An NLP-based Tool for Software Artifacts Analysis

Andrea Di Sorbo*, Corrado A. Visaggio*, Massimiliano Di Penta*,
Gerardo Canfora*, Sebastiano Panichella†
*University of Sannio, Italy
†Zurich University of Applied Sciences, Switzerland
{disorbo, visaggio, dipenta, canfora}@unisannio.it, panc@zhaw.ch

***Abstract*—Software developers rely on various repositories and communication channels to exchange relevant information about their ongoing tasks and the status of overall project progress. In this context, semi-structured and unstructured software artifacts have been leveraged by researchers to build recommender systems aimed at supporting developers in different tasks, such as transforming user feedback in maintenance and evolution tasks, suggesting experts, or generating software documentation. More specifically, Natural Language (NL) parsing techniques have been successfully leveraged to automatically identify (or extract) the relevant information embedded in unstructured software artifacts. However, such techniques require the manual identification of patterns to be used for classification purposes. To reduce such a manual effort, we propose an NL parsing-based tool for software artifacts analysis named NEON that can automate the mining of such rules, minimizing the manual effort of developers and researchers. Through a small study involving human subjects with NL processing and parsing expertise, we assess the performance of NEON in identifying rules useful to classify app reviews for software maintenance purposes. Our results show that more than one-third of the rules inferred by NEON are relevant for the proposed task.**
Demo webpage: https://github.com/adisorbo/NEON_tool

***Index Terms*—Unstructured Data Mining, Natural Language Parsing, Software maintenance and evolution**

I. Introduction

Software developers intensively rely on of software repositories [1], [9], [29] and written communication channels [7], [24] for exchanging relevant information about the ongoing development tasks and the status of overall project progress. Development teams contributing to the software repositories [1], [9], [29] are globally distributed [7]: they often (i) communicate in unstructured form through mailing lists [5] and chats [2], (ii) discuss problems and related solutions over issue trackers [20], [21] and Question & Answer forums (*e.g.,* Stack Overflow) [31], and (iii) acquire feedback posted by users on dedicated channels, as in the case of mobile apps (*e.g.,* app stores) [14]. Therefore, valuable information is disseminated (and scattered) over semi-structured and unstructured software artifacts [7], [9], [24].

Semi-structured and unstructured software artifacts have been leveraged by researchers to build automated approaches aimed at supporting developers dealing with several software engineering (SE) tasks [1], [3], [4], [10], [20], [30]. In this context, the information embedded in unstructured software artifacts has been analyzed with approaches relying on information retrieval models (*e.g.,* VSM [6], LSI [13], or LDA [8]). However, such approaches treat unstructured text as a bag of word (or, in the best case, infer latent topics/concepts from them). This makes them ineffective when a deeper level of detail in the text analysis and interpretation is needed [16]–[18].

To overcome the limitations of approaches based on bag-of-words representations, and to automatically identify textual patterns in informal software documents that are relevant to different evolution tasks, in previous work we proposed an approach named *intention mining* [16], which leverages Natural Language (NL) parsing techniques. Such an approach has been successfully applied for classification [17], [25], [27], summarization [15], [28], or quality assessment [11], [32] purposes, where it turned out to be more accurate than models based on bag-of-words representations.

The main challenge of leveraging approaches based on NL parsing techniques is that they require the manual definition of sets of NL rules [15], [16], [25] to recognize natural language patterns. This manual task has proven to be effort-intensive and error-prone, since it requires specific domain-knowledge in natural language parsing [18]. For this reason, recent research [19] attempted to automate and generalize intention mining by experimenting with deep learning-based methods. However, while deep learning-based approaches avoid the manual tagging of textual information, they hampers the interpretability of the results, making it difficult to understand the specific linguistic patterns that have been identified. Such patterns are indeed crucial to support several tasks, *e.g.,* automated detection of inconsistencies between source code and API documents [32].

