Can I Solve It? Identifying APIs Required to Complete OSS Tasks

Fabio Santos,[1] Igor Wiese,[2] Bianca Trinkenreich,[1] Igor Steinmacher,[1,2] Anita Sarma,[3] Marco A. Gerosa[1]
[1]*Northern Arizona University, USA*, [2]*Universidade Tecnológica Federal do Paraná, Brazil*,[3] *Oregon State University, USA*

Fabio-Marcos.De-Abreu-Santos@nau.edu, igor@utfpr.edu.br, bt473@nau.edu, igorfs@utfpr.edu.br
anita.sarma@oregonstate.edu, marco.gerosa@nau.edu

***Abstract*—Open Source Software projects often add labels to their open issues to help contributors in choosing tasks. However, manually labeling issues is time-consuming. Further many current automatic approaches for creating labels are limited to classifying issues as bug/non-bug. Little is known about the feasibility and relevancy of predicting labels based on API categories. APIs have already been explored to provide recommendations, offer tips, or define developers' skills. In this paper, we leverage the issues' description and the project history to build prediction models. We found that API-category labels can be predicted with precision up to 82% and recall up to 97.8%. We also ran an experiment (n=74) to assess these labels' relevancy for potential contributors. The results show that the labels are often used to choose the tasks, and the API-category labels were selected more often than the existing architecture-based labels. Our results may inspire the creation of tools to automatically label issues, helping developers to find tasks that better match their skills.**

***Index Terms*—API identification, Tagging, Labelling, Onboarding, Expertise, Skills, Multi-Label Classification, Mining Software Repositories, Case Study**

I. Introduction

Finding tasks to contribute in Open Source projects is challenging [1, 2, 3, 4, 5]. Open tasks vary in complexity and required skills, which can be hard for an outsider to determine when reading the task descriptions only [6, 7, 8]. Adding labels to the issues (a.k.a tasks, bug reports) may support newcomers when they are choosing their tasks [9]. However, community managers argue that labeling issues is challenging and time-consuming [10] because projects require skills in different languages, frameworks, databases, and Application Programming Interfaces (APIs).

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

Given this context, in this paper, we investigate the feasibility of automatically labeling issues with categories of APIs to facilitate choosing tasks. Since an issue may require knowledge in multiple APIs, we applied a multi-label classification approach, which has been used in software engineering to classify questions in Stack Overflow Xia et al. [11] and detect types of failures Feng et al. [12] and code smells Guggulothu and Moiz [13].

By employing an exploratory case study and an experiment, we aimed to answer the following research questions:

RQ1: To what extent can we predict the categories of the APIs present in the implementation that solves an issue? To answer RQ1, we employed a multi-label classification approach to predict the API-categories. 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 Random Forest algorithm can predict the API-category labels with up to 82% precision and up to 97.8% of recall. This configuration outperformed recent approaches reported in the literature [14].

RQ2. How relevant are the API-category labels to outside 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 they would like to contribute, we presented a follow-up survey to determine what information was relevant to make the decision. We compared the group with access to the API-category labels with a control group (which used only the pre-existing labels). The participants considered API-category labels more relevant than the name of components originally labeled by the project—with a large effect size.

These results indicate that labeling issues with API-categories is feasible and relevant for new contributors who need to determine which issues to contribute.

II. Related Work

Newcomers need specific guidance on what to contribute [9, 15]. In particular, finding an appropriate issue can be a daunting task, which can discourage contributors [2]. Social coding platforms like GitHub[1] encourages projects to label issues that are easy for newcomers, what is already done by several communities (e.g LibreOffice,[2], KDE [3], and Mozilla [4]). However, community managers argue that manually labeling issues is difficult and time-consuming [10].

[1]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

Several studies have proposed ways to automatically label issues as bug/non-bug, combining text mining techniques with classification to mitigate this problem. For example, Antoniol et al. [16] compared text-based mining with Naive Bayes (NB), Logistic Regression (LR), and Decision Trees (DT) to process data from titles, descriptions, and discussions and achieved a recall up to 82%. Pingclasai et al. [17] used the same techniques to compare a topic and word-based approach and found F-measures from 0.68 to 0.81 using the topic-based approach. More recently, Zhou et al. [18] used two-stage processing, introducing the use of structured information from the issue tracker, improving the recall obtained by Antoniol et al. [16]. Kallis et al. [19] simplified the data mining step to produce a tool able to classify issues on demand. They used the title and body to create a bag of words used to classify issues as “bug report”, “enhancement”, and “question”. El Zanaty et al. [14] applied type detection on issues and tried 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. [11] employed a multi-label classification using text data from Stack Overflow questions, obtaining recall from 0.59 to 0.77. As opposed to these related work—which focuses mostly on classifying the type of issue (e.g., bug/non-bug)—our work focuses on identifying the categories of APIs used in the implementation code, which might reflect skills needed to complete a task.

Regarding APIs, recent work focuses on understanding the crowd’s opinion about API usage [20], understanding and categorizing the API discussion [21], creating tutorial sections that explain a given API type [22], generating/understanding API documentation [23, 24], providing API recommendations [25, 26, 27], offering API tips to developers [28], and defining the skill space for APIs, developers, and projects [29]. In contrast to these previous work, we focus on predicting the category of the API that may be needed to solve an issue.

