Exploiting Natural Language Structures in Software Informal Documentation

Andrea Di Sorbo, Sebastiano Panichella,
Corrado A. Visaggio, Massimiliano Di Penta, Gerardo Canfora, Harald C. Gall

Abstract—Communication means, such as issue trackers, mailing lists, Q&A forums, and app reviews, are premier means of collaboration among developers, and between developers and end-users. Analyzing such sources of information is crucial to build recommenders for developers, for example suggesting experts, re-documenting source code, or transforming user feedback in maintenance and evolution strategies for developers. To ease this analysis, in previous work we proposed DECA (Development Emails Content Analyzer), a tool based on Natural Language Parsing that classifies with high precision development emails' fragments according to their purpose. However, DECA has to be trained through a manual tagging of relevant patterns, which is often effort-intensive, error-prone and requires specific expertise in natural language parsing. In this paper, we first show, with a study involving Master's and Ph.D. students, the extent to which producing rules for identifying such patterns requires effort, depending on the nature and complexity of patterns. Then, we propose an approach, named NEON (Nlp-based softwarE dOcumentation aNalyzer), that automatically mines such rules, minimizing the manual effort. We assess the performances of NEON in the analysis and classification of mobile app reviews, developers discussions, and issues. NEON simplifies the patterns' identification and rules' definition processes, allowing a savings of more than 70% of the time otherwise spent on performing such activities manually. Results also show that NEON-generated rules are close to the manually identified ones, achieving comparable recall.

Index Terms—Mining Unstructured Data, Natural Language Processing, Empirical Study

1 Introduction

Nowadays the use of written unstructured communication channels is assuming increasing importance in software development [5], [30]. On the one side developers, especially when they are globally distributed [6], communicate through mailing lists, chats, and issue trackers [32]. Also, they discuss recurring problems and seek solutions on Questions & Answers Forums such as Stack Overflow [12], [43]. On the other side, end-users provide feedback and suggestions for improving software applications by means of user review posted on stores, such as those of mobile apps (*e.g.,* Apple Store, Google Play, or Microsoft Store) [17].

These communication channels are a precious source of information for developers, for example, to understand the rationale of development choices [3], [6], [33], to identify experts [2] or mentors [8] based on previous change or bug-fixes, or to decide what features should be included in the next version of an application [17].

Researchers have proposed several approaches to extract and classify pieces of information from unstructured communication channels to support software engineering (SE) tasks. For example, Antoniol *et al.* [1] used machine learning techniques to classify issue reports. Bacchelli *et al.* [3] proposed the use of island parsers to separate different elements of emails (*e.g.,* natural language text, source code, or logs), while Cerulo *et al.* [10] leveraged, for the same purposes, Hidden Markov Models.

The main limitation of most proposed approaches is that they classify textual content by mainly considering the associated word frequencies, thus analyzing or measuring the information relying on information retrieval models such as Vector Space Models [4], Latent Semantic Indexing [16], or Latent Dirichlet Allocation (LDA) [7]. With these techniques, textual information tends to be treated as a bag of words. While this is a simple and relatively effective strategy, it suffers from weaknesses and may turn out to be inadequate in several circumstances. For example, consider the following two sentences:

1) *We could use a leaky bucket algorithm to limit the bandwidth.*
2) *The leaky bucket algorithm fails in limiting the bandwidth.*

A topic analysis will reveal that these two sentences are likely to discuss the same topics: *"leaky bucket algorithm"* and *"bandwidth"*. However, these two sentences have completely different *intentions*: in the sentence (1) the writer proposes a solution for a specific problem, while in the sentence (2) the writer points out a problem. Thus, they could be useful in different contexts. This example highlights that understanding the *intentions* in developers' communication could add valuable information for guiding developers in detecting text content useful to accomplish different and specific maintenance and evolution tasks.

To overcome the limitations of approaches based on information retrieval, we proposed an approach named DECA (Development Email Content Analyzer) [19] that leverages natural language syntactical patterns contained in

- *A. Di Sorbo, C. A. Visaggio, M. Di Penta and G. Canfora are with the Department of Engineering, University of Sannio, Benevento, Italy. E-mail: {disorbo, visaggio, dipenta, canfora}@unisannio.it*
- *S. Panichella is with Zurich University of Applied Science (ZHAW), Switzerland. E-mail: panc@zhaw.ch*
- *H.C. Gall is with the Department of Informatics, University of Zurich, Switzerland. E-mail: gall@ifi.uzh.ch*

development emails, and use such patterns to classify the sentences' intent. A similar approach, by Panichella *et al.* [34], combined with other features (*e.g.*, sentiment) has been also used to classify mobile app reviews for the purpose of better release planning. Chaparro *et al.* [11] combined the extraction of natural language patterns with machine learning to identify missing elements in bug reports *i.e.*, observed behavior, expected behavior or steps to reproduce.

The aforementioned approaches turned out to be more accurate than approaches based on bag-of-words representations. Also, they had *the advantage of identifying the particular linguistic pattern used in a specific text fragment*. This not only makes it possible to clearly classify the intent of the fragment but also paves the way towards other applications, *e.g.*, automated support for error/request reporting.

The main challenge of such approaches based on natural language patterns is that they require the definition of a set of manually-labeled patterns from known sentences [17], [19], [34]. This task can be effort-intensive, error-prone, and requires that taggers have specific skills, *i.e.*, the ability to recognize linguistic patterns in text sentences. Moreover, the difficulty in recognizing such patterns may vary depending on the length and complexity of the sentence.

To have a better understanding of the effort needed by humans to annotate patterns, in this paper we conduct a study aimed at empirically investigating the difficulties that may arise during the patterns' identification process (RQ1). The study involves 8 Master's students and 2 Ph.D. students, who have been asked to define patterns and related heuristics aimed at recognizing the intent of sentences we provided them. Results of the study indicate that, while in some cases the heuristics are relatively simple to identify, the required effort increases proportionally to the number and types of conditions that have to be considered in the heuristics' definition.

To reduce the effort associated with the tedious manual extraction of linguistic patterns, we propose to investigate approaches to automatically infer patterns in software informal documentation, in order to support processes aimed at (i) recognizing and classifying the intent in different kinds of informal documents, and (ii) extracting useful fragments for supporting specific tasks from these kinds of documents. To achieve this goal we formulated the following research question:

Can an automated approach assist researchers and practitioners in identifying recurrent linguistic patterns appearing in software informal documentation?

We present an approach, named NEON (Nlp-based softwarE dOcumentation aNalyzer), to automatically mine patterns from software informal documents. NEON analyses the syntactical information of natural languages sentences received as input, with the purpose of identifying recurrent grammatical structures appearing in the input texts. The output of NEON is a list of heuristics aimed at automatically detecting all the identified patterns.

To assess the effectiveness of NEON we carry out three different studies. The first one, conducted on a data collection of mobile app reviews [34], is aimed at evaluating the time-saving capability of NEON when used for extracting patterns useful for classification purposes (RQ2). The second study, conducted on development emails [19], is aimed at evaluating (i) the classification performances of the patterns inferred through our automated approach ($RQ3_a$), and (ii) the extent to which the automatically extracted patterns are similar to those extracted by developers ($RQ3_b$). The last study, conducted on a dataset containing descriptions of bugs mined from GitHub, is aimed at demonstrating the practical usefulness of the proposed approach for the SE research (RQ4). Figure 1 illustrates the dimensions investigated in each conducted study. On the one hand, results of the studies indicate that NEON allows saving more than 70% of the time otherwise spent in manually mining patterns useful for identifying relevant information in software informal documents, achieving nearly the same performance of DECA in terms of recall, while it is less precise (20% less) in terms of precision. On the other hand, the identified patterns are very similar to the ones manually identified, highlighting how an automated inference of intent classification patterns is, indeed, possible.

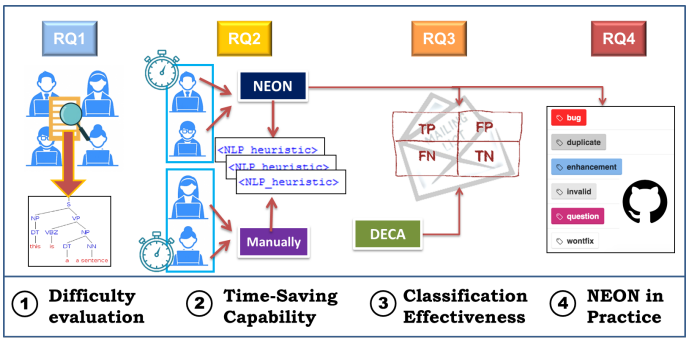


Fig. 1. Study design

Paper replication package. We make publicly available a replication package[1] with (i) the material and the working datasets of all our studies, (ii) the complete results of the study with students, and (iii) raw data (for replication purposes and to support future studies).

Paper structure. Section 2 reports the study we performed to investigate the difficulties human experts encounter when identifying patterns to employ for the classification of the development intent. Then, Section 3 describes NEON, the approach we proposed to automatically identify relevant linguistic patterns for intent classification. Section 4 reports the studies performed to evaluate NEON, highlighting its practical usefulness in a specific SE tasks. Section 5 discusses the related literature, while Section 6 concludes the paper.