To reduce the manual work required for manually identifying patterns in textual documents, in this paper we present a tool named NEON (Nlp-based softwarE dOcumentation aNalyzer), able to automatically mine recurring patterns from software informal documents. NEON, by implementing an approach that we presented in previous work [18], inspects natural language sentences received as input and identifies the recurrent grammatical structures appearing in them. NEON is able to output a list of syntactical rules and each rule has the purpose of detecting a specific natural language pattern. In previous work [18], we showed that NEON allows for saving more than 70% of time otherwise spent in the manual identification and definition of NL rules.

Through a small study involving human subjects with NL parsing expertise, we assess the performance of NEON in identifying rules relevant for classifying user intents in the

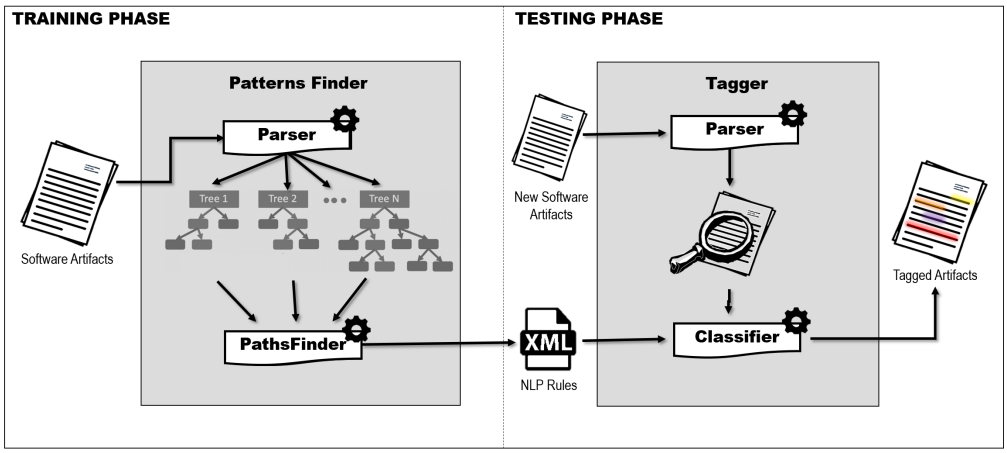

Fig. 1. The NEON's architecture.

context of app reviews. Such an experiment demonstrates that NEON can efficiently automate the detection of rules useful for intention mining.

II. The NEON's Approach and Tool

NEON has the goal of minimizing the manual effort to gather the relevant rules (or patterns) for identifying (or extracting) the information of interest embedded in software informal documents.

Figure 1 shows the high-level architecture of NEON. The analysis of software documentation is performed through two main phases: *i.e.,* the *training phase* and the *testing phase*. In the training phase, a set of software artifacts of a specific type (*e.g.,* app reviews or issue reports) is inspected to identify rules for capturing recurrent NL patterns [16]. In the testing phase, the inferred rules are leveraged to recognize the information of interest in a different corpus of software artifacts. Specifically, NEON encompasses two independent software components that allow automating both phases: (i) the `Patterns Finder`, and (ii) the `Tagger`.

The `Patterns Finder` automatically identifies relevant rules for detecting NL patterns occurring in a set of unstructured texts (*i.e., training set*). The `Tagger` exploits such rules, stored in an XML file, to automatically label the relevant pieces of information appearing in the documents whose sentences need to be classified (*i.e.,* the *test set*).

The `Patterns Finder` is composed of two main software modules: the `Parser` and the `PathsFinder`. The `Parser` (i) preprocesses the text through sentence splitting and tokenization, and (ii) generates the semantic graph[1] of each sentence present in the input text. Such semantic graphs are then provided to the `PathsFinder` module, which implements the approach presented in our previous work [18]. More specifically, given two generic semantic graphs, G_1 and G_2, that share a common grammatical structure, the `PathsFinder` (i) selects the nodes of the verb and noun types from both graphs, (ii) identifies the pairs of similar nodes

[1]https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/semgraph/SemanticGraph.html

```
<NLP_heuristic>
    <sentence type="declarative"/>
    <type>aux/neg/dobj/nsubj</type>
    <text>[something] [auxiliary] not open [something].</
        text>
    <conditions>
        <condition>aux.governor="open"</condition>
        <condition>aux.governor=neg.governor</condition>
        <condition>neg.governor=dobj.governor</condition>
        <condition>dobj.governor=nsubj.governor</condition>
    </conditions>
    <sentence_class>PROBLEM DISCOVERY</sentence_class>
</NLP_heuristic>
```

