num_discussants > 2 (Values close to the average in PowerToys). While Audacity and Jabref kept only 11 and three rows, respectively, PowerToys kept 196 for wordiness and 20 and 18, and 219 for num_discussants. Since the social metrics require some developers debating the issues, we continued the study with

only the PowerToys dataset (PowerToys num_comments, wordiness, num_discussants, betweenness_avg, and betweenness_sum have normal distributions— Shapiro-Wilk test with p=1.83, 9.43, 3.45, 2.94, 3.04, respectively. Audacity and JabRef also have normal distributions on the same metrics).

TABLE 7.2: Averages from social metrics in the studied datasets

Metric AVG/Project	Audacity	JabRef	PowerToys
Comments	1.41	1.73	**3.83**
Discussants	1.58	1.47	**2.23**
Wordiness	81.69	157.88	**297.91**
Edges	4.60	10.73	**31.38**
Density	0.73	0.58	**1.52**
Diameter	0.43	0.56	**0.70**
Betweenness avg	1077.72	2042.21	**208451.44**
Betweenness max	1418.27	2648.70	**350726.26**
Betweenness sum	1684.45	3304.60	**418619.28**
Closeness avg	0.56	**0.64**	0.56
Closeness max	0.60	**0.68**	0.61
Closeness sum	0.92	1.15	**1.25**

To understand the impact of each predictor in the predictions, we evaluated the model running only with the social metrics (without the TF-IDF weights). We used the method "feature_importances_" from the package `sklearn.ensemble` [11] to rank the importance of each metric. The three most important predictors in the PowerToys project are wordiness (W), betweenness_sum (BS), and closeness_sum (CS). They also appear in the top three ranks for the other projects (Table 7.3).

Next, we filtered out the issues based on the three best PowerToys predictors (Table 7.4). We tested three different filters: wordiness > {300, 500, 700}, betweennness_sum > {450K, 600K, 800K, 1000K}, and closeness_sum > {1.5, 1.75, 2.0, 2.25}. We compared the baseline ("Non-Social") and the predictions without filtering ("Social"). Using the PowerToys dataset, we start from a threshold close to the average of the three mentioned

[11] `https://scikit-learn.org/stable/`

Table 7.3: Feature Importances by Project

Feature	PowerToys	JabRef	Audacity
num_discussants	0.02908	0.04451	0.04451
num_comments	0.05488	0.03591	0.03591
wordiness	**0.26509**	**0.14283**	**0.57612**
edges	0.04970	0.03200	0.03650
diameter	0.03479	0.03462	0.01299
density	0.06147	0.02540	0.01758
betweenness_avg	0.08943	**0.12441**	0.04366
betweenness_max	0.07232	0.11745	0.04077
betweenness_sum	**0.09793**	**0.11788**	0.05049
closeness_avg	0.08836	0.11094	**0.06220**
closeness_max	0.06718	0.11440	0.04495
closeness_sum	**0.08977**	0.09964	**0.06166**

metrics (Tables 7.4 and 7.5 and Figure 7.4), increasing the value and filtering to a point where it produced a precision drop.

Table 7.4: Comparison - social metrics filtering - PowerToys

Filter	Number of issues	Precision	Recall	F-Measure
W300	196	0.904	0.938	0.920
W500	99	**0.922**	0.962	0.941
W700	56	0.909	**0.978**	**0.942**
BS450K	292	0.823	0.878	0.849
BS600K	243	0.834	0.895	0.863
BS800K	177	0.832	0.854	0.842
BS1000K	146	0.807	0.895	0.847
CS1.50	216	0.856	0.905	0.879
CS1.75	142	0.867	0.936	0.899
CS2.00	106	0.908	0.917	0.911
CS2.25	80	0.905	0.949	0.926

Table 7.4 and Figure 7.4 show the results after filtering by wordiness, betweennness_avg, and betweennness_sum. Filtering with one feature, we obtained the best precision with wordiness $>$ 500 (W500). The best recall and F-measure were reached with wordiness $>$ 700 (W700). Filtering with a combination of best features (e.g., W700+BS600K+CS2.00) reduced the dataset to a few rows, and, therefore, we discarded the results. We tested different configurations with two and three filters, and we could not find better results.

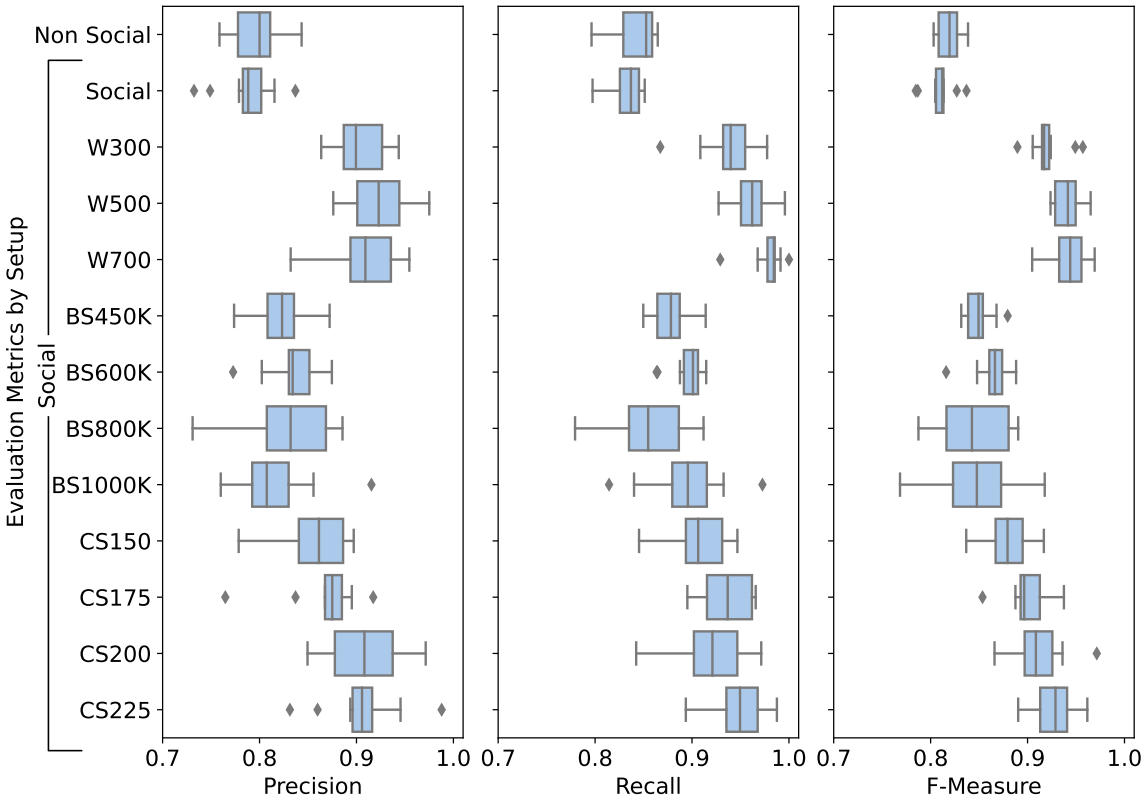

FIGURE 7.4: Social metrics setups comparison: TF-IDF - PowerToys

TABLE 7.5: Cliff's Delta for F-Measure and Precision: comparison of corpus models by setup - TF-IDF.

Corpus Comparison	**Cliff's delta**			
	Precision		**F-measure**	
Non Social x Social	0.18	'small'	0.28	'small'
Social: W300 x W500	-0.38	'medium'	-0.68	'large' *
Social: W500 x W700	0.16	'small'	-0.08	'negligible'
Social: BS450K x BS600K	-0.28	'small'	-0.5	'large'
Social: BS600K x BS800K	0.0	'negligible'	0.16	'small'
Social: BS800K x BS1000K	0.3	'small'	-0.12	'negligible'
Social: CS1.50 x CS1.75	-0.22	'small'	-0.48	'large'
Social: CS1.75 x CS2.00	-0.5	'large'	-0.22	'small'
Social: CS2.00 x CS2.25	-0.02	'negligible'	-0.36	'medium'
Social x W500	-1.0	'large' ***	-1.0	'large' ***
Social: W500 x BS600K	1.0	'large' ***	1.0	'large' ***
Social: W500 x CS2.00	0.23	'small'	0.64	'large' *

** $p \leq 0.05$; ** $p \leq 0.01$; *** $p \leq 0.001$*

Precision and F-measure tend to fall for PowerToys when the dataset reaches the lower bound (roughly below 100 issues), except when the dataset reaches a point where the filters reduce the feature space and facilitate the work of the machine-learning algorithm. We reject the hypothesis that the distributions are the same in precision and F-measure using the Mann-Whitney test and comparing Social (dataset without filters) `x` W500 (large effect size), W500 `x` BS600K (large effect size), and only in F-measure for W500 `x` CS2.00 (small effect size for precision and large for F-measure) and W500 `x` W300 (medium effect size for precision and large for F-measure), as can be observed in Table 7.5.

Next, we compared the different hypotheses in predicting the API-domain labels using the PowerToys project.

Using the filter, wordiness > 500, we predicted the API-domain labels using sets of social metrics regarding the defined hypotheses (Section 7). Looking at Figure 7.5, we can see the model created with all social metrics and the model based on H_0^1, H_0^2, and H_0^3 had similar results and no statistical difference (for precision: H_0^1 `x` H_0^2 p=0.61 and H_0^2 `x` H_0^3 p=0.42). However, when we compare the model only with TF-IDF and all social metrics filtered by wordiness > 500, the Mann-Whitney and Cliff's Delta showed a difference (p=0.00018, and large effect size, delta=1.0). Observing Table 7.6, we can see that, although we fail to reject the null hypothesis (all have H_0^n where $n = 1..3$ had the same distribution), H_0^2 performed slightly better on precision, recall, and F-measure. H_0^2 model with six features conveyed almost the same result as the model with all features (Table 7.5), thus should be preferred when the dataset is considerably large.

Table 7.6: Comparison of Hypotheses.

Hypotheses	**Precision**	**Recall**	**F-Measure**
H_0^1	0.913	0.957	0.934
H_0^2	**0.920**	**0.962**	**0.940**
H_0^3	0.909	0.961	0.934

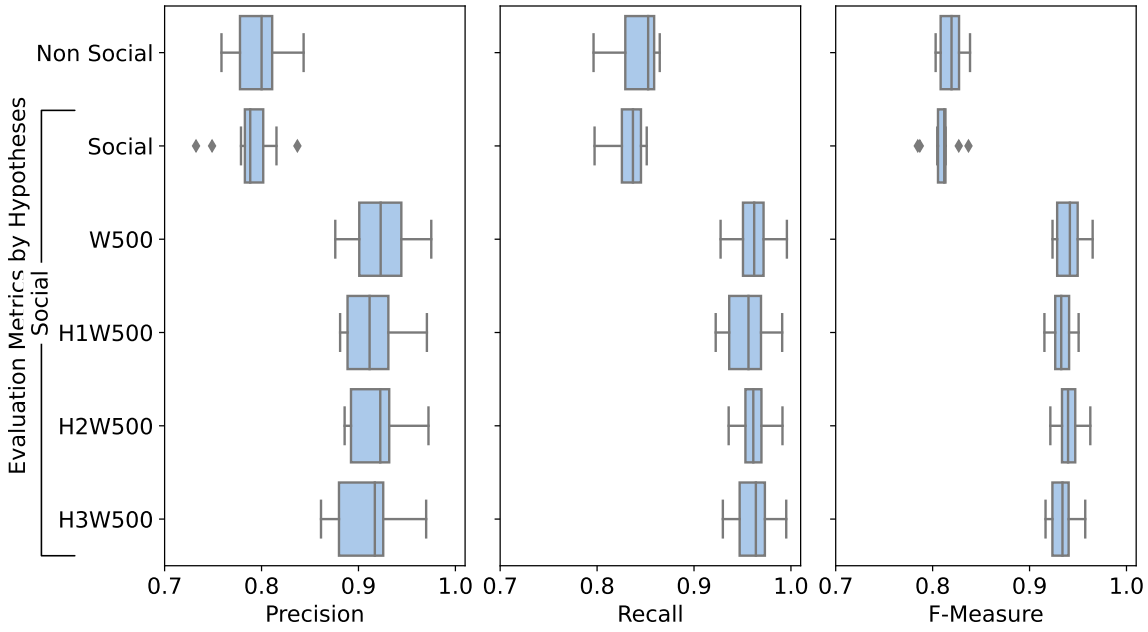

FIGURE 7.5: Social metrics hypotheses setups comparison: PowerToys Project

RQ1 Summary. Employing all features, it is possible to predict the API-domain labels with the social metrics and TF-IDF, filtering the dataset when wordiness > 500, with precision = 0.922, recall = 0.962 and F-measure = 0.941.

7.3.2 RQ2. To what extent can we transfer learning among projects using the social metrics to predict the API-domain labels?

To answer RQ2, we verified whether it is possible to transfer learning from PowerToys to Audacity and JabRef. Transferring from PowerToys to Audacity and JabRef required us to normalize the labels in common for each pair of projects since we cannot predict for Audacity/JabRef what we cannot find in the PowerToys dataset.

The training dataset was filtered with wordiness > 500, repeating the best configuration found. This makes some label occurrences disappear from the training dataset. For

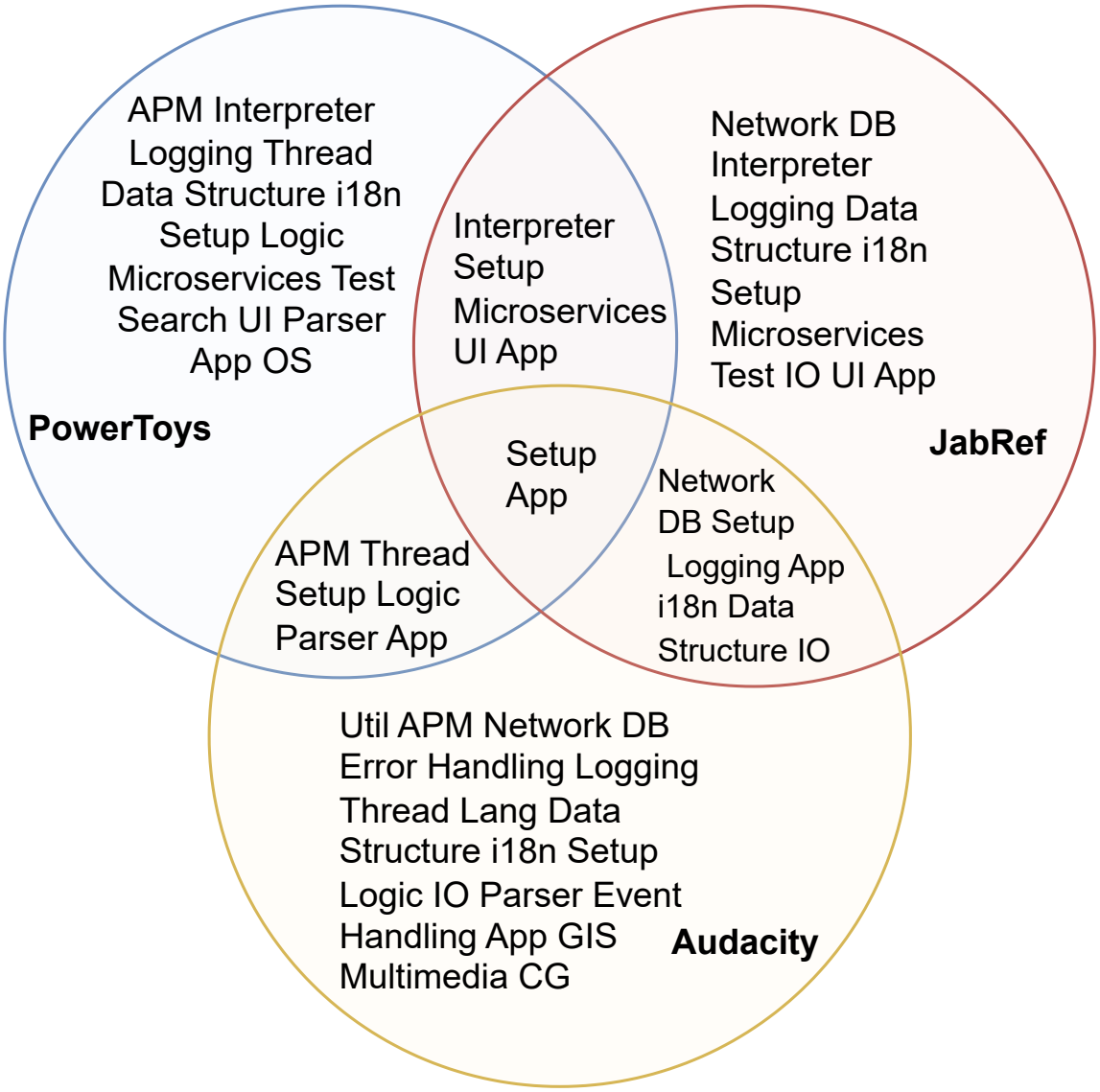

FIGURE 7.6: Labels intersections from the studied projects.

example, "i18n" is present in PowerToys and JabRef projects, but no issue in the dataset prevails after filtering (Figure: 7.6).

Table 7.7 shows the result of the transfer learning. The metrics are far behind the ones obtained by training and testing with the same dataset.

TABLE 7.7: Comparison of corpus models with transfer learning.