2 Problem: Discovering Recurrent Language Patterns

In our previous work [19], we showed that developers use recurrent linguistic patterns to express their *intents*, such as asking/providing helps, proposing a new feature or reporting/discussing a bug. Such recurrent patterns can be automatically identified through the definition of NLP heuristics relying on the analysis of texts' grammatical structures. Specifically, each NLP heuristic is a collection of

1. https://github.com/adisorbo/NEONReplicationPackage

rules able to detect a specific predicate-argument structure in natural language sentences that evokes or suggests the intent of the writer [19]. Once heuristics have been defined, they can be used for classification purposes. Their accuracy is reasonably high (about 79%) and better than machine learning approaches [19].

The main limitation of such an approach is that the identification of recurring patterns and its implementation as a heuristic for intent classification require the manual analysis of a large number of sentences, as well as the availability of suitable expertise in natural language analysis. This is because the formalization of an NLP heuristic involves the following activities: (i) the identification of the relevant details that make a grammatical structure recognizable, (ii) the generalization of unnecessarily complex structures, and (iii) the filtering of the grammatical structures not providing any useful information for the linguistic pattern identification.

Our main aim is to study more in depth the processes of language patterns identification and heuristic definition, to find a possible solution to accelerate these processes.

With the *goal* of studying the factors affecting the language patterns identification and heuristics definition processes, and therefore understanding the difficulties humans encounter in such a process, we conduct an empirical study aimed at answering the following research question:

RQ1: *Is it possible to estimate the effort required for identifying/defining an NLP heuristic?*

In Section 2.1 we describe the design of the study we conducted to answer RQ1 while Section 2.2 reports the results. Section 2.3 discusses the threats that could affect the achieved results.

2.1 Study design

To answer RQ1, we conducted a study involving 10 participants: (i) 8 Master's students in Computer Engineering, who have all attended courses in software engineering and software evolution, and (ii) 2 Ph.D. students in Software Engineering. All the participants are familiar — from specialized courses they attended — with the Stanford Typed Dependencies (SD) representation [15]. The SD representation was designed to provide a simple description of the grammatical relationships in a sentence. Each typed dependency is a binary relation; it represents the grammatical relation existing between two words of the sentence: a governor (also known as regent or head) and a dependent. These dependencies map onto a directed graph representation, in which words in the sentence are nodes in the graph, and grammatical relations are edge labels [15]. Moreover, during a training session, the study participants have been introduced to the ways in which NLP heuristics can be defined to automatically recognize linguistic patterns. In particular, an NLP heuristic able to recognize a particular path in the Stanford Typed Dependencies tree of a generic sentence can be defined through the XML grammar illustrated in Listing 1. The `<conditions>` represent the core part of the heuristic (*i.e.*, they describe the path to be searched in the SD representation of the sentence under analysis).

In general, three types of conditions can be defined through our XML grammar:

Listing 1. Example of a NLP heuristic's definition

```
<NLP_heuristic>
  <sentence type=''declarative''/>
  <type>nsubj/aux/dobj</type>
  <text>[someone] can run/launch [something]</text>
  <conditions>
    <condition>nsubj.governor=''run launch''</condition>
    <condition>nsubj.governor=aux.governor</condition>
    <condition>aux.dependent=''can could''</condition>
    <condition>not:aux.governor=neg.governor</condition>
    <condition>aux.governor=dobj.governor</condition>
  </conditions>
  <sentence_class>INFORMATION GIVING</sentence_class>
</NLP_heuristic>
```

1. **Conditions of type 1**, in which specific lemmas have to appear in precise grammatical roles (*e.g.*, `nsubj.governor=''need require''`.
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.*, `nsubj.governor=aux.governor`).
3. **Conditions of type 3**, which establish that a particular typed dependency has (i) to not present a specific lemma in the governor or dependent role, or (i) to not expose a particular relation with a second typed dependency (*e.g.*, `not:aux.dependent=''could''` or `not:aux.governor=neg.governor`).

We performed a preliminary session where we talked to the participants about the three types of conditions. This was relevant to clarify any questions raised by the participants. After such a session, each participant was requested to practice, for a period of two weeks, his/her ability in defining NLP heuristics using the XML grammar. The exercises were related to different kinds of natural language sentences from social media posts, including app user reviews, issues, malware descriptions, and software vulnerability descriptions.

Afterward, the participants have been divided into two groups, Group 1 and Group 2, each one composed of 5 subjects. In order to balance the abilities of the two groups, we assigned one Ph.D. student per group. Every participant was asked to complete three tasks having similar difficulties.

In each task, 3 sentences along with their Stanford Typed Dependencies representation were shown and participants were required to define an NLP heuristic able to automatically recognize the three sentences or a subset of them (for further details please refer to our replication package). Once defined the required heuristic, participants were asked to provide feedback about the difficulty encountered in defining the heuristic (for this purpose, a Likert scale ranging from *"Very Low"* to *"Very High"* has been used) and the time spent for the heuristic's definition has been recorded.

Tasks A, B, and C were assigned to participants belonging to Group 1, while the remaining tasks (*i.e.*, Tasks D, E, and F) were allotted to subjects of Group 2.

For each participant and each task, we collected:

- The *NLP heuristic* defined in order to complete the task. This heuristic has been considered *suitable* if it was able to properly recognize all the sentences indicated in the task's instruction. Otherwise, we considered the heuristic *incorrect*.
- The *time required* for completing the heuristic's definition task. We measured the time elapsed from the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2930519, IEEE Transactions on Software Engineering

TABLE 1
Aggregated answers about tasks' difficulty

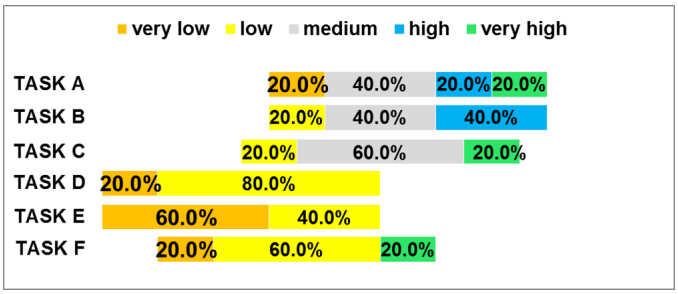


TABLE 2
Tasks' completion times expressed in minutes and seconds (mm:ss)

Subject	Task A	Task B	Task C	Task D	Task E	Task F
Subject 1	07:53	18:26	02:03			
Subject 2	03:04	25:20	07:45			
Subject 3	18:58	02:46	01:45			
Subject 4				06:40	03:29	05:25
Subject 5				07:04	05:22	05:19
Subject 6	17:50	03:21	03:51			
Subject 7				07:52	03:55	03:24
Subject 8				23:44	11:11	48:34
Subject 9	05:35	07:49	02:14			
Subject 10				03:19	02:10	02:35
Average	10:40	11:34	03:32	09:44	05:13	13:03

visualization of the task's instructions (along with the SD representation of the sentences) till the submission of the required response (*i.e.*, the heuristic).
- The *difficulty rating* (from *"Very Low"* to *"Very High"*) provided by the participant to indicate the difficulty encountered in completing the task.

2.2 Results

The results obtained in our study can be summarized as follows:

- For the Task F, all the participants belonging to the Group 2 defined *suitable* NLP heuristics, while (i) for the Tasks C, D and E, we received one (out of 5) *incorrect* heuristic, and (ii) for the Task B, two of the participants provided an *incorrect* heuristic. Moreover, for The task A, the majority of participants (3 out of 5) were not able to identify a *suitable* NLP heuristic. The study results indicated that *the identification and definition of NLP heuristics is not trivial* and approaches able to better support developers in accomplishing these activities are required.
- *The number of conditions in an NLP heuristic reflects the time required for identifying/defining it*. As a matter of fact, the Task E, for which 4 out of the 5 participants belonging to Group 2 defined an NLP heuristic containing only one condition (of type 1), is the task with the shortest average completion time (*i.e.*, 5 minutes and 13 seconds, as showed in Table 2) of the tasks assigned to participants of Group 2. Moreover, this task (*i.e.*, Task E) resulted the simplest task among those assigned to participants of Group 2, since 3 (out of 5) of them judged the *difficulty* for completing the task as *Very Low*, while the remaining 2 participants rated it as *Low*.
- *Conditions of type 1 are the simplest to be identified.* In most cases, the study's participants defined heuristics containing only conditions of this type. For example, (i) for the Task C, all the participants only defined conditions of this type and it was the task for which participants of Group 1 collectively assigned the lowest level of difficulty (see Table 1), (ii) for the Tasks B and E, 4 out of 5 subjects only identified conditions of this type, and (iii) for the Tasks A and F, 3 out of 5 subjects defined exclusively conditions of type 1.
- *The identification/definition of conditions of type 2 increases the heuristic's average definition time*. In the following, we report some evidence supporting this observation:
  - *Subject 2*: For completing the Task B, this subject defined an NLP heuristic with a condition of type 2, while for all other tasks Subject 2 provided heuristics containing only conditions of type 1. Noteworthy, Task B is the one for which this participant spent the longest time for identifying/defining the required heuristic (as reported in Table 2).
  - *Subjects 4, 5 and 10*: For completing the Task D, these subjects defined an NLP heuristic with a condition of type 2, while for all other tasks they provided heuristics not containing any condition of this type. Task D is the one for which these participants spent the longest time for identifying/defining the required heuristic (see Table 2).
  - *Subject 8*: For completing the Tasks D and E, Subject 8 defined NLP heuristics containing a condition of type 2, while for accomplishing the Task F this subject provided a heuristic containing two conditions of type 2. Among the subjects of the Group 2, this is the participant spending the longest time for completing all three tasks (*i.e.*, this subject spent more than 48 minutes to define the heuristic for completing the task F and rated the difficulty to complete this task as *Very High*).
- *Conditions of type 3 are harder to be identified* since this kind of conditions requires that the SD representation of the candidate sentences should not contain specific typed dependencies with particular characteristics (*e.g.*, two specific typed dependencies must not expose the same governor). As a matter of fact, among all the answers provided for completing the various tasks, only one of the study participants (*i.e.*, *Subject 10*) defined an NLP heuristic containing a condition of this type. Moreover, Task F, for which one of the participants provided a heuristic containing a condition of type 3 and another subject defined a heuristic containing two conditions of type 2, was the task for which participants of Group 2 assigned the highest level of difficulty (see Table 1).