III. Method

This study comprises three phases, as summarized in Fig. 1: mining software repository, build the classifiers, and evaluating the API-category labels with developers. To foster reproducibility, we provide a publicly available dataset[5] containing the raw data, the Jupyter notebook scripts that build and test the models, and the survey data.

We conducted an exploratory case study [30] using the JabRef project as our case. We chose JabRef [31] because we have access to the project’s contributors and because of its characteristics. JabRef is an open-source bibliography reference manager developed by a community of volunteers, including contributors with different background and diverse set of skills (with and without computer science background)—this helped us evaluate the approach with a diverse set of contributor profiles. JabRef is a mature and active project created in 2003 (migrated to GitHub in 2014), with 15.7k commits, 42 releases, 337 contributors, 2.7k closed issues, and 4.1k closed pull requests. JabRef is frequently investigated in scientific studies [32, 33, 34, 35, 36].

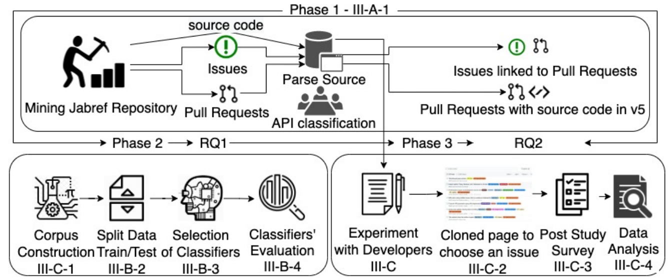

Fig. 1. Research Design

A. *Phase 1 - Mining JabRef Repository*

We used the GitHub API to collect data from JabRef issues and pull requests (PR), including title, description (body), comments, and submission date. We also collected the name of the files changed in the PR and the commit message associated with each commit. The data was collected in April 2020.

After collecting the data, we filtered out open issues and pull requests not explicitly linked to issues. To find the links between pull requests and issues, we searched for symbol #issue_number in the pull request title and body and checked the URL associated with each link. We manually inspected a random sample of issues to check whether the data was correctly collected and reflected what was shown on the GitHub interface. Two authors manually examined 50 issues, comparing the collected date with the GitHub interface. All records were consistent. We also filtered out issues linked to pull requests without at least one Java file (e.g., those associated only with documentation files). Our final dataset comprises 705 issues and their corresponding pull requests.

In parallel, we parsed all Java files from the project and extracted 796 APIs. In total, 1,692 import statements from 1,472 java sources had been mapped to 796 distinct APIs. The parser identified all classes, including the complete namespace from each import statement. Then, using the source code changed in the pull requests identified in the previous step, we filtered out APIs not found in the latest version of the source code (JabRef 5.0) to avoid recommending APIs that were no longer used in the project.

Then, we employed a card-sorting approach to manually classify the imported APIs into higher level categories based on their purpose. For instance, we classified “java.nio.x” as “IO’, “java.sql.x” as “Database”, and “java.util.x” as “Utils”’. To perform this classification, we counted with the input of a contributor of JabRef and of two authors (one of the authors is an expert Java developer). They analyzed the APIs identified in the previous step and each person individually classified the API and discussed to reach a consensus. After classifying all the APIs, the researchers conducted a second round to revise the classification (8 hours). During this step, we excluded some labels and aggregated some others. The final set of categories of APIs contains: Google Commons, Test, OS,

[5]https://doi.org/10.5281/zenodo.4439808

IO, UI, Network, Security, OpenOffice Documents, Database, Utils, PDF, Logging, and Latex. We used these labels, assigned to the 705 issues previously collected based on the presence of the corresponding APIs in the changed files, as our training and test sets for the multi-label classification models.

B. Phase 2 - Building the Multi-label Classifiers

1) 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 using the Python nltk package. We also filtered issue and pull request templates[6] since templates were not consistently used among all the issues and we found in our exploratory studies that their repetitive structure introduced too much noise.

Next, similar to other studies [37, 38, 39], 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 for each issue. The vector length is the number of terms used to calculate the TF-IDF plus the 13 labels in the dataset. Each label received a binary value (0 or 1), indicating whether the corresponding API category is present in the issue and each term received the TF-IDF score.

We evaluated different configurations for the number of terms used to compute the TF-IDF features. The number of tokens limited the "number of max features" by selecting a slice with the most frequent words of the corpus. We started with all 5,728 words of the corpus, then we sucessively divided the number of terms in the corpus by 2 until reaching a configuration with less than 100 words. This resulted in 7 different models (with the 5,728, 2,864, 1,432, 716, 358, 179, 90 most frequent words). The goal of using different number of terms was to test if the corpus reduction would lead to better results compared to a baseline configuration—which has 899 words extracted from the title.

2) Training/Test Sets: We split the data into training and test sets using the ShuffleSplit method [40], which is a model selection technique that emulates cross-validation for multi-label classifiers. We randomly split our 705 issues into a training set with 80% (564) of the issues and a test set with the remaining 20% (142 issues). To avoid overfitting, we ran each experiment ten times, using ten different training and test sets to match a 10-fold cross validation. To improve the balance of the data set, we ran the SMOTE algorithm for multi-label approach [41].

3) 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), MPLC Classifier (neural network multilayer perceptron), MLkNN (multi-label lazy learning approach based on the traditional K-nearest neighbor algorithm) [40, 42], and Logistic Regression. We ran the classifiers using the Python sklearn package and tested several parameters. For the RandomForestClassifier, the best classifier (see Section IV), we kept the following parameters: $criterion =' entropy'$,$max\_depth = 50$, $min\_samples\_leaf = 1$, $min\_samples\_split = 3$,$n\_estimators = 50$.