Fig. 2. Example of NEON rule in XML

(*e.g.,* nodes containing the same or *similar* lemmas) present in G_1 and G_2, and (iii) analyzes the children and the labeled arcs outgoing from the pairs of similar nodes, to specify the rule aimed at recognizing the common grammatical structure. In particular, a rule able to recognize a grammatical structure in a generic sentence is defined through the XML grammar illustrated in Figure 2. The `<conditions>` represent the core part of the rule and they describe the path to be searched in the Stanford Dependencies (SD) representation [12] of a sentence under analysis. In general, the `PathsFinder` is able to automatically infer two types of conditions:

1. **Conditions of type 1**, in which specific lemmas have to appear in precise grammatical roles (*e.g.,* `aux.governor=``open"`).
2. **Conditions of type 2**, which localize typed dependencies sharing particular regents (*i.e.,* governors) or arguments (dependents) or typed dependencies whose dependent represents the governor of a second dependency (*e.g.,* `aux.governor=neg.governor`).

It is worth noting that the rule set can evolve incrementally. Indeed, when inspecting new artifacts through the `Patterns Finder`, new rules might be discovered. Hence, the tool provides the ability to integrate the newly identified rules into the existing rule set stored in a specific XML file.

The `Tagger` generalizes the approach used in previous NLP-based classification tools [17], [26]. It is composed of two main software modules: the `Parser` and the

`Classifier`. By relying on the Stanford CoreNLP API [23], the `Parser` performs sentence splitting, tokenization, and, for each sentence, it generates the Stanford Dependencies (SD) representation. The `Classifier` leverages such a representation and the set of rules (defined in an XML file) to detect the presence of text structures that match one (or more) of the defined rules. Whenever a rule is matched, the `Classifier` creates a result instance, together with information about (i) the sentence in the text that matched the rule, (ii) the sentence class (on the basis of the matched rule), and (iii) the specific rule matched. The `Classifier` labels only the sentences that match known rules, assuming that all other sentences are too generic or have negligible contents [17].

III. Using NEON

NEON provides an intuitive Graphical User Interface (GUI). It can also be used programmatically as a Java library. Here, we mainly describe the features of the GUI, while code examples on how to use the `NEON.jar` library are reported in our online repository[2]. As shown in previous work [18], NEON can be useful to analyze a variety of software artifacts (*e.g.,* issue reports, development emails, etc.). For simplicity, in this section, we focus on a specific usage scenario, *i.e.,* app review analysis. Given a set of app reviews, NEON helps researchers (or practitioners) to (i) discover, during the *training phase*, the NL patterns (and the respective detection rules) occurring in these artifacts and connected with information of interest (*e.g., feature requests*), and (ii) leverage the inferred rules to automatically recognize/extract such information when inspecting further app reviews (*i.e., testing phase*).

Training phase. By selecting the `Add new heuristics...` option from the `File` menu, a new window will open. This new window includes a text area in which the user can paste the training documents (or load them by selecting the `Open text file...` option from the `File` menu). The `Find Common Patterns` button located below the text area can be used to start the analysis of the training app reviews. At the end of the analysis, an interactive table containing all the inferred rules is generated and visualized by NEON. As illustrated in Figure 3, each row in the table reports the conditions and the details related to a specific rule. By inspecting such a table, the researchers (or practitioners) can easily analyze, modify, and decide if each given rule is relevant for a given purpose (by acting on a checkbox). The table also offers the possibility to specify the category to which each relevant rule is related (*i.e.,* the sentences matching the given rule will fall within the assigned category). For instance, the table in Figure 3 shows that the selected rules (useful for detecting patterns like *"It lacks"* or *"I would like"*) can be later used to identify app reviews requesting new features or enhancements. Once finished reviewing the interactive table, by clicking on the `Add Selected Patterns` button, all the rules marked as relevant are stored in an output XML file specified by the end-user. This feature fosters the reuse of knowledge, providing end-users with the opportunity of sharing and using sets of rules extracted for different purposes (or from different documents).