To investigate the relationship existing between the time spent by each participant to identify/define an NLP heuristic and the number of the defined conditions in the correspondent NLP heuristics, we use the Kendall rank correlation coefficient. Similarly, we investigate the relationship existing between the level of difficulty assigned to a task and the number of the defined conditions, using the same criteria.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2930519, IEEE Transactions on Software Engineering

Kendall's *tau* coefficient is an index (τ) to assess the ordinal association between two measured quantities [27]. Since the majority of heuristics submitted by the participants have 1 or 2 conditions and the ranking of the number of conditions will have a lot of ties, we compute the τ_B coefficient that makes adjustments for ties [37].

We interpret the strength of the correlation as *small* for $0.10 \leq |\tau_B| \leq 0.29$, *medium* for $0.30 \leq |\tau_B| \leq 0.49$, and *large* for $|\tau_B| \geq 0.50$, as recommended by Cohen's standard [14].

Specifically, for each task t and each participant i, we considered: (i) the time spent by the participant i for completing the task t, $T_{(i,t)}$, (ii) the difficulty rating assigned by the participant i to the task t, $D_{(i,t)}$, and (iii) the number of conditions defined by the participant i to complete the task t, $CN_{(i,t)}$. Twenty-nine data points for each variable have been collected, since one of the participants (Subject 6) was not able to provide any heuristic for completing the Task A (we thus discarded the values $T_{(6,1)}$, $D_{(6,1)}$ and $CN_{(6,1)}$). For $D_{(i,t)}$ we assigned values ranging from 1 (when the participant rated the difficulty of the completed task as "Very Low") to 5 (when the participant rated the difficulty of the completed task as "Very High").

We obtained (i) a small level of correlation ($\tau_B = 0.2547$) between $T_{(i,t)}$ and $CN_{(i,t)}$, but it is only marginally significant ($p = 0.0877$), while (ii) we found that the correlation between $D_{(i,t)}$ and $CN_{(i,t)}$ is small ($\tau_B = 0.2203$) and not statistically significant ($p = 0.1770$). Also the correlation between $T_{(i,t)}$ and $D_{(i,t)}$ is small ($\tau_B = 0.2377$) and marginally significant ($p = 0.0970$). Indeed, as reported in Table 2, the time spent for identifying/defining a NLP heuristic is fairly subjective (*e.g.*, Subjects 8 and 10 produce the same NLP heuristic for Task D but their times are very different)

Interestingly, results change when considering not only the number of conditions identified $CN_{(i,t)}$, but also their types. To this aim, we assigned different weights to the different types of conditions:

- To conditions of type 1, we assign a weight of 0.5 points for each lemma appearing in the related list (*e.g.*, the condition `nsubj.governor=''add provide''` weighs 1).
- To conditions of type 2, we assign a weight of 2 points (*e.g.*, the condition `nsubj.governor=aux.governor` weighs 2).
- To conditions of type 3, we assign a basic weight of 0.5 points. Moreover, the weight of the encompassed clause (which could be a condition of type 1, or a condition of type 2) has to be added. For example, to the condition `not:aux.governor=neg.governor` a weight of $0.5 + 2 = 2.5$ is assigned.

The complexity of defining a condition could be due to a higher number of dependencies encountered during the analysis of SD. The distinct weights assigned to the 3 types of conditions are based on the number of the different typed dependencies that must be considered for actually defining the condition. Specifically, we assign a weight of 0.5 to each distinct typed dependency inspected during the condition definition. In particular, given two generic sentences, S_1 and S_2, and the respective SD representations G_1 and G_2, we consider *similar* two typed dependencies, T_i from G_1 and T_j from G_2, if they present the same relation name and the same lemma appears in the governor or the dependent position. Otherwise T_i and and T_j are *different*. For conditions of type 1 (*e.g.*, `nsubj.governor=''add''`) with just one lemma in the list, the same typed dependency (or a *similar* one) will appear in both G_1 and G_2, and, thus, just one typed dependency has to be considered to define the condition. For this reason, we assign a weight of 0.5. Then, 0.5 points will be added according to each additional lemma in the list (since each lemma appears in a different typed dependency, such typed dependency has to be considered for defining the condition). For conditions of type 2 (*e.g.*, `nsubj.governor=dobj.governor`), 2 typed dependencies in G_1 and 2 typed dependencies (*different* from the first two) in G_2 have to be considered, in order to define the condition, and we thus assign a weight of 2 (*i.e.*, 2*0.5 + 2*0.5=2). For conditions of type 3 (*e.g.*, `not:nsubj.governor=''add''`), a specific typed dependency encompassed in the not clause could appear in either G_1 or G_2, and such a dependency must be considered during the definition of the condition. Therefore, we assign a basic weight of 0.5 for this type of conditions. Moreover, the weight of the encompassed clause (that could be a condition of type 1, or a condition of type 2) has to be added.

For each task t and each participant i, we also considered the sum of weights assigned to the conditions defined by the participant i to complete the task t, $WC_{(i,t)}$. Since only one condition of type 3 has been provided by participants, we computed the Kendall's τ_B coefficient between $WC_{(i,t)}$ and $D_{(i,t)}$ and between $WC_{(i,t)}$ and $T_{(i,t)}$ without considering the specific data point in which such condition has been defined. We obtained a (i) medium correlation ($\tau_B = 0.3794$) between $D_{(i,t)}$ and $WC_{(i,t)}$, that is statistically significant ($p = 0.0155$), and (ii) a small association ($\tau_B = 0.2916$) between $T_{(i,t)}$ and $WC_{(i,t)}$, that is statistically significant ($p = 0.0426$).

> **RQ1 summary**: *Although the time required for identifying/defining an NLP heuristic may be fairly subjective, the difficulty in accomplishing such task is correlated with the number and the types of conditions the heuristic comprises.*

2.3 Threats to validity

Threats to *construct validity* concern the relation between the theory behind the experiment and the observation. In our study, we considered the correctness of responses and the time taken to produce these responses, as indicators of the tasks' difficulty. This could represent a threat to construct validity, as difficulty could depend on further factors that we did not consider. For alleviating this threat we asked participants to also rate the difficulty they encountered in completing each task. However, it is difficult to ensure that the tasks A, B, C, D, E, and F present the same level of actual or perceived difficulty by the involved groups of participants. The study performed in Section 4 allowed us to understand the extent to which NEON impact the time required by humans to identify natural language patterns, by involving groups of (2) external evaluators working on the same task. We argue that this additional study mitigates in part the threat concerning the potential difference, in terms of perceived/actual difficulty, of between Tasks A, B, C, D, E, and F.

Threats to *internal validity* concern any confounding factor that may affect the results. In our experiment we also

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2930519, IEEE Transactions on Software Engineering

enrolled two Ph.D. students having previous experience in text mining and natural language processing. This may represent a threat to internal validity, as groups with heterogeneous abilities could have been created. To mitigate this threat and ensure a good balancing of ability levels, we assigned one Ph.D. student to each group.

Conclusion validity focuses on how sure we can be that the treatment we used in an experiment really is related to the actual outcome we observed. In order to quantitatively corroborate the results emerged from a qualitative analysis, we studied the correlations existing between (i) the time spent for defining an NLP heuristic, (ii) the difficulty rate, and (iii) the number and types of conditions enclosed in the corresponding heuristic. The time spent for completing the task could depend on other factors (*e.g.*, distraction) than the intrinsic effort required, and this could represent a threat to conclusion validity. To alleviate this issue, we asked participants to complete each task in the shortest possible time and the study was conducted in a controlled environment.