4) Classifiers Evaluation: To evaluate the classifiers, we employed the following metrics (also calculated using the scikit-learn package):

- **Hamming loss** measures the fraction of the wrong labels to the total number of labels.
- **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.
- **Matthews correlation coefficient - MCC** calculates the Pearson product-moment correlation coefficient between actual and predicted values. It is an alternative measure unaffected by the unbalanced dataset issue [43].

5) Data Analysis: To conduct the data analysis, we used the aforementioned evaluation metrics and the confusion matrix logged after each model's execution. 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 in [44], i.e. $|d|<0.147$ "negligible", $|d|<0.33$ "small", $|d|<0.474$ "medium", otherwise "large".

6) Dataset Analysis: Multi-label datasets are usually described by label cardinality and label density [40]. 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 3.04. The density is 0.25. These values consider the 705 distinct issues and API-category 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 [45].

For the remainder of our analysis, we removed the API label "Utils" since we found that this label was present in 96% of the issues in our final dataset and has an overly generic meaning. The API-category labels "IO", "UI", and "Logging" had 492, 452, and 417 occurrences respectively. These last three occurred in approximately 60% of the issues. We also observed that "Test", "Network", and "Google Commons" appeared in almost 29% of the issues (212, 208, and 206 times). "SO", "Database", "PDF", "Open Office", "Security", and "Latex" were less common, with 56, 31, 21, 21, 20, and 14 occurrences respectively.

Finally, we checked the distribution of the number of labels per issue (Fig. 2). We found 140 issues with five labels, 132

[6]https://help.GitHub.com/en/GitHub /building-a-strong-community/about-issue-and-pull-request-templates

issues with three labels, 121 issues with two labels, and 117 issues with four labels. Only 8.5% of issues have one label, which confirms we face a multi-label classification problem.

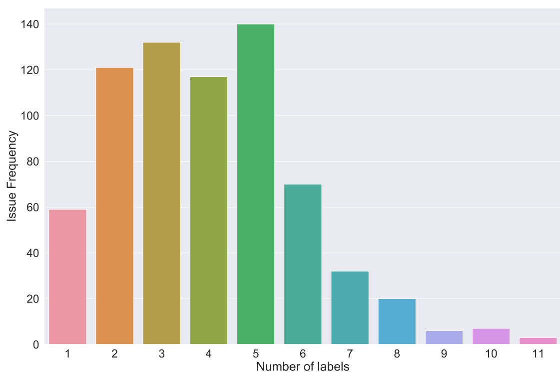

Fig. 2. Number of labels per issue

C. Phase 3 - Evaluating the API-Category Labels with Developers

To evaluate the relevancy of the API-category labels from an outsider's perspective, we conducted an experimental study 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 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) Participants: We recruited participants from both industry and academia. We reached out to our own 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-size IT startup hosted in Brazil and the IT department of a large and global company. Students included undergraduate and graduate computer science students from one university in the US and two others 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 III-C1. We offered an Amazon Gift card to incentive 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 "P", followed by a sequential number and a character representing the location where they were recruited (university: P,I,N & Industry: C,B), and "T" for Treatment and "C" for Control groups.

The participants were randomly split into two groups: Control and Treatment. From the 120 participants that started the

Population	Quantity	Percentage
Industry	41	55.5
Student	33	44.5

Tenure	Quantity	Percentage
Expert	19	25.7
Novice	55	74.3

survey, 74 (61.7%) finished all the steps and were considered in the analysis. We ended up with 33 and 41 participants in the Control and Treatment groups, respectively.

2) 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, trying to keep similar distributions to the number of API-category labels predicted per issue and the counts of predicted API-category labels (see Section III-B6). 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 those API-category labels. These pages are available in the replication package.

3) 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 deciding which issues to pick (Fig: 3).
- 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 skill level related to the technologies used in JabRef.

Fig. 3 shows an example of an issue details page and an issue entry on an issue list page. After selecting the issues to contribute, the participant was presented with this page to select what information (region) was relevant to the previous issue selection.

4) Survey Data Analysis: Next, to understand participants' perceptions about what information (regions) they considered important and the relevancy of the API-category 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, resulting in the subgroups presented in Table III-C1. We calculated the odds ratio to check how likely it would be to get similar responses from both groups. We used a 2x2 contingency table for each comparison—for instance, industry practitioners vs. students and experienced vs. novice coders. We used the following formula to calculate the odds ratio [46]:

$$OddsRatio(OR) = \frac{(a/c)}{(b/d)}$$

Probabilities >1 mean that the first subgroup is more likely to report a type of label, while probabilities less than 1 mean that the second group has greater chances (OR) [47].

Fig. 3. Survey question about the regions relevance

To understand the rationale behind the label choices, we qualitatively analyzed the answers of the open question ("Why was the information you selected relevant?"). We selected representative quotes to illustrate the participants perceptions about the labels' relevancy.

IV. RESULTS

In the following, we report the results grouped by research question.

A. RQ1. To what extent can we predict the categories of the APIs present in the implementation that solves an issue?

To predict the API categories identified in the files changed in each issue (RQ1), we started by testing a simple configuration, which we used as a baseline to compare the machine learning algorithms. For this baseline model, we used only the issue "TITLE" as input to build the model. The corpus comprised all the terms found in the titles (899 distinct words), after applying stemming, stop word removal, and cleaning (see Section III-B1). In this baseline model, we used the Random Forest (RF) algorithm, since it usually yields good results in software engineering studies [48, 49, 50, 51] and is insensitive to parameter settings [52].