Testing phase. By acting on the GUI's main window, the user can classify relevant sentences (*e.g., feature requests*) contained in app reviews different from the ones used for training. Such documents have to be imported in the text area of the GUI (by selecting the `Open text file...` option from the `File` menu). By pressing the `Classify` button and selecting the desired XML file containing the set of rules enabling the classification, the *testing phase* is triggered. NEON will highlight all the recognized sentences (*i.e.,* containing one or more relevant NL patterns) using different colors for different categories. The classification results can be exported (by selecting the `Export results...` option from the `File` menu) in an XML file for further analysis.

[2]https://github.com/adisorbo/NEON_tool

IV. Evaluation

In addition to the empirical evaluation reported in our research paper [18], we performed a small study, whose *goal* is to assess the NEON's capability of identifying rules useful to automatically classify app software documents (in particular, app reviews) along two categories, *i.e., feature request*, and *problem discovery*. The study *context* consists of subjects, *i.e.,* three participants, and objects, *i.e.,* 100 app reviews. The three subjects are (i) one professional software engineer (*Subject 1*), (ii) one software engineering master student (*Subject2*), and (iii) an author of the paper (*Subject 3*). All of them had prior knowledge about NL processing and parsing.

To identify the study objects, we leveraged a dataset from previous work [25], which comprises 1,390 mobile app review sentences, and each sentence is labeled with one of the *feature request, problem discovery, information giving*, and *information seeking* categories. In particular, we sampled 100 sentences from the aforementioned dataset, 50 of them belonging to the *feature request* category and the remaining 50 sentences falling in the *problem discovery* category. The choice of only sampling 100 sentences was motivated by the need for limiting the manual task duration (*i.e.,* three hours at maximum), as the human raters involved in our study had limited time availability, as well as to mitigate fatigue effects. Moreover, we decided to only focus on documents of the *feature request* or *problem discovery* categories, as software engineers are more likely familiar with phrases of these types, often analyzed during software maintenance tasks [15].

We processed all the 100 sentences in the sample with NEON, and obtained a list of 241 candidate grammatical rules. We asked *Subject 1* and *Subject 2* to independently inspect the candidate rules provided by NEON and judge whether each rule was relevant (or not) for identifying sentences belonging to one of the two categories (*i.e., feature request* or *problem discovery*). Table I reports the results of the manual inspection performed by the raters involved in our study.

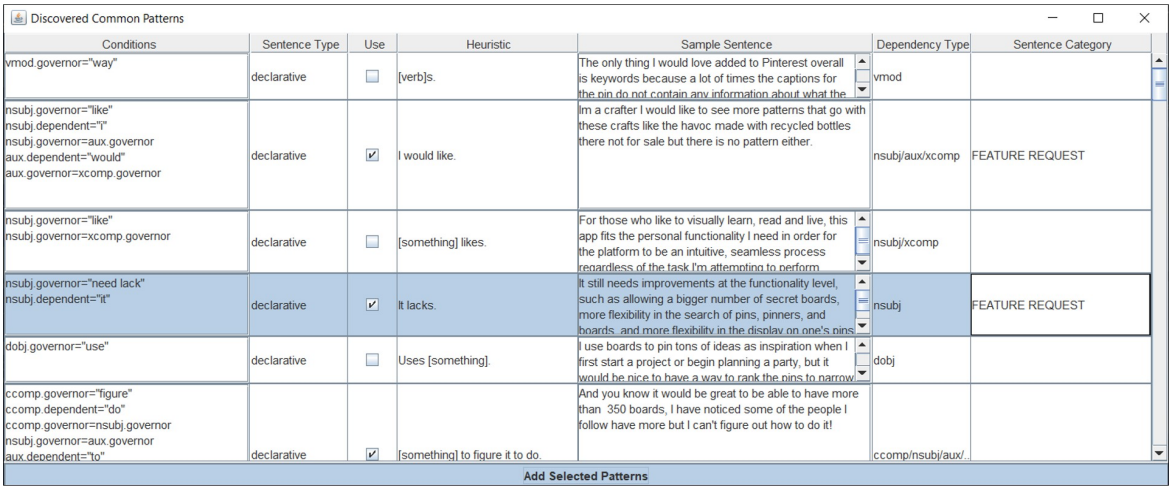

Fig. 3. The interactive table for inspecting the results of the rule mining process