Threats to *external validity* concern the generalization of our findings. Our study is exploratory and a large-scale validation is needed to achieve statistically significant conclusions. Also, the results obtained in our controlled experiment may be specific to the syntactic features contained in the sentences selected in each task. To mitigate this issue, the selected sentences (i) belong to different categories (*e.g.*, feature request, bug signaling) and (ii) vary in terms of size, lexicon and contents.

Moreover, the subjects involved in our experiment are all Master's and Ph.D. students. This could be a potential threat to validity because experienced NLP researchers could differently perceive the effort required for completing the tasks. However, to alleviate this threat, we enrolled subjects that were already familiar with text analysis (this is also the reason why we were unable to find a larger number of participants) and asked them to exercise their ability in NLP heuristics identification/definition for a period of 15 days. According to previous work [21], when a software engineering research experiment is aimed at studying basic programming and comprehension skills, students behavior is sufficiently similar to the one exhibited by professional developers. Nevertheless, further studies involving a larger set of experimental materials and professional subjects are needed, in order to generalize our results.

3 Approach: Automated Extraction of Natural Language Patterns

Results of the study reported in Section 2.2 highlighted how the manual identification of recurrent linguistic patterns and NLP heuristics for classifying intent in development communication is a difficult and time-consuming task. We argue that an approach able to automatically recognize recurring grammatical patterns from a set of natural language sentences can support the difficult task of devising intent classification heuristics.

We propose an approach, named NEON (*i.e.*, **N**lp-based softwar**E** d**O**cumentation a**N**alyzer), which is aimed at automatically identifying similar grammatical frames present in a set of natural language sentences. Our approach is

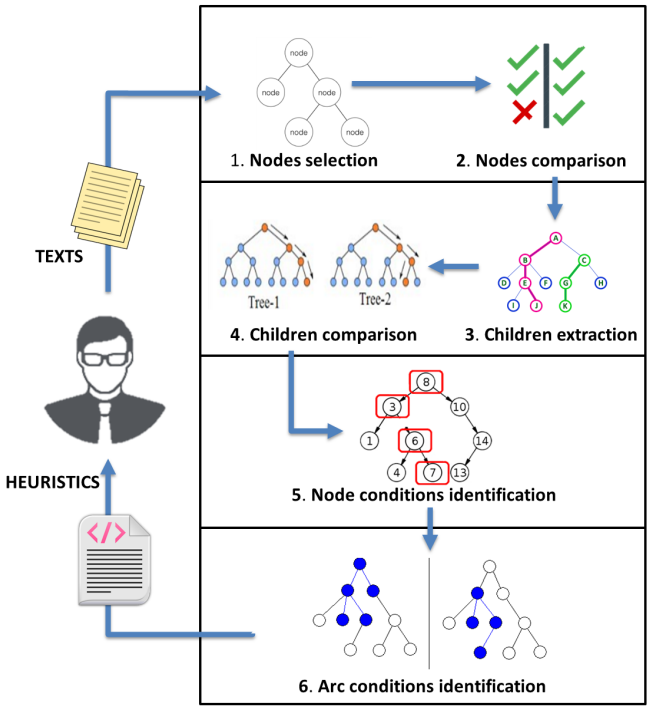


Fig. 2. The NEON approach.

configured as a computer-aided solution to the patterns' identification problem presented in Section 2.1, since it is also able to automatically recommend the NLP heuristics for automatically recognizing the identified linguistic patterns and therefore classify the intent of a sentence.

The semantic graph of a sentence is a graph in which the nodes represent the words of the sentence and a typed dependency between two words is represented through a labeled arc (the label reports the typed dependency name) directed from the governor node to the dependent node. Given two generic sentences S_1 and S_2 and their semantic graphs, G_1 and G_2, respectively, the process for automatically gathering similar grammatical structures appearing in S_1 and S_2, is illustrated in Figure 3, and comprises the following steps.

1. **Nodes selection:** Through the analysis of G_1, all the non-leaf nodes, V_{G1}, of the verb or noun types (*i.e.*, having POS tags starting with "VB" or "NN" [39]) are selected. The algorithm focuses on these kinds of nodes since nouns and verbs are generally considered to be more important than other types of terms appearing in a sentence, such as articles, adjectives or adverbs [9], [40], [42]. All the non-leaf nodes of the verb or noun types, V_{G2}, are also selected from G_2.
2. **Nodes comparison:** All the nodes in V_{G1} are compared with nodes in V_{G2}, with the purpose of finding similar nodes. Two nodes are similar, if, once stemmed, (i) the words represented in the two nodes are equal, or (ii) the *Wu & Palmer semantic relatedness* (WUP) [44] between them is higher than a threshold of 0.9 (after experimenting several threshold values for this metric, 0.9 proved to be the most appropriate one in this specific application).
3. **Children extraction:** For each pair of similar nodes, V_i in

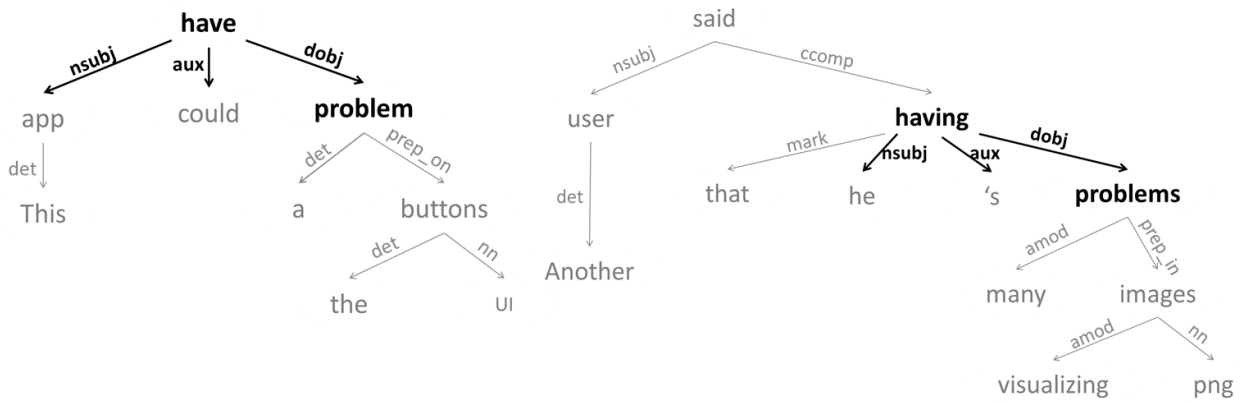


Fig. 3. Semantic graphs of Example Sentences: (i) *"This app could have a problem on the ui buttons"* (on the left), and (ii) *"Another user said that he's having many problems in visualizing png images"* (on the right)

V_{G1} and V_j in V_{G2}, the child pairs P_i of V_i are extracted. Each pair P_x in P_i contains a child node, $P_{(x,N)}$, of V_i and the labeled arc, $P_{(x,E)}$, directed from V_i to $P_{(x,N)}$. Similarly, the child pairs P_j are extracted from V_j.

4. **Children comparison:** Each pair P_x in P_i is compared with every pair P_y in P_j, in order to fill a P_{shared} set containing shared grammatical structures depending on similar nodes. A pair P_y is added to the P_{shared} set when the arc $P_{(y,E)}$ from V_j to $P_{(y,N)}$ has the same label, L, (*i.e.,* the relation name) of the arc $P_{(x,E)}$ directed from V_i to $P_{(x,N)}$.
5. **Node conditions identification:** When P_y is added to P_{shared} a clause indicating that P_y has V_j as parent node is appended to the conditions list C. In particular, a type 1 condition establishing that a typed dependency of the type L has to present one of the stemmed words in V_i and V_j, (if they are different), or the common word (if the same stemmed word appears in both V_i and V_j) as governor (*e.g.,* `aux.governor=''need require''` or `aux.governor=''need''`) is added to C. Moreover, if the nodes $P_{(y,N)}$ and $P_{(x,N)}$ are similar, a type 1 condition establishing that a typed dependency of the type L has to present one of the stemmed words in $P_{(y,N)}$ and $P_{(x,N)}$ (if they are different), or the shared word (if the same stemmed word appears in both $P_{(y,N)}$ and $P_{(x,N)}$) as dependent (*e.g.,* `aux.dependent=''can may''` or `aux.dependent=''can''`) is added to C.
6. **Arc conditions identification:** To the aim of identifying this kind of conditions, all the pairs in P_{shared} are analyzed. In particular, given two consecutive pairs, P_k and P_{k+1}, in P_{shared}, the following situations may occur:
   - if the parent nodes of $P_{(k,N)}$ and $P_{(k+1,N)}$ match, a clause indicating that the relation (*i.e.,* typed dependency) of the type $P_{(k,E)}$ needs to have a governor that coincides with the governor of a relation of the type $P_{(k+1,E)}$ is appended to C (*e.g.,* `nsubj.governor=aux.governor`);
   - if the node $P_{(k,N)}$ coincides with the parent node of $P_{(k+1,N)}$, the algorithm add to the conditions' list, C, a clause indicating that the dependency of the type $P_{(k,E)}$ needs to have a dependent that coincides with the governor of a relation of the type $P_{(k+1,E)}$ (*e.g.,* `dobj.dependent=prep_on.governor`);
   - if the parent node of $P_{(k,N)}$ matches the node $P_{(k+1,N)}$, the algorithm adds to the conditions' list, C, a clause indicating that the dependency of the type $P_{(k,E)}$ needs to expose a governor that matches with the dependent of a relation of the type $P_{(k+1,E)}$ (*e.g.,* `prep_with.governor=dobj.dependent`).