Then, we evaluated the corpus configuration alternatives, varying the input information: only TITLE (T), only BODY (B), TITLE and BODY, and TITLE, BODY, and COMMENTS. To compare the classifiers, we created Random Forest models

TABLE I
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

and used the Mann-Whitney U test with the Cliffs-delta effect size.

As can be seen in Table I, all alternative settings provided better results than the baseline, improving precision, recall, and F-measure. When using Title, Body, and Comments, we reached Precision of 75.5%, Recall of 74.7%, and F-Measure of 75.1%.

TABLE II
CLIFF'S DELTA FOR F-MEASURE AND PRECISION: COMPARISON BETWEEN CORPUS MODELS - SECTION III-B-5. TITLE(T), BODY(B) AND COMMENTS (C)

corpus Comparison	Cliff's delta F-measure		Cliff's delta 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

The Mann-Whitney U test showed statistical difference between the results using TITLE and the other corpus configurations for both F-measure (p-value varying from 0.0006 to 0.01) and precision (p-value in range 0.0002 to 0.0005) with large effect size. We found the same statistical distribution for F-measure and precision with the effect size varying from negligible to small, from all the models except the one based on TITLE. Results suggest to use only the body to predict API-category labels, since it achieves similar results with less effort. In general, this model only presented 14% incorrect predictions (hamming loss metric) for all 12 labels. Table II shows the Cliffs-delta comparison between each pair of corpus configuration.

Next, we investigated the use of bi-grams, tri-grams, and four-grams and the number of words, comparing the results to the use of unigrams. We used the corpus with only issue body for this analysis, since this configuration performed best in the last step. Fig. 4 and Table III present how the Random Forest model performs for each n-gram configuration. The unigram configuration outperformed the others with large effect size. We did not find statistical significant difference among the models with 90, 179, 358, 716, 899, 1432, and 2864 words. The model with 5728 words has different distribution with "large" effect size (Cliffs-delta in the range from $|d|$=0.7 to 0.88; however, the model with 899 words had the lowest Hamming loss mean (0.145) and we used it as baseline.

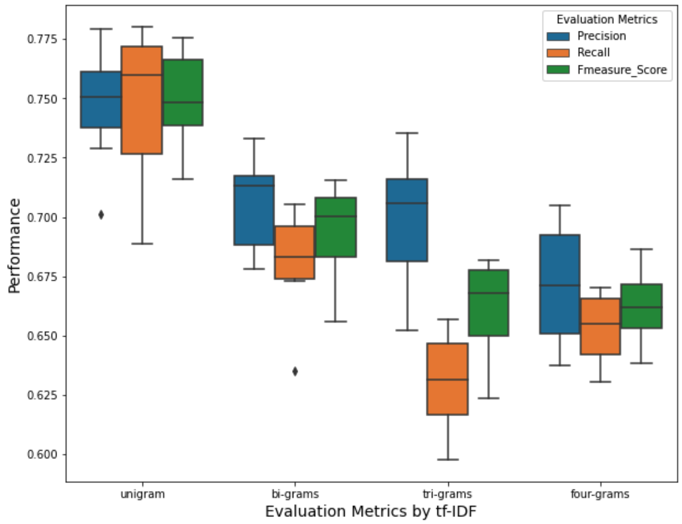

Fig. 4. Comparison between unigram model and n-grams models

TABLE III
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

Finally, to investigate the influence of the machine learning (ML) classifier, we compared several options using the title with unigrams as a corpus: Random Forest (RF), Neural Network Multilayer Perceptron (MLPC), Decision Tree (DT), LR, MlKNN, and a Dummy Classifier with strategy "most_frequent". We used the implementation from the Python package scikit-learn. Fig. 5 shows the comparison among the algorithms, and Table IV presents the pair-wise statistical results comparing F-measure and precision using Cliffs delta.

TABLE IV
CLIFFS 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

Random Forest (RF) and Neural Network Multilayer Perceptron (MLPC) were the best models when compared to Decision Tree (DT), LR, MlKNN, and Dummy algorithms.

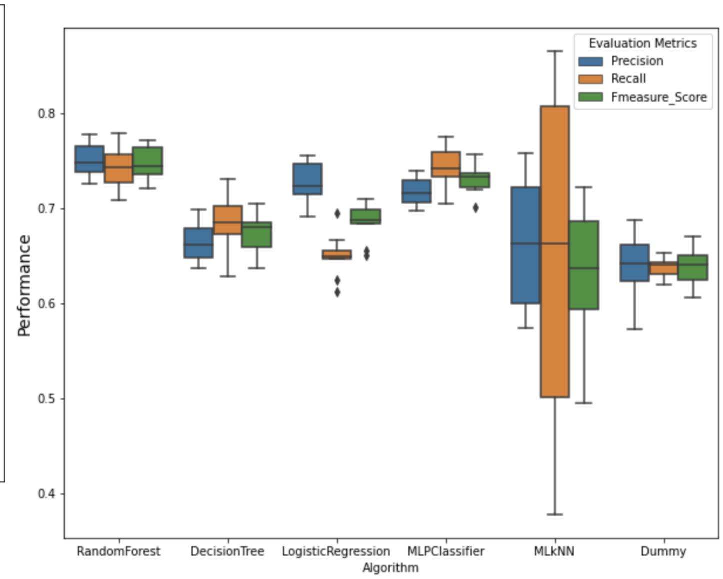

Fig. 5. Comparison between the baseline model and other machine learning algorithms

RF outperforms these four algorithms with large effect sizes considering F-measure and precision.