Training	Test	Precision	Recall	F-Measure
PowerToys	Audacity	0.392	0.434	0.412
PowerToys	JabRef	0.328	0.643	0.434

TABLE 7.8: Labels distribution

Labels distribution	Label counts		Labels counts normalized	
Label names	Before filtering	After filtering	Before filtering	After filtering
Logging	6	1	0.008	0.01
Data Structure	13	1	0.01	0.01
Logic	14	7	0.01	0.07
Setup	337	49	0.46	0.49
Microservices	214	34	0.29	0.34
Test	6	1	0.008	0.01
App	502	75	0.69	0.75
Search	394	61	0.54	0.61
i18n	4	1	0.005	0.01
Parser	42	3	0.06	0.03
APM	201	36	0.26	0.36
UI	498	68	0.69	0.68
Thread	25	12	0.02	0.12
OS	521	75	0.71	0.75
Interpreter	22	4	0.02	0.04
rows	721	99		

RQ2 Summary. Transfer learning did perform poorly and resulted in precision = 0.392, recall = 0.643, and F-measure = 0.434.

7.4 Discussion

Do Social Metrics Matter?

Our results showed the social metrics are relevant to predict API-domain labels. The predictions using social metrics improved the precision of the models up to 15.82% and the F-measure up to 15.89% when filtering wordiness > 500). We could not observe an increase in the performance of the classifiers in the projects, with few comments and discussants actively participating in the tasks. The results are coherent with the intuition behind the research, as the predictions are based on the presence of social interaction.

We observed the increase of the relative counts in some labels after filtering—with emphasis on "Thread" from 0.02 to 0.12 and "APM" from 0.26 to 0.36. "Thread" issues have approximately six comments, 18 discussants, and 947 words on average, which is far above the PowerToys averages in Table 7.2. "Thread" labels are present when the discussions are more protracted and might point to more specific characteristics of the issues (Table 7.8).

To what extent does the model transfer learning?

Transfer learning is paramount when projects do not have enough data for training, not enough time to prepare the training dataset, the infrastructure is unavailable to develop their models, or the costs to run the models are prohibitive. Moreover, while we can access features from projects in the OSS communities to run API predictions, the same availability is not always possible in the industry. The source code may be restricted, and there is no way to prepare the ground truth with the APIs declared. Besides, the conversations are often protected by privacy laws. Therefore, despite the relevance of transfer learning, the results are far from the regular training and testing, and we should investigate ways to improve them. One possible reason is the diverse domains of the projects that predicted different API-domain labels.

How can software practitioners benefit from the results?

This approach can benefit developers who want to find an appropriate issue, which is the case of newcomers who have a hard time searching for a task [35]. Once the maintainers foment the communication in the projects' issue tracker, the discussions around issues, which hold the project skills, may be used to train our model to predict the API-domain labels and, thus, assist newcomers in picking issues and helping them learn about the tasks. Indeed, communication is a recipe for newcomers to "Keep the community informed about decisions" [6]. Reporting the advances on a task should attract comments from core members in charge of the module and with the skillset related to the task. Steinmacher et

al. [6] guide newcomers to "Do not be afraid of the community". Reporting problems in appropriate community channels may lead contributors with the required skills to join the discussion. The expansion of communication to produce better social metrics will assist indirectly in breaking barriers defined by Steinmacher et al. [6]. The communication strategy is second in primacy to maintainers [107] and meets multi-teaming research whose hints include improving communication to address the coordination weakness, lack of member stability, and hierarchy in the highly-dynamic OSS projects [132].

7.5 Final Considerations

The metrics were improved significantly with the social predictors. However, one question remains: despite all the improvements, are the API-domain labels impacting the contributors when they are starting and a first issue as a newcomer? The next chapter aims to answer that question.

CHAPTER 8: FINAL EXPERIMENT

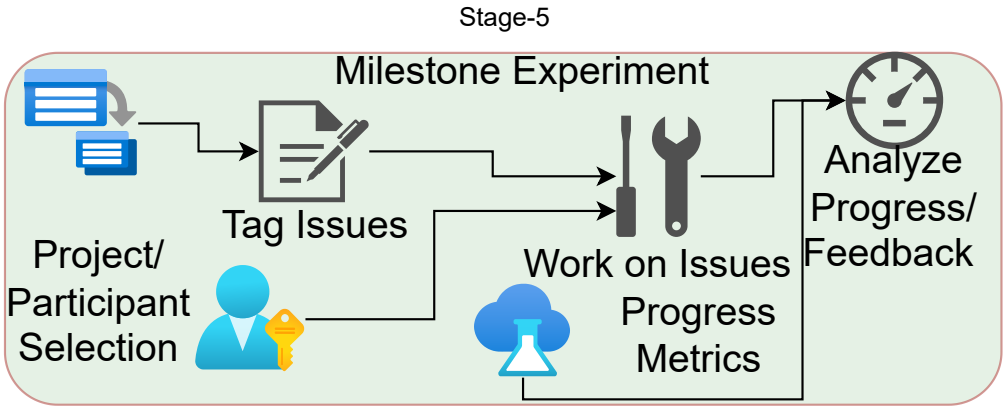

FIGURE 8.1: The Research Method - Stage 5 - Milestone Experiment

In this study, we tested the impact of classifying issues by API-Domain through a rigorous experiment. This study aimed to determine how helpful the labels are in issue selection and completion.

To answer our research questions, we carried out an empirical experiment that began by showing an issue list page to participants. We asked each participant to pick one issue to work on. The participants were timed as they completed contribution milestones, such as selecting an issue, describing the problem, and finding the component, file, method, and lines to be updated to solve their chosen issue. In addition, we measured the correctness of the participant's solution and the participant's perception of the complexity, confidence, and skills needed to solve the issue. Similar to the empirical study carried out for the case study, the participants were divided into two groups, one with the additional API-domain labels on the issues page (treatment group) and the other with only the original project labels present (control group). We determined the impact of the API domain labels by measuring the differences between these two groups, using metrics such as issue selection time, issue completion time, the confidence of the participants, and the correctness of each participant's solution (Figure 8.1). We answered the following research questions:

RQ1: To what extent do the API-domain labels assist the contribution progress?

RQ2: What do API-domain labels provide to potential contributors?

8.1 Experiment Design

The experiment involve short contributions in a time frame of one hour. The time frame considered the availability of the participants, the researchers, and the infrastructure (laboratory space and video recording). The expected participants are students and practitioners from the industry. We conducted in-person contributions to have more control over the experiment.

We randomly divided the participants into two groups:

1. Control group participants: select issues from a web view list that are NOT labeled by the API domains (they still had issue labels as extracted from the project).

2. Experimental group participants: select issues from a web view list of labeled issues that included pre-existing labels and the API-domain labels we generated

Progress checkpoints were timed and based on the three proposed strategies for newcomers by [26]. We excluded two strategies: "Set up the Environment," and "Communicate with the Community". Set up the Environment is the step prior to attempting a contribution and is not influenced by issue selection. In the same direction, "Communicate with the Community" is a step that happens before, during, and after a contribution. The research team provided some communication assistance to ask questions about the overall experiment and contribution progress, and without giving tips about the solution so as to avoid biasing the experiment.

The newcomers' strategies and steps contemplated by the experiment are "Understand the Issue," "Understand the Context," and "Understand What Needs to be Changed." We added one more step to complement the newcomers' strategy to find an issue to start, described below as: "Code the Solution." The extra step is necessary to work on the issue after the newcomer decides to contribute.

1. Understand the issue. (Write the problem.)
2. Understand the context. (Explain the overall organization of the bug/feature.)
3. Understand what needs to be changed. (Find the pieces. Write it down.)
   (a) Find the component. (Name the component.)
   (b) Find the unit. (File) (Name the files.)
   (c) Find the class. (Name the class.)
   (d) Find the method. (Name the method.)
   (e) Find the lines to be changed. (Identify the lines.)
   (f) Propose a solution in natural language. (Write down what to do.)
4. Code the solution to the problem and resolve the issue.

After the experiment, we assessed the solution. We accepted different solutions to the problem even when the proposed solution differs from the one presented in the pull request that solved the issue.

8.1.1 Selecting Issues

The experiment is based on the JabRef project. We selected JabRef for our empirical study because of the project's characteristics (it is an accessible and understandable

domain: a desktop project with popular libraries and limited complexity) and because we have access to the project's contributors. We adopted JabRef version 5 because two researchers contributed to this version and therefore understand the impact of rolling back the PRs and evaluating the proposed contributions.

We selected issues based on several factors: 1) the perceived difficulty of the issue; 2) the amount of code needed to solve it; and 3) the feasibility of rolling back the changes addressed by that issue.

We rated difficulty based on the amount of JabRef project knowledge needed to solve it and the complexity of the changes themselves. The issues must be relatively simple to solve as the participants only have one hour to select and progress in the milestones.

The amount of code needed and the number of files changed to solve the issue are related to the difficulty, as significant changes are not feasible to complete in the allotted time frame. Three researchers carried out the evaluation, two of them were contributors to the JabRef project.

We selected 30 candidates of existing JabRef issues to build mock GitHub pages for control and treatment groups based on the obtained predictions. We selected issues based on: if they had changes we could roll back without breaking the application; their difficulty level and whether they could be completed or have significant progress made within the one-hour experiment; and the aim to maintain similar distributions of the number of API-domain labels predicted per issue and the counts of predicted API-domain labels.

From the initial 30 issues selected as candidates, 15 were analyzed in depth after a filtering step carried out by reading titles and descriptions. Each of the 15 candidate issues was rolled back to verify its feasibility for the experiment. Two researchers tested the application to verify the general system operation after the rollback and the manifestation of the issue description. From the 15 issues selected, seven were selected to start the experiment.

Table 8.1: Issues Selection *rollback break the app

Issues	Files	Lines	Difficulty
5679 (PR 5680)	1	32	Easy
5653 (PR 5671)	1	5	Easy-Medium
5532 (PR 5533)	1	31	Medium *
5485 (PR 5495)	1	2	Easy
4612 (PR 5229)	1	2	Easy
5194 (PR 5195)	2	10	Easy
5069 (PR 5472)	1	24	Medium
4886 (PR 4964)	2	9	Easy-medium
5277 (PR 5454)	5	26	Medium
5071 (PR 5539)	7	>100	Hard
4288 (PR 4355)	6	>100	Hard
3189 (PR 4429)	3	9	Easy *
2811 (PR 3248)	9	>100	Hard
5002 (PR 5123)	31	>100	Hard
4735 (PR 4839)	7	>100	Hard

After ten participants finished the experiment, two issues were removed from the issue pages because they did not attract any participants and to ease the concentration of the control and treatment groups' evaluations on the same issues and to allow comparison. Finally, five were retained to compose the user interface experiment (Table 8.1).

8.1.2 Preparing the User Interface Experiment

To create a similar user interface to GitHub, we copied the issue page structure and the page for each issue selected for the experiment by using the httrack [1] tool.

With the HTML code in hand, we cleaned up all the information that revealed that the issue had been closed and merged. We also removed external links to prevent involuntary external navigation while participating in the experiment.

Next, we manually inserted all the predicted labels in the user interface and hosted the website (Figure 8.2).

[1] https://www.httrack.com/

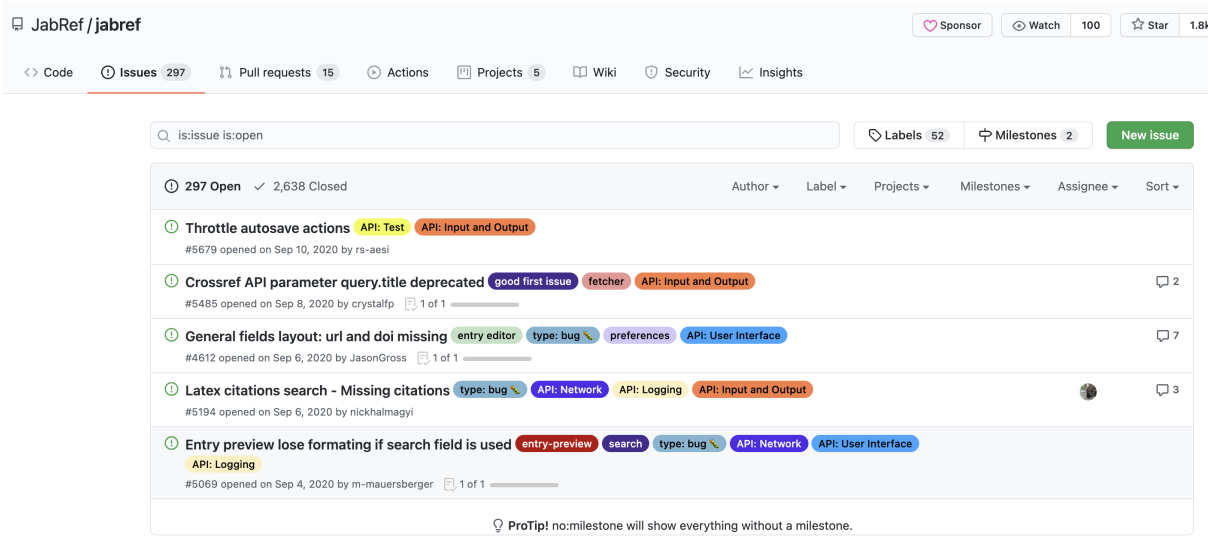

FIGURE 8.2: Mocked Issue Page - Treatment Group

8.1.3 Preparing the Local Infrastructure

The experiment was conducted in person in a lab where the researchers set up the environment for the participants.

The lab was composed of two connected rooms. The reception room and the experiment room. The participants were welcomed in the reception room. They received the first instructions about the experiment and watched the tutorial video ($\approx 6min$).

The tutorial video [2]introduced the participants to aspects of the JabRef project (its architecture, IntelliJ IDE, how to compile and execute the project), as well as the experiment execution and the survey[3]. After the tutorial video, a QA section was carried out to permit the participants to ask questions.

Once a participant finished the tutorial, they were admitted to the experiment room. The participant then received one MacBook Pro with 16GB RAM in the experiment room to perform the task. The researchers demonstrated the experiment environment in the

[2]`https://drive.google.com/file/d/1-e0bzgk38nEqcpmZShzvH0BC7Vo7ug4R/view?usp=sharing`

[3]The research protocol was approved by the institutional review board (IRB) of the author's institution.

workstation focusing on the Java IDE, the browser tabs used in the experiment, and the MacOS. The goal of showing the developer environment was to compare the IntelliJ IDE with the tutorial video, which pointed to where the JabRef project maps its main components and how to edit, compile, and run the project. Next, the researchers assisted the participants in familiarizing themselves with the MacOS operating system, including the keyboard, shortcut keys, and navigation. Finally, the researchers showed tabs for the tutorial video, JabRef documentation (as pointed out in the tutorial video), the survey, and the JabRef issue page used in the experiment.

8.1.4 Planning the Experiment Process

We pulled previously merged issues via a pull request and then presented those issues to participants to select and solve. The available issues were previously merged into the JabRef project, but we rolled them back to present a version of JabRef that contained all of the code and content except for those issues.

We created an online questionnaire to be answered by participants during and after the experiment. We informed participants they should not worry about being unable to solve the issue, because the experiment intended to measure the tool and not judge their development abilities. We presented the first part of the questions after choosing an issue to which to contribute, but before starting to solve the issue. Those questions included the reasons for choosing the selected issue, which issue labels (if any) were helpful to support their selection, which skills they believed were needed to work on the issue (User Interface, Databases, Network development, etc.), and finally a self-assessment about their confidence in having the necessary skills to solve the issue. We settled the environment to avoid infrastructure issues and focused on the study's goal - issue selection and solving.

The second part of the questions was created to be answered while solving the issue, with progress milestones to achieve the solution. The questions were based on the strategies maintainers proposed for a contributor to select and solve an issue [26]. They covered a brief explanation about the problem or feature request, where the code is located (folder, file, class, method, and line of code), and what solution ideas the participant has to solve it. Two in-person researchers assisted participants during the experiment, measuring each participant's progress by the time the milestone questions were answered. These milestones measure how close the participant came to a full solution to their selected issue. When a participant completed a milestone, we recorded the time that they did so to gauge how quickly they made progress. The milestones recorded were: selecting an issue; identifying the problem/feature requested; finding the folder, file, class, method, and code lines that need to be changed; explaining a high-level solution to the problem; and implementing the code to solve the issue and running the JabRef environment we created to test the fix. We then compared the solutions to each of these milestones to the true PR that solved the issue to gauge the correctness of each participant's solution.

The last part of the survey was prepared for the participant to answer after they completed the experiment (regardless of their success in solving the issue). The questions covered how well they believed they performed, the tool's usefulness, and how valuable the labels and information were for selecting and solving the issue.

One undergraduate and one graduate student executed a pilot study before we started the experiment. The pilot helped to understand the participants' difficulties related to the macOS environment, IntelliJ interface (including shortcuts, debugging, navigation, and execution), and the milestones and survey questions. The instructional video was updated to address the difficulties.

8.1.5 Recruiting Participants

We recruited participants from academia since students are potential contributors to the JabRef project. We reached out to our own students in addition to instructors and asked them to help in inviting participants. Participants included undergraduate and graduate computer science students from one university in the US.

8.1.6 Selection Criteria

The first selection criterion required participants to have experience with object-oriented programming in the Java programming language and the IntelliJ IDE. We chose this selection criterion as the JabRef project is coded in Java and recommends IntelliJ as the tool contributors use to work on the project.