The steps 3, 4, and 5 are repeated for each pair of similar nodes, V_i in V_{G1} and V_j in V_{G2}, identified in the step 2 of the algorithm. Once analyzed all the pairs, V_i and V_j, of similar nodes, Step 6 is executed. At the end of the process, the generated clauses contained in the conditions' list C are ordered to be compliant with the XML grammar illustrated in Section 2.1.

For example, the following sentences match the linguistic pattern *"[something] [auxiliary] have problem"*:

1. *This app could have a problem on the UI buttons.*
2. *Another user said that he's having many problems in visualizing png files.*

The process illustrated above, when working with these two sentences, is able to recognize the nodes and arcs highlighted in Figure 3, and generate the following conditions to capture the pattern:

```
nsubj.governor=''have''
nsubj.governor=aux.governor
aux.governor=dobj.governor
dobj.dependent=''problem''
```

This example shows how NEON detects sentences sharing similar linguistic patterns, even if they concern different topics. It is important to note that the process determines the sentences that have similar linguistic patterns without considering the actually frequency of the extracted heuristics, thus assigning to all heuristics the same importance. Clearly, a limitation of our method is that the identification of linguistic patterns and the definition of NLP heuristics of NEON requires that at least two sentences share a common linguistic pattern (*i.e.,* they can be detected by the same NLP heuristic).

NEON is able to provide *suitable* NLP heuristics for three of the tasks illustrated in Section 2.1 (*i.e.,* tasks C, D and F). Moreover, the outputs of the approach for the sentences of tasks B and E, require simple changes to become *suitable* (*i.e.,* the addition of a lemma in a type 1 condition, *e.g.,* `cop.governor=''problem'' ⟶ cop.governor=''problem bug''`). Finally, to make *suitable* the response of our approach for the sentences of the task A, a type 3 condition needs to be added (*i.e.,* `not:xcomp.dependent=''do''`). Since the algorithm is unable to locate something that has to not appear in the semantic graph of a sentence, our approach cannot identify type 3 conditions. However, as the results of our studies show (see Section 4.1 and Section 4.2), type 3 conditions are rarely used also by human subjects. Indeed, the results of the study in Section 4.2 suggest that for the Feature Request category the conditions of type 3 (manually defined by researchers) are 6 out of 205 overall conditions, *i.e.,* about 2.3%, while for the Problem Discovery category the type 3 conditions (manually defined by researchers) are 2 out of 108 overall conditions (about 1.85%).

4 Evaluation

The *goal* of our study is to evaluate NEON, for the *purpose* of investigating the extent to which it could replace, or at least reduce, the manual work for identifying recurring linguistic patterns in informal documentation. Such patterns can then be used for intent recognition and classification, and, as previous work has shown, can be valuable to build recommender systems to support software development, evolution [19], [34], or testing [22], [35].

To evaluate NEON, we perform three studies. The first study aims at comparing the time spent by humans to define NLP heuristics useful for classification purposes when supported by our tool with the time required for performing the NLP heuristics identification and definition processes manually. The second study is aimed at investigating the classification capabilities of heuristics extracted by NEON on development mails, as well as the extent to which such heuristics are similar to the ones manually identified by researchers. The third study tests NEON in action, and aims at showing how patterns automatically identified by NEON can be used to analyze the extent to which developers consistently report issues through issue tracking-systems.

4.1 Mining of Recurrent Patterns in Mobile App Reviews

For evaluating the NEON's effectiveness and estimating its time-saving capability, we conducted a study aimed at answering the following research question.

RQ2: *To what extent does NEON impact the time required by humans to identify natural language patterns useful for classification purposes?*

Context of the study. To carry out the investigation, we employed the data available from our previous work [34]. In particular, this dataset encompasses 1390 mobile app review sentences extracted from popular apps. Each sentence is labeled with one of the following categories:

TABLE 3
Review sentences distribution.

Category	Sentences	Proportion
Feature Request	192	13.81%
Problem Discovery	494	35.54%
Other	704	50.65%
Total	**1390**	**100%**

- **Information Giving**: sentences that inform or update users or developers about an aspect related to the app.
- **Information Seeking**: sentences related to attempts to obtain information or help from other users or developers.
- **Feature Request**: sentences expressing ideas, suggestions or needs for improving or enhancing the app or its features.
- **Problem Discovery**: sentences describing issues with the app or unexpected behaviors.

In particular we exclusively took into account the sentences belonging to the Feature Request and Problem Discovery categories, and we assigned a generic *Other* label to the sentences that do not belong to these categories. Beyond being the main relevant categories from a software maintenance perspective [17], software engineers can easily recognize sentences requiring features or describing issues. Consequently, they can more easily identify recurrent language structures used for introducing these types of sentences, and this makes the sentences belonging to the Feature Request and Problem Discovery categories the best candidates for our purposes.

Table 3 illustrates the number of sentences in the data collection belonging to each of the two categories, as well as the number of sentences falling in the *Other* category.

Analysis Method. To answer **RQ2**, we carried out a study involving four external subjects: (i) one professional software engineer, and (iii) three software engineering master students. Three of them (*i.e., Subject 2, Subject 3*, and *Subject 4*) were experienced in natural language processing and dependency parsing, while the remaining subject (*i.e., Subject 1*) has prior knowledge about natural language processing but not about dependency parsing. We were able to only involve four subjects, due to the specific skills (*i.e.,* natural language processing background) required to participate. Moreover, the potentially-long time required to perform the tasks discouraged further subjects to participate in the study.

According to the need of ensuring reasonable times for tasks' completion, we extracted 100 sentences from the dataset: 50 belonging to the Feature Request category, and 50 belonging to the Problem Discovery category. These 100 sentences represent the training set, $T_{training}$, while the remaining 1290 sentences represent the test set, T_{test}. Then, we divided the subjects in two groups: *Subjects 1* and *2* have been assigned to the *Group 1*, while *Subjects 3* and 4 have been assigned to the *Group 2*.

The sentences in $T_{training}$ have been used to automatically generate (through NEON) a list of 241 candidate recurrent linguistic patterns. We asked each subject of *Group 1* to analyze the NEON's output and manually select (and

possibly modify) the patterns that, in their opinion, were relevant for identifying sentences belonging to one of the categories of interest (*i.e.,* Feature Request and Problem Discovery). In contrast, we asked each subject of *Group 2* to inspect the 100 sentences in $T_{training}$ (along with their SD representation) to manually define a set of NLP heuristics for recognizing sentences belonging to the Feature Request and Problem Discovery categories. Before starting the tasks, all the subjects have been introduced to (i) the XML grammar for defining the NLP heuristics, and (ii) the ways such heuristics can be used to automatically recognize linguistic patterns.

For each subject, we collected (i) the list of identified NLP heuristics, and (ii) the time required to identify such heuristics. We used the list of NLP heuristics produced by each subject to automatically classify the sentences contained in T_{test}, and assessed the classification performance of identified heuristics relying on widely adopted metrics of Information Retrieval: Precision, Recall and F-measure.

4.1.1 Results of RQ2: NEON time-saving capability

Table 4 reports for each subject: (i) the overall time (expressed in minutes) required to complete the task (T), (ii) the number of identified NLP heuristics (H_{ALL}), (iii) the number of identified NLP heuristics associated with the Feature Request category (H_{FR}), and (iv) the number of identified NLP heuristics associated with the Problem Discovery category (H_{PD}). In Table 5 the classification results obtained through the NLP heuristics identified by each subject are reported. The use of NEON allowed to save more than the 70% of time for identifying the required NLP heuristics: the two subjects who selected the NLP heuristics analyzing the NEON's output took about an hour and a half and two and a half hours, respectively, to complete the identification task, while the subjects who manually defined the NLP heuristics spent about nine and ten hours to complete the process, respectively. As reported in Table 5, *Subject 4*, who is the one spending the longest time for completing the task, obtained the best F-measure result for the Feature Request category (*i.e.,* 0.341), while *Subject 3* achieved the best F-measure value for the Problem Discovery category (*i.e.,* 0.618). However, the NLP heuristics identified by both subjects using NEON achieved comparable F-measure values for both categories (*i.e.,* higher than 0.27 for Feature Request, and about 0.5 for Problem Discovery). In general, NEON-generated heuristics obtained (i) reasonable recall values (*i.e.*, on average 0.426), outperforming of about 18% the recall values obtained by both *Subjects 3* and *4* for the Feature Request category, but (ii) lower precision results (*i.e.,* higher than 0.6 for Problem Discovery and around 0.2 for Feature Request) than the ones obtained by *Subjects 3* and *4*. The differences in the results obtained for the two different categories could be due to the fact that Problem Discoveries contain more recognizable and recurrent structures, while common patterns for detecting Feature Requests are harder to be identified [34].

This means that NEON allowed to drastically reduce the times required for the manual inspection at the cost of a classification accuracy loss ranging between 7% and 14%.

It is worth pointing out that *Subject 1* did not modify (by adding further conditions) any of the relevant patterns selected from the NEON's output, while *Subject 2* did modify just 4 out of the overall 78 patterns selected. Considering these results, we believe that the NEON-generated heuristics could represent a good initial set of heuristics that humans can further refine to achieve better classification performance. For instance, *Subject 1* marked the pattern *"way to [something]"* as relevant for the Feature Request class, but such pattern may produce many false positives (*e.g., "Easy to use, great way to kill time at a doctor's office and such.."*). However, the refinement of this pattern through the addition of further conditions so that the pattern could more likely introduce an actual request (*e.g.,, "wish there was a way to [something]"*) would result in a lower number of false positives, and, consequently, a better precision. Clearly, the poor classification performance (for all subjects) are also related to the low numbers of samples in $T_{training}$ (26.04% of overall sentences belonging the Feature Request class and 11.26% of overall sentences belonging to the Problem Discovery category).

After completing the tasks, we interviewed all the subjects and asked them for impressions about the performed task and suggestions on how to possibly improve NEON. *Subject 1* (with no prior knowledge on dependency parsing) encountered some difficulties in completing the task (*"...The background knowledge required for the task was a bit too much"*), but the human-readable patterns accompanying the heuristic definitions provided a useful guide to manage the task's complexity (*"The human readable pattern (Column Text) was fairly clear to me and I mostly only looked at this column"*). *Subject 2* did not encounter any problem during the task (*"I've not encountered particular problems during the task"*). Both *Subject 1* and *Subject 2* (who defined rules based on the

TABLE 4
Rule identification Times with (Group 1) and without NEON (Group 2).

Group	Subj.	Time	H_{ALL}	H_{FR}	H_{PD}
1	#1	93 mins	74	32	42
(with NEON)	#2	152 mins	78	34	44
2	#3	540 mins	37	20	17
(w.o. NEON)	#4	602 mins	56	29 27	

TABLE 5
Classification performance obtained by each subject: True Positives (TP), False Negatives (FN), False Positives (FP), True Negatives (TN), Recall (R), Precision (P), and F-measure (F1).

Class	TP	FN	FP	TN	R	P	F1
Subject 1							
Feature Request	62	80	250	898	0.437	0.199	0.273
Problem Discovery	193	251	101	745	0.435	0.656	0.523
Subject 2							
Feature Request	63	79	261	887	0.444	0.194	0.270
Problem Discovery	172	272	108	738	0.387	0.614	0.475
Subject 3							
Feature Request	37	105	104	1044	0.261	0.262	0.261
Problem Discovery	238	206	88	758	0.536	0.730	0.618
Subject 4							
Feature Request	36	106	33	1115	0.254	0.522	0.341
Problem Discovery	219	225	66	780	0.493	0.768	0.601

NEON's output), found the information provided by NEON very useful to identify the NLP heuristics for automating the recognition of relevant data contained in software informal documents: *"...this can certainly be a great help to human activities in terms of time effort..."*, *"...the tool can surely help in the future to automate for example issue tracking etc"*. Both *Subject 3* and *4* (who manually defined the NLP heuristics) found the XML grammar for defining the heuristics easy or very simple to use. In particular, for both subjects, the difficulty in the definition of a condition depends on its type (*e.g.,* both *Subject 3* and *Subject 4* reported a higher difficulty rate in identifying conditions of types 2 and 3). The main obstacle encountered by subjects working without NEON was to identify *"the most significant grammatical relations to be included"* in heuristics, especially for sentences belonging to both Feature Request and Problem Discovery categories or *"for sentences that contain syntax errors"*.

RQ2 summary: *NEON allowed to save more than 70% of the time when compared to the manual identification and definition of a set of NLP heuristics useful for classification purposes. However, the NLP heuristics automatically extracted by NEON achieved a classification accuracy between 7% and 14% lower than the manually defined ones.*

4.1.2 Threats to validity

Construct validity. In this study, the identification of relevant patterns useful for classifying reviews belonging to the Feature Request and Problem Discovery categories is performed by human subjects. Thus, such an identification process could be biased, and this may represent a threat to construct validity. Moreover, the time required for identifying such patterns could be subjective and might not depend on the actual effort required for identifying them. To alleviate these issues, for each of the two tasks we involved at least two different subjects and analyzed their results separately.

Internal validity. Four different subjects are involved in our study. Their different levels of background knowledge risk to weaken the obtained results, and this may represent a threat to internal validity. To alleviate this threat, we only selected subjects with previous experience in natural language processing and instructed all of them on how to correctly identify and define NLP heuristics useful for classification purposes.

Conclusion validity. As previously mentioned, given the relatively difficult (*i.e.,* requiring specific skills not all developers possess) and effort-prone task, it was not possible to conduct a study involving a large number of participants (nor to conduct a crossover experiment where each subject worked with both treatments), and statistically assess the differences between the two groups (*i.e.,* using NEON or not). Therefore, we could only qualitatively assess the observed differences, as well as the feedback reported by the study participants. We could not exclude that the observed differences are due to chance or to the participants' specific skills. Nevertheless, as observed efforts are 3.5-6.5 times larger when NEON was not used, it is unlikely this could have happened by chance.

External validity. The patterns' identification tasks have been performed on a training set ($T_{training}$) that is small in scale (*i.e.,* 100 app review sentences) due to the need of ensuring reasonable tasks' completion times. This could represent a threat to external validity since some of the identified patterns could be due to the specific sentences selected. Consequently, this may result in a lower classification effectiveness. To counteract this issue, the sentences in our training set belong to seven different apps [34] and different categories (*i.e.,* Feature Request and Problem Discovery).

TABLE 6
Development mailing lists samples

Project	Messages	Posting Period
Qt Project	102	June 2014
Qt Project	100	May 2014
Qt Project	20	March 2014
Qt Project	20	April 2014
Qt Project	20	July 2014
Qt Project	20	August 2014
Qt Project	20	September 2014
Ubuntu	20	September 2004
Ubuntu	20	October 2004
Ubuntu	20	November 2004
Ubuntu	20	December 2004
Ubuntu	20	January 2005

4.2 Mining of Recurrent Patterns in Developers' Discussions

This study aims at addressing the following research questions:

RQ3$_a$*: To what extent are NLP heuristics extracted by NEON effective in classifying intent in development emails?*

RQ3$_b$*: To what extent are automatically-extracted NLP heuristics similar to the ones manually extracted by human annotators?*

Context of the study. To answer these questions we used the dataset available from previous work [19]. Specifically, this dataset consists of messages contained in the mailing lists of two popular open source projects, Qt and Ubuntu. A detailed list of the sampled messages in our dataset is shown in Table 6.

Moreover, as reported in our previous work [19], all the messages of our dataset were manually analyzed by two researchers, who assigned each relevant email sentence to one of the categories of the taxonomy in Table 7. Such a taxonomy is aimed at modeling categories of sentences describing the relevant textual fragments from software maintenance and evolution perspective which developers report in development emails.

Analysis Method. To answer **RQ3**$_a$, we considered the sentences belonging to the Problem Discovery and Feature Requests categories, available from the *training sets* of experiments I, II, III (*i.e.,* a total of 302 emails of the Qt Project mailing list exchanged during 2014) of our previous work [19]. In particular, such sentences have been provided as input to NEON, to automatically mine the NLP heuristics able to recognize sentences in these two categories. Finally, we evaluated the classification performances obtained through these heuristics on the *test set* available from experiment III (*i.e.,* 100 emails of the Ubuntu mailing list) of our previous work [19].

TABLE 7
Development mailing lists categories

Category	Description
Feature Request	Sentences related to ideas/suggestions/needs for improving or enhancing the product/service or its features (*e.g., "We should add a new button"*)
Opinion Asking	Sentences for requiring someone to explicitly express his/her point of view about something (*e.g., "What do you think about the aspect of the main panel?"*)
Problem Discovery	Sentences related to issue definitions and unexpected behaviors (*e.g., "the problem occurs when I try to access to database"*)
Solution Proposal	Sentences used to describe possible solutions for known problems (*e.g.,, "let's try to use a new method to compute costs"*)
Information Seeking	Sentences related to attempts to obtain information or help from other users (*e.g., "can you explain how the code works?"*)
Information Giving	Sentences used to inform/update other users about something (*e.g., "the plan is to release new updates in this week"*)

To avoid using too generic heuristics for the classification task, we performed manual filtering of the heuristics proposed by NEON. Specifically, two authors of the paper separately marked the relevance of each of the heuristics proposed by NEON with respect to the Problem Discovery and Feature Requests categories with *yes*, *no*, or *maybe*. For completing this task, the two subjects spent 39 and 43 minutes, respectively. The inter-rater agreement (Cohen's Kappa) was $k = 0.498$ (170 items out of a total of 248). All the items for which the two annotators initially disagreed (or to which both annotators assigned the label *maybe*) were discussed among them and, consequently, promoted to either *yes* or *no*.

Because of the initial divergence between the annotators (Cohen's Kappa < 0.5) we decided to involve a third annotator for a further check. The third annotator (another author of the paper) expressed his judgment on the relevance of each heuristic in all the cases (*i.e.*, 78) in which a discussion between the two initial annotators was necessary. In the end, 73 heuristics out of a total of 129 (56.6%) were classified as meaningful for the Feature Request category, while for the Problem Discovery category, 57 heuristics out of 119 (47.9%) were selected. All the remaining heuristics were discarded.

To answer **RQ3**$_b$, we compare the heuristics inferred by NEON with those encoded in DECA [19], [20]. A manual inspection of the automatically extracted heuristics was conducted, to establish whether each of them was *semantically linked* to one or more of the heuristics originally identified in our previous research [20]. More specifically, given the set of sentences S in our *test set*, we consider two heuristics, H_1 (from the set of heuristics extracted by NEON) and H_2 (from the set of heuristics encoded in DECA) as *semantically linked* if the intersection between the sentences in S which H_1 and H_2 are able to recognize is a non-empty subset of sentences and all the sentences identified by both heuristics exhibit a shared typed dependency subtree. This manual analysis to address RQ3$_b$ required for the two annotators about five hours of work.

TABLE 8
Results obtained by automatically extracted (through NEON) and manually selected heuristics: True Positives (TP), False Positives (FP), False Negatives (FN), Recall (R), Precision (P), and F-Measure (F1).

Class	TP	FP	FN	R	P	F1
Feature Request	28	28	26	0.519	0.500	0.509
Problem Discovery	25	8	24	0.510	0.758	0.610
Other	125	45	33	0.791	0.735	0.762
Total	178	81	83	0.682	0.687	0.685

TABLE 9
Results obtained by DECA' s heuristics: True Positives (TP), False Positives (FP), False Negatives (FN), Recall (R), Precision (P), and F-Measure (F1).

Class	TP	FP	FN	R	P	F1
Feature Request	46	9	8	0.852	0.836	0.844
Problem Discovery	39	2	10	0.796	0.951	0.867
Other	98	9	60	0.620	0.916	0.740
Total	183	20	78	0.701	0.901	0.789

4.2.1 Results of RQ3$_a$: NEON classification effectiveness

Table 8 shows the classification results obtained when relying on the heuristics automatically extracted by NEON, while Table 9 reports the results previously obtained using heuristics defined by researchers. Note that in both Tables 8 and 9 the *Other* category includes all the items not belonging to Feature Request or Problem Discovery categories. We decided to merge all these categories in the *Other* class because the main relevant categories for maintenance and evolution perspectives are Feature Request and Problem Discovery.

The first result shown in Tables 8 and 9 is that the best classification accuracy is obtained when relying on NLP heuristics manually defined by researchers. In particular, for the Problem Discovery and Feature Request categories the ranges of values achieved by NEON in terms of recall, precision and F-measure are $0.51 - 0.52$, $0.50 - 0.76$ and $0.51 - 0.61$ respectively. In the case of human-generated NLP templates, the achieved results in terms of recall, precision and F-measure are $0.80 - 0.85$, $0.84 - 0.95$ and $0.84 - 0.87$ respectively. On average, considering all the sentence categories, we can notice how human-generated heuristics (DECA) outperform automatically generated ones (NEON) by about 2% (0.70 vs. 0.68) in terms of recall, 21% (0.90 vs. 0.69) in terms of precision, and 10% (0.79 vs. 0.69) in terms of F-measure.

This result is not completely surprising, considering that, while the automatic approach we have defined is based on syntactic information (*i.e.*, NEON compares the syntactic trees of the sentences to extract NLP heuristics), a human subject has, in general, a broader knowledge base and full understanding of semantics and context in which these heuristics are defined. However, as reported in previous work [19], a major limitation of DECA is that the definition of the human-generated heuristics is a very error-prone and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2930519, IEEE Transactions on Software Engineering

TABLE 10
Results obtained by the experimented ML classifiers.

Class	Recall	Precision	F-measure
functions.Logistic			
Feature Request	0.261	0.158	0.197
Problem Discovery	0.326	0.200	0.246
Other	0.448	0.670	0.537
Weighted Avg.	0.395	0.502	0.429
functions.SimpleLogistic			
Feature Request	0.022	0.500	0.042
Problem Discovery	0.000	0.000	0.000
Other	0.994	0.660	0.794
Weighted Avg.	0.659	0.523	0.530
trees.J48			
Feature Request	0.000	0.000	0.000
Problem Discovery	0.000	0.000	0.000
Other	1.000	0.659	0.794
Weighted Avg.	0.659	0.434	0.524
trees.RandomForest			
Feature Request	0.043	0.080	0.056
Problem Discovery	0.047	0.154	0.071
Other	0.849	0.655	0.739
Weighted Avg.	0.575	0.471	0.509
trees.FT			
Feature Request	0.174	0.182	0.178
Problem Discovery	0.209	0.290	0.243
Other	0.715	0.661	0.687
Weighted Avg.	0.536	0.516	0.524
rules.NNge			
Feature Request	0.087	0.070	0.078
Problem Discovery	0.047	0.095	0.063
Other	0.686	0.645	0.665
Weighted Avg.	0.475	0.453	0.462

effort-intensive activity, which can lead in most cases to heuristics that could be very specific to a project and thus, not generalizable. In Section 4.1 we discussed in more detail how developers could benefit from NEON by automatically obtaining an initial set of heuristics that they could further refine.

Concerning the comparison with machine learning classifiers based on n-grams, Table 10 reports the results of the classification achieved by six different machine learning algorithms, considering as *training and test sets* the ones used in previous work upon evaluating DECA [19]. The automatic classification of sentences is performed using the Weka tool[2] experimenting with six different machine learning techniques, namely the Logistic Regression, Simple Logistic, J48, Random Forest, FT, and Nnge. The choice of these techniques was not random, they were successfully used for bug reports classification and for defect prediction in many previous works [19].

By analyzing the results in Table 10, we can confirm that, as a positive effect, the intent classification performed through NLP heuristics identified by NEON is substantially more effective than classical text categorization based on lexicon analysis and machine learning algorithms experimented in recent work [19]. Indeed, automatically extracted heuristics obtain classification performance (see Table 8) higher than all the considered machine learning algorithms.

2. https://www.cs.waikato.ac.nz/ml/weka/

```
<NLP_heuristic>
<sentence type="declarative"/>
<type>nsubj/aux/dobj</type>
<text>[something] could have [something]</text>
<conditions>
<condition>nsubj.governor="have"</condition>
<condition>nsubj.governor=aux.governor</condition>
- <condition>aux.dependent="could should might may"</condition>
+ <condition>aux.dependent="could"</condition>
<condition>aux.governor=dobj.governor</condition>
</conditions>
</NLP_heuristic>
```

Fig. 4. Comparison of a NEON-generated and a manually identified heuristic of the Feature Request category.

Specifically, NEON outperforms overall results obtained through functions.Logistic algorithm (which resulted the best performing machine learning model for the *Feature Request* and *Problem Discovery* categories) of about 28% (0.68 vs. 0.40) in terms of recall, about 19% (0.69 vs. 0.50) in terms of precision and about 25% (0.68 vs. 0.43) in terms of F-measure.

> **RQ3$_a$ summary**: *Manually-defined heuristics of DECA outperform the ones automatically generated by NEON by about 2% in terms of recall, 21% in terms of precision and 10% in terms of F-measure. The heuristic-based intent classification provided by NEON is substantially more effective than classical text categorization based on lexicon analysis and machine learning algorithms experimented in the literature.*

4.2.2 Results of RQ3$_b$: Similarity of NEON NLP heuristics with the ones extracted by researchers

Among the 130 heuristics automatically extracted and marked as relevant by the three annotators, 77 (59.23%) of these resulted semantically linked to heuristics previously defined by researchers [19]. Specifically, for the Feature Request category, 41 out of 73 (56.16%) of the automatically extracted and selected heuristics resulted *semantically linked* to one of the previously implemented heuristics. Concerning the Problem Discovery category 36 out of 57 (63.16%) of the automatically extracted and selected heuristics are *semantically linked* to one of the heuristics originally implemented for this category. In Figures 4 and 5 are reported some examples of heuristics extracted with NEON and considered semantically linked to heuristics previously identified by researchers. In particular, given two semantically linked heuristics, H_1 (NEON-generated) and H_2 (manually defined), in Figures 4 and 5, to emphasize similarities and differences existing between H_1 and H_2 we annotate with a $+$ symbol the conditions present in H_1 but missing in H_2, while with a $-$ symbol we annotate conditions present in H_2 but missing in H_1.

The obtained results are encouraging if we consider that around 52% of the heuristics extracted by NEON are considered relevant (*i.e.*, they are related to the Feature Request and Problem Discovery categories), and about 60% of such heuristics (the ones that are relevant) are *semantically linked* to heuristics extracted by humans. However, future research should be devoted in studying the patterns that were not recognized as semantically-related to the manual defined heuristics.

```
  <NLP_heuristic>
  <sentence type="declarative"/>
  <type>nsubj</type>
- <text>Problem/issue occurs</text>
+ <text>[something] happens/occurs</text>
  <conditions>
- <condition>nsubj.governor="issue problem mistake bug"</condition>
- <condition>nsubj.governor="occur"</condition>
+ <condition>nsubj.governor="occur happen"</condition>
- <condition>not:nsubj.governor=neg.governor</condition>
  </conditions>
  </NLP_heuristic>
```

Fig. 5. Comparison of a NEON-generated and a manually identified heuristic of the Problem Discovery category.

RQ3$_b$ summary: *Around 60% of the heuristics extracted by NEON and marked as relevant are semantically linked to heuristics manually identified by humans. This means that around 30% of the identified patterns were semantically linked to manually-written patterns.*

4.2.3 Threats to validity

Construct validity. Semantic labels have been assigned to sentences in our dataset according to the judgment of human annotators. A threat to construct validity could be due to a possible level of subjectivity caused by the manual classification of entities. As also reported in our previous work [19], to reduce this threat, two human judges (two authors of this paper) separately assigned each sentence in our dataset to one of the categories in the taxonomy, and only predictions, which both judges agreed upon, formed our truth set for the experiment. This threat is in part mitigated by the study in Section 4, where we discuss the effort required by external subjects in defining the NLP heuristics with and without NEON.

Internal validity. We used a set of machine learning (ML) algorithms as a baseline to estimate our results. This represents a threat to internal validity, as the obtained results could be due to the particular ML technique experimented. For mitigating this issue, we trained different ML algorithms and compared the results achieved by our approach with the results obtained by each ML classifier.

External validity. This study makes use of messages sampled from a single project (Qt) for the composition of the training set and focuses on two categories. It is possible that the specific, recurring issues in this project could bias (and limit the generalizability of) the derived heuristics. This concern is mitigated by showing that the approach produces acceptable results on unseen data from a different project, using as test set emails belonging to another project, Ubuntu for precision's sake.

4.3 Mining of Recurrent Patterns in Similar Issues

The main research question that guided this second study is:

RQ4: *To what extent do developers' belonging to different projects use similar patterns to describe the same type of issues in an issue tracker?*

This investigation aims at studying whether NEON can be applied to identify the possible use, in similar circumstances, of common linguistic patterns by different developers. In particular, we investigate whether developers working in different projects employ common linguistic patterns to title issues of the same type. For example, bugs, feature requests, or enhancements may exhibit recurring linguistic regularities in their titles.

Issue titles have been proven to be important features for the detection of duplicated bugs [26] and previous research [28], [38] demonstrated that issue titles have characteristics or features that can be exploited to assign the correct severity level to a bug.

Context of the study. To answer **RQ4**, we collected issues from 20 open source projects hosted on GitHub. Table 11 reports the issues we collected from these 20 projects. In particular, the selection process we applied was not random and consisted of the following steps:

1. In order to consider projects with a reliable number of issues, the 100 most popular ones (*i.e.,* top starred) have been selected from GitHub.
2. From the list of projects obtained in the previous step, we selected those written in C#, Java, Javascript, and Ruby.
3. For each of the aforementioned programming languages, we selected the top 5 projects, for a total of 20 projects.
4. We analyzed the *issue labels* associated with all these projects and grouped the semantically-similar labels in five issue types, as shown in Table 11.
5. Issues having at least one of the selected labels assigned have been extracted from these 20 projects.

Analysis Method. As shown in Table 11, we collected all issue labels related to five common issue types: *"bug*, *enhancement*, *question*, *area build* and *documentation"*. We selected such 5 issue types since they are widespread in all the analyzed projects, and also because we want to understand whether developers of different projects use similar linguistic patterns while describing similar issues. It is important to mention that each project assigned similar but not the same names to the *"bug"* (*e.g.,* "bug", "issue-bug", "defect", "type: defect", etc.), *"enhancement"* (*e.g.,* "enhancement","improvement", "issue-enhancement", etc.), *"question"* (*e.g.,* "question", "contains potential question keywords", etc.), *"area build"* ("area-build", "Area-Build") and *"documentation"* (*e.g.,* "documentation", "docs", "type: docs", etc.) issue types.

To identify recurring patterns for each issue type, we selected, for each issue type, projects having at least 100 issues. In this way, for the *"bug"* issue type we have selected the projects AngularJS, CoreCLR and PowerShell; while for the *"enhancement"* issue type, we selected the projects CoreCLR, Guava, PowerShell and Spring Boot. For the remaining issue types, we did not find enough issues. As a further step, for each selected project, we used NEON to extract linguistic patterns from the titles of the issues associated to the *"bug"* and *"enhancement"* issue types. Hence, to answer our RQ4, for both issue types, we observe the number of *patterns with shared conditions* extracted from the issue titles of the selected projects. To perform this task we leverage the following definitions:

- **Definition 1**. Given two NLP heuristics, H_1 and H_2, a condition C_i, defined in H_1, and a condition C_j, defined in H_2, are equivalent if $i = j$, as well as C_i and C_j involve (i) the same typed dependencies, (ii) the same *arguments* (*i.e.,* governors and/or dependents), and (iii) the same *lemmas*. More formally, (i) let TD_i and TD_j

TABLE 11
RQ4: Dataset of collected issues.

Project	Language	bug	enhancement	question	area build	documentation
AngularJS	Javascript	194	0	0	26	48
CoreClr	C#	147	494	54	58	36
Dapper	C#	12	24	0	5	6
Dubbo	Java	9	8	0	6	6
fastlane	Ruby	14	0	0	3	3
fpm	Ruby	13	0	0	2	5
freeCodeCamp	Javascript	9	8	0	33	0
grape	Ruby	0	0	0	8	0
Guava	Java	52	102	0	0	62
Nancy	C#	8	9	0	0	1
OkHttp	Java	32	61	0	12	3
PowerShell	C#	120	266	90	74	0
React	Javascript	57	39	0	6	6
React Native	Javascript	0	0	0	18	2
redux-form	Javascript	30	30	0	16	11
RestSharp	C#	14	0	0	15	1
Robolectric	Java	57	51	0	1	0
RuboCop	Ruby	38	27	0	10	4
Sass	Ruby	20	0	0	43	8
Spring Boot	Java	36	338	0	0	36
Total		**862**	**1457**	**144**	**336**	**238**

be the sets of typed dependencies (with the related *arguments*) involved in the conditions C_i and C_j, respectively, and (ii) let W_i and W_j be the sets of *lemmas* involved in C_i and C_j, respectively: C_i and C_j are equivalent if (i) $i = j$, (ii) $(TD_i \cup TD_j) - (TD_i \cap TD_j) = \emptyset$, (iii) $(W_i \cup W_j) - (W_i \cap W_j) = \emptyset$.

- **Definition 2**. Given two NLP heuristics, H_1, which incorporates m conditions, and H_2, containing n conditions:
  - H_1 and H_2 are *equivalent*, if $m = n$ and each C_i ($i \in \{1, 2, ..., m\}$) condition defined in H_1 is equivalent to one and only one C_j ($j \in \{1, 2, ..., n\}$) condition defined in H_2.
  - H_1 is *contained* in H_2, if $m < n$ and each C_i ($i \in \{1, 2, ..., m\}$) condition of H_1 is equivalent to one of the first m conditions, C_j ($j \in \{1, 2, ..., m\}$), of H_2.
  - H_2 is *contained* in H_1, if $m > n$ and each C_j ($j \in \{1, 2, ..., n\}$) condition of H_2 is equivalent to one of the first n conditions, C_i ($i \in \{1, 2, ..., n\}$), of H_1.

Hence, given two lists of heuristics L_i and L_j separately extracted for the P_i and P_j projects ($i \neq j$), we evaluated:

- the number of heuristics in L_i that are *contained* in heuristics belonging to L_j ($H_i \subset H_j$)
- the number of heuristics in L_j that are *contained* in heuristics belonging to L_i ($H_j \subset H_i$)
- the number of heuristics in L_i and L_j that are *equivalent* ($H_i \equiv H_j$).

4.3.1 Results of RQ4: shared linguistic patterns among different developers' communities

Results related to patterns extracted from the issue titles belonging to the selected projects are reported in Tables 12 and 13. Specifically, in Tables 12 and 13 we report: (i) the names of the considered projects ($P_1 - P_2$), (ii) the number of heuristics extracted from P_1 (PH_1), (iii) the number of heuristics extracted from P_2 (PH_2), (iv) the number of heuristics extracted from P_1 *contained* in heuristics mined from P_2 ($H_i \subset H_j$), (v) the number of heuristics extracted from P_2 *contained* in heuristics mined from P_1 ($H_j \subset H_i$), (vi) the number of heuristics from P_1 that are *equivalent* to heuristics mined from P_2 ($H_i \equiv H_j$), and (vii) the total number of patterns with shared conditions (SCH = $H_i \subset H_j$ + $H_j \subset H_i$ + $H_i \equiv H_j$).

TABLE 12
Patterns with shared conditions extracted from different projects for issues of type "bug".

P_1 - P_2	PH_1	PH_2	$H_i \subset H_j$	$H_j \subset H_i$	$H_i \equiv H_j