TABLE I
RESULTS OF THE RULES' RELEVANCE ASSESSMENT TASK

Subject 1	Subject 2	Subject 3	# Patterns
yes	yes	-	52
yes	no	yes	15
yes	no	no	8
no	yes	yes	17
no	yes	no	9
no	no	-	140

Both initial raters marked as relevant 52 of the rules recommended by NEON, while they agreed on the irrelevance of 140 items. However, there were conflicts in 49 cases. Indeed, the inter-rater agreement (Cohen's Kappa) was $k = 0.531$ (192 items out of a total of 241). According to the guidelines in [22], a moderate agreement between the two validators was achieved. We decided to involve a third annotator for a further check. The third annotator (*i.e., Subject 3*) was asked to express her independent judgment on the relevance of the rules provided by NEON in all the 49 cases in which a disagreement between the two initial raters was observed. Specifically, the third annotator independently marked as relevant 32 out of the 49 inspected rules. Some examples of relevant rules that NEON was able to automatically mine from unstructured texts in our sample are the following: *[something] should have, [something] is missing, [something] [auxiliary] use [something], problem with [something] is, it crashes, [something] needs to [verb], [auxiliary] fix [something], [something] fails to [verb]*. These patterns are very similar to those manually identified in our previous work [25].

After the finalization of the evaluation process, we can conclude that more than one third of the rules (*i.e.,* 84 patterns out of a total of 241) automatically extracted and recommended by NEON were judged useful by at least two out of three human validators experienced in natural language parsing. Furthermore, as shown in our previous work [18], NEON reduces over 70% the time spent for the identification and definition of NL rules.

V. CONCLUSION

In this paper, we presented NEON, a tool aimed at (i) automatically inferring rules for identifying natural language patterns in software artifacts, and (ii) leverage such rules for information classification (or extraction) purposes. A small study involving real-world app reviews and human subjects experienced in natural language parsing demonstrated that more than one third (*i.e.,* 35%) of the rules inferred by our tool are relevant for identifying app review sentences reporting bugs or requesting enhancements. In previous work [18], we also demonstrated that NEON is time-saving and useful for inferring rules from different types of software artifacts (*e.g.,* development emails, and issue reports). However, the effectiveness of NEON might degrade when dealing with sentences containing mixtures of code elements and natural language [27] or incomplete sentences (*e.g.,* commit messages, chats).

NEON is designed to be used by both researchers and practitioners in a wide set of software engineering-related contexts. For instance, it can be leveraged to develop (or improve) recommender systems supporting specific software engineering tasks such as requirements elicitation, issue management, task prioritization, and artifacts analysis. NEON's ability to identify (and extract) precise information embedded in different kinds of sources can indeed help developers and researchers in improving the effectiveness of systems that require the analysis of semi-structured (*e.g.,* code comments, code documentation, etc.) and unstructured artifacts, *e.g.,* developer communications, issue- or vulnerability-related documents, Q&A posts, or user feedback.

ACKNOWLEDGEMENTS

We thank the participants in our experiment. We thank Prof. Dr. Harald Gall for supporting this project and providing lab facilities for the development of this tool. Sebastiano Panichella gratefully acknowledges the Innosuisse support for the project *ARIES* (Exploiting User Journeys for Supporting Mobility as a Service Platforms), Project No.45548.1.

REFERENCES

[1] C. V. Alexandru, S. Panichella, S. Proksch, and H. C. Gall. Redundancy-free analysis of multi-revision software artifacts. *Empir. Softw. Eng.*, 24(1):332–380, 2019.

[2] R. Alkadhi, J. O. Johanssen, E. Guzman, and B. Bruegge. REACT: an approach for capturing rationale in chat messages. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, pages 175–180, 2017.

[3] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y. Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering, CASCON 2018, Markham, Ontario, Canada, October 29-31, 2018*, pages 2–16, 2018.