RQ1 Summary. It is possible to predict the API-categories with a precision of 0.755, a recall of 0.747, an F-measure of 0.751, and 0.142 of Hamming loss using the Random Forest algorithm, title, body, and comments as a corpus, and unigrams with 899 words.

B. RQ2. How relevant are the API-category labels to outside contributors?

To answer this research question, we analyzed the answers of the 74 participants of the experiment.

What information is used while selecting a task? Understanding the type of information that participants use to make their decision of selecting an issue to work on can help projects better organize the information on their issue pages. Fig. 6 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 was only marginally useful and participants had to also review the code. In contrast, in the Treatment group the top four regions of interest in priority order 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 useful than those 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 participants from Treatment selected 1.6x more labels than the Control groups (p=0.05). The odds ratio analysis suggests labels were perceived more relevant in the Treatment groups.

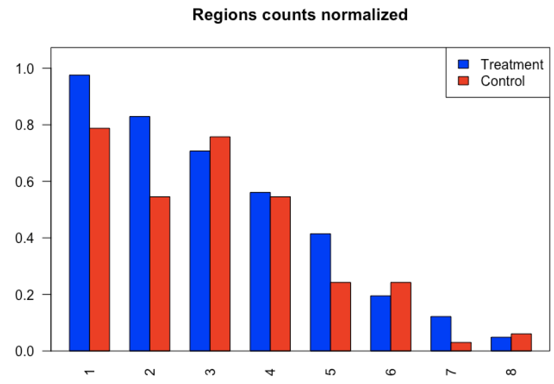

Fig. 6. The regions counts (normalized) of the issue's information page selected as most relevant by participants from Treatment and Control groups. 1-Title,2-Label,3-Body,4-Code,5-Comments,6-Author,7-Linked issues,8-Participants.

Qualitative analysis of the reason behind the choice of participants in the Treatment group reveals that the Title and the Labels together provided a comprehensive view of the issue. For instance, P4BT 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 to be useful in pointing out the specific topic about the issue, as P14BT stated: *"[labels are] hints about what areas have 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 divide 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 the 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-category (new labels): the labels that were generated by our classifier (IO, UI, Network, Security, etc.). These labels were available only to the Treatment group.

TABLE V
LABEL DISTRIBUTIONS AMONG THE CONTROL AND TREATMENT GROUP

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

Table V compares the labels that participants considered relevant (section III-C-3) across the Treatment and Control groups, distributed across these label types. As we can see in the Control group, a majority of selected labels (56.4%) relate to the type of issue (e.g., Bug or Enhancement) to make their decision as compared to the code component type. In the Experimental group, however, this number drops down to 36.8%, with API-category labels being in majority (42.7%), followed by Code component labels (20.6%). This difference in distributions alludes to the usefulness of the API-category labels.

To better understand the usefulness of the API-category labels as compared to the other types of labels, we further investigate the label choices among the Experimental group participants. Figure 7 presents two violin plots comparing (a) API-category labels against code component labels and (b) and type of issue. 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 labels are more frequently chosen (median of 5 labels) as compared to Code Component labels (median of 2 labels), with a large effect size ($|d|$ = 0.52). However, the distribution of the Issue Type and API-category labels are similar; confirmed by a negligible effect size ($|d|$ = 0.1). These results indicate that the type of issue (bug fix, enhancement, suitable for newcomer) is important as it allows developers to understand the type of task they would be involved in. Understanding the technical (API) requirements of solving the task is equally important in developers making their decision of selecting the task.

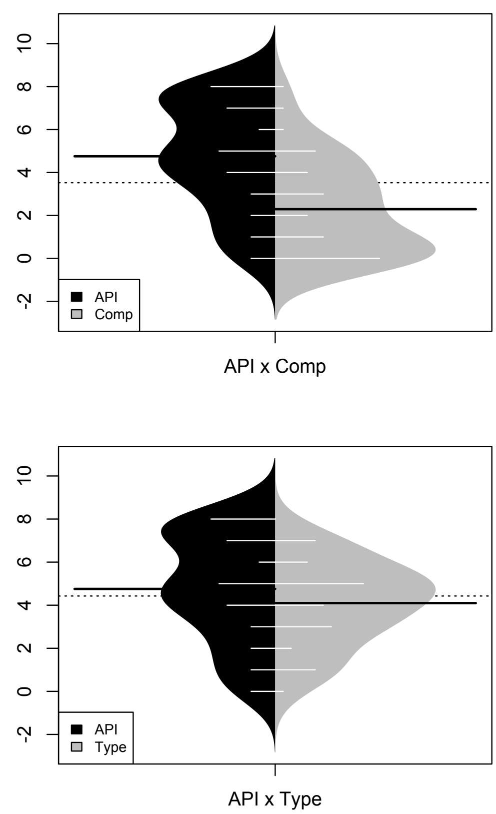

Fig. 7. Density Probability Labels (Y-Axis): API-category x Components x Types.

Finally, we analyzed whether the demographic subgroups had different perceptions about the API-category labels (Table VI). When comparing Industry vs. Students, we found participants from industry selected 1.9x (p-value=0.001) more API-category labels than students when we control by components 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-category labels than novice coders (p-value=0.01) when we control by type labels.

The odds ratio analysis suggests the API-category labels are more likely to be perceived relevant from practitioners and experienced than by students and novice coders.

TABLE VI
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 to work on. API-category labels increased the perception of the labels' relevancy. API-category labels are specially relevant for industry and experienced coders.

V. DISCUSSION

Are API-category labels relevant? Our results show that labels become more relevant for selecting issues when we introduce API-category labels. From a newcomer's point of view, these labels are preferred over the current code component labels. The fact that these labels are more generic than code components, usually not exposing details about the technology, may explain the results. Future interviews can help investigate this topic.

API-category labels showed to be more relevant for experienced coders. Additional research is necessary to provide effective ways to help novice contributors in onboarding. One could investigate, for example, labels that are less related to the technology and inform about difficult levels, priorities, estimated time to complete, contact for help, required/recommended academic courses, etc.

Finding an appropriate issue involves multiple aspects, one of which is knowing the types of APIs required, which our labels are about. Our findings show that participants consider these labels relevant in selecting issues. Future work can investigate if these labels lead to better decisions in terms of finding appropriate issues that match the developers' skills.

Does the size of the corpus matter?

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. Possibly bad titles (and bodies) impacted the ability of the classifier to produce good labels reducing the precision.

API-category imbalance and co-occurrence play a role. 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 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-category labels for ”Latex” and ”Open Office Document” appeared only in five java files, while ”Security” appears in only six files.

Therefore, 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-category 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, not considering possible co-occurrence of labels.

The more specific the API-category is, the harder it is to label. 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 the ”UI” and ”IO.” ”Logging” is a useful API category for all files and therefore explains its high occurrence. On the other hand, some specific API categories that are supposedly highly relevant to JabRef—such as "Latex", "PDF", and "Open Office Documents"—are not well represented in the predictions.

How could we label issues with API categories that are rare in our dataset? Looking to the Table VII and comparing with the co-occurrence mentioned above 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 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. 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 [53]. Thus, we can find an association rule between the previously classified labels.

TABLE VII
OVERALL METRICS FROM MODELS TO EVALUATE THE CORPUS SIZE

API Category	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%	66.6%
OpenOffice	165	6	3	5	0.0%	0.0%
Database	154	3	6	16	84.2%	72.7%
PDF	164	5	4	6	54.5%	60%
Logging	19	32	18	110	77.4%	85.9%
Latex	170	1	1	7	87.5%	87.5%

What information is relevant when selecting an issue to contribute to? Participants often selected TITLE, BODY, and LABELS to look for information when deciding which issue to contribute to. This result is in line with what we observed in the design of the machine learning algorithm.

Practical implications A practical application of this study is the capability of labeling issues with the most prominent API categories—in this case, ”UI”, ”Logging”, ”IO”, ”Network”, and ”Test.” This can be noticed in Table VII with precision and recall up to 77% and 95%, respectively.

Our results can be used to propose better layouts to the issue list and detail pages, prioritizing them against other information regions (3). In the issue detail page on GitHub, for instance, the label information is out of the main contributor focus, on the right of the screen.

Some wrong predictions in our study might be possibly caused by titles and body with little useful information. Templates can guide GitHub users in filling the issues' body to create patterns that help classifiers use the information to predict API labels.

This research has other implications for different stakeholders.

OSS contributors. API-category labels can help open source contributors, enabling them to review the skills needed to work on the issues up front. This is specially useful for newcomers and casual contributors [54, 55], who has no previous experience with the project terminology.

Project maintainers. Automatic API-category labeling can help maintainers distribute the team effort to address project tasks based on required expertise.

Researchers. The scientific community can extend the proposed approach to other languages and projects, including more data and different algorithms. Our approach can also be used in tools aimed at creating task recommendations that match newcomers' skills and career goals (e.g., [56]).

Educators. Educators who assign contributions to OSS as part of their coursework [57] can also benefit from the findings of this study. Labeling issues on OSS projects can help them select examples or tasks for their classes, bringing a practical perspective to the learning environment.

VI. Threats to Validity

One of the threats to the validity of this study is the API categorization. We acknowledge the threat that different individuals can create different categorizations, which may cause some bias in our results. To mitigate this problem, three individuals, including a Java Developer expert and a contributor to the JabRef project, created the API-category labels. In the future, we can leverage it with (semi-)automated or collaborative approaches (e.g., [58, 59]).

Another concern is the number of issues in our dataset and the link between issues and pull requests. To include an issue in the dataset, we needed to link it to its solution submitted via pull request. Thus, we include all the 705 issues with links. By linking the issue with its correspondent pull request, we could identify the APIs used to create the labels and define our ground truth (check Section III-A). To ensure that the link was correctly identified, we selected a random sample of 50 issues and manually checked for consistency. All of the issues in this validation set were correctly linked to their pull requests.

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.

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

Further, we did not evaluate the impact of API misclassification on users. However, we believe the impact is minimal since in the three most selected issues, out of 11 recommendations only 1 was a misclassification.

Generalization is also a limitation since we conducted a case study. The outcomes could differ to other projects or programming languages ecosystems and we expect to extend this approach in that direction in future work. Nevertheless, the case study in a real world system showed how a multi-label classification approach can be useful for predicting API-category labels and how relevant such kind of label can be to outsider developers. In the future, these API categories can generalize to other projects that use the same APIs across multiple project domains (Desktop and Web applications). JabRef adopts a common architecture (MVC) and frameworks (JavaFX, JUnit, etc.), which makes it similar to a large number of other projects. As described by Qiu et al. [60], projects adopt common APIs, accounting up to 53% of the APIs used. Moreover, our data can be used as a training set for automated API-category label generation in other projects.

VII. Conclusion

In this paper, we investigate to what extent we can predict API-category labels. To do that, we mined data from 705 issues from the JabRef project and predicted 12 API-category labels over this dataset. The model created using the Random Forest algorithm, unigrams, and data from the issue body offered the best results. The labels most present in the issues can be predicted with high precision.

To investigate whether API-category labels are helpful to contributors, we built an experiment to present a mocked list of open issues with the API-category labels mixed with the original labels. We found that industry practitioner and experienced coder select API-category labels more often than students and novice coders. Participants also preferred API-category labels over code component labels, which were already used in the project.

This study is a step toward helping outsiders match their API skills with each task and better identify an appropriate task to start their onboarding process into an OSS project. For future work, we will explore new projects, a word embedding approach (Word2vec) with domain trained data to vectorise the issues, and investigate an unified API label schema capable of accurately mapping the skills needed to contribute to OSS projects.

Acknowledgment

This work is partially supported by the National Science Foundation under Grant numbers 1815486, 1815503, 1900903, and 1901031, CNPq grant #313067/2020-1. We also thank the developers who spent their time answering our survey.

References

[1] J. Wang and A. 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*. ACM, 2011, pp. 76–79.

[2] I. Steinmacher, T. U. Conte, and M. A. 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*. IEEE, 2015, pp. 5299–5308.

[3] I. Steinmacher, M. A. G. Silva, M. A. Gerosa, and D. F. Redmiles, "A systematic literature review on the barriers faced by newcomers to open source software projects," *Information and Software Technology*, vol. 59, pp. 67–85, 2015.

[4] I. Steinmacher, T. Conte, M. A. Gerosa, and D. 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*, ser. CSCW '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1379–1392. [Online]. Available: https://doi.org/10.1145/2675133.2675215

[5] C. Stanik, L. Montgomery, D. Martens, D. Fucci, and W. 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)*. IEEE, 2018, pp. 172–182.

[6] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, 2010.

[7] N. Bettenburg, S. Just, A. Schröter, C. Weiß, R. Premraj, and T. Zimmermann, "Quality of bug reports in eclipse," in *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '07. New York, NY, USA: ACM, 2007, pp. 21–25. [Online]. Available: http://doi.acm.org/10.1145/1328279.1328284

[8] L. Vaz, I. Steinmacher, and S. Marczak, "An empirical study on task documentation in software crowdsourcing on topcoder," in *2019 ACM/IEEE 14th International Conference on Global Software Engineering (ICGSE)*. IEEE, 2019, pp. 48–57.

[9] I. Steinmacher, C. Treude, and M. A. Gerosa, "Let me in: Guidelines for the successful onboarding of newcomers to open source projects," *IEEE Software*, vol. 36, no. 4, pp. 41–49, 2018.

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

[11] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 287–296.

[12] Y. Feng, J. Jones, Z. Chen, and C. 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)*. IEEE, 2018, pp. 320–330.

[13] T. Guggulothu and S. A. Moiz, "Code smell detection using multi-label classification approach," *Software Quality Journal*, vol. 28, no. 3, pp. 1063–1086, 2020.

[14] F. El Zanaty, C. Rezk, S. Lijbrink, W. van Bergen, M. Côté, and S. McIntosh, "Automatic recovery of missing issue type labels," *IEEE Software*, 2020.

[15] Y. Park and C. 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*, ser. VISSOFT '09. IEEE, Sep. 2009, pp. 3–10.

[16] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. 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*, 2008, pp. 304–318.
[17] N. Pingclasai, H. Hata, and K.-i. Matsumoto, "Classifying bug reports to bugs and other requests using topic modeling," in *2013 20Th asia-pacific software engineering conference (APSEC)*, vol. 2. IEEE, 2013, pp. 13–18.
[18] Y. Zhou, Y. Tong, R. Gu, and H. Gall, "Combining text mining and data mining for bug report classification," *Journal of Software: Evolution and Process*, vol. 28, no. 3, pp. 150–176, 2016.
[19] R. Kallis, A. Di Sorbo, G. Canfora, and S. Panichella, "Ticket tagger: Machine learning driven issue classification," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 406–409.
[20] G. Uddin and F. Khomh, "Automatic mining of opinions expressed about apis in stack overflow," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.
[21] D. Hou and L. Mo, "Content categorization of API discussions," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 60–69.
[22] 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*, vol. 1, 2015, pp. 869–879.
[23] 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)*, 2016, pp. 392–403.
[24] 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)*, 2017, pp. 27–37.
[25] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. 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*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 293–304. [Online]. Available: https://doi.org/10.1145/3238147.3238191
[26] H. Zhong and H. Mei, "An empirical study on API usages," *IEEE Transactions on Software Engineering*, vol. 45, no. 4, pp. 319–334, 2019.
[27] 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*, pp. 1–1, 2018.
[28] 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)*, 2019, pp. 321–332.
[29] T. Dey, A. Karnauch, and A. Mockus, "Representation of developer expertise in open source software," *arXiv preprint arXiv:2005.10176*, 2020.
[30] R. K. Yin, *Case Study Research: Design and Methods*, ser. Applied social research methods series. Beverly Hills, CA: Sage Publications, 1984.
[31] JabRef, "JabRef project," 2019. [Online]. Available: https://jabref.org/
[32] T. Olsson, M. Ericsson, and A. Wingkvist, "The relationship of code churn and architectural violations in the open source software jabref," in *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, 2017, pp. 152–158.
[33] A. Mayr, R. Plösch, and C. Körner, "A benchmarking-based model for technical debt calculation," in *2014 14th International Conference on Quality Software*. IEEE, 2014, pp. 305–314.
[34] S. Herold, "An initial study on the association between architectural smells and degradation," in *European Conference on Software Architecture*. Springer, 2020, pp. 193–201.
[35] Z. Shi, J. Keung, and Q. Song, "An empirical study of bm25 and bm25f based feature location techniques," in *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices*, 2014, pp. 106–114.
[36] S. Feyer, S. Siebert, B. Gipp, A. Aizawa, and J. Beel, "Integration of the scientific recommender system mr. dlib into the reference manager jabref," in *European Conference on Information Retrieval*. Springer, 2017, pp. 770–774.
[37] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242. Piscataway, NJ, 2003, pp. 133–142.
[38] D. Behl, S. Handa, and A. 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)*. IEEE, 2014, pp. 294–299.
[39] S. L. Vadlamani and O. Baysal, "Studying software developer expertise and contributions in Stack Overflow and GitHub," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 312–323.
[40] F. Herrera, F. Charte, A. J. Rivera, and M. J. del Jesus, *Multilabel Classification: Problem Analysis, Metrics and Techniques*, 1st ed. Springer Publishing Company, Incorporated, 2016.
[41] F. Charte, A. J. Rivera, M. J. del Jesus, and F. Herrera, "Mlsmote: approaching imbalanced multilabel learning through synthetic instance generation," *Knowledge-Based Systems*, vol. 89, pp. 385–397, 2015.
[42] M.-L. Zhang and Z.-H. Zhou, "Ml-knn: A lazy learning

approach to multi-label learning," *Pattern Recognition*, vol. 40, no. 7, pp. 2038 – 2048, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0031320307000027
[43] D. Chicco and G. Jurman, "The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation," *BMC genomics*, vol. 21, no. 1, p. 6, 2020.
[44] J. Romano, J. 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*, 2006, pp. 1–3.
[45] A. Blanco, A. Casillas, A. Pérez, and A. D. de Ilarraza, "Multi-label clinical document classification: Impact of label-density," *Expert Systems with Applications*, vol. 138, p. 112835, 2019.
[46] M. Szumilas, "Explaining odds ratios," *Journal of the Canadian academy of child and adolescent psychiatry*, vol. 19, no. 3, p. 227, 2010.
[47] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, 5th ed. Chapman & Hall, 2020.
[48] D. Petkovic, M. Sosnick-Pérez, K. Okada, R. Todtenhoefer, S. Huang, N. Miglani, and A. Vigil, "Using the random forest classifier to assess and predict student learning of software engineering teamwork," in *2016 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2016, pp. 1–7.
[49] E. Goel, E. Abhilasha, E. Goel, and E. Abhilasha, "Random forest: A review," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 7, no. 1, 2017.
[50] T. 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)*. IEEE, 2014, pp. 1–5.
[51] S. M. Satapathy, B. P. Acharya, and S. K. Rath, "Early stage software effort estimation using random forest tech-
[54] G. Pinto, I. Steinmacher, and M. A. Gerosa, "More common than you think: An in-depth study of casual contributors," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER*
nique based on use case points," *IET Software*, vol. 10, no. 1, pp. 10–17, 2016.
[52] 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*, vol. 45, no. 7, pp. 683–711, 2019.
[53] R. Agrawal, T. Imieliundefinedski, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Rec.*, vol. 22, no. 2, p. 207–216, Jun. 1993. [Online]. Available: https://doi.org/10.1145/170036.170072
2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1, 2016, pp. 112–123.
[55] S. Balali, I. Steinmacher, U. Annamalai, A. Sarma, and M. A. Gerosa, "Newcomers' barriers. . . is that all? an analysis of mentors' and newcomers' barriers in OSS projects," *Comput. Supported Coop. Work*, vol. 27, no. 3–6, p. 679–714, Dec. 2018. [Online]. Available: https://doi.org/10.1007/s10606-018-9310-8
[56] A. Sarma, M. A. Gerosa, I. Steinmacher, and R. 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*, 2016, pp. 932–935.
[57] G. H. L. Pinto, F. Figueira Filho, I. Steinmacher, and M. A. Gerosa, "Training software engineers using open-source software: the professors' perspective," in *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 2017, pp. 117–121.
[58] 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)*, 2018, pp. 239–246.
[59] Y. Lu, G. Li, Z. Zhao, L. Wen, and Z. Jin, "Learning to infer API mappings from API documents," in *International Conference on Knowledge Science, Engineering and Management*. Springer, 2017, pp. 237–248.
[60] D. Qiu, B. Li, and H. Leung, "Understanding the API usage in Java," *Information and software technology*, vol. 73, pp. 81–100, 2016.