Automatic Detection of Five API Documentation Smells: Practitioners' Perspectives

Junaed Younus Khan[a], Md. Tawkat Islam Khondaker[a], Gias Uddin[b] and Anindya Iqbal[a]
[a]Bangladesh University of Engineering and Technology and [b]University of Calgary

***Abstract*—The learning and usage of an API is supported by official documentation. Like source code, API documentation is itself a software product. Several research results show that bad design in API documentation can make the reuse of API features difficult. Indeed, similar to code smells or code anti-patterns, poorly designed API documentation can also exhibit 'smells'. Such documentation smells can be described as bad documentation styles that do not necessarily produce an incorrect documentation but nevertheless make the documentation difficult to properly understand and to use. Recent research on API documentation has focused on finding content inaccuracies in API documentation and to complement API documentation with external resources (e.g., crowd-shared code examples). We are aware of no research that focused on the automatic detection of API documentation smells. This paper makes two contributions. First, we produce a catalog of five API documentation smells by consulting literature on API documentation presentation problems. We create a benchmark dataset of 1,000 API documentation units by exhaustively and manually validating the presence of the five smells in Java official API reference and instruction documentation. Second, we conduct a survey of 21 professional software developers to validate the catalog. The developers agreed that they frequently encounter all five smells in API official documentation and 95.2% of them reported that the presence of the documentation smells negatively affects their productivity. The participants wished for tool support to automatically detect and fix the smells in API official documentation. We develop a suite of rule-based, deep and shallow machine learning classifiers to automatically detect the smells. The best performing classifier BERT, a deep learning model, achieves F1-scores of 0.75 - 0.97.**

***Index Terms*—API Documentation, Smell, Benchmark, Survey, Shallow Learning, Deep Learning.**

I. Introduction

APIs (Application Programming Interfaces) are interfaces to reusable software libraries and frameworks. Proper learning of APIs is paramount to support modern day rapid software development. To support this, APIs typically are supported by official documentation. An API documentation is a product itself, which warrants the creation and maintenance principles similar to any existing software product. A good documentation can facilitate the proper usage of an API, while a bad documentation can severely harm its adoption [6], [61], [62].

A significant body of API documentation research has focused on studying API documentation problems based on surveys and interviews of software developers [6], [11], [13], [26], [27], [42], [61], [62], [64], [91]. Broadly, API documentation problems are divided into two types, what (i.e., what is documented) and how (i.e., how it is documented) [7], [81]. Tools and techniques are developed to address the 'what' problems in API documentation, such as detection of code comment inconsistency [57], [75], [84], [92], natural language summary generation of source code [44], [47], [65], [71], adding description of API methods by consulting external resources (e.g., online forums) [5], detecting obsolete API documentation by comparing API version [16], [17], and complementing official documentation by incorporating insights and code examples from developer forums [72], [78]. In contrast, not much research has focused on the automatic detection of 'how' problems, e.g., bad design in API documentation that can make the reuse of API features difficult due to lack of usability [81]. Recently, Treude et al. [77] find that not all API documentation units are equally readable. This finding reinforces the needs to automatically detect API documentation presentation issues as 'documentation smells', as previously highlighted by Aghajani et al. [6]. Unfortunately, we are not aware of any research on the automatic detection of such API documentation smells.

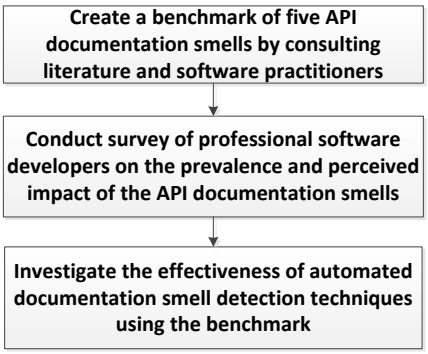

Fig. 1. The three major phases used in this study.

As a first step towards developing techniques to detect smells in API documentation, in this paper, we follow three phases (see Fig. 1). First, we identify five API documentation smells by consulting API documentation literature [7], [81] (Section II). Four of the smells (bloated, fragmented and tangled description of API documentation unit, and excess structural info in the description) are reported as presentation problems by Uddin and Robillard [81]. The other smell is called 'Lazy documentation' and it refers to inadequate description of an API documentation unit (e.g., no explanation of method parameters). Such incomplete documentation is reported in literature [7] and in online discussions. We exhaustively explore official API documentation to find the

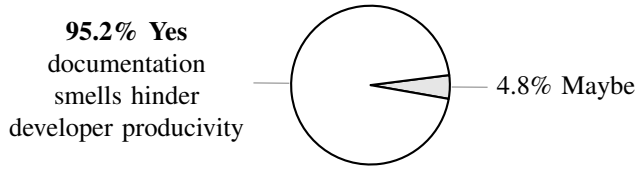

Fig. 2. Survey responses from professional developers on whether the presence of the smells in API documentation hinders productivity.

occurrences of the five smells. The focus was to develop a benchmark of *smelly* API documentation units. A total of 19 human coders participated in this exercise. This phase resulted in a benchmark of 1,000 API documentation units, where 778 units have at least one of the five smells. To the best of our knowledge, this is the first benchmark with real-world examples of the five documentation smells.

In the second phase (Section III), we conducted a survey of 21 professional software developers to validate our catalog of API documentation smells. All the participants reported that they frequently encounter the five API documentation smells. More than 95% of the participants (20 out of 21) reported that the presence of the five smells in API documentation negatively impacts their productivity (see Fig. 2). The participants asked for tool support to automatically detect and fix the smells in API official documentation. These findings corroborate previous research that design and presentation issues in API documentation can hinder API usage [6], [81]. In the third phase (Section IV), we investigate a suite of rule-based, shallow and deep machine learning models using the benchmark to investigate the feasbility of automatically detecting the five smells. The best performing classifer BERT, a deep learning model, achieves F1-scores of 0.75 - 0.97. To the best of our knowledge, ours are the first techniques to automatically detect the five API documentation smells. The machine learning models can be used to monitor and warn about API documentation quality by automatically detecting the smells in real-time with high accuracy.

Replication Package with benchmark, code, and survey is shared at https://github.com/disa-lab/SANER2021-DocSmell

II. A Benchmark of API Documentation Smells

We describe the methodology to create our benchmark of API documentation smells (Section II-A) and then present the benchmark with real-world examples (Sections II-B - II-C).

A. Benchmark Creation Methodology

Code and design smells are relatively well studied fields of software engineering. However, to the best of our knowledge, this is the first research on API documentation smells. As such, we needed to investigate both the literature on API documentation [6], [7], [62], [81] and the diverse API documentation resources (e.g., Java SE docs) during the creation of our catalog of API documentation smells. We followed a three-step process, which closely mimics the standard approaches followed in code/design smell formulation studies [2], [3]. The three steps are outlined in Fig. 3 and are explained below.

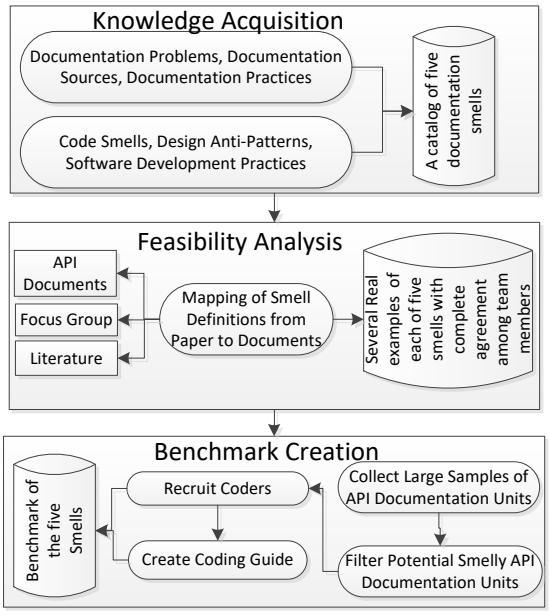

Fig. 3. The three major steps in benchmark creation process.