[4] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 361–370, 2006.

[5] A. Bacchelli, T. D. Sasso, M. D'Ambros, and M. Lanza. Content classification of development emails. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 375–385, 2012.

[6] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[7] A. Begel and N. Nagappan. Global software development: Who does it? In *3rd IEEE International Conference on Global Software Engineering, ICGSE 2008, Bangalore, India, 17-20 August, 2008*, pages 195–199, 2008.

[8] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *The Journal of Machine Learning Research*, 3:993–1022, 2003.

[9] C. Brown and C. Parnin. Understanding the impact of github suggested changes on recommendations between developers. In P. Devanbu, M. B. Cohen, and T. Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1065–1076. ACM, 2020.

[10] G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. Who is going to mentor newcomers in open source projects? In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 44, 2012.

[11] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. D. Penta, A. Marcus, G. Bavota, and V. Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 396–407, 2017.

[12] M. de Marneffe and C. D. Manning. The stanford typed dependencies representation. In *Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation@COLING 2008, Manchester, UK, August 23, 2008*, pages 1–8, 2008.

[13] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[14] A. Di Sorbo, G. Grano, C. A. Visaggio, and S. Panichella. Investigating the criticality of user-reported issues through their relations with app rating. *J. Softw. Evol. Process.*, 33(3), 2021.

[15] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall. What would users change in my app? Summarizing app reviews for recommending software changes. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 499–510, 2016.

[16] A. Di Sorbo, S. Panichella, C. A. Visaggio, M. Di Penta, G. Canfora, and H. C. Gall. Development emails content analyzer: Intention mining in developer discussions (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 12–23, 2015.

[17] A. Di Sorbo, S. Panichella, C. A. Visaggio, M. Di Penta, G. Canfora, and H. C. Gall. DECA: development emails content analyzer. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 641–644, 2016.

[18] A. Di Sorbo, S. Panichella, C. A. Visaggio, M. Di Penta, G. Canfora, and H. C. Gall. Exploiting natural language structures in software informal documentation. *IEEE Trans. Software Eng.*, pages 1–1, 2019.

[19] Q. Huang, X. Xia, D. Lo, and G. C. Murphy. Automating intention mining. *IEEE Trans. Software Eng.*, 46(10):1098–1119, 2020.

[20] 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 2019, Cleveland, OH, USA, September 29 - October 4, 2019*, pages 406–409, 2019.

[21] R. Kallis, A. D. Sorbo, G. Canfora, and S. Panichella. Predicting issue types on github. *Sci. Comput. Program.*, 205:102598, 2021.

[22] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

[23] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, System Demonstrations*, pages 55–60, 2014.

[24] S. Panichella, G. Bavota, M. D. Penta, G. Canfora, and G. Antoniol. How developers' collaborations identified from different sources tell us about code changes. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 251–260. IEEE Computer Society, 2014.

[25] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In R. Koschke, J. Krinke, and M. P. Robillard, editors, *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 281–290. IEEE Computer Society, 2015.

[26] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall. Ardoc: app reviews development oriented classifier. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 1023–1027, 2016.

[27] P. Rani, S. Panichella, M. Leuenberger, A. Di Sorbo, and O. Nierstrasz. How to identify class comment types? a multi-language approach for class comment classification. *J. Syst. Softw.*, 181:111047, 2021.

[28] E. R. Russo, A. Di Sorbo, C. A. Visaggio, and G. Canfora. Summarizing vulnerabilities' descriptions to support experts during vulnerability assessment activities. *J. Syst. Softw.*, 156:84–99, 2019.

[29] V. N. Subramanian. An empirical study of the first contributions of developers to open source projects on github. In G. Rothermel and D. Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Companion Volume, Seoul, South Korea, 27 June - 19 July, 2020*, pages 116–118. ACM, 2020.

[30] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora. CODES: mining source code descriptions from developers discussions. In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pages 106–109, 2014.

[31] S. Wang, D. Lo, and L. Jiang. An empirical study on developer interactions in stackoverflow. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 1019–1024, 2013.

[32] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. C. Gall. Analyzing apis documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 27–37, 2017.