The next selection criteria asked about participants' years of programming experience as well as their contributions to open-source software projects. We gathered this data to ensure that the treatment and control groups were balanced in the average amount of programming experience as well as the number of contributions. Finally, we aimed to filter out participants who significantly contributed to the JabRef project to avoid confounding the experiment's results.

We assigned participants to each group based on their years of programming experience. The participants with similar programming experiences were randomly assigned to the groups. We assigned participants in this manner as we wanted to have an even amount of average experience between groups. The treatment group had 2.8 average years of experience, while the control group had 2.5 average years of experience. The average number of contributions in the groups was respectively 0.8 and 1 for treatment and control, and both groups had no participants who had previously contributed to the JabRef project.

The participants with similar years of experience were randomly split into control and treatment groups. From the 85 participants who started the survey, 60 (70.5%) finished all the questions, and we only considered these participants in the selection criteria. Five answers were invalid, and 55 were invited to schedule the experiment on the available dates in the calendar. 15 participants confirmed and scheduled the appointment. We ultimately had 7 and 8 participants in the control and treatment groups, respectively. One result was discarded since a participant from the control group accessed the pull request in the project.

8.1.7 Experiment Execution

In total, we had 15 participants (eight treatment; seven control). One participant did not complete the post-experiment survey, so there were 14 responses. The experiment sessions were conducted in a room with two researchers.

The experiment started by having the participant watch a training JabRef video. The content included an explanation of the JabRef project, documentation, and organization, how to access the project's assets using the IDE, and how to build the JabRef application. Next, the video included an explanation about how to answer the questionnaire and about the milestones survey. After the video, the participants had the opportunity to ask questions to the researchers.

During the experiment, the two researchers registered times for the completion of the milestones, verified the progress, and asked questions about how to fill out the survey correctly. No answers were given to JabRef maintenance questions, JabRef architecture, Java programming, or help to pick up an issue for the contributions.

Once the participant reached a milestone, the researchers confirmed and registered the times. The researchers also kept track of the progress to register milestones even if the participant did not communicate it explicitly.

The sessions were recorded to permit the verification of the progress and confirm the timing of the milestones after the experiment.

8.1.8 Data Analysis

To examine the timing of the milestones, we ran the same statistical tests we used in previous studies, verifying the statistical significance of the differences between groups of variables. We employed the Mann-Whitney U test to compare the classifier metrics, followed by Cliff's delta effect size test. The Cliff's delta magnitude was assessed using the thresholds provided by Romano et al. [102], i.e., $|d| < 0.147$ "negligible," $|d| < 0.33$ "small," $|d| < 0.474$ "medium," otherwise "large."

Correctness was evaluated by examining the proposed answers and code. When the milestone did not need code, we verified it as a perfect match (e.g., file changed). When the milestone had a code associated and was available in the answer, we ran the code and examined the proposed changes. The code was first compared with the merged closed pull request. The proposed solution (written) was also examined to verify its soundness by two researchers (one JabRef contributor). The researchers individually evaluated the correctness and soundness, and when there was no agreement (only one case) they discussed to reach a consensus. The examination and evaluations took place after the experiments and took around one hour each.

Next, the code was analyzed independently to verify a possible alternative solution. In the case that an alternative solution was approved by the researchers, the previous milestones were re-evaluated to verify the correctness. For example: if a participant answered a class

that must be updated and it did not match the PR updated class, it was recorded as an incorrect milestone. However, if in the proposed solution the participant suggested a sound alternative to the solution and implemented it, all the previous milestones were checked again regarding their correctness.

To understand the rationale behind the participants' contribution, we qualitatively analyzed responses to open-ended questions ("Explain difficulty," "Explain confidence level," and "Why did you choose?"). We selected representative quotes to illustrate participants' perceptions of the relevance of labels.

We qualitatively analyzed the responses by inductively applying open coding in groups. We identify why the participant considered the information provided relevant and what information the participant would have liked to be provided. We built post-formed codes as the analysis progressed and associated them with the respective parts of the transcribed text to encode the relevance of the information according to the participants' perspectives.

Two researchers met weekly to discuss coding and categorization until we reached a consensus on the meaning and relationships between the codes.

8.2 Results

This section discusses the experiments' results. We start with the timed milestones, followed by the pre-survey and the post-survey.

8.2.1 RQ1: To what extent do the API-domain labels assist the contribution progress?

8.2.1.1 Timing Results

The participants from both the treatment and control groups first chose an issue. Next, they almost tied in the time to describe the problem and find the folder, file, and class. The difference started to grow again when identifying the method for the update and finding the lines to change. We found a statistical difference when comparing the times to describe a possible solution and implement it. Table 8.2 and Figure 8.3 show the timing and the statistical comparison among the milestones.

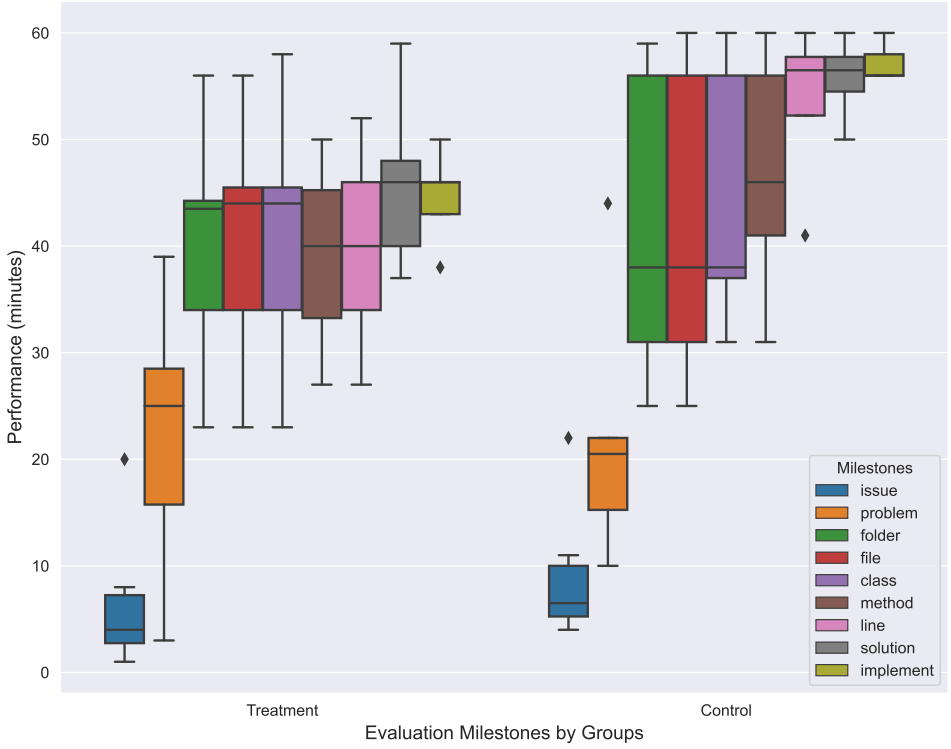

FIGURE 8.3: Timed milestones per group

TABLE 8.2: Cliff's Delta for F-Measure and Precision: comparison of corpus models by setup - TF-IDF.

Milestone	Cliffs Delta	
Issue	0	'negligible'
Problem	-0.06	'negligible'
Folder	0.73	'large'
File	0.73	'large'
Class	0.73	'large'
Method	0.73	'large'
Line	0.73	'large'
Solution	1	'large'*
Implementation	1	'large'*

** $p \leq 0.05$; ** $p \leq 0.01$; *** $p \leq 0.001$*

8.2.1.2 Milestone Correctness Results

Overall we observed an increase in correct milestones among participants in the treatment group. Figure 8.4 shows the increased performance of the treatment group in terms of milestone correctness. Wider sections of the violin plot represent a higher probability of observations taking a given value. The thinner sections correspond to a lower probability. The plots show that API-domain labels are more frequently related to milestone correctness when present in an issue (median is 1 correct milestone and 2.37 on average) as compared to regular labels in the same situation (median is 0 correct milestone and 0.71 on average). However, the distribution of the correctness is similar, as confirmed by

$$p$$

$= 0.11$. These results indicate that awareness of the technical (API) requirements of solving the task is essential to speed up the solution proposal and the change code contribution progress and ultimately produce more correct results.

Table 8.3 shows the statistical difference in milestone correctness between groups and the mean and median number of milestones correct for each group. Overall, both the median and the mean number of correct milestones are higher in the alternative group.

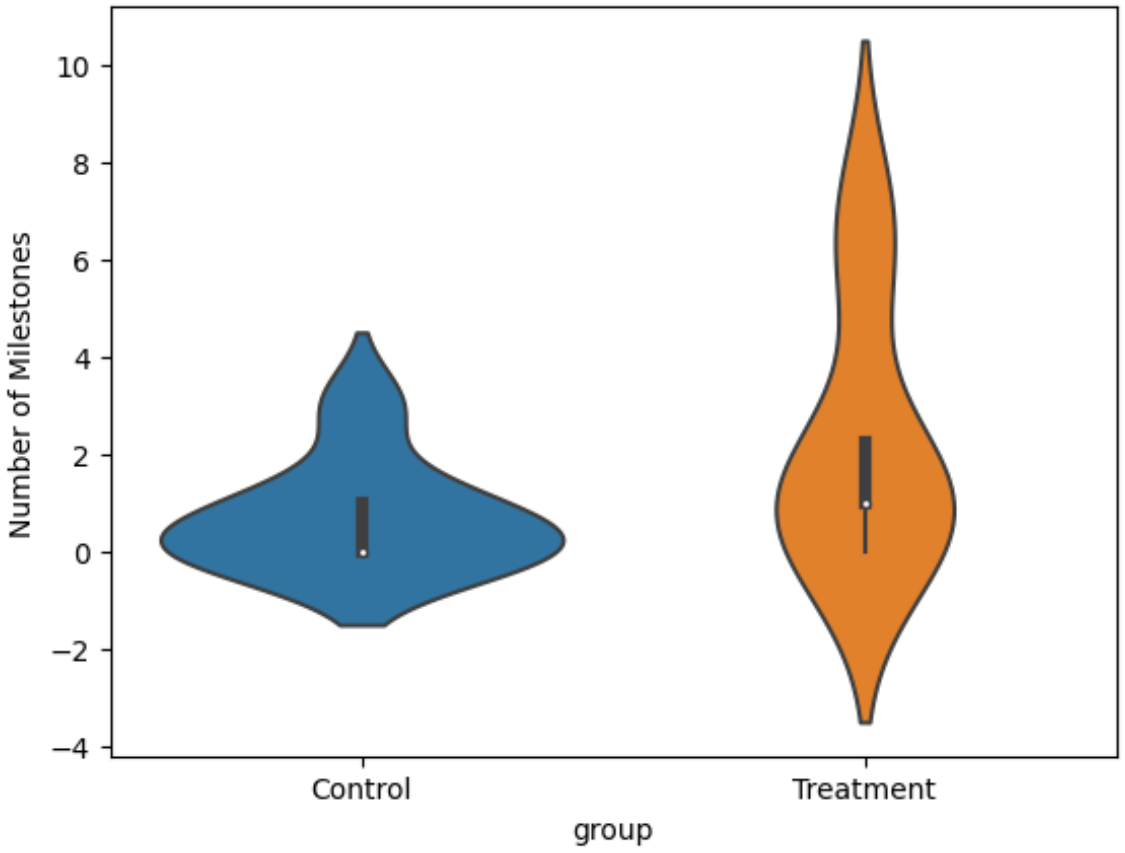

FIGURE 8.4: Number of milestones by groups. Control: black. Treatment: gray.

TABLE 8.3: Number of correct milestones in each group

Group	Mean	Median
Control	0.714	0.0
Treatment	2.37	1.0

This data suggests that participants in the treatment group were better able to complete their issues as they had more correctly completed milestones.

RQ.1 Summary. The API-domain labels significantly improved the time to reach the milestones "Propose a solution," and "Code the solution."

8.2.2 RQ2: What do API domain labels provide potential contributors?

8.2.2.1 Survey Analysis

We qualitatively analyzed the answers to the questionnaire regarding the experiment participation. The analysis included the participants' reasons to select the issue; the difficulties they faced while solving (or trying to solve) the issue; their feelings about being able (or unable) to solve the issue; how the API labels helped (or not) during the process; and their feedback about what could be improved for better support to solve the issue.

Reasons to select the issue We found four categories of reasons to select the issue: PERCEIVED EASE, SKILLS MATCHING, CURIOSITY, AND MORAL.

Six participants (P2, P3, P5, P6, P12, P13) mentioned reasons for their PERCEIVED EASE to solve the issue. This category of reason was observed when participants explained their selection in terms of the issue seeming *"doable"* (P5), *"easy to contribute"* (P2), suitable for a newcomer (according to the *"good first issue label"* (P12)), or because it was *"an issue widely found across many projects these days"* (P3). Perceived ease can be achieved when the issues are well described, as mentioned by P6: *"clear understanding from the issue comments and text"*, and also when there is a good match with the contributor's skills. P2 mentioned that *"[the selected] issue looked the most easy issue"*. Five participants (P4, P7, P8, P11, P14) indicated the reason for their selection as due to the match between issues and their skills. The SKILLS MATCHING reason brings self-confidence when contributors *"were able to find the bug"* (P14); i.e., having the suitable knowledge to solve the issue. P8 described an information processing cognitive style [154] of reading the issues and

"comparing [the selected issue] to other issues [he] thinks [he] may have some knowledge to solve this issue".

Three participants (P7, P10, P11) indicated having CURIOSITY towards the issue, either by pondering that the issue *"sounded interesting"* (P11) or because they *"wanted to know"* (P7) about the solution (in P7 case, *"which layout is missing"*). MORAL reasons to improve the software quality were mentioned by three participants (P1, P3, P9). P1 stated that he thinks the issue *"should"* be solved: *"URL should be visible to the end users and should fix that bug"*. P9 *"felt that fixing the issue would lead to better optimization"*, and P3 stated that the selected issue should be prioritized to be solved because it *"could lead to huge problems when not addressed on time"*.

Three participants from the treatment groups were able to match their skills versus two from the control group. Four participants from the treatment group perceived ease compared to two from the control group. Only the control group had participants mention curiosity as a reason.

Difficulties faced while solving (or trying to solve) the issue Perceived difficulty was measured through a 5-point Likert-scale item that ranged from "extremely easy" to "extremely difficult." The majority of participants (9 out of 15) considered the issue difficult to solve (including the answers "somewhat difficult" and "extremely difficult"). Only three (out of 15) participants considered the issue easy to solve (including the answers "extremely easy" and "somewhat easy").

The categories of difficulties included: ISSUE COMPLEXITY, FINDING THE ISSUE IN CODE, FAMILIARITY WITH THE CODEBASE, ENVIRONMENT SETUP, DEBUG PROCESS, AND INSUFFICIENT SKILLS.

P12 considered it "extremely easy" to solve the issue due to LOW COMPLEXITY, as (*"it was a very small change"*), whereas P4 considered it "somewhat easy" because he was

able to FIND THE ISSUE IN CODE when he *"found the error message in the source code"*. By contrast,P10 reported *"search for the files and for the code* as "somewhat difficult." P7 was *"able to locate the folder and classes"*, but considered it "extremely difficult" to *"find the location where the code needs to be changed"*. While P4 considered it "somewhat easy" to find the issue, P10 found it "somewhat difficult" to *"search for the files and for the code*. P6 also associated his difficulty with ENVIRONMENT SETUP: not having the *"proper steps on how to reproduce the issue"*. Both P9 and P14 linked difficulty with the DEBUG PROCESS. While P9 reported needing more *"time to debug the code and understand the way the functions are called and how it interacts with others"*, P14 regretted not having used *"debug points"* that could have helped on the solution. P5 and P8 felt they did NOT HAVE ENOUGH SKILLS to work on the issue. P5 explicitly mentioned the lack of knowledge of the programming language, and both P1 and P13 found it difficult NOT BEING FAMILIAR WITH THE CODEBASE (P1) and not understanding *"how the flow of the program works"* (P13). Two participants considered it neither easy nor difficult (the neutral answer), and one participant did not answer.

Dividing by groups, the control group evaluated the contributions in three instances as "somewhat difficult" to solve and once as "extremely difficult." In contrast, the treatment group had five instances of rating the contributions as difficult (though only "somewhat difficult") and one instance of an "extremely easy" rating. Both groups had one instance each of rating the contribution "somewhat easy" and "neither easy nor difficult."

Feelings of being able (or not) to solve the issue The feelings of being able to solve the issue were measured through a multiple choice question with yes, no, and not sure. Only one participant (P4) answered "yes", describing the perceived accomplishment as having *"found the code snippet raising the issue"*.

Seven participants answered "no," and six answered "not sure." The control group participants answered "no" three times (3/6), and the treatment group chose "no" four times

(4/8). Only the treatment groups had a participant who answered "yes" to this question. Both groups answered "not sure" three times.

Suggestions to improve the labels P7 and P12 suggested more directions to help understand the issue and how to reproduce it. P10 and P11 claimed more support in finding the issue in the code. P7 suggested visual cues with *"pictures of how to reproduce [the issue]"* and P12 offered more general advice about having *"some example of the issue"*. P5, P6, P8, and P9 desired better code readability (P8), more documentation (P5,P9), and comments in code (P6,P9).

RQ.2 Summary. We could not find evidence that the API-domain labels provided more confidence or reduced the perceived difficulty of the contributions.

8.3 Final Considerations

After investigating the impact of the API-domain labels, we aimed to provide OSS communities with a tool based on our approach. The next section presents the OSS demonstration tool that partially implements the approach proposed by this research.

CHAPTER 9: DEMONSTRATION TOOL

Following the idea to develop a tool for use by OSS communities and industry, we implemented *GiveMeLabeledIssues* to classify issues for potential contributors. *GiveMeLabeledIssues* is a web tool that indicates the API-domain labels for open issues in registered projects. Currently, *GiveMeLabeledIssues* works with three open-source projects: JabRef, PowerToys, and Audacity. The tool enables users to select a project, input their skill set (in terms of our categories), and receive a list of open issues that would require those skills.

9.1 GiveMeLabeledIssues Architecture

GiveMeLabeledIssues leverages prediction models trained with closed issues that are linked with merged pull requests. On top of these models, we built a platform that receives user input and queries the open issues based on the users' skills and desired projects. In the following, we detail the model's construction process, the issue classification process, and the user interface that enables users to receive the recommendation. *GiveMeLabeledIssues* is structured in two layers: the frontend web interface[1] and the backend REST API[2].

9.2 Model training

In the current implementation, a model must be built for each project using data collected from project issues linked with merged pull requests. The tool maps the issue text data to the APIs used in the source code that solved the issues (Figure 9.1).

[1]https://https://github.com/fabiojavamarcos/GiveMeLabeledIssuesUI
[2]https://github.com/fabiojavamarcos/GiveMeLabeledIssuesAPI

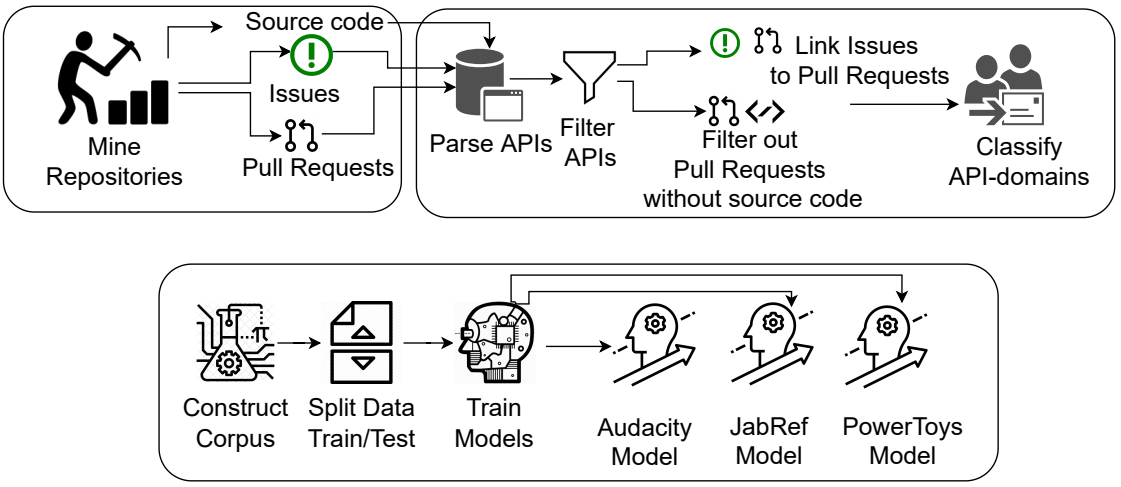

FIGURE 9.1: The Process of Training a Model

We used similar procedures from the generalization study to mine, parse, clear, and train the model, with a few adaptations to match the tool needs. For the sake of clarity, we summarized and pointed to differences in the following sections.

9.2.1 Mining repositories

We used the 18,482 issues and 3,129 pull requests (PRs) collected from JabRef, PowerToys, and Audacity projects up to November 2021. We used GitHub REST API v3 to collect the title, body, comments, closure date, name of the files changed in the PR, and commit messages.

9.2.2 APIs parsing

We used the parser from the generalization study to process all source files from the projects and to identify the APIs used in the source code affected by each pull request. In total, we found 3,686 different APIs in 3,108 source files. The parser looked for specific commands, i.e., import (Java), using (C#), and include (C++). The parser identified all classes, including the complete namespace from each import/using/include statement.

9.2.3 Dataset construction

To map issue data to the APIs used in the source code files changed to close the issue, we kept only the data from issues linked with merged and closed pull requests. To find the links between pull requests and issues, we searched for the symbol `#issue_number` in the pull request title and body and checked the URL associated with each link. We also filtered out issues linked to pull requests without at least one source code file (e.g., those associated only with documentation files), since they do not provide the model with content related to any API.

9.2.4 API categorization

We use the API domain categories defined by Santos et al. [24]. Experts defined these 31 categories to encompass APIs from several projects (e.g., UI, IO, Cloud, DB, etc.—see our replication package[3].

9.2.5 Corpus construction

The TF-IDF is not only used during the training. We can use models trained and previously saved or we can train the model each time we run the application with the data recently found in the extractor.

We used the issue title and body as our corpus to train our model since they performed well in our previous analysis [23]. Similar to other studies [80, 81], we applied TF-IDF as a technique for quantifying word importance in documents, assigning a weight to each word following the same process described in the previous work [23]. TF-IDF returns a vector whose length is the number of terms used to calculate the scores. Before calculating the

[3]https://doi.org/10.5281/zenodo.7575116

scores, we converted each word to lowercase and removed URLs, source code, numbers, and punctuation. After that, we removed templates and stop-words and stemmed the words, mimicking the cleaning procedures adopted by [23]. These TF-IDF scores are then passed to the Random Forest classifier (RF) as features for prediction. RF was chosen since it obtained the best results in previous work [23]. The ground truth has a binary value (0 or 1) for each API domain, indicating whether the corresponding domain is present in the issue solution.

We also offer the option of using a BERT model in *GiveMeLabeledIssues*. We reproduced the procedures used in the generalization study and created two separate CSV files to train BERT: an input binary with expert API-domain labels paired with the issue corpus and a list of the possible labels for the specific project. BERT directly labels the issue using the corpus text and lists possible labels without needing an additional classifier (such as Random Forest).

9.2.6 Building the model

The BERT model should be built offline due to the amount of processing time needed. The BERT model was built using the Python package fast-bert[4], which builds on the Transformers[5] library for Pytorch. Before training the model, the optimal learning rate was computed using a LAMB optimizer [101]. Finally, the model was fit over 11 epochs and validated every epoch. The BERT model was trained on an NVIDIA Tesla V100 GPU with an Intel(R) Xeon(R) Gold 6132 CPU within a computing cluster.

TF-IDF and BERT models were trained and validated for every fold in a ShuffleSplit 10-fold cross-validation. Once trained, the models were hosted on the backend. The replication package contains instructions on registering a new project by running the

[4]https://github.com/utterworks/fast-bert
[5]`https://huggingface.co/docs/transformers/index`

model training pipeline that feeds the demo tool. The models can then quickly output predictions without continually retraining with each request. TF-IDF model is fast to build and may have the option to be trained with fresh data.

For the RandomForestClassifier (TF-IDF), the best classifier, we kept the following parameters: criterion = 'entropy,' max depth = 50, min samples leaf = 1, min samples split=3, and n estimators = 50.

9.3 Issue Classification Process

GiveMeLabeledIssues classifies currently open issues for each registered project. The tool combines the title and body text and sends it to the classifier. The classifier then determines which domain labels are relevant to the gathered issues based on the inputted issue text. The labeled issues are stored in an SQLite database for future requests, recording the issue number, title, body, and domain labels outputted by the classifier.

The open issues for all projects registered with *GiveMeLabeledIssues* are reclassified daily to ensure that the database is up to date as issues are opened and closed. Figure 9.2 outlines the daily classification procedure.

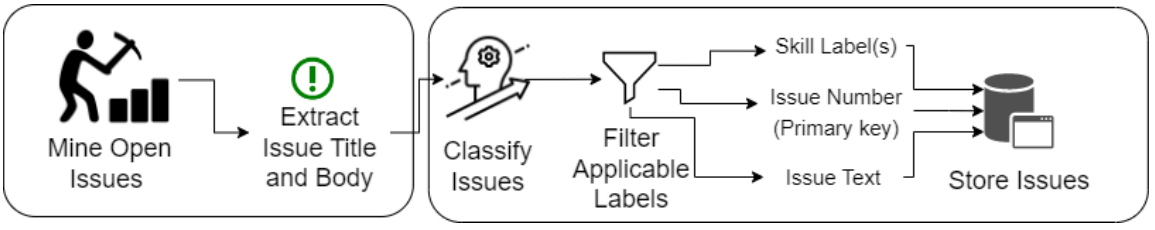

FIGURE 9.2: The Process of Classifying and Storing Issues

9.3.1 User Interface

GiveMeLabeledIssues outputs the labeled issues to the User Interface. The user interface is implemented using the Angular web framework. To use the tool, users provide the project name and select API-domain labels that interest them. This information is sent to the backend REST endpoint via a GET request. The backend processes the request, recommending a set of relevant issues for the user.

The backend REST API is implemented using the Django Rest Framework. It houses the trained TF-IDF and BERT text classification models and provides an interface to the labeled issues. When receiving the request, the backend queries the selected project for issues that match the user's skills. Once the query is completed, the backend returns the labeled issues to the user interface. Each labeled issue includes a link to the open issue page on GitHub and the issue's title, number, and applicable labels. The querying process is shown in Figure 9.3.

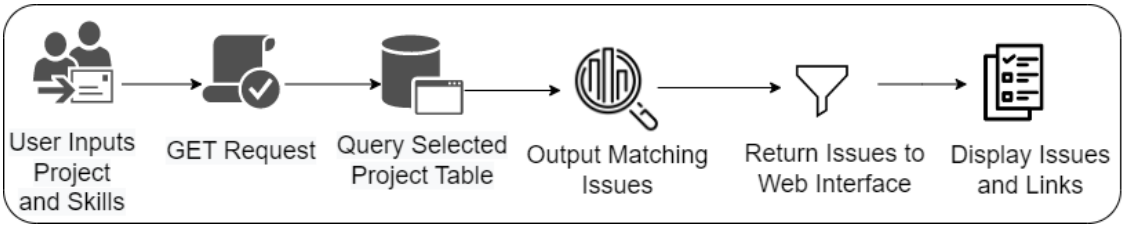

FIGURE 9.3: The Process of Outputting Issues

Figure 9.4 shows JabRef selected as the project and "Utility," "Databases," "User Interface," and "Application" as the API domains provided by the user. Figure 9.5 shows the results of this query, which displays all JabRef open issues that match those labels.

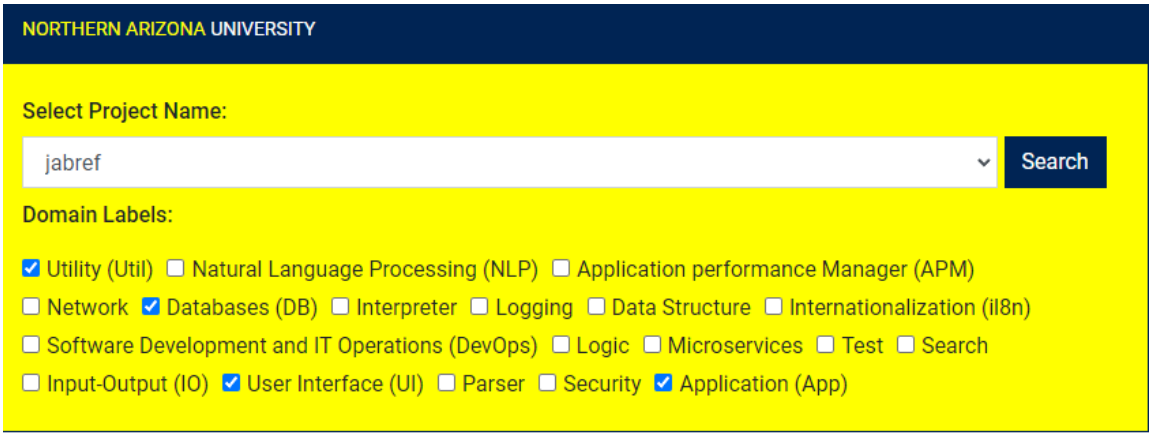

FIGURE 9.4: Selection of a project and API domains

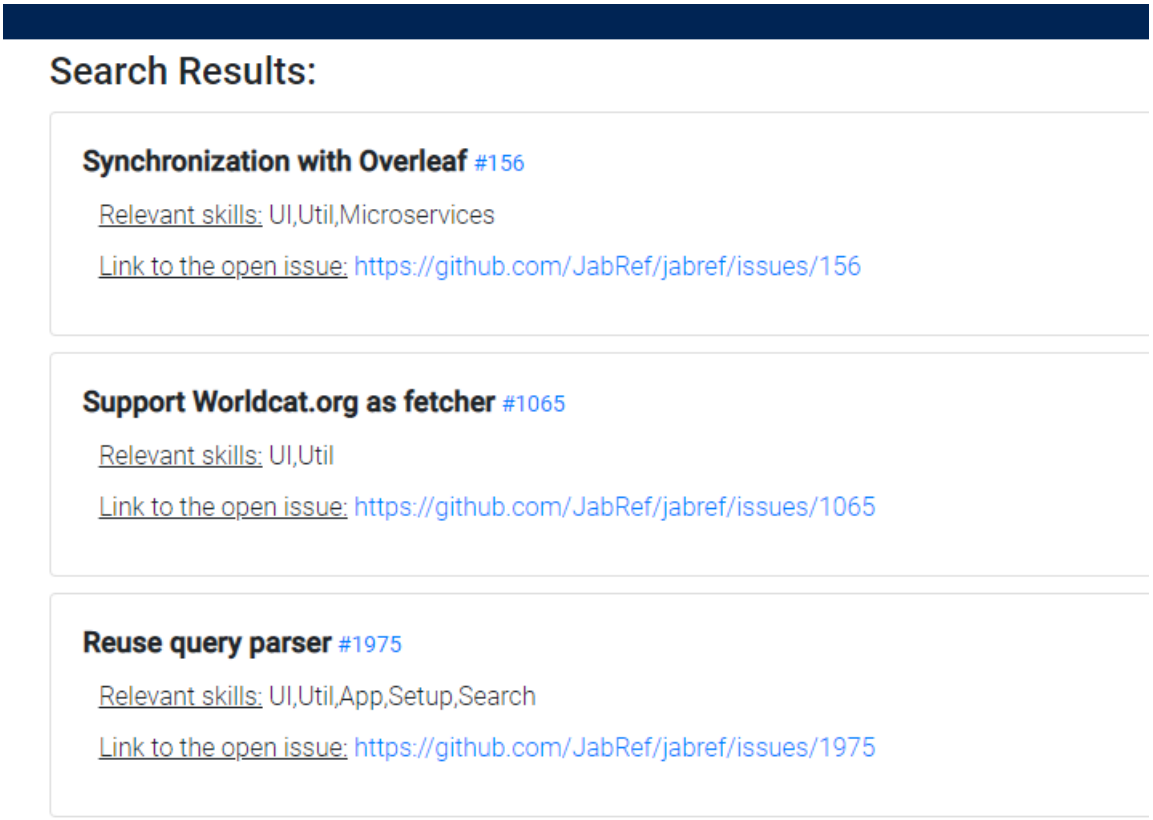

FIGURE 9.5: Labeled Issues Outputted for JabRef with the Utility, Databases, User Interface, and Application skills Selected

9.4 Results

9.4.1 Evaluation

We have evaluated the performance of the models used to output API-domain labels using a dataset comprised of 18,482 issues, 3,129 PRs, 3,108 source code files, and 3,686

Table 9.1: Overall metrics from models -averages. RF-Random Forest* Hla - Hamming Loss**

Model averages	**O**ne/**M**ulti projects	Precision	Recall	F-measure	Hla**
RF* TF-IDF	O	**0.839**	0.799	0.817	**0.113**
BERT	O	0.595	0.559	0.568	0.269
RF* TF-IDF	M	0.659	0.785	0.573	0.153
BERT	M	0.593	0.725	0.511	0.219
Izadi et al. [155]	M	0.796	0.776	0.766	NA
Kallis et al. [10]	M	0.832	**0.826**	**0.826**	NA
Santos et al. [23]	O	0.755	0.747	0.751	0.142

Table 9.2: Overall metrics from models - by projects. RF-Random Forest* Hla - Hamming Loss**

Model by project	Project	Precision	Recall	F-measure	Hla**
RF* TF-IDF	Audacity	**0.872**	**0.839**	**0.854**	0.103
RF* TF-IDF	JabRef	0.806	0.782	0.793	0.143
RF* TF-IDF	PowerToys	0.84	0.776	0.805	**0.094**
BERT	Audacity	0.382	0.511	0.434	0.42
BERT	JabRef	0.791	0.606	0.686	0.192
BERT	PowerToys	0.619	0.643	0.626	0.187

distinct APIs, chosen from active projects and diverse domains: Audacity (C++), PowerToys (C#), and JabRef (Java).[6] Table 9.1 shows the results with precision, recall, F-measure, and Hamming loss values. We trained models to predict API-domain labels using individual issue datasets from each project and a single dataset that combined the data from all the projects.

As shown in Table 9.1, TF-IDF overcame BERT both in the per-project analysis and for the complete dataset. The difference was quite large when using single projects. The results were closer when we used the combined dataset that included data from all projects (3,736 linked issues and pull requests). We hypothesize that the sample size influenced the classifiers' performance. This aligns with previous research on issue labeling that showed that BERT performed better than other language models for datasets larger than

[6]The model training replication package is available at `https://zenodo.org/record/7726323#.ZA5oy-zMIeY`

5,000 issues [156]. TF-IDF performs very well when the dataset is from a single project because the vocabulary used in the project is very contextual, and the frequency of terms can identify different aspects of each issue. When we include the dataset from all the projects, the performance of TF-IDF drops, as the context is not unique. These results outperformed the results from the API-domain labels case study conducted by Santos et al. [23]. The project metrics (Table 9.2) varied less than 6% (e.g., the recall: 0.839 (Audacity) - 0.776 (PowerToys). Audacity had the best scores for all metrics except Hamming loss.

CHAPTER 10: THREATS TO VALIDITY

There are some limitations related to our research results.

Internal Validity. One of the threats to the validity of this study is the API domain categorization. We acknowledge the threat that different individuals can create different categorizations, possibly introducing bias in our results. To mitigate this, three individuals, including two senior developers and a contributor to the JabRef project, created the API-domain labels categories, aiming to generalize to many projects by using generic instead of tailored labels for the project. In the future, we can improve this classification process with a collaborative approach (e.g., [157, 158]). A limitation of this approach is that NLP suggestions might not work as expected in languages like C++, where the information about the API packages is not present in the "include" declaration. For researchers interested in extending our work for projects from these languages, we recommend using another source of information beyond the API namespace, such as comments and documentation, to achieve proper library categorization.

Although participants with different profiles participated in the JabRef empirical study, the sample cannot represent the entire population, and the results can be biased. The study link randomly assigned a group to each participant. However, some participants did not finish the questionnaire, and the groups ended up not being balanced. Also, the way we created subgroups can introduce bias in the analysis. The practitioners' classification as industry and students was done based on the location of the recruitment, and some students could also be industry practitioners and vice-versa. However, the results of this analysis were corroborated by the aggregation by experience level.

Construct Validity. Another concern is the number of issues in our dataset and the link between issues and pull requests. To include an issue/key/tracking ID in the dataset, we linked it to its solution submitted via pull request (or "revision," "trouble ID"). By linking them, we could identify the APIs used to create the labels and define our ground truth

(check Section 5.1). We manually inspected a random sample of issues (or "keys", and "tracking ids") to check whether the data was correctly collected and reflected what was shown on the ITS interface. Two authors manually examined 100 tasks from the projects, comparing the collected data with the GitHub interface. All records were consistent, and all of the issues in this validation set were correctly linked to their pull requests. When the linked data has more than one correspondence, we concatenated all data using the appropriated corpus entry (title, body, comments, description, and summary). Some of the linked data sometimes had repeated text, which can overfit our model. Future versions may improve the data cleaning step. Unlike the other projects, Cronos had multiple linked data through the following columns: "pai" and "ramo," "linked issue" and "key," and "key" and "ramo." This creates a recursive situation where we can link each update with many "keys" in different ways. We preferred to keep it simple, using only the linked data that was similar to the other projects: "key" and "ramo."

When an issue is closed without leaving a trace on the ITS, we cannot track it using this method, and therefore the issue is discarded. The need to have a solution to the issue also introduces a bias to the generability of our results. We only tested the predictions for issues that had a linked PR to be able to establish the ground truth. When using our approach in practice to label open issues, some issues may not be related to code solutions or may have different characteristics than those that have a linked PR and may receive incorrect labels, decreasing the performance observed in our study.

We used the presence of an API in the file changed by a PR to define the ground truth of the API domains. The API does not necessarily affect the lines of code changed by a PR. Future work can explore more sophisticated ways to link issues and API domains, increasing the accuracy of the labels.

Despite the feature importance test highlighting some features as the best predictors, this was observed when we isolated the social metrics as features. When mixing all the

features (social metrics + TF-IDF weights), the feature importance among the social metrics decreased.

In prediction models, overfitting occurs when a prediction model has random error or noise instead of an underlying relationship. During the model training phase, the algorithm used information not included in the test set. To mitigate this problem, we also used a shuffle method to randomize the training and test samples.

Further, we acknowledge that we did not investigate in the case study whether the labels helped the users find the most appropriate tasks. It was not part of the user study to evaluate how effective the API labels were in finding a match with user skills. Additionally, we did not evaluate how false positive labels would impact task selection or ranking. Our focus was on understanding the relevance that the API-domain labels have on the participants' decisions. However, we believe the impact is minimal, since in the three most selected issues, out of 11 recommendations in the JabRef project, only one label was a false positive. Investigating the effectiveness of API labels for skill matching and the problems that misclassification causes are potential avenues for future work.

The contribution empirical experiment relied on a few issues to permit the groups' performance comparison. It also was limited to issues that can be addressed in a limited time frame. Both adaptations can bias the experiment since they reduced the issue population evaluated. The evaluation of the correctness of the milestones was carried out by two researchers. Some proposed written solutions and code were considered satisfactory but they can hide undiscovered bugs. The selected issues also did not contemplate all possible labels (31) and therefore the evaluation of the progress cannot be claimed for all issues.

External Validity. Generalization is also a limitation of the JabRef case study. The outcomes could differ for other projects, programming languages ecosystems, or even issues written in a different language. To address this limitation, we extended the previous study in that direction, mining different projects, including three programming languages

and two vocabularies. Nevertheless, this study showed how a multi-label classification approach could be useful for predicting API-domain labels and how relevant such a label can be to new contributors. Moreover, the API-domain labels that we identified can generalize to other projects that use the same APIs across multiple project domains (Desktop and Web applications), since the chances that most APIs in a project had been previously categorized increase as our work is extended to new APIs and projects. Many projects adopt a typical architecture (MVC) and frameworks (JavaFX, JUnit, etc.), which makes them similar to many other projects. As described by Qiu et al. [111], projects adopt common APIs, accounting for up to 53% of the APIs used. Moreover, our data can be used as a training set for automated API-domain label generation in other projects. Although we included in this study projects encompassing several programming languages, domains, sizes, and languages, the sample may not necessarily be representative of reality.

Generalizability. One can argue that a majority of our interviewees identified as men. Although it is a similar distribution to typical OSS gender demographics [159–161], we could have found new insights with a more diverse distribution of gender. The strategies uncovered in our study are not meant to be exhaustive, and further research into different types of projects will likely uncover other strategies. Furthermore, we acknowledge that our sample may be biased in unknown ways, and our results are only valid for our respondents. Additionally, the results presented in this paper are related to open source communities. Thus, we do not expect that the strategies found in our study will be directly applicable to other software domains. Nevertheless, to allow replication of our study, we carefully describe our research method steps.

Replicability in qualitative research is hard, since human behaviors, feelings, and perceptions change over time. Merriam [162] suggests checking the consistency of the results and inferences. Consistency refers to ensuring that the results consistently follow the data and the data analysis can support all inferences. To increase consistency, we performed data analysis in pairs, which was consistently revised by two experienced researchers. We

held weekly meetings to discuss and adjust codes and categories until we reached an agreement. We also performed member checking with four participants, who confirmed our interpretation with minor changes. Moreover, we provide the codebook for traceability and increased comprehensibility and repeatability.

Theoretical saturation. A potential limitation in qualitative studies is not reaching theoretical saturation. The quality, rather than the size, of the sample of participants, is essential to increase our confidence in the results. In this study, we interviewed 17 participants with different perspectives and perceptions about the studied phenomenon. Our participants were diverse in terms of the number of years with OSS and roles. Further, these participants represent 26 different OSS projects of different sizes. The number of projects is higher than the number of interviewees, as some of them contribute to more than one project in parallel. The number of interviewed participants was adequate to uncover and understand the core categories in an all-defined cultural domain or study of lived experience [119]. While we cannot claim saturation, our population has helped us uncover a consistent and comprehensive account of the strategies.

Inappropriate participation. As described in Section 6.1.8, we employed several filtering and inspecting strategies to reduce the possibility of fake data; however, it is not possible to claim that our data is completely free of this threat. From the 12 answers indicating no previous contributions, only two respondents had no coding experience but have contributed in other ways to projects they work on (possibly as non-coders). Since non-coder contributions are also valuable, we decided to include these answers. Some participants did not answer the name/number of the projects they contributed to as it was not a mandatory answer. To verify the commitment and experience we relied on the questions: How frequently do you contribute to OSS projects? How many years contributing to OSS projects? How many years of programming experience do you have?

The milestones empirical experiment recruited participants and tried to avoid unbalanced

participants by filtering the ones with no and large experience or contributions. We also randomly selected them for the groups. Despite those measures, we can not guarantee bias nonexistence.

CHAPTER 11: IMPLICATIONS

Implications for Researchers. (i) Our work shows the importance of using social metrics as prediction factors, which is aligned with the idea that software engineering is a sociotechnical activity. Many studies use prediction models based solely on technical attributes and miss the opportunity to improve the results with social metrics. We expect our results to inspire further investigation of social metrics in other contexts and applications. (ii) Our results illustrate the value of replicating previous work not only to confirm previous findings but to extend with other ideas to improve the state of the art. (iii) The scientific community can extend our approach to predict contributors' skills using a similar approach as we used for issues. The community can also build tools that use our approach to recommend tasks according to contributors' skills and career goals (e.g., as proposed in the literature [118]). Automatic matchmaking can use the historical contributions data, when available, to compare the skills of tasks and contributors. We published the software, scripts, and data we used to facilitate replication, use, and extension. (iv) We used just one source of information to build our social metrics. We hope our results will inspire the software engineering community to explore the multitude of tools developers use to collaborate and use/propose new metrics and communication channels as information sources to calculate the metrics. Many prediction models end up not being used in practice due to suboptimal performance. Exploring ways to improve state of the art is paramount to transferring the scientific results into practice.

We understand that further studies are necessary before adopting our approach in practice. Our study is an essential step toward improving the predictions and testing the approach in different projects. In the following, we discuss some implications, assuming the practical adoption of the approach.

Implications to new contributors. Social metrics can potentially improve label prediction, which increases the chance of adopting our approach in practice to facilitate task

selection. The literature shows that newcomers find labels in the issue tracker useful to help find their tasks [163]. Facilitating the choice of a task—and consequently the onboarding of newcomers—is crucial for newcomers to overcome the various barriers the literature has shown they face, which often leads them to give up contributing [6].

Implication for project maintainers. Our approach can be used by maintainers to automatically label issues on the issue tracker system of their projects. OSS maintainers are known to be busy, and issues end up being poorly labeled due to the lack of time to label them manually. Providing an approach that can automatically label issues has the potential of supporting their job, making the issues easier to find and enabling contributors to shortlist the issues by filtering those that match their skills with the tasks. Ultimately, this can facilitate the onboarding and engagement of contributors, which is vital for the sustainability of the projects [164, 165].

Educators. Educators often recommend their students contribute to OSS projects. Educators also adopt this activity as a course assignment. However, the literature has shown that students struggle to find suitable tasks [6]. Improving the automatic labeling of issues is a step toward improving task selection, which is very important for these students. Our labels are related to API domains, which map to skills students may want to develop (e.g., testing, security, UI, etc.) or are comfortable with.

CHAPTER 12: FUTURE WORK

In future work, we will have the opportunity to follow some exciting directions, such as:

1. Matching skills among contributors and tasks. Contributors can be biased when choosing an issue to start, even with the available API-domain labels. Therefore, a similar approach can be used to predict the contributors' skills using each contributor's API-domain expertise. The skills also can be obtained by mining the technical roles [166].

2. Multi-level Skill Mining. So far, our skills are based on the 31 "flat" domains elected by the experts. However, the skills may be aggregated and ranked in levels. [167] proposed a way to level the social skills we can use to rank the skills while we have to mine different data to build our rank model.

3. Matching Large Network/Conceptual Models Structures. Matching multi-level skills from one project with team resources can be done without processing time issues. What if we want to match all projects in a global company with all available human resources to optimize workforce allocation? For such a large-scale matching problem, comparing all possible contributors and tasks sounds unfeasible. [168] proposed a combined stochastic and algebraic method to avoid the Cartesian product exploring the data distribution and the existence of duplicate information to be compared.

4. Future work should reach maintainers to receive feedback about how communities can adopt the strategies and how to automate them. Additional research may also propose ways to improve productivity in OSS communities by analyzing multi-team research that possibly shares problems with OSS projects, like team churn and poor team coordination. Reusing mature strategies to create robust contribution processes, collaborations and supporting the integration of new team members seem

to liaise with the challenges faced by the OSS communities. Additional research can also uncover the sequence and prerequisites of the strategies. Finally, another interesting future work would be handling the choice paradox by suggesting a step-by-step project–a customized process newcomers can follow to track progress and avoid getting lost in selecting an issue.

12.1 Matching contributor and task skills

The background chapter 2 discussed the complexity of determining skills and how to use an ontology model to hold the concepts related to the skills. The skill ontology also provides a way to extend our work, adding information about levels of the skill (or competence) and paving a research roadmap to support the matching.

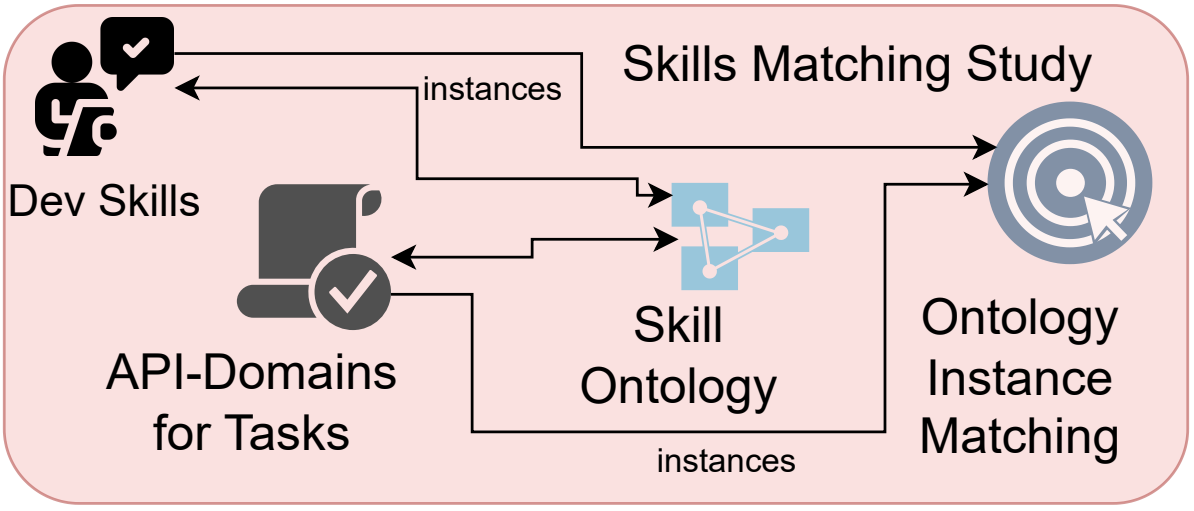

FIGURE 12.1: The research design future work - skills matching

Method - Skills Matching The skills method comprises the mining repository, skill ontology selection, skill ontology creation, and skills matching (Figure 12.1).

This study aimed to address the research questions:

- **RQ3.** To what extent can contributors match their skills with tasks?

Mining Repositories The mining repository procedures should be the same principles observed in the previous study to mine the tasks' skills [23]. In addition, we will mine the authors' skills to match them with the skills of the task. To define the authors' skills we will aggregate all the issues and skills committed by the author. Indeed the authors' skills will be the sum of all the issues skills that were solved by a commit from each author. Suppose author "A" solved issues 333 and 444 and issue 333 predicted the API-domain labels "UI" and "IO" and issue 444 has predictions "DB" and "Cloud", therefore we will suppose author "A" has skills in "UI", "IO", "DB", and "Cloud".

Ontology Selection Many ontology projects embrace competence management (CM). Some of them are specific to domains or embrace different aspects of CM, like skills evolution, task requirements, and workplace role. Many authors propose the use of CM systems, supported in the background by ontologies and allowing the management of competencies at the institutional level [169–175]. Some are generic and may be applied in a broad context [176, 177] or focus on high-level abstractions [178].

To pick the best option for this study, we must find one that can describe the skills of individuals and tasks. For instance, since we are not interested in modeling the evolution of individual skills in time or career, it is not mandatory to choose one ontology covering these aspects.

Ontology Engineering We will build the ontology instances model, employing the selected ontology. [179] proposes NeOn, an ontology engineering method. Our goal is to reuse ontological resources, what the authors called "scenario 3," seizing the opportunity from the work of [180] instead of proposing a new ontology to address our research problem. They proposed an ontology based on gUFO, a lightweight implementation of

the Unified Foundational Ontology (UFO) [181] to model skills in an enterprise context. Reusing a skill or competence management ontology prevents us from reinventing the wheel and building a new ontology from scratch, which takes time and should be relied on only when one can not find a suitable ontology to use. UFO ontologies define high-level abstractions to support domain-driven ontologies that can anchor their concepts properly.

Populating the Ontology After the ontology is selected, we need to instantiate ontology instances with the results produced by the classifier in section 4. Manually populating the ontologies with instances is possible. However, it should be unfeasible for a large number of instances. Here we have some options depending on the target programming language: OWL-API [182] and OWLReady [183]. Many other tools have limited scope, support some operations, and deserve brief research—for example, owlapy, neo4j, and AllegroGraph (https://github.com/pysemtec /semantic-python-overview). We will prioritize OSS libraries.

The OWL-API (https://www.w3.org/2001/sw/wiki/OWLAPI) was developed in Java and implements the W3C Web Ontology Language OWL, including OWL 2, which encompasses OWL-Lite, OWL-DL, and some elements of OWL-Full. OWL is a W3 standard.

OWLReady and OWLReady2 are Python implementations of the OWL.

The choice will be made based on the API capabilities of building and exporting the OWL file to the matchers.

Planning the Measurement Instrument We will use it to measure the experiments– a matcher that can find correspondences using input ontology instances. The following matchers are candidates for the experiment: LogMap [184], Lily [185], and AML

[186]. They have been used in the OAEI (Ontology Alignment Evaluation Initiative - https://oaei.ontologymatching.org/) [187] with good results.

The expected matching is depicted in figure 12.2. Ana and Luan are candidate contributors for two new open issues: 333 and 444. Ana has skills in Java, Cloud, and user interface (UI), while Luan has skills in Python, databases (DB), and Cloud. However, the task requirements are 333: Java, DB, Cloud, and UI. Furthermore, 444 requires Python and UI. So, the instance matching could be estimated as $< Ana, 333, 0.75 >$, $< Ana, 444, 0.50 >$, $< Luan, 333, 0.50 >$, $< Luan, 444, 0.50 >$. Indeed, the match was possible traversing the concepts until the more generic ones: UI front-end competence, language back-end competence, DB back-end competence, and cloud Bbck-end competence. The orange (Luan) and blue (Ana) dotted lines show the instance alignments. The orange and blue unbroken lines show the relationship between the instances and the concepts. The green dotted lines show the path traveled to anchor the instances to the main concepts.

Analyzing Data After submitting the ontologies from the individual and the task, we will analyze the metrics to compare them with a threshold. The threshold will determine whether or not the skills match.

To evaluate the matchers, we employed the following metrics (calculated by the matchers)

- **Precision** measures the proportion between the number of correctly predicted labels and the total number of predicted labels.
- **Recall** corresponds to the percentage of correctly predicted labels among all truly relevant labels.

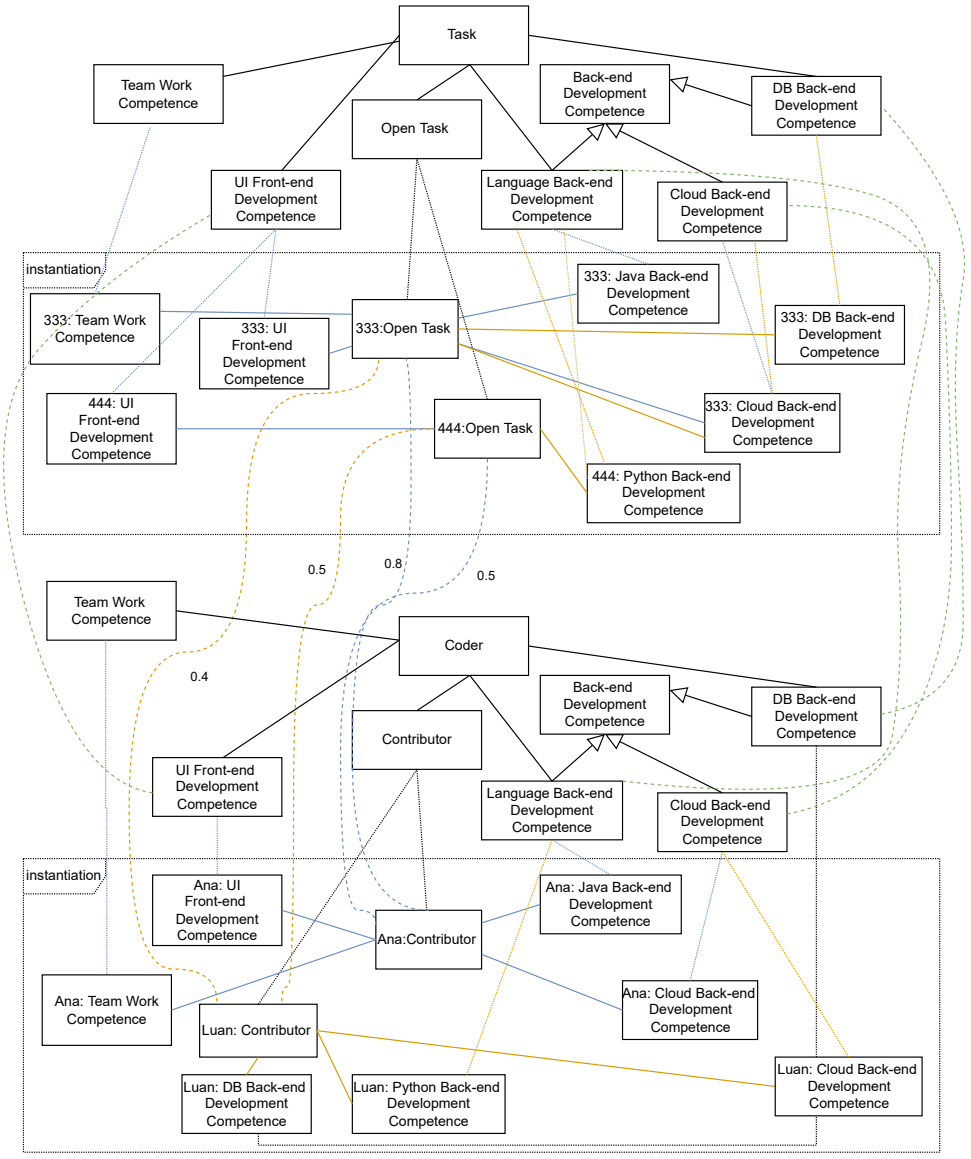

FIGURE 12.2: Ontology Instance Matching. Skills Example.

- **F-measure** calculates the harmonic mean of precision and recall. F-measure is a weighted measure of how many relevant labels are predicted and how many of the predicted labels are relevant.

CHAPTER 13: CONCLUSION

When joining a new project, newcomers face difficulty choosing an issue with which to start. Several studies addressed this problem by evaluating the developers' expertise or the required skills in tasks to inform and recommend issues for a contribution. Towards this goal, APIs correlate to skills to update and fix source code, and knowing which skills are needed for a possible solution may assist a newcomer in selecting an issue to unravel.

We investigated whether newcomers use API-domain labels to select an issue and what information newcomers use to decide what issue to which to contribute. We found that industry practitioners and experienced coders prefer API-domain labels more often than students and novice coders. Participants prefer API-domain labels over component labels already used in the project. Users would like to see labels with information about issue type, priority, programming language, complexity, technology, and API and choose an issue based on the title, body, comments, and labels.

We interviewed maintainers from diverse OSS projects and identified 27 strategies (grouped in five categories) that a newcomer uses to choose a task and 40 strategies (grouped in seven categories) communities employ to help newcomers. Next, we surveyed maintainers, newcomers, and frequent contributors to rank the newcomers' and maintainers' strategies. Using a Schulze method, we ranked the relative importance of the strategies to elucidate which ones are seen as more relevant for contributors in different roles (newcomers, frequent contributors, and maintainers), highlighting how they diverge. We found maintainers and newcomers diverge on the importance of the onboarding process, the improvement of the contribution process, and team communication. Overall, stakeholders agreed on the priority of project quality, good documentation, correctly reading and understanding the problem, and identifying what changes need to be made.

Prior works proposed several guidelines, mitigation strategies, and processes to overcome the initial barrier faced by the newcomers. Our ranking might be used to prioritize the

management effort in OSS projects or support goals to improve the onboarding process. Strategies that converge in serving the various stakeholders can decrease existing gaps in perspectives, obviating the problem of the expectation gulf.

We also investigate to what extent we can predict API-domain labels. We mined data from 22,231 issues from five projects and predicted 31 API-domain labels. Training and testing the projects separately, TF-IDF with the Random Forest algorithm (RF) and unigrams obtained a precision of 84% and over- came BERT (precision of 62%). Data from the issue body offered the best results. However, when predicting the API-domain labels for all projects together, RF precision decreased to 78%, and BERT increased to 72%, suggesting the positive sensibility of the BERT technique when applied to larger datasets. Transferring learning from diverse sources and targets resulted in a decrease in evaluation metrics with an extensive range of values regarding precision and recall. Future work should investigate ways to determine when or how to apply transfer learning to API-domain labels among projects.

Developers agreed that up to 64.4% of the API-domain labels are important to identify the skills and therefore should help to solve the issues if they are available.

We investigated the performance of the social metrics to improve the previously proposed API-domain labels. We found a set of predictors able to enhance the model's performance. Subsequent studies should unravel ways to discover when a project is sensible to a specific metric threshold to avoid exhaustive possibilities. Our results suggest that the conversations on issues convene interested discussants or experts on those subjects. We substantially improve the API-domain label predictions up to 0.922 (precision), 0.978 (recall), and 0.942 (F-measure) using the conversation data when the project has enough participants and dialogs to create a consistent dataset. The outcomes of this study compared with Santos et al. [23], whose results reached an increase of 15.82% and F-measure by 15.89%. With these results in mind, maintainers and communities can encourage and

leverage project communication to calculate social metrics. Better classifier performance can encourage the practical adoption of automatic issue labeling.

We verified the contribution progress was impacted by the adoption of the API-domain labels in the last two steps: formulating a solution and coding it. More extensive investigation is needed to confirm the findings in different projects and experiment with longer and more complex issues.

Finally, we demonstrated the usefulness of the approach by implementing an OSS tool able to recommend issues based on what the contributor selects on the user interface.

CHAPTER A: APPENDIX TITLE

Additional data from results. Some data were presented with box plots in Sections 5.5.2, 5.5.3 and 5.5.4. The redundant data (and more detailed) about the experiments are available here in tables.

TABLE A.1: overall metrics from models created to evaluate the corpus. Hla*

Model	Corpus	Precision	Recall	F-measure	Hla
TD-IDF	Title (T)	0.830	0.794	0.809	0.117
TD-IDF	Body (B)	0.840	0.786	0.811	0.116
TD-IDF	T, B	0.839	0.799	0.817	0.113
TD-IDF	T, B, Comments	0.831	0.796	0.812	0.116
BERT	Title (T)	0.616	0.592	0.596	0.277
BERT	Body (B)	0.599	0.598	0.591	0.27
BERT	T, B	0.595	0.559	0.568	0.269
BERT	T, B, Comments	0.597	0.587	0.582	0.266

** Hla - Hamming Loss*

TABLE A.2: overall metrics (Section 5.5.2) from models created to evaluate the number of grams.

Model	Precision	Recall	F-measure	Hla*
unigrams (1,1)	0.841	0.829	0.834	0.115
bigrams (2,2)	0.844	0.809	0.825	0.119
trigrams (3,3)	0.841	0.809	0.822	0.123
quadrigrams (4,4)	0.845	0.798	0.819	0.125

** Hla - Hamming Loss*

TABLE A.3: overall metrics (Section 5.5.2) from models created to evaluate the algorithms. Hla*

Model	Hla	Precision	Recall	F-measure
DecisionTree	0.105	0.861	0.837	0.847
Dummy	0.202	0.749	0.658	0.698
LogisticRegression	0.120	0.858	0.792	0.822
MLPClassifier	0.107	0.853	0.846	0.848
MLkNN	0.126	0.837	0.801	0.816
RandomForest	0.107	0.864	0.836	0.849

** Hla - Hamming Loss*

Table A.4: overall metrics (Section 5.5.3) from model created by training all projects together evaluate the algorithms.

Model	Hla*	Precision	Recall	F-measure
DecisionTree	0.157	0.768	0.576	0.654
LogisticRegression	0.167	0.779	0.504	0.611
MLPClassifier	0.154	0.766	0.595	0.666
MLkNN	0.179	0.709	0.555	0.617
RandomForest	0.153	0.785	0.573	0.659

** Hla - Hamming Loss*

We also include the confusion matrix for all projects trained and tested alone (Tables A.5–A.10). The confusion matrix for the RTTS project is in Table 5.12 on Section 7.4.

Table A.5: Overall performance from the selected model - JabRef project

API-domain	TN	FP	FN	TP	Precision	Recall
Network	13	6	8	19	0.76	0.70
DB	43	1	0	2	0.67	1
Interpreter	12	9	5	20	0.69	0.8
Logging	0	7	0	39	0.85	1
Data Structure	45	0	0	1	1	1
i18n	40	0	5	1	1	0.17
Setup	33	3	1	9	0.75	0.9
Microservices	42	0	4	0	0	0
Test	41	0	3	2	1	0.4
IO	0	6	0	40	0.87	1
UI	4	2	0	40	0.95	1
App	41	1	2	2	0.67	1

Table A.6: Overall performance from the selected model - Powertoys project

API-domain	TN	FP	FN	TP	Precision	Recall
APM	72	3	11	7	0.70	0.39
Interpreter	91	0	1	1	1	0.50
Logging	92	1	0	0	0	0
Data Structure	90	1	0	2	0.67	1
i18n	92	0	1	0	0	0
Setup	47	6	11	29	0.83	0.72
Logic	87	1	0	5	0.83	1
Microservices	70	1	15	7	0.88	0.32
Test	91	1	0	1	0.50	1
Search	41	6	8	38	0.86	0.83
UI	25	15	1	52	0.78	0.98
Parser	87	1	2	3	0.75	0.60
App	22	12	0	59	0.83	1

Table A.7: Overall performance from the selected model - Audacity project

API-domain	TN	FP	FN	TP	Precision	Recall
Util	38	3	2	13	0.81	0.87
APM	50	1	2	3	0.75	0.60
Network	54	0	0	2	1.00	1.00
DB	49	1	2	4	0.80	0.67
Error Handling	37	3	2	14	0.82	0.88
Logging	52	0	1	3	1.00	0.75
Thread	46	1	0	9	0.90	1.00
Lang	54	0	0	2	1.00	1.00
Data Structure	15	8	2	31	0.79	0.94
i18n	48	2	0	6	0.75	1.00
Setup	13	5	2	36	0.88	0.95
Logic	3	0	0	53	1.00	1.00
IO	10	3	6	37	0.93	0.86
UI	6	2	0	48	0.96	1.00
Parser	51	1	0	4	0.80	1.00
Event Handling	28	6	1	21	0.78	0.95
App	29	4	4	19	0.83	0.83
GIS	50	1	2	3	0.75	0.60
Multimedia	15	4	5	32	0.89	0.86
CG	50	0	1	5	1.00	0.83

Table A.8: Overall performance from the selected model - MTT project

API-domain	TN	FP	FN	TP	Precision	Recall
NLP	34	0	9	9	1.00	0.50
APM	9	0	0	43	1.00	1.00
DB	37	0	4	11	1.00	0.73
Lang	10	1	2	39	0.97	0.95
DevOps	0	5	0	47	0.90	1.00
Setup	18	6	8	20	0.77	0.71
IO	43	0	4	5	1.00	0.56
UI	0	2	0	50	0.96	1.00
Security	9	4	2	37	0.90	0.95

Table A.9: Confusion matrix and performance: Project JabRef - transfer learning.

API-domain	TN	FP	FN	TP	Precision	Recall
Network	65	7	43	3	0.30	0.07
DB	107	8	3	0	0	0
Interpreter	71	0	47	0	0	0
Logging	45	3	63	7	0.70	0.10
Data Structure	98	16	4	0	0	0
i18n	103	0	15	0	0	0
Setup	43	72	0	3	0.04	1
Microservices	77	24	12	5	0.17	0.29
Test	77	13	23	5	0.28	0.18
IO	33	0	67	18	1.00	0.21
UI	2	35	13	68	0.66	0.84
App	9	90	0	19	0.17	1

TABLE A.10: Confusion matrix and performance: Project Audacity - transfer learning.

API-domain	TN	FP	FN	TP	Precision	Recall
APM	127	1	9	0	0	0
Network	132	1	4	0	0	0
DB	117	0	20	0	0	0
Error Handling	105	0	32	0	0	0
Logging	88	41	7	1	0.02	0.13
Thread	122	0	15	0	0	0
Lang	133	0	4	0	0	0
Data Structure	37	0	100	0	0	0
i18n	118	0	19	0	0	0
Setup	28	1	94	14	0.93	0.13
Logic	5	10	60	62	0.86	0.51
IO	31	8	65	33	0.80	0.34
UI	0	18	9	110	0.86	0.92
Parser	126	1	10	0	0	0
Event Handling	68	0	69	0	0	0
App	58	33	12	34	0.51	0.74

TABLE A.11: Independent Variables - **Control - *Treatment

Name	Type	Description	Scale type
TF-IDF* **	Corpus	TF-IDF weights representing words in the corpus (title, body, comments on the issue)	number
Doc2Vec*	Corpus	Doc2Vec vectors representing documents words in the corpus (title, body, comments on the issue)	number
Issue comments*	Communication context	The size of the conversation in responses	number
Issue commenters*	Communication context	The size of the conversation in participants	number
Issue wordiness*	Communication context	The size of the conversation in words	number
Betweeness*	Developer's Role in Communication	The number of times a node lies on the shortest path between other nodes	number
Closeness*	Developer's Role in Communication	A measure of independence from potential control by intermediaries	number
Constraint*	Structural Hole of Communication	A measure of the extent to which a contact controls other individuals. This measure is based on the degree of unique connections. Low values indicate there are few alternatives to access a single neighbor	number
Hierarchy*	Structural Hole of Communication	A measure for the constraint to a single node. High values indicate that the restriction is concentrated on one single node neighbor	number
Effective Size*	Structural Hole of Communication	Measures the portion of a node's non-redundant neighbors. High values represent that many nodes among the node's neighbors are not redundant	number
Efficiency*	Structural Hole of Communication	Measure to normalize the effective size by the number of neighbors of the node. High values show that many neighbors are not redundant	number
Size*	Communication Network Properties	The size of the conversational network in participants	number
Edges*	Communication Network Properties	The size of the conversational network in answers	number
Diameter*	Communication Network Properties	The diameter of the conversational network. Calculated as the biggest shortest path in the network	number
Density*	Communication Network Properties	Determined by its ratio of links to nodes. The higher the ratio, the denser the network. Typically, the higher the density of a conversational network, the more powerful its network effects are	number

REFERENCES

[1] Ana Carolina Tomé Klock, Isabela Gasparini, and Marcelo Soares Pimenta. 5w2h framework: a guide to design, develop and evaluate the user-centered gamification. In *Proceedings of the 15th Brazilian Symposium on Human Factors in Computing Systems*, pages 1–10, 2016.

[2] Igor Steinmacher, Marco Aurelio Graciotto Silva, Marco Aurelio Gerosa, and David F Redmiles. A systematic literature review on the barriers faced by newcomers to open source software projects. *Information and Software Technology*, 59: 67–85, 2015.

[3] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiß, Rahul Premraj, and Thomas Zimmermann. Quality of bug reports in eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '07, pages 21–25, New York, NY, USA, 2007. ACM. ISBN 978-1-60558-015-9.

[4] Luis Vaz, Igor Steinmacher, and Sabrina Marczak. An empirical study on task documentation in software crowdsourcing on topcoder. In *2019 ACM/IEEE 14th International Conference on Global Software Engineering (ICGSE)*, pages 48–57. IEEE, 2019.

[5] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010.

[6] Igor Steinmacher, Christoph Treude, and Marco Aurelio Gerosa. Let me in: Guidelines for the successful onboarding of newcomers to open source projects. *IEEE Software*, 36(4):41–49, 2018.

[7] A. Barcomb, K. Stol, B. Fitzgerald, and D. Riehle. Managing episodic volunteers in free/libre/open source software communities. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[8] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement? a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, pages 304–318, 2008.

[9] Qiang Fan, Yue Yu, Gang Yin, Tao Wang, and Huaimin Wang. Where is the road for issue reports classification based on text mining? In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 121– –130. IEEE, 2017.

[10] Rafael Kallis, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella. Ticket tagger: Machine learning driven issue classification. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 406–409. IEEE, 2019.

[11] Ahmed Fawzi Otoom, Sara Al-jdaeh, and Maen Hammad. Automated classification of software bug reports. In *Proceedings of the 9th International Conference on Information Communication and Management*, pages 17–21, 2019.

[12] Nitish Pandey, Debarshi Kumar Sanyal, Abir Hudait, and Amitava Sen. Automated classification of software issue reports using machine learning techniques: an empirical study. *Innovations in Systems and Software Engineering*, 13(4):279–297, 2017.

[13] Natthakul Pingclasai, Hideaki Hata, and Ken-ichi Matsumoto. Classifying bug reports to bugs and other requests using topic modeling. In *2013 20Th asia-pacific software engineering conference (APSEC)*, volume 2, pages 13–18. IEEE, 2013.

[14] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Tag recommendation in software information sites. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 287–296. IEEE, 2013.

[15] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald Gall. Combining text mining and data mining for bug report classification. *Journal of Software: Evolution and Process*, 28(3):150–176, 2016.

[16] Yuxiang Zhu, Minxue Pan, Yu Pei, and Tian Zhang. A bug or a suggestion? an automatic way to label issues. *arXiv preprint arXiv:1909.00934*, 2019.

[17] Nicolas Ducheneaut. Socialization in an open source software community: A socio-technical analysis. *Computer Supported Cooperative Work (CSCW)*, 14(4):323–368, 2005.

[18] Johan Linåker, Hussan Munir, Krzysztof Wnuk, and Carl-Eric Mols. Motivating the contributions: An open innovation perspective on what to share as open source software. *Journal of Systems and Software*, 135:17–36, 2018.

[19] Igor Scaliante Wiese, Reginaldo Ré, Igor Steinmacher, Rodrigo Takashi Kuroda, Gustavo Ansaldi Oliva, Christoph Treude, and Marco Aurélio Gerosa. Using contextual information to predict co-changes. *Journal of Systems and Software*, 128: 220–235, 2017.

[20] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540, Leipzig, Germany, 2008. ACM.

[21] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23, Atlanta, Georgia, USA, 2008. ACM.

[22] Igor Wiese, Filipe Côgo, Reginaldo Ré, Igor Steinmacher, and Marco Gerosa. Social metrics included in prediction models on software engineering: a mapping study. In *International Conference on Predictive Models in Software Engineering*, 2014.

[23] Fabio Santos, Igor Wiese, Bianca Trinkenreich, Igor Steinmacher, Anita Sarma, and Marco A Gerosa. Can i solve it? identifying apis required to complete oss tasks. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 346–257. IEEE, 2021.

[24] Fabio Santos, Jacob Penney, João Felipe Pimentel, Igor Wiese, Bianca Trinkenreich, Igor Steinmacher, and Marco A Gerosa. Tell me who are you talking to and i will tell you what issues need your skills. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023.

[25] Fabio Santos, Joseph Vargovich, Bianca Penney, Trinkenreich, Italo Santos, Jacob, Ricardo Britto, João Felipe Pimentel, Igor Wiese, Igor Steinmacher, Anita Sarma, and Marco A Gerosa. Tag that issue: Applying api-domain labels in issue tracking systems. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023.

[26] Fabio Santos, Bianca Trinkenreich, João Felipe Pimentel, Igor Wiese, Igor Steinmacher, Anita Sarma, and Marco A Gerosa. How to choose a task? mismatches in perspectives of newcomers and existing contributors. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 114–124, 2022.

[27] Joseph Vargovich, Fabio Santos, Jacob Penney, Bianca Trinkenreich, Igor Steinmacher, and Marco A Gerosa. Givemelabeledissues: An open source issue recommendation system. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR - Data and Tool Showcase)*, 2023.

[28] Burrhus Frederic Skinner. *The behavior of organisms: An experimental analysis*. BF Skinner Foundation, 2019.

[29] Albert Bandura and Richard H Walters. Social learning and personality development. 1963.

[30] Jean Piaget. Le mécanisme du développement mental et les lois du groupement des opérations [the mechanism of mental development and the laws of grouping of operations]. *Archives de Psychologie*, 28:215–285, 1941.

[31] John H Flavell. Stage-related properties of cognitive development. *Cognitive Psychology*, 2(4):421–453, 1971.

[32] Harry Beilin. Developmental stages and developmental processes. *Measurement and Piaget*, pages 172–188, 1971.

[33] Kurt W Fischer. A theory of cognitive development: The control and construction of hierarchies of skills. *Psychological review*, 87(6):477, 1980.

[34] Kenneth Lovell. A follow-up study of inhelder and piaget's the growth of logical thinking. *British Journal of Psychology*, 52(2):143–153, 1961.

[35] Igor Steinmacher, Tayana Uchôa Conte, and Marco Aurélio Gerosa. Understanding and supporting the choice of an appropriate task to start with in open source software communities. In *2015 48th Hawaii International Conference on System Sciences*, pages 5299–5308. IEEE, 2015.

[36] Colin Potts and Lara D. Catledge. Collaborative conceptual design: A large software project case study. *Computer Supported Cooperative Work*, 5(4):415–445, 1996.

[37] Yutaka Yamauchi, Makoto Yokozawa, Takeshi Shinohara, and Toru Ishida. Collaboration with lean media: how open-source software succeeds. In *Proceedings of*

the 2000 ACM conference on Computer supported cooperative work, pages 329–338, 2000.

[38] Karl Fogel and M Bar. Open source development with cvs: Learn how to work with open source software. *The Coriolis Group*, 1999.

[39] Wei Zhang and John Storck. Peripheral members in online communities. 2001.

[40] Oded Maimon and Lior Rokach. Data mining and knowledge discovery handbook. 2005.

[41] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[42] G. Uddin and F. Khomh. Automatic mining of opinions expressed about apis in stack overflow. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[43] D. Hou and L. Mo. Content categorization of API discussions. In *2013 IEEE International Conference on Software Maintenance*, pages 60–69, 2013.

[44] G. Petrosyan, M. P. Robillard, and R. De Mori. Discovering information explaining API types using text classification. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 869–879, 2015.

[45] C. Treude and M. P. Robillard. Augmenting API documentation with insights from stack overflow. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 392–403, 2016.

[46] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall. Analyzing apis documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 27–37, 2017.

[47] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. Api method recommendation without worrying about the task-api knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 293–304, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375.

[48] H. Zhong and H. Mei. An empirical study on API usages. *IEEE Transactions on Software Engineering*, 45(4):319–334, 2019.

[49] Y. Zhou, C. Wang, X. Yan, T. Chen, S. Panichella, and H. C. Gall. Automatic detection and repair recommendation of directive defects in Java API documentation. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[50] S. Wang, N. Phan, Y. Wang, and Y. Zhao. Extracting API tips from developer question and answer websites. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 321–332, 2019.

[51] Tapajit Dey, Andrey Karnauch, and Audris Mockus. Representation of developer expertise in open source software. *arXiv preprint arXiv:2005.10176*, 2020.

[52] Jianguo Wang and Anita Sarma. Which bug should i fix: helping new developers onboard a new project. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, pages 76–79, New York, NY, USA, 2011. ACM.

[53] Yunrim Park and Carlos Jensen. Beyond pretty pictures: Examining the benefits of code visualization for open source newcomers. In *Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, VISSOFT '09, pages 3–10. IEEE, September 2009.

[54] Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. Categorizing the content of github readme files. *Empirical Software Engineering*, 24(3):1296–1327, 2019.

[55] Omar Elazhary, Margaret-Anne Storey, Neil Ernst, and Andy Zaidman. Do as i do, not as i say: Do contribution guidelines match the github contribution process? In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 286–290. IEEE, 2019.

[56] Ann Barcomb, Klaas-Jan Stol, Brian Fitzgerald, and Dirk Riehle. Managing episodic volunteers in free/libre/open source software communities. *IEEE Transactions on Software Engineering*, 2020.

[57] Jianguo Wang and Anita Sarma. Which bug should i fix: helping new developers onboard a new project. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 76–79. ACM, 2011.

[58] Yunrim Park and Carlos Jensen. Beyond pretty pictures: Examining the benefits of code visualization for open source newcomers. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 3–10. IEEE, 2009.

[59] Ann Barcomb, Klaas-Jan Stol, Dirk Riehle, and Brian Fitzgerald. Why do episodic volunteers stay in floss communities? In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 948–959. IEEE, 2019.

[60] Gillian J Greene and Bernd Fischer. Cvexplorer: Identifying candidate developers by mining and exploring their open source contributions. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 804–809, 2016.

[61] Weizhi Huang, Wenkai Mo, Beijun Shen, Yu Yang, and Ning Li. Cpdscorer: Modeling and evaluating developer programming ability across software communities. In *SEKE*, pages 87–92, 2016.

[62] Gunnar R Bergersen, Dag IK Sjøberg, and Tore Dybå. Construction and validation of an instrument for measuring programming skill. *IEEE Transactions on Software Engineering*, 40(12):1163–1184, 2014.

[63] Joao Eduardo Montandon, Luciana L. Silva, and Marco Tulio Valente. Identifying experts in software libraries and frameworks among GitHub users. In *16th International Conference on Mining Software Repositories (MSR)*, pages 276–287, 2019.

[64] Jose Ricardo da Silva, Esteban Clua, Leonardo Murta, and Anita Sarma. Niche vs. breadth: Calculating expertise over time through a fine-grained analysis. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 409–418. IEEE, 2015.

[65] John Anvik and Gail C Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):1–35, 2011.

[66] C. d. S. Costa, J. J. Figueiredo, J. F. Pimentel, A. Sarma, and L. G. P. Murta. Recommending participants for collaborative merge sessions. *IEEE Transactions on Software Engineering*, pages 1–1, 2019. doi: 10.1109/TSE.2019.2917191.

[67] Joao Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. Identifying experts in software libraries and frameworks among github users. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 276–287. IEEE, 2019.

[68] Catarina Costa, Jair Figueirêdo, João Felipe Pimentel, Anita Sarma, and Leonardo Murta. Recommending participants for collaborative merge sessions. *IEEE Transactions on Software Engineering*, 47(6):1198–1210, 2019.

[69] Farida El Zanaty, Christophe Rezk, Sander Lijbrink, Willem van Bergen, Mark Côté, and Shane McIntosh. Automatic recovery of missing issue type labels. *IEEE Software*, 2020.

[70] Jordi Cabot, Javier Luis Cánovas Izquierdo, Valerio Cosentino, and Belén Rolandi. Exploring the use of labels to categorize issues in open-source software projects. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 550–554. IEEE, 2015.

[71] Joao P Diniz, Daniel Cruz, Fabio Ferreira, Cleiton Tavares, and Eduardo Figueiredo. Github label embeddings. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 249–253. IEEE, 2020.

[72] Nicolas Bettenburg and Ahmed E Hassan. Studying the impact of social interactions on software quality. *Empirical Software Engineering*, 18(2):375–431, 2013.

[73] Irwin Kwan, Marcelo Cataldo, and Daniela Damian. Conway's law revisited: The evidence for a task-based perspective. *IEEE software*, 29(1):90–93, 2011.

[74] Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. Using dynamic and contextual features to predict issue lifetime in github projects. In *2016 ieee/acm 13th working conference on mining software repositories (msr)*, pages 291–302, Austin, TX, USA, 2016. IEEE, ACM.

[75] Igor Steinmacher, Tayana Conte, Marco Aurélio Gerosa, and David Redmiles. Social barriers faced by newcomers placing their first contribution in open source software

projects. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, page 1379–1392, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450329224.

[76] Christoph Stanik, Lloyd Montgomery, Daniel Martens, Davide Fucci, and Walid Maalej. A simple nlp-based approach to support onboarding and retention in open source communities. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 172–182. IEEE, 2018.

[77] Yang Feng, James Jones, Zhenyu Chen, and Chunrong Fang. An empirical study on software failure classification with multi-label and problem-transformation techniques. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 320–330. IEEE, 2018.

[78] Thirupathi Guggulothu and Salman Abdul Moiz. Code smell detection using multi-label classification approach. *Software Quality Journal*, 28:1063–1086, 2020.

[79] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 133–142. Piscataway, NJ, 2003.

[80] Diksha Behl, Sahil Handa, and Anuja Arora. A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf. In *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*, pages 294–299. IEEE, 2014.

[81] Sri Lakshmi Vadlamani and Olga Baysal. Studying software developer expertise and contributions in Stack Overflow and GitHub. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 312–323. IEEE, 2020.

[82] Francisco Herrera, Francisco Charte, Antonio J. Rivera, and Mara J. del Jesus. *Multilabel Classification: Problem Analysis, Metrics and Techniques*. Springer Publishing Company, Incorporated, 1st edition, 2016. ISBN 3319411101.

[83] Min-Ling Zhang and Zhi-Hua Zhou. Ml-knn: A lazy learning approach to multi-label learning. *Pattern recognition*, 40(7):2038–2048, 2007.

[84] Grigorios Tsoumakas, Ioannis Katakis, and Ioannis Vlahavas. Random k-labelsets for multilabel classification. *IEEE transactions on knowledge and data engineering*, 23(7):1079–1089, 2010.

[85] Emmanuel Gbenga Dada, Stephen Bassi Shafi'i Muhammad ABDULHAMID, and Madhu Puvvula. A comparative study between naïve bayes and neural network (mlp) classifier for spam email detection. 2014.

[86] Stefanie Beyer, Christian Macho, Massimiliano Di Penta, and Martin Pinzger. What kind of questions do developers ask on stack overflow? a comparison of automated approaches to classify posts into question categories. *Empirical Software Engineering*, 25:2258–2301, 2020.

[87] Takaya Saito and Marc Rehmsmeier. The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets. *PloS one*, 10(3):e0118432, 2015.

[88] Peter A Flach and Meelis Kull. Precision-recall-gain curves: Pr analysis done right. In *NIPS*, volume 15, 2015.

[89] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 45(7):683–711, 2019.

[90] Dragutin Petkovic, Marc Sosnick-Pérez, Kazunori Okada, Rainer Todtenhoefer, Shihong Huang, Nidhi Miglani, and Arthur Vigil. Using the random forest classifier

to assess and predict student learning of software engineering teamwork. In *2016 IEEE Frontiers in Education Conference (FIE)*, pages 1–7. IEEE, 2016.

[91] Eesha Goel, Er Abhilasha, E Goel, and E Abhilasha. Random forest: A review. *International Journal of Advanced Research in Computer Science and Software Engineering*, 7(1), 2017.

[92] TP Pushphavathi, V Suma, and V Ramaswamy. A novel method for software defect prediction: hybrid of fcm and random forest. In *2014 International Conference on Electronics and Communication Systems (ICECS)*, pages 1–5. IEEE, 2014.

[93] Shashank Mouli Satapathy, Barada Prasanna Acharya, and Santanu Kumar Rath. Early stage software effort estimation using random forest technique based on use case points. *IET Software*, 10(1):10–17, 2016.

[94] Rakesh Agrawal, Tomasz Imieliundefinedski, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, June 1993. ISSN 0163-5808.

[95] Stéphane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4): 573–591, 2009.

[96] Anthony Savidis and Crystallia Savaki. Software architecture mining from source code with dependency graph clustering and visualization. In *IVAPP*, 12 2021. doi: 10.5220/0010896800003124.

[97] Sudharsan Ravichandiran. *Getting Started with Google BERT: Build and train state-of-the-art natural language processing models using BERT*. Packt Publishing Ltd, 2021.

[98] Maliheh Izadi, Abbas Heydarnoori, and Georgios Gousios. Topic recommendation for software repositories using multi-label classification algorithms. *Empirical Software Engineering*, 26, 09 2021. doi: 10.1007/s10664-021-09976-2.

[99] Alberto Blanco, Arantza Casillas, Alicia Pérez, and Arantza Diaz de Ilarraza. Multilabel clinical document classification: Impact of label-density. *Expert Systems with Applications*, 138:112835, 2019.

[100] Francisco Charte, Antonio J Rivera, María J del Jesus, and Francisco Herrera. Mlsmote: approaching imbalanced multilabel learning through synthetic instance generation. *Knowledge-Based Systems*, 89:385–397, 2015.

[101] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=Syx4wnEtvH`.

[102] J. Romano, J.D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In *annual meeting of the Florida Association of Institutional Research*, pages 1–3, 2006.

[103] Anselm Strauss and Juliet Corbin. *Basics of qualitative research techniques*. Thousand oaks, CA: Sage publications, 1998.

[104] Taiichi Ohno. How the toyota production system was created. *Japanese Economic Studies*, 10(4):83–101, 1982.

[105] Hana Pacaiova. Analysis and identification of nonconforming products by 5w2h method. *Center for Quality*, 2015.

[106] Maarten Van Gompel and Antal Van Den Bosch. Efficient n-gram, skipgram and flexgram modelling with colibri core. *Journal of Open Research Software*, 4(1), 2016.

[107] Fabio Santos, Bianca Trinkenreich, João Felipe Nicolati Pimentel, Igor Wiese, Igor Steinmacher, Anita Sarma, and Marco Gerosa. How to choose a task? mismatches in perspectives of newcomers and existing contributors. *Empirical Software Engineering and Measurement*, 2022.

[108] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, 1993.

[109] Maliheh Izadi, Siavash Ganji, and Abbas Heydarnoori. Topic recommendation for software repositories using multi-label classification algorithms. *Empir. Softw. Eng.*, 26:93, 2021.

[110] Jun Wang, Xiaofang Zhang, and Lin Chen. How well do pre-trained contextual language representations recommend labels for GitHub issues? *Knowledge-Based Systems*, 232:107476, 2021. ISSN 0950-7051. doi: https://doi.org/10.1016/j.knosys.2021.107476. URL `https://www.sciencedirect.com/science/article/pii/S0950705121007383`.

[111] Dong Qiu, Bixin Li, and Hareton Leung. Understanding the API usage in Java. *Information and software technology*, 73:81–100, 2016.

[112] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *2013 35th international conference on software engineering (ICSE)*, pages 382–391. IEEE, 2013.

[113] Chun-Wei Seah, Ivor W Tsang, and Yew-Soon Ong. Transfer ordinal label learning. *IEEE transactions on neural networks and learning systems*, 24(11):1863–1876, 2013.

[114] Igor Steinmacher, Igor Wiese, Ana Paula Chaves, and Marco Aurélio Gerosa. Why do newcomers abandon open source software projects? In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 25–32. IEEE, 2013.

[115] Yuekai Huang, Junjie Wang, Song Wang, Zhe Liu, Dandan Wang, and Qing Wang. Characterizing and predicting good first issues. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, Bari, Italy, 2021. ACM.

[116] Xin Tan, Minghui Zhou, and Zeyu Sun. A first look at good first issues on github. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 398–409, Virtual Event, USA, 2020. ACM.

[117] Sogol Balali, Umayal Annamalai, Hema Susmita Padala, Bianca Trinkenreich, Marco A Gerosa, Igor Steinmacher, and Anita Sarma. Recommending tasks to newcomers in oss projects: How do mentors handle it? In *Proceedings of the 16th International Symposium on Open Collaboration*, pages 1–14, 2020.

[118] Anita Sarma, Marco Aurélio Gerosa, Igor Steinmacher, and Rafael Leano. Training the future workforce through task curation in an OSS ecosystem. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 932–935, 2016.

[119] H Russell Bernard. *Research methods in anthropology: Qualitative and quantitative approaches*. Rowman & Littlefield, 2017.

[120] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, 25(4):557–572, 1999.

[121] Anselm Strauss and Juliet M. Corbin. *Basics of Qualitative Research : Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 3rd edition, 2007. ISBN 0803959400.

[122] D Garrison, Martha Cleveland-Innes, Marguerite Koole, and James Kappelman. Revisiting methodological issues in transcript analysis: Negotiated coding and reliability. *The Internet and Higher Education*, 9(1):1–8, 2006.

[123] Claes Wohlin and Aybüke Aurum. Towards a decision-making structure for selecting a research design in empirical software engineering. *Empirical Software Engineering*, 20(6):1427–1455, 2015.

[124] Markus Schulze. A new monotonic and clone-independent single-winner election method. *Voting matters*, 17(1):9–19, 2003.

[125] Markus Schulze. A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method. *Social choice and Welfare*, 36 (2):267–303, 2011.

[126] CRAN. Cran repository policy, 2018. URL `https://cran.r-project.org/web/packages/votesys/index.html`.

[127] Marissa L Shuffler and Matthew A Cronin. The challenges of working with "real" teams: Challenges, needs, and opportunities, 2019.

[128] Igor Steinmacher, Sogol Balali, Bianca Trinkenreich, Mariam Guizani, Daniel Izquierdo-Cortazar, Griselda G Cuevas Zambrano, Marco Aurelio Gerosa, and Anita Sarma. Being a mentor in open source projects. *Journal of Internet Services and Applications*, 12(1):1–33, 2021.

[129] Marco Gerosa, Igor Wiese, Bianca Trinkenreich, Georg Link, Gregorio Robles, Christoph Treude, Igor Steinmacher, and Anita Sarma. The shifting sands of motivation: Revisiting what drives contributors in open source. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1046–1058. IEEE, 2021.

[130] Minghui Zhou, Qingying Chen, Audris Mockus, and Fengguang Wu. On the scalability of linux kernel maintainers' work. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 27–37, 2017.

[131] Matthijs den Besten, Jean-Michel Dalle, and Fabrice Galia. The allocation of collaborative efforts in open-source software. *Information Economics and Policy*, 20 (4):316–322, 2008.

[132] Pranav Gupta and Anita Williams Woolley. Productivity in an era of multi-teaming: The role of information dashboards and shared cognition in team performance. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–18, 2018.

[133] Mariam Guizani, Thomas Zimmermann, Anita Sarma, and Denae Ford. Attracting and retaining oss contributors with a maintainer dashboard. *CoRR*, abs/2202.07740, 2022.

[134] Barry Schwartz. *The paradox of choice: Why more is less*. HarperPerennial, New York, NY, 2004.

[135] Gail Steptoe-Warren, Douglas Howat, and Ian Hume. Strategic thinking and decision making: literature review. *Journal of Strategy and Management*, 4(3):238–250, 2011.

[136] Stefan Meyer, Philip Healy, Theo Lynn, and John Morrison. Quality assurance for open source software configuration management. In *2013 15th International*

Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pages 454–461, Timisoara, Romania, 2013. IEEE, IEEE Computer Society.

[137] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6), 2011.

[138] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and software technology*, 55(8):1397–1418, 2013.

[139] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *2009 20th International Symposium on Software Reliability Engineering*, pages 109–119, Mysuru, Karnataka, India, 2009. IEEE.

[140] Igor Scaliante Wiese, Rodrigo Takashi Kuroda, Douglas Nassif Roma Junior, Reginaldo Ré, Gustavo Ansaldi Oliva, and Marco Aurelio Gerosa. Using structural holes metrics from communication networks to predict change dependencies. In *CYTED-RITOS International Workshop on Groupware*, pages 294–310, Santiago, Chile, 2014. Springer, Springer.

[141] Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[142] Linton C Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1978.

[143] Serdar Biçer, Ayşe Başar Bener, and Bora Çağlayan. Defect prediction using social network analysis on issue repositories. In *Proceedings of the 2011 International Conference on Software and Systems Process*, ICSSP '11, page 63–71, New York,

NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307307. doi: 10.1145/1987875.1987888. URL https://doi.org/10.1145/1987875.1987888.

[144] Frank Harary. *Graph Theory*. Westview Press, Boulder, CO, USA, 1994. ISBN 9780201410334. URL https://openlibrary.org/books/OL7407411M/Graph_Theory.

[145] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice Hall, Upper Saddle River, NJ, USA, 2001.

[146] R Diestel. *Graph Theory*. Springer, 3rd edition, 2005. ISBN 9783540261834.

[147] JabRef. JabRef project, 2019. URL https://jabref.org/.

[148] Mitch Rees-Jones, Matthew Martin, and Tim Menzies. Better predictors for issue lifetime. *arXiv preprint arXiv:1702.07735*, 1702(07735), 2017.

[149] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, 74:204–218, 2016.

[150] Donghwa Kim, Deokseong Seo, Suhyoun Cho, and Pilsung Kang. Multi-co-training for document classification using various document representations: Tf–idf, lda, and doc2vec. *Information Sciences*, 477:15–29, 2019.

[151] Metin Bilgin and İzzet Fatih Şentürk. Sentiment analysis on twitter data with semi-supervised doc2vec. In *2017 international conference on computer science and engineering (UBMK)*, pages 661–666. Ieee, 2017.

[152] Denis Eka Cahyani and Irene Patasik. Performance comparison of tf-idf and word2vec models for emotion text classification. *Bulletin of Electrical Engineering and Informatics*, 10(5):2780–2788, 2021.

[153] Grigorios Tsoumakas, Ioannis Katakis, and Ioannis Vlahavas. Mining multi-label data. *Data mining and knowledge discovery handbook*, pages 667–685, 2009.

[154] Nancy H Leonard, Richard W Scholl, and Kellyann Berube Kowalski. Information processing style and decision making. *Journal of Organizational Behavior: The International Journal of Industrial, Occupational and Organizational Psychology and Behavior*, 20(3):407–420, 1999.

[155] Maliheh Izadi, Kiana Akbari, and Abbas Heydarnoori. Predicting the objective and priority of issue reports in software repositories. *Empirical Software Engineering*, 27(2):1–37, 2022.

[156] Jun Wang, Xiaofang Zhang, and Lin Chen. How well do pre-trained contextual language representations recommend labels for GitHub issues? *Knowledge-Based Systems*, 232:107476, 2021.

[157] M. Ferreira Moreno, W. H. Sousa Dos Santos, R. Costa Mesquita Santos, and R. Fontoura De Gusmao Cerqueira. Supporting knowledge creation through has: The hyperknowledge annotation system. In *2018 IEEE International Symposium on Multimedia (ISM)*, pages 239–246, 2018. doi: 10.1109/ISM.2018.00034.

[158] Yangyang Lu, Ge Li, Zelong Zhao, Linfeng Wen, and Zhi Jin. Learning to infer API mappings from API documents. In *International Conference on Knowledge Science, Engineering and Management*, pages 237–248. Springer, 2017.

[159] F. Sharan. Asf committer diversity survey. accessed: 2020-10-16. `https://cwiki.apache.org/confluence/display/COMDEV/ASF+Committer+Diversity+Survey+-+2016`, 2016.

[160] Bitergia. Gender-diversity analysis of the linux kernel technical contributions. accessed: 2020-10-16. `https://blog.bitergia.com/2016/10/11/`

gender-diversity-analysis-of-the-linux-kernel-technical-contributions, 2016.

[161] Bianca Trinkenreich, Igor Wiese, Anita Sarma, Marco Gerosa, and Igor Steinmacher. Women's participation in open source software: A survey of the literature. *arXiv preprint arXiv:2105.08777*, 2021.

[162] Sharan B Merriam and Elizabeth J Tisdell. *Qualitative research: A guide to design and implementation.* John Wiley & Sons, 2015.

[163] Jan Willem David Alderliesten and Andy Zaidman. An initial exploration of the "good first issue" label for newcomer developers. In *2021 IEEE/ACM 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 117–118. IEEE, 2021.

[164] Hyuga Horiguchi, Itsuki Omori, and Masao Ohira. Onboarding to open source projects with good first issues: A preliminary analysis. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 501–505. IEEE, 2021.

[165] Ifraz Rehman, Dong Wang, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Newcomer oss-candidates: Characterizing contributions of novice developers to github. *Empirical Software Engineering*, 27(5):109, 2022.

[166] João Eduardo Montandon, Marco Tulio Valente, and Luciana L Silva. Mining the technical roles of github users. *Information and Software Technology*, 131:106485, 2021.

[167] Jenny T Liang, Thomas Zimmermann, and Denae Ford. Towards mining oss skills from github activity. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 106–110, 2022.

[168] Fabio Santos and Carlos E Mello. Matching network of ontologies: a random walk and frequent itemsets approach. *IEEE Access*, 10:44638–44659, 2022.

[169] Fotis Draganidis and Gregoris Mentzas. Competency based management: a review of systems and approaches. *Information management & computer security*, 2006.

[170] Fotis Draganidis, Paraskevi Chamopoulou, and Gregoris Mentzas. A semantic web architecture for integrating competence management and learning paths. *Journal of knowledge Management*, 2008.

[171] Ana Gjorgjevik, Riste Stojanov, and Dimitar Trajanov. Semccm: course and competence management in learning management systems using semantic web technologies. In *Proceedings of the 10th International Conference on Semantic Systems*, pages 140–147, 2014.

[172] Sergio Miranda, Francesco Orciuoli, Vincenzo Loia, and Demetrios Sampson. An ontology-based model for competence management. *Data & Knowledge Engineering*, 107:51–66, 2017.

[173] Gilbert Paquette. An ontology and a software framework for competency modeling and management. *Journal of Educational Technology & Society*, 10(3):1–21, 2007.

[174] Kalthoum Rezgui and Hédia Mhiri. Modeling competencies in competency-based learning: Classification and cartography. In *2018 JCCO Joint International Conference on ICT in Education and Training, International Conference on Computing in Arabic, and International Conference on Geocomputing (JCCO: TICET-ICCA-GECO)*, pages 1–8. IEEE, 2018.

[175] Uldis Zandbergs, Jānis Grundspeņķis, Jānis Judrups, and Signe Briķe. Development of ontology based competence management model for non-formal education services. *Applied Computer Systems*, 24(2):111–118, 2019.

[176] Marie-Hélène Abel. Competencies management and learning organizational memory. *Journal of knowledge management*, 2008.

[177] Giovanni Acampora, Matteo Gaeta, Francesco Orciuoli, and Pierluigi Ritrovato. Exploiting semantic and social technologies for competency management. In *2010 10th IEEE International Conference on Advanced Learning Technologies*, pages 297–301. IEEE, 2010.

[178] Andreas Schmidt and Christine Kunzmann. Towards a human resource development ontology for combining competence management and technology-enhanced workplace learning. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 1078–1087. Springer, 2006.

[179] Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez, and Mariano Fernández-López. The neon methodology for ontology engineering. In *Ontology engineering in a networked world*, pages 9–34. Springer, 2012.

[180] Rodrigo F Calhau, Carlos LB Azevedo, and João Paulo A Almeida. Towards ontology-based competence modeling in enterprise architecture. In *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 71–81. IEEE, 2021.

[181] JPA Almeida, G Guizzardi, RA Falbo, and Tiago Prince Sales. gufo: a lightweight implementation of the unified foundational ontology (ufo). *URL http://purl.org/nemo/doc/gufo*, 2019.

[182] Jérôme Euzenat. An api for ontology alignment. In *International Semantic Web Conference*, pages 698–712. Springer, 2004.

[183] Jean-Baptiste Lamy. Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. *Artificial intelligence in medicine*, 80:11–28, 2017.

[184] E Jimenez-Ruiz. Logmap family participation in the oaei 2021. In *CEUR Workshop Proceedings*, volume 3063, pages 175–177, 2021.

[185] Shiyi Zou, Jiajun Liu, Zherui Yang, Yunyan Hu, and Peng Wang. Lily results for oaei 2021. 2021.

[186] Daniel Faria, Catia Pesquita, Teemu Tervo, Francisco M Couto, and Isabel F Cruz. Aml and amlc results for oaei 2021. *OM@ ISWC*, 2019, 2020.

[187] Nikooie Pour, Alsayed Algergawy, Reihaneh Amini, Daniel Faria, Irini Fundulaki, Ian Harrow, Sven Hertling, Ernesto Jiménez-Ruiz, Clement Jonquet, Naouel Karam, et al. Results of the ontology alignment evaluation initiative 2020. In *Proceedings of the 15th International Workshop on Ontology Matching (OM 2020)*, volume 2788, pages 92–138. CEUR-WS, 2020.