Knowledge Acquisition. Similar to code and design smells that do not directly introduce a defect or a bug into a software system, documentation smells refer to presentation issues that do not make a documentation incorrect, rather they hinder its proper usage due to the lack of quality in the design of the documented contents. As such, we studied extensively the API documentation literature that reported issues related to API documentation presentation and usability [7], [81]. For example, the most recent paper on this topic was by Aghajani et al. [6], [7], who divided the 'how' problems in API documentation into four categories: maintainability (e.g., lengthy files), readability (e.g., clarity), usability (e.g., information organization like dependency structure), and usefulness (e.g., content not useful in practice). Previously, Uddin and Robillard [81] studied 10 common problems in API documentation by surveying 323 IBM developers. They observed four common problems related to presentation, i.e., bloated (i.e., too long description), tangled (complicated documentation), fragmented (i.e., scattered description), and excessive structural information (i.e., information organization like dependency structure). Given that the four problems appeared in both studies, we included each as a documentation smell in our study. In addition, we added lack of proper description of an API method as a 'lazy' documentation smell, because incomplete documentation problems are discussed in literature [6], [81] as well as in online developer discussions (see Fig. 4).

Feasibility Analysis. Once we decided on the five smells, we conducted a feasibility study by looking for real-world examples of the smells in official and instructional API documentation. This was important to ensure that the smells are prevalent in API documentation and that we can find

Fig. 4. Tweet complaining about lazy documentation of API method.

those with reasonable confidence, because otherwise there is no way we can design automated techniques to detect those automatically. We combined our knowledge of the five smells gained from API documentation literature with active exploration of the five smells in the API official documentation. We conducted multiple focus group discussions where all the four authors discussed together by analyzing potential examples of the five smells in API documentation and by mapping the characteristics of such API documentation with the description of the smells in the literature/developer discussions. Before every such focus group meeting, the first two authors created a list of 50 API documentation units with their labels of the five smells in the units. The four authors discussed those labels together, refined the labels, and identified/filtered the labeling criteria. This iterative process led to increased understanding among the group members on the specific characteristics of the five documentation smells. From multiple discussion sessions, the final output was a list of 50 labeled datapoints.

Benchmark Creation. In the last step of the benchmark creation process, we expanded our initial list of 50 API documentation units with smell labels as follows. We collected documentations of over 29K methods belonging to over 4K classes of 217 different packages. We extracted these documentations from the online JAVA API Documentation website [1] through web crawling and text parsing techniques. Since a documentation can contain multiple smells at the same time, this is a multi-labeled dataset. We produced the benchmark as follows. First, all the authors mutually discussed the documentation smells. Then, we randomly selected 950 documentations from a total of 29K that we extracted. Then the first two authors labeled the first 50 documentations separately. When they finished, they consulted other co-authors and resolved the disagreement based on the discussion. Then they continued with the next 50 documentations and repeated the same process. Their agreement of labeling has been recorded using Cohen's Kappa Coefficient [45] for each iteration, i.e., labeling 50 documentations (Table I). After the third iteration, both the authors reached a perfect agreement level with Cohen's Kappa Coefficient of 0.83. Then they prepared a coding guideline for the labeling task which was later presented to 17 computer science undergraduate students. The students labeled the remaining 800 documentation units. During the entire coding sessions by the 17 coders, the first two authors remained available to them via Skype/Slack. Each coder consulted their labels with the two authors. This ensured quality and mitigated subjective bias in the manual labeling of the benchmark.

TABLE I
MEASURE OF AGREEMENT BETWEEN TWO LABELERS

Iteration ID	Documentation Unit #	Cohen κ
1	50	.49
2	50	.67
3	50	.83

B. The Five Documentation Smells in the Benchmark

Bloated Documentation Smell. By 'Bloated' we mean the documentation whose description (of an API element type) is verbose or excessively elaborate. It is difficult to understand or follow a lengthy documentation [81]. Moreover, it cannot be effectively managed that makes it hard to modify when needed, e.g., in case of any update in the API source code. In our benchmark, we found many documentations that are larger than necessary. For example, the documentation shown in Fig. 5 is so verbose and lengthy that it is hard to follow and use it. Hence, it is a bloated documentation.

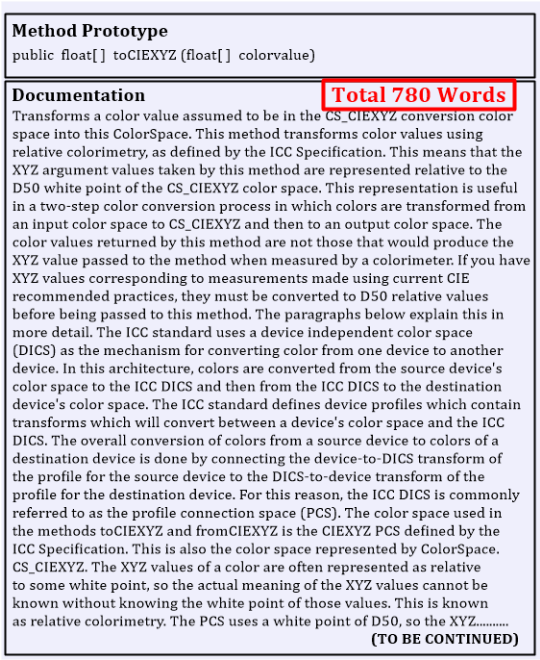

Fig. 5. Example of Bloated Smell.

Excess Structural Information Smell. Such a description of a documentation unit (e.g., method) contains too many structural syntax or information, e.g., the Javadoc of the java.lang.Object class. Javadoc lists all the hundreds of subclasses of the class. In our study, we find this type of documentation to contain many class and package names. For instance, the documentation of Fig. 6 contains many structural information (marked in red rectangle) that are quite unnecessary for the purpose of understanding and using the underlying method.

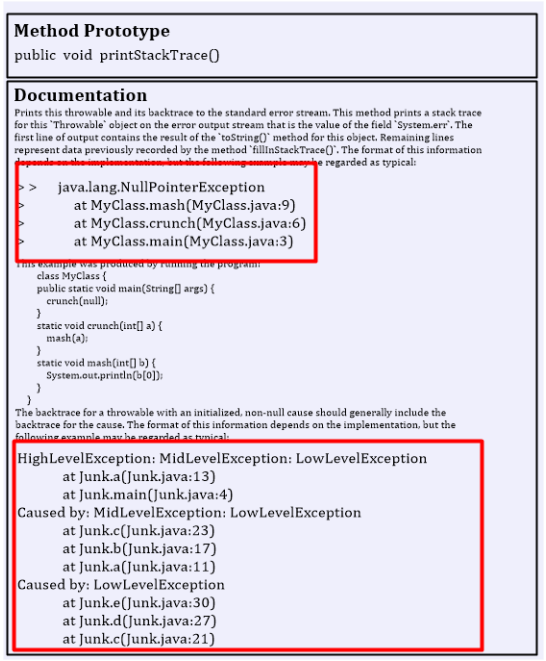

Fig. 6. Example of Excess Structural Information.

Tangled Documentation Smell. A documentation of an API element (method) is 'Tangled' if it's description is *tangled* with various information (e.g., from other methods). This makes it complex and thereby reduces the readability and understandability of the description. Fig. 7 depicts an example of tangled documentation which is hard to follow and understand.

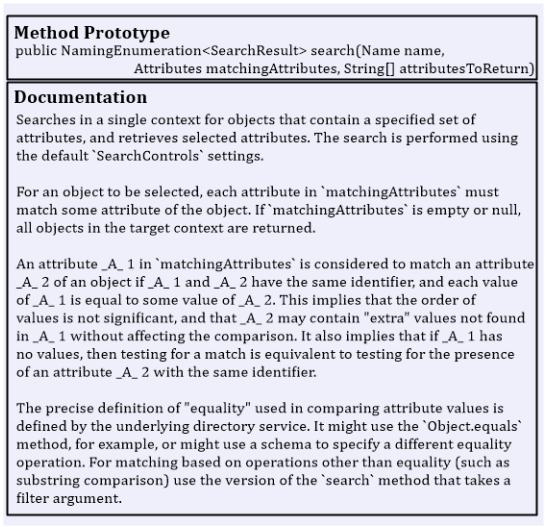

Fig. 7. Example of Tangled Smell.

Fragmented Documentation Smell. Sometimes it is seen that the information of documentation (related to an API element) is scattered (i.e., fragmented) over too many pages or sections. In our empirical study, we found a good number of documentation that contain many URLs and references that indicate possible fragmentation smell. For example, the documentation of Fig. 8 is fragmented as it refers the readers to other pages or sections for details.

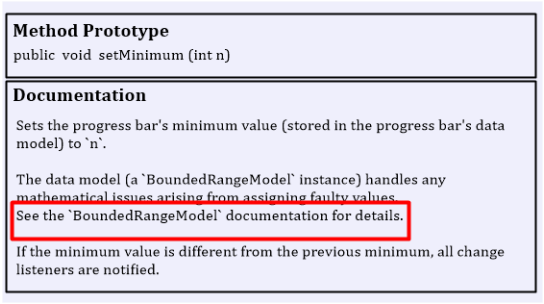

Fig. 8. Example of Fragmented Smell.

Lazy Documentation Smell. We categorize a documentation as 'Lazy' if it contains very small information to convey to the readers. In many cases, it is seen that the documentation does not contain any extra information except what can be perceived directly from the function name. Hence, this kind of documentation does not have much to offer to the readers. We see a lazy documentation in Fig. 9 where the documentation says nothing more about the underlying method than what is suggested by the prototype itself.

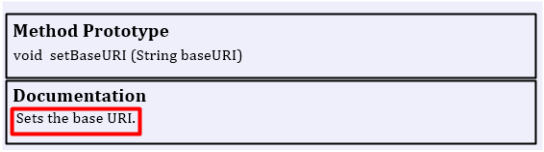

Fig. 9. Example of Lazy Smell.

C. Distribution of API Documentation Smells in Benchmark

We calculated the total number of smells in our dataset (Fig. 10). We found that 778 documentations (almost 78%) of our dataset contain at least one smell. While most (524) of the smelly documentations contain only one type of smell, a small number (19) of documentations show as high as four smells at the same time. We also determined the distribution of different smells in our dataset (Fig. 11). It shows that all the five types of smells discussed occur in the dataset with a considerable frequency where the most frequent smell in our dataset is 'Lazy' with 275 occurrences and the least frequent smell is 'Bloated' with 141 occurrences.

In multi-label learning, the labels might be interdependent and correlated [31]. We used Phi Coefficients to determine such interdependencies and correlations between different documentation smells. The Phi Coefficient is a measure of association between two binary variables [15]. It ranges from -1 to +1, where ±1 indicates a perfect positive or negative correlation and 0 indicates no relationship. We report the Phi Coefficients between each pair of labels in Fig. 12. We find that there is almost no correlation between 'Fragmented' and any other smell (except 'Lazy'). By definition, the information

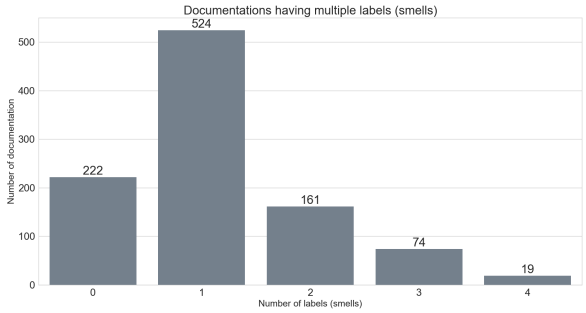

Fig. 10. Smell distribution by # of documentation units.

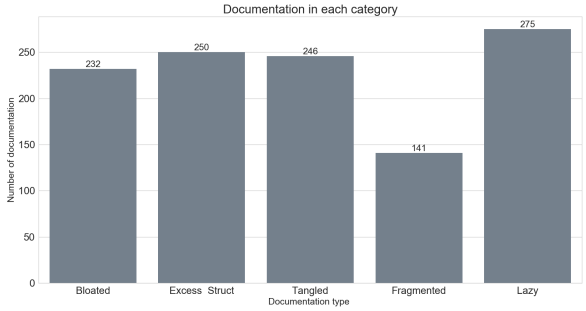

Fig. 11. Distribution of different smells in our dataset.

of fragmented documentation is scattered in many sections or pages. Hence, it has little to do with smells like 'Bloated', 'Excess Structural Information', or 'Tangled'. We also observe that there is a weak positive correlation (+0.2 to +0.4) among the 'Bloated', 'Excess Structural Information', and 'Tangled' smells. One possible reason might be that if a documentation is filled with complex and unorganized information (Tangled) or unnecessary structural information (Excess Structural Information), it might be prone to become bloated as well. On the other hand, 'Lazy' smell has a weak negative correlation (-0.2 to -0.3) with all other groups since these kinds of documentation are often too small to contain other smells. However, none of these coefficients is high enough to imply a strong or moderate correlation between any pair of labels. Hence, all types of smells in our study are more or less unique in nature.

III. Developers' Survey of Documentation Smells

Four out of the five API documentation smells in our study were previously reported as commonly observed by IBM developers [81]. The other smell (lazy documentation) is reported as a problem in API documentation in multiple studies [6], [61]. Given that we extended previous studies by creating a benchmark of the smells with real-world examples, we needed to further ensure that our collected examples of smelly documentation units do resonate with software developers. We, therefore, conducted a survey of professional software developers (1) to validate our catalog of the five API documentation smells and (2) to understand whether, similar to previous research, developers agree with the negative impact

Fig. 12. Correlation between different documentation smells in our benchmark. Red, Blue, and Gray mean positive, negative, and no correlation. Intensity of color indicates the level of correlation.

of the documentation smells. In particular, we explore the following two research questions:

RQ1. How do software developers agree with our catalog and examples of the five API documentation smells?

RQ2. How do software developers perceive the impact of the detected documentation smells?

A. Survey Setup

We recruited 21 professional software developers who are working in the software industry. We ensured that each developer is actively involved in daily software development activities like API reuse and documentation consultation. The participants were collected through personal contacts. First, each participant had to answer two demographic questions: current profession and years of experience in software development. We then presented each participant two Javadoc examples of each smell and asked him/her whether they agreed that this documentation example belonged to that particular smell. Then, we asked them about how frequently they faced these documentation smells. Finally, we inquired them of the negative impact of the documentation smells on their overall productivity during software development. Out of the 21 participants, 14 participants had experience less than 5 years and the rest had more than 5 years. Majority of the participants had experience less than 5 years because they are likely to be more engaged in studying API documentation as part of their software programming responsibility. Developers with experience more than five years are more engaged in design of the software and its architecture.

B. How do software developers agree with our catalog and examples of the five API documentation smells? (RQ1)

We showed each participant two examples of each smell, i.e., 10 examples in total. For each example, we asked two questions: (1) Do you think the documentation mentioned above is [smell, e.g., lazy]? The options are in Likert scale, i.e., strongly agree, agree, neutral, disagree, and strongly disagree. and (2) Based on your experience of the last three months,

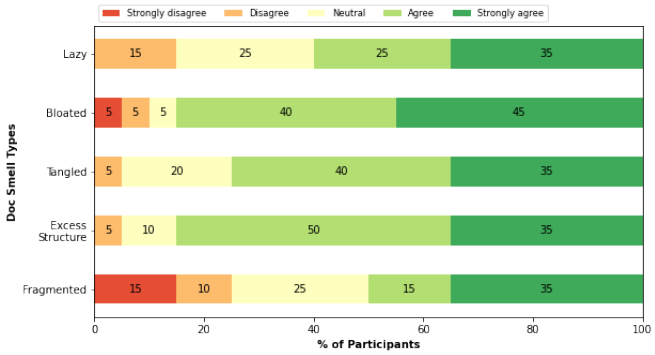

Fig. 13. Survey response on whether the software developers agreed with our labeled documentation smell examples.

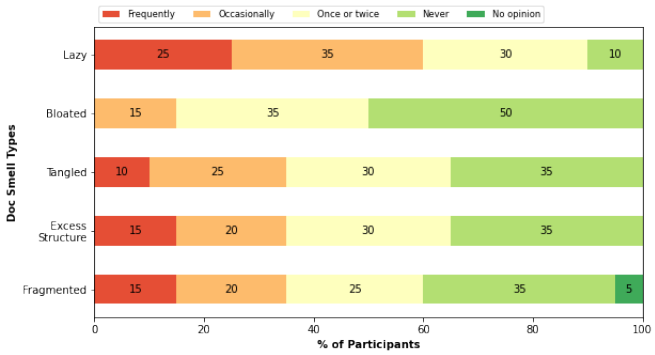

Fig. 14. Survey response on how frequently the participants faced the documentation smells in the last three months.

how frequently did you observe this [smell, e.g., lazy] in documentation? The options are: never, once or twice, occasionally, frequently, and no opinion. The options are picked from literature [81]. Two examples per smell ensure increased confidence on the feedback we get from each participant.

Fig. 13 shows the responses of the participants to the first question. More than 75% participants agreed to the examples of three smells: bloated, tangled, and excess structural info. At least 50% of the participants agreed to the examples of the other two smells. Only 5-25% of the participants disagreed to the examples. Overall, each example of the API documentation smell was agreed by at least 50% of the participants. This validates out catalog of API documentation smells based on feedback from the professional developers.

Fig. 14 shows the frequency of the documentation smells the developers observed in the last three months (second question). We found that 50% of the participants had faced all the smells and lazy smell was the most frequently encountered. On the other hand, half of the participants did not face bloated documentation smells in the last three months, while 60%-65% of the participants faced tangled, excess structural info, and fragmented API documentation. This study reveals that API documentation is becoming less explicable, more complex, and unnecessarily structured to keep the documentation short. To solve this problem, API documentation needs to be more understandable and elaborated to explain the API functionality.

C. How do software developers perceive the impact of the detected documentation smells? (RQ2)

We asked the participants how severely the documentation smells impact their development tasks. The responses were taken on a scale of five degrees: "Blocker", "Severe", "Moderate", "Not a Problem", and "No opinion". The options were picked from similar questions on API documentation presentation problems from literature [81].

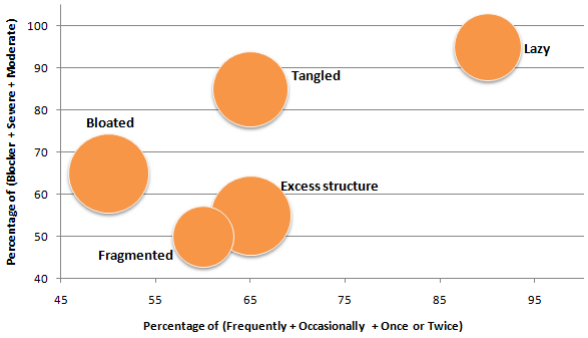

Fig. 15. The perceived impact of the five documentation smells by severity and frequency. Circle size indicates the percentage of participants who strongly agreed or agreed to the smells.

We analyzed the impact of the documentation smells with respect to the frequency of the smells the participants had observed over the past three months (see Fig. 15). For each smell, we compute the frequency scale (x-axis) as the percentage of response "Frequently", "Occasionally", and "Once or twice". For example, regarding whether the participants had observed lazy documentation in the past three months, 25% answered "Frequently", 35% answered "Occasionally", and 30% answered "Once or twice", leading to a total 90% in the frequency scale. We constructed the severity scale (y-axis) by combining the percentage of the participants responded with "Blocker", "Severe", and "Moderate". For example, due to fragmented documentation smells, 5% of the participants could not use that particular API and picked another API ("Blocker"), 20% of the participants believed that they wasted a lot of time figuring out the API functionality ("Severe"), and 25% of the participants felt irritated ("Moderate") with the fragmented documentation. The circle size indicates the percentage of the participants "Strongly Agree" or "Agree" with the examples containing documentation smells.

From Fig. 15, we observed that lazy documentation had the most frequent and the most negative impact (90%). Tangled documentation was identified as the second most severe smell (85%). Although bloated documentation was considered more severe (65%) than excess structural info (55% severity) and fragmented (50% severity) documentation, bloated occurred less frequently than the later two. The most important finding of this survey is that the coordinates of all the circles (referring to documentation smells) in Fig. 15 were above or equal to

50. This indicates that according to the majority of the participants, these documentations smells are occurring frequently and hindering the productivity of the development tasks.

IV. Automatic Detection of The Smells

The responses from the survey validate our catalog of API documentation smells. The perceived negative impact of the smells on developers' productivity, as evidenced by the responses from our survey participants, necessitates the needs to fix API documentation by removing the smells. To do that, we first need to detect the smells automatically in the API documentation. The automatic detection offers two benefits: (1) we can use the techniques to automatically monitor and warn about bad documentation quality and (2) we can design techniques to fix the smells based on the detection. In addition, manual effort can also be made for improving detected examples. With a view to determine the feasibility of techniques to detect API documentation smells using our benchmark, we answer three research questions:

RQ3. How accurate are rule-based classifiers to automatically detect the documentation smells?

RQ4. Can the shallow machine learning models outperform the rule-based classifiers?

RQ5. Can the deep machine learning models outperform the other models?

The shallow and deep learning models are supervised, for which we used 5-fold iterative stratified cross-validation as recommended for a multilabel dataset in [67]. Traditional k-fold cross-validation is a statistical method of evaluating machine learning algorithms which divides data into k equally sized folds and runs for k iterations [59]. In each iteration, each of the k folds is used as the held-out set for validation while the remaining $k-1$ folds are used as training sets. Stratified cross-validation is used to make sure that each fold is an appropriate representative of the original data by producing folds where the proportion of different classes is maintained [51]. However, stratification is not sufficient for multi-label classification problems as the number of distinct labelsets (i.e., different combinations of labels) is often quite large. For example, there can be 32 combinations of labels in our study as there are 5 types of documentation smells. In such cases, original stratified k-fold cross-validation is impractical since most groups might consist of just a single example. Iterative stratification, proposed by [67], solves this issue by employing a greedy approach of selecting the rarest groups first and adding them to the smallest folds while splitting.

We report the performances using four standard metrics in information retrieval [43]. Accuracy (A) is the ratio of correctly predicted instances out of all the instances. Precision (P) is the ratio between the number of correctly predicted instances and all the predicted instances for a given smell. Recall (R) represents the ratio of the number of correctly predicted instances and all instances belonging to a given class. F1-score ($F1$) is the harmonic mean of precision and recall.

$$P = \frac{TP}{TP+FP},\ R = \frac{TP}{TP+FN},\ F1 = 2 * \frac{P*R}{P+R},\ A = \frac{TP+TN}{TP+FP+TN+FN}$$

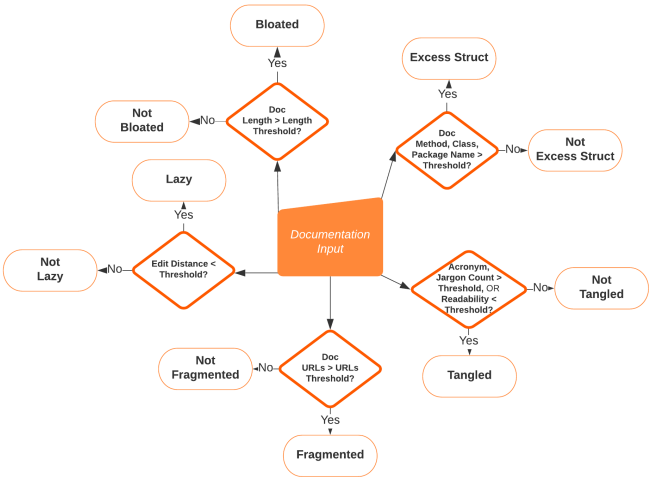

Fig. 16. Flowchart of rule-based classification approach.

TP = Correctly classified as a smell, FP = Incorrectly classified as a smell, TN = Correctly classified as not a smell, FN = Incorrectly classified as not a smell.

A. Performance of Rule-Based Classifiers (RQ3)

Based on manual analysis of a statistically significant random sample of our benchmark dataset (95% confidence interval and 5 levels), we designed six metrics to establish five rule-based classifiers as described below.

1) Rule-based Metrics: (a) *Documentation Length.* We use the length of every documentation in order to capture the extensiveness of the bloated documentations. (b) *Readability Metrics.* We measure Flesch readability metrics [25] for the documentations to analyze the understandability of documentation. This feature might be useful to detect tangled documentations. (c) *Number of Acronyms and Jargons.* Since acronyms and jargons increase the complexity of a reading passage [10], we use the number of acronyms and jargons in every documentation to detect the tangled documentation. (d) *Number of URLs* is computed because URLs are hints of possible fragmentation in the documentation. (e) *Number of function, class, and package name mentioned* in documentation is computed to capture excess structural information smell. (f) *Edit Distance.* The edit distance (i.e., measure of dissimilarity) between the description of a lazy documentation and its' corresponding unit definition (i.e., method prototype) can be smaller than non-lazy documentations. We calculate the Levenshtein distance [38] between the documentation description and method prototype.

2) Rule-based Classifiers: Fig. 16 shows flowchart of the rule-based classification approach. For each metric, we study average, 25^{th}, 50^{th}, 75^{th}, and 90^{th} percentiles as thresholds.

3) Results: In Table II, we reported the performances of the baseline models for each documentation smell. Different thresholds of features achieved higher performance for different documentation smells. For example, taking 90^{th} percentiles of the features' values, baseline model achieved the higher performance for bloated documentation detection, while lazy

TABLE II
CLASS-WISE PERFORMANCE OF RULE-BASED BASELINE MODELS BY THE METRIC THRESHOLDS (P STANDS FOR PERCENTILE)

		Bloated				**Lazy**				**Excess Struct**				**Tangled**				**Fragmented**			
Model	**Threshold**	A	P	R	F1	A	P	R	F1	A	P	R	F1	A	P	R	F1	A	P	R	F1
Rule Based	**AVG**	.77	.38	.86	.52	.58	.39	.71	.51	.68	.35	.34	.34	.49	.13	.18	.15	.65	.33	.52	.40
	25P	.39	.18	.64	.29	.96	.96	.93	**.95**	.67	.38	.30	.34	.54	.09	.09	.09	.52	.31	.90	**.47**
	50P	.64	.28	.79	.41	.77	.55	.84	.66	.75	.37	.50	**.42**	.45	.20	.40	.26	.61	.34	.71	.46
	75P	.89	.56	.93	.70	.52	.36	.67	.47	.65	.32	.31	.31	.37	.25	.75	.37	.67	.29	.27	.28
	90P	.95	.97	.85	**.90**	.37	.30	.56	.39	.75	.50	.17	.25	.33	.26	.96	**.41**	.72	.27	.10	.15

and excess structural information smell detection required 25^{th} percentile and 50^{th} percentile, respectively. Notably, the performance of the baseline models in detecting bloated (.90 F1-score) and lazy (F1 = .95) documentation were higher than detecting excess structural info (F1 = .42), tangled (F1 = .41), and fragmented (F1 = .47) documentations.

B. Performance of Shallow Learning Models (RQ4)

1) Shallow Learning Models: Since documentation smell detection is multi-label classification problem, we employed different decomposition approaches: One-Vs-Rest (OVR), Label Powerset (LPS), and Classifier Chains (CC) [18], [79], [80], [89] with Support Vector Machine (SVM) [14] as the base estimator. We chose SVM and OVR-SVM since those are successfully used for multi-label text classification [22], [23], [28], [35], [79]. Each model trains a single classifier per class, with the samples of that class as positive samples and all other samples as negatives. Each individual classifier then separately gives predictions for unseen data. We used linear kernel for the SVM classifiers as recommended by earlier works [86], [90]. [33], [87]. We also evaluated adapted approaches like Multi label (ML) *k*NN [88] in this study. It finds the *k* nearest neighborhood of an input instance using *k*NN, then uses Bayesian inference to determine the label set of the instance. We studied this method because it has been reported to achieve considerable performance for different multi-label classification tasks in previous studies [8], [88]. For each algorithm, we picked the best model using standard practices, e.g., hyper parameter tuning in SVM as recommended by Hsu [83], choice of K in ML-kNN as recommended by [8].

2) Studied Features: We used two types of features: (1) rule-based metrics (described in Section IV-A1) and (2) bag of words (BoW) [32]. Bag of words (BoW) is a common feature extraction procedure for text data and has been successfully used for text classification problems [46], [66].

3) Results: Table III presents the performance of the shallow learning models. The best performer is OVR-SVM, followed closely by CC-SVM. CC-based models are generally superior to OVR-based models because of the capability of capturing label correlation [58]. Since the labels (types) of the presentation smells are not correlated (see Section II-C), the CC-based SVM could not exhibit higher performance than the OVR-based SVM. Using rule-based features, OVR-SVM achieved a higher F1-score (0.88) than the other models for bloated documentation detection. Because documentation length (a rule-based) was more effective in detecting bloated documentation than bag of words. On the other hand, LPS-SVM achieved a higher F1-score (0.58) for fragmented documentation detection using bag of words, as bag of words more successfully determined whether the documentation was referring to other documentation than any rule-based features. Overall, the shallow models outperformed the rule-based classifiers for four smell types (except for lazy documentation smell). Therefore, the documentation smell detection does not normally depend on a single rule-based metric, rather, it depends on a combination of different metrics and their thresholds. The shallow learning models attempted to capture this combination of thresholds, and therefore, achieved better performances than the baseline models.

4) Feature Importance Analysis: We verified the importance of our rule-based features by applying permutation feature importance technique [9], [24] in the best performing shallow model, i.e., OVR-SVM. We first train OVR-SVM with all the features. While testing, we randomly shuffle the values of one feature at a time while keeping other feature values unchanged. A feature is important if shuffling its values affects the model performance. We calculate the change in performance in two ways. First, we measure the change in the average F1-score of the OVR-SVM model for the permutation of a feature. Second, we report the change of the specific class that the feature was intended for (i.e., 'Documentation Length' for 'Bloated'). We observe that the permutation of any of our rule-based features degrades the model performance (see Table IV). For example, after permutation of the values of the 'Documentation Length' of test data, the average F1-score decreases by 0.17 (from 0.62 to 0.45) and the F1-score of the desired class (i.e., 'Bloated') decreases by 0.46 (from 0.88 to 0.42). This analysis confirms the importance of combining rule-based metrics as features in the models.

C. Performance of Deep Learning Models (RQ5)

1) Deep Learning Models: We evaluated two deep learning models, Bidirectional LSTM (Bi-LSTM) and Bidirectional Encoder Representations from Transformers (BERT). We picked Bi-LSTM, because it is more capable of exploiting contextual information than the unidirectional LSTM [30]. Hence, the Bi-LSTM network can detect the documentation smell by capturing the information of the API documentations from both directions. BERT is a pre-trained model which was designed to learn contextual word representations of unlabeled

TABLE III
CLASS-WISE PERFORMANCE OF SHALLOW MACHINE LEARNING MODELS

		Bloated				Lazy				Excess Struct				Tangled				Fragmented			
Feature	**Model**	A	P	R	F1	A	P	R	F1	A	P	R	F1	A	P	R	F1	A	P	R	F1
Rule Based Feats	**OVR-SVM**	.96	.88	.89	**.88**	.94	.86	.94	.90	.74	.45	.23	.31	.82	.67	.56	**.61**	.80	.69	.25	.37
	LPS-SVM	.94	.86	.70	.77	.91	.77	.97	.86	.74	.44	.21	.28	.80	.70	.40	.51	.81	.73	.32	.45
	CC-SVM	.96	.88	.87	**.88**	.92	.79	.97	.87	.75	.47	.24	.32	.82	.68	.54	.60	.80	.71	.27	.39
	ML-*k*NN	.93	.73	.89	.80	.91	.86	.80	.83	.75	.49	.31	.38	.80	.63	.54	.58	.79	.57	.50	.53
BoW Feats	**OVR-SVM**	.93	.84	.66	.74	.95	.87	.96	**.91**	.75	.49	.47	.48	.78	.57	.54	.56	.79	.55	.54	.55
	LPS-SVM	.93	.89	.63	.74	.94	.83	.97	.89	.75	.50	.49	**.50**	.79	.59	.58	.58	.80	.59	.58	**.58**
	CC-SVM	.93	.85	.67	.75	.94	.85	.96	.90	.74	.48	.47	.48	.78	.57	.54	.56	.78	.54	.54	.54
	ML-*k*NN	.93	.86	.60	.71	.88	.75	.83	.79	.73	.44	.29	.35	.79	.59	.53	.56	.80	.63	.41	.50

TABLE IV
OVR-SVM PERFORMANCE DECREASE IN FEATURE PERMUTATION

Permuted Feature	**Desired Class C**	**Decrease in F1**	
		Overall	**Desired C**
Doc Length	Bloated	.17	.46
Readability	Tangled	.06	.11
#Acronym&Jargon	Tangled	.05	.07
#URLs	Fragmented	.03	.11
#Method, Class, Package	Excess Struct	.17	.08
Edit Distance	Lazy	.09	.37

texts [21]. We picked BERT, because it is found to significantly outperform other models in various natural language processing and text classification tasks [4], [29], [39], [40], [48], [52], [76]. We constructed a Bi-LSTM model with 300 hidden states. We used ADAM optimizer [37] with an initial learning rate of 0.001. We trained the model with batch size 256 over 10 epochs. We used BERT-Base for this study which has 12 layers with 12 attention heads and 110 million parameters. We trained it on benchmark for 10 epochs with a mini-batch size of 32. We used early-stop to avoid overfitting [56] and considered validation loss as the metric of the early-stopping [55]. The maximum length of the input sequence was set to 256. We used AdamW optimizer [41] with the learning rate set to $4e^{-5}$, $\beta 1$ to 0.9, $\beta 2$ to 0.999, and ϵ to $1e^{-8}$ [21], [73]. We used binary cross-entropy to calculate the loss [63].

2) Studied Features: We used word embedding as feature which is a form of word representation that is capable of capturing the context of a word in a document by mapping words with similar meaning to a similar representation. For Bi-LSTM, we used 100-dimensional pre-trained GloVe embedding which was trained on a dataset of one billion tokens (words) with a vocabulary of four hundred thousand words [53]. We used the pre-trained embedding in BERT model [21].

3) Results: Table V shows the performance of the deep learning models. BERT outperformed Bi-LSTM, the shallow, and rule-based classifiers to detect each smell (F1-score). The increase in F1-score in BERT compared to the best performing shallow learning model per smell is as follows: bloated (5.7% over OVR-SVM Rule), lazy (6.6% over OVR-SVM BoW), Excess Structural Information (52% over ML-kNN BoW), tangled (36.1% over OVR-SVM Rule), and fragmented (36.4% over OVR-SVM BoW). SVM and kNN-based models produced more false-negative results because the number of positive instances for an individual smell type is lower than the number of negative instances for that type. As a result, SVM and kNN-based models showed low recalls for some types (Excess structural information, Tangled, and Fragmented) and consequently resulted in low F1-scores. On the other hand, Bi-LSTM and BERT achieved better performance because they focused on capturing generalized attributes for each smell type. We manually analyzed the misclassified examples of Excess Structural Information and fragmented documentation where BERT achieved below 0.8 accuracy. For the Excess Structural Information smell detection, BERT falsely considered some java objects and methods as structural information; therefore, the model produced some false positive cases. In some examples, BERT could not identify whether the information of documentation was referring to other documentation. As a result, the model misclassified the fragmented documentation.

V. DISCUSSIONS

Implications of Findings. Thanks to the significant research efforts to understand API documentation problems using empirical and user studies, we now know with empirical evidence that the quality of API official documentation is a concern both for open source and industrial APIs [7], [26], [27], [62], [81]. The five API documentation smells we studied in this paper are frequently referred to as documentation presentation/design problems in the literature [7], [81]. Our comprehensive benchmark of 1,000 API official documentation units has 778 units each exhibiting one or more of the smells. The validity of the smells by professional software developers proves that this benchmark can be used to foster a new area of research in software engineering on the automatic detection of API documentation quality - which is now an absolute must due to the growing importance of APIs and software in our daily lives [54], [62]. The superior performance of our machine learning classifiers, in particular the deep learning model BERT, offers promise that we can now use such tools to automatically monitor and warn about API documentation quality in real-time. Software companies and open source community can leverage our developed model to analyze the quality of their API documentation. Software developers could save time by focusing on good quality API documentation

TABLE V
CLASS-WISE PERFORMANCE OF DEEP LEARNING MODELS

		Bloated				Lazy				Excess Struct				Tangled				Fragmented			
Feature	**Model**	A	P	R	F1	A	P	R	F1	A	P	R	F1	A	P	R	F1	A	P	R	F1
Word Embed	**Bi-LSTM**	.92	.92	.92	.91	.89	.90	.89	.90	.76	.72	.76	.73	.78	.74	.78	.74	.67	.64	.67	.63
	BERT	.93	.93	.93	**.93**	.97	.97	.97	**.97**	.76	.75	.76	**.76**	.83	.83	.83	**.83**	.75	.75	.75	**.75**

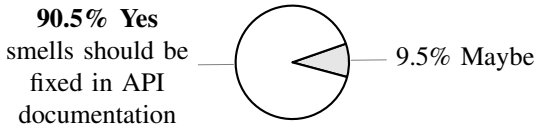

Fig. 17. Survey responses on whether the five documentation smells should be fixed to improve API documentation quality.

instead of the bad ones as detected by our model. Based on such real-time feedback, tools can be developed to improve the documentation quality by fixing the smells. Indeed, when we asked our survey participants (Section III) whether the five smells need to be fixed, more than 90% responded with a 'Yes', 9.5% with a 'Maybe', 0% with a 'No' (see Fig. 17).

Threats to Validity. *Internal validity* threats relate to authors' bias while conducting the analysis. We mitigated the bias in our benchmark creation process by taking agreement from 17 coders and co-authors and by consulting API documentation literature. The machine learning models are trained, tested, and reported using standard practices. There was no common data between the training and test set. *Construct validity* threats relate to the difficulty in finding data to create our catalog of smells. Our benchmark creation process was exhaustive, as we processed more than 29K unit examples from official documentation. *External validity* threats relate to the generalizability of our findings. We mitigated this threat by corroborating the five smells in our study with findings from state-of-the-art research in API documentation presentation and design problems. Our analysis focused on the validation and detection of five API documentation smells. Similar to code smell literature, additional documentation smells can be added into our catalog as we continue to research on this area.

VI. RELATED WORK

Related work is divided into **studies** on understanding (1) documentation problems and (2) how developers learn APIs using documentation, and developing **techniques** (3) to detect errors in documentation and (4) to create documentation.

Studies. Research shows that traditional Javadoc-type approaches to API official documentation are less useful than example-based documentation (e.g., minimal manual [13]) [68] Both code examples and textual description are required for better quality API documentation [19], [26], [49]. Depending of the types of API documentation, reability and understandability of the documentation can vary [77]. Broadly, problems in API official documentation can be about 'what' contents are documented and 'how' the contents are presented [6], [7], [61], [62], [81]. Literature in API documentation quality discussed four desired attributes of API documentation: completeness, consistency, usability and accessibility [77], [91]. Several studies show that external informal resources can be consulted to improve API official documentation [20], [34], [36], [50], [74], [82], [85]

The five documentation smells studied in this paper are taken from five commonly discussed API documentation design and presentation issues in literature [6], [81]. In contrast to the above papers that aim to understand API documentation problems, we focus on the development of techniques to automatically detect documentation smells.

Techniques. Tools and techniques are proposed to automatically add code examples and insights from external resources (e.g., online forums) into API official documentation [5], [72], [78]. Topic modeling is used to develop code books and to detect deficient documentation [12], [69], [70]. API official documentation and online forum data are analyzed together to recommend fixes API misuse scenarios [60]. The documentation of an API method can become obsolete/inconsistent due to evolution in source code [16], [84]. Several techniques are proposed to automatically detect code comment inconsistency [57], [75], [92]. A large body of research is devoted to automatically produce natural lanugage summary description of source code method [44], [47], [65], [71].

Unlike previous research, we focus on the detection of five API documentation smells that do not make a documentation inconsistent/incorrect, but nevertheless make the learning of the documentation difficult due to the underlying design/presentation issues. We advance state-of-the-art research on API documentation quality analysis by offering a benchmark of real-world examples of five documentation smells and a suite of techniques to automatically detect the smells.

VII. CONCLUSIONS

The learning of an API is challenging when the official documentation resources are of low quality. We identify five API documentation smells by consulting API documentation literature on API documentation design and presentation issues. We present a benchmark of 1,000 API documentation units with five smells in API official documentation. Feedback from 21 industrial software developers shows that the smells can negatively impact the productivity of the developers during API documentation usage. We develop a suite of machine learning classifiers to automatically detect the smells. The best performing classifier BERT, a deep learning model, achieves F1-scores of 0.75 - 0.97. The techniques can help automatically monitor and warn about API documentation quality.

REFERENCES

[1] *Javadoc SE 7*. https://docs.oracle.com/javase/7/docs/api/, 2020.

[2] M. Abidi, M. Grichi, F. Khomh, and Y. G. Guéhéneuc. Anti-patterns for multi-language systems. In *24th European Conference on Pattern Languages of Programs*, page Article No. 42, 2019.

[3] M. Abidi, M. Grichi, F. Khomh, and Y. G. Guéhéneuc. Code smells for multi-language systems. In *24th European Conference on Pattern Languages of Programs*, page Article No. 12, 2019.

[4] A. Adhikari, A. Ram, R. Tang, and J. Lin. Docbert: Bert for document classification. *arXiv preprint arXiv:1904.08398*, 2019.

[5] E. Aghajani, G. Bavota, M. Linares-Vásquez, and M. Lanza. Automated documentation of android apps. *IEEE Transactions on Software Engineering*, page 17, 2019.

[6] E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd. Software documentation: The practitioners' perspective. In *42nd International Conference on Software Engineering*, page 12, 2020.

[7] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza. Software documentation issues unveiled. In *41st International Conference on Software Engineering*, page 1199–1210, 2019.

[8] W. Alkhatib, C. Rensing, and J. Silberbauer. Multi-label text classification using semantic features and dimensionality reduction with autoencoders. In *International Conference on Language, Data and Knowledge*, pages 380–394. Springer, 2017.

[9] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[10] O. M. Bullock, D. Colón Amill, H. C. Shulman, and G. N. Dixon. Jargon as a barrier to effective science communication: Evidence from metacognition. *Public Understanding of Science*, 28(7):845–853, 2019.

[11] I. Cai. *Framework Documentation: How to document object-oriented frameworks. An Empirical Study*. PhD in Computer Sscience, University of Illinois at Urbana-Champaign, 2000.

[12] J. C. Campbell, C. Zhang, Z. Xu, A. Hindle, and J. Miller. Deficient documentation detection: A methodology to locate deficient project documentation using topic analysis. In *Proceedings of the 10th International Working Conference on Mining Software Repositories*, pages 57–60, 2013.

[13] J. M. Carroll, P. L. Smith-Kerker, J. R. Ford, and S. A. Mazur-Rimetz. The minimal manual. *Journal of Human-Computer Interaction*, 3(2):123–153, 1987.

[14] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[15] H. Cramér. *Mathematical methods of statistics*, volume 43. Princeton university press, 1999.

[16] B. Dagenais. *Analysis and Recommendations for Developer Learning Resources*. PhD in Computer Sscience, McGill University, 2012.

[17] B. Dagenais and M. P. Robillard. Using traceability links to recommend adaptive changes for documentation evolution. *IEEE Transactions on Software Engineering*, 40(11):1126–1146, 2014.

[18] A. C. de Carvalho and A. A. Freitas. A tutorial on multi-label classification techniques. In *Foundations of computational intelligence volume 5*, pages 177–195. Springer, 2009.

[19] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 68–75, 2005.

[20] F. Delfim and M. M. Klérisson Paixão, Damien Cassou. Redocumenting apis with crowd knowledge: a coverage analysis based on question types. *Journal of the Brazilian Computer Society*, 29(1), 2016.

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[22] S. Dumais et al. Using svms for text categorization. *IEEE Intelligent Systems*, 13(4):21–23, 1998.

[23] A. Elisseeff and J. Weston. A kernel method for multi-labelled classification. In *Advances in neural information processing systems*, pages 681–687, 2002.

[24] A. Fisher, C. Rudin, and F. Dominici. All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously. *Journal of Machine Learning Research*, 20(177):1–81, 2019.

[25] R. Flesch and A. J. Gould. *The art of readable writing*, volume 8. Harper New York, 1949.

[26] A. Forward and T. C. Lethbridge. The relevance of software documentation, tools and technologies: A survey. In *Proc. ACM Symposium on Document Engineering*, pages 26–33, 2002.

[27] G. Garousi, ahid Garousi-Yusifoğlu, G. Ruhe, J. Zhi, M. Moussavi, and B. Smith. Usage and usefulness of technical software documentation: An industrial case study. *Information and Software Technology*, 57:664–682, 2015.

[28] T. F. Gharib, M. B. Habib, and Z. T. Fayed. Arabic text classification using support vector machines. *Int. J. Comput. Their Appl.*, 16(4):192–199, 2009.

[29] S. González-Carvajal and E. C. Garrido-Merchán. Comparing bert against traditional machine learning text classification. *arXiv preprint arXiv:2005.13012*, 2020.

[30] A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural networks*, 18(5-6):602–610, 2005.

[31] Q. Gu, Z. Li, and J. Han. Correlated multi-label feature selection. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1087–1096, 2011.

[32] Z. S. Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[33] C.-W. Hsu, C.-C. Chang, C.-J. Lin, et al. A practical guide to support vector classification, 2003.

[34] H. Jiau and F.-P. Yang. Facing up to the inequality of crowdsourced api documentation. *ACM SIGSOFT Software Engineering Notes*, 37(1):1–9, 2012.

[35] T. Joachims. Text categorization with support vector machines: Learning with many relevant features. In *European conference on machine learning*, pages 137–142. Springer, 1998.

[36] D. Kavaler, D. Posnett, C. Gibler, H. Chen, P. Devanbu, and V. Filkov. Using and asking: Apis used in the android market and asked about in stackoverflow. In *In Proceedings of the INTERNATIONAL CONFERENCE ON SOCIAL INFORMATICS*, pages 405–418, 2013.

[37] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[38] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[39] X. Li, L. Bing, W. Zhang, and W. Lam. Exploiting bert for end-to-end aspect-based sentiment analysis. *arXiv preprint arXiv:1910.00883*, 2019.

[40] Y. Liu. Fine-tune bert for extractive summarization. *arXiv preprint arXiv:1903.10318*, 2019.

[41] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[42] H. V. D. Maij. A critical assessment of the minimalist approach to documentation. In *Proc. 10th ACM SIGDOC International Conference on Systems Documentation*, pages 7–17, 1992.

[43] C. D. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*. Cambridge Uni Press, 2009.

[44] P. W. McBurney and C. McMillan. Automatic documentation generation via source code summarization of method context. In *22nd International Conference on Program Comprehension*, pages 279 – 290, 2014.

[45] M. L. McHugh. Interrater reliability: the kappa statistic. *Biochemia medica: Biochemia medica*, 22(3):276–282, 2012.

[46] M. McTear, Z. Callejas, and D. Griol. Spoken language understanding. In *The Conversational Interface*, pages 161–185. Springer International Publishing, 2016.

[47] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for Java classes. In *Proceedings of the 21st IEEE International Conference on Program Comprehension*, pages 23–32, 2013.

[48] M. Munikar, S. Shakya, and A. Shrestha. Fine-grained sentiment classification using bert. In *2019 Artificial Intelligence for Transforming Business and Society (AITB)*, volume 1, pages 1–5. IEEE, 2019.

[49] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, and M. Gordon. What programmers really want: Results of a needs assessment for SDK documentation. In *Proc. 20th Annual International Conference on Computer Documentation*, pages 133–141, 2002.

[50] C. Parnin and C. Treude. Measuring api documentation on the web. In *Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering*, pages 25–30, 2011.

[51] V. L. Parsons. Stratified sampling. *Wiley StatsRef: Statistics Reference Online*, pages 1–11, 2014.

[52] Y. Peng, S. Yan, and Z. Lu. Transfer learning in biomedical natural language processing: An evaluation of bert and elmo on ten benchmarking datasets. *arXiv preprint arXiv:1906.05474*, 2019.

[53] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[54] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Prompter: Turning the IDE into a self-confident programming assistant. *Empirical Software Engineering*, 21(5):2190–2231, 2016.

[55] L. Prechelt. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks*, 11(4):761–767, 1998.

[56] L. Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998.

[57] F. Rabbi and M. S. Siddik. Detecting code comment inconsistency using siamese recurrent network. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 371–375, 2020.

[58] J. Read, B. Pfahringer, G. Holmes, and E. Frank. Classifier chains for multi-label classification. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 254–269. Springer, 2009.

[59] P. Refaeilzadeh, L. Tang, and H. Liu. Cross-validation. *Encyclopedia of database systems*, 5:532–538, 2009.

[60] X. Ren, J. Sun, Z. Xing, X. Xia, and J. Sun. Demystify official api usage directives with crowdsourced apimisuse scenarios, erroneous code examples and patches. In *42nd International Conference on Software Engineering*, page 12, 2020.

[61] M. P. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):26–34, 2009.

[62] M. P. Robillard and R. DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[63] L. Rosasco, E. D. Vito, A. Caponnetto, M. Piana, and A. Verri. Are loss functions all the same? *Neural Computation*, 16(5):1063–1076, 2004.

[64] M. B. Rosson, J. M. Carrol, and R. K. Bellamy. Smalltalk scaffolding: a case study of minimalist instruction. In *Proc. ACM SIGCHI Conf. on Human Factors in Computing Systems*, pages 423–430, 1990.

[65] A. M. S. Haiduc, J. Aponte. Supporting program comprehension with source code summarization. In *In Proceedings of the 32nd International Conference on Software Engineering*, pages 223–226, 2010.

[66] F. Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[67] K. Sechidis, G. Tsoumakas, and I. Vlahavas. On the stratification of multi-label data. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 145–158. Springer, 2011.

[68] F. Shull, F. Lanubile, and V. R. Basili. Investigating reading techniques for object-oriented framework learning. *IEEE Transactions on Software Engineering*, 26(11):1101–1118, 2000.

[69] L. Souza, E. Campos, , and M. Maia. On the extraction of cookbooks for apis from the crowd knowledge. In *Proceedings of the 28th Brazilian Symposium on Software Engineering*, pages 21–30, 2014.

[70] L. B. Souza, E. C. Campos, F. Madeiral, K. P. ao, A. M. Rocha, and M. de Almeida Maia. Bootstrapping cookbooks for apis from crowd knowledge on stack overflow. *Information and Software Technology*, 111:3749, 2019.

[71] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52, 2010.

[72] S. Subramanian, L. Inozemtseva, and R. Holmes. Live API documentation. In *Proceedings of 36th International Conference on Software Engineering*, pages 643–652, 2014.

[76] I. Tenney, D. Das, and E. Pavlick. Bert rediscovers the classical nlp pipeline. *arXiv preprint arXiv:1905.05950*, 2019.

[73] C. Sun, X. Qiu, Y. Xu, and X. Huang. How to fine-tune bert for text classification? In *China National Conference on Chinese Computational Linguistics*, pages 194–206. Springer, 2019.

[74] J. Sunshine, J. D. Herbsleb, , and J. Aldrich. Searching the state space: A qualitative study of api protocol usability. In *Proceedings of the International Conference on Program Comprehension*, pages 82–93, 2015.

[75] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *International Conference on Software Testing, Verification, and Validation*, pages 260 – 269, 2012.

[77] C. Treude, J. Middleton, and T. Atapattu. Beyond accuracy: Assessing software documentation quality. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering - Vision and Reflections Track*, page 4, 2020.

[78] C. Treude and M. P. Robillard. Augmenting api documentation with insights from stack overflow. In *Proc. IEEE 38th International Conference on Software Engineering*, pages 392–402, 2016.

[79] G. Tsoumakas and I. Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining (IJDWM)*, 3(3):1–13, 2007.

[80] G. Tsoumakas and M.-L. Zhang. Learning from multi-label data. 2009.

[81] G. Uddin and M. P. Robillard. How API documentation fails. *IEEE Softawre*, 32(4):76–83, 2015.

[82] W. Wang and M. W. Godfrey. Detecting api usage obstacles: A study of ios and android developer questions. In *In Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 61–64, 2013.

[83] C. wei Hsu, C. chung Chang, and C. jen Lin. A practical guide to support vector classification.

[84] F. Wen, C. Nagy, G. Bavota, and M. Lanza. A large-scale empirical study on code-comment inconsistencies. In *27th International Conference on Program Comprehension*, page 53–64, 2019.

[85] D. Yang, A. Hussain, and C. V. Lopes. From query to usable code: an analysis of stack overflow code snippets. In *In Proceedings of the 13th International Conference on Mining Software Repositories*, pages 391–402, 2016.

[86] Y. Yang and X. Liu. A re-examination of text categorization methods. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 42–49, 1999.

[87] B. Yekkehkhany, A. Safari, S. Homayouni, and M. Hasanlou. A comparison study of different kernel functions for svm-based classification of multi-temporal polarimetry sar data. *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 40(2):281, 2014.

[88] M.-L. Zhang and Z.-H. Zhou. Ml-knn: A lazy learning approach to multi-label learning. *Pattern recognition*, 40(7):2038–2048, 2007.

[89] M.-L. Zhang and Z.-H. Zhou. A review on multi-label learning algorithms. *IEEE transactions on knowledge and data engineering*, 26(8):1819–1837, 2013.

[90] W. Zhang, T. Yoshida, and X. Tang. Text classification based on multi-word with support vector machine. *Knowledge-Based Systems*, 21(8):879–886, 2008.

[91] J. Zhia, V. Garousi-Yusifoǵlubc, B. Sun, G. Garousi, S. Shahnewaz, and G. Ruhe. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software*, 99:175–198, 2015.

[92] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall. Analyzing apis documentation and code to detect directive defects. In *39th International Conference on Software Engineering*, pages 27 – 37, 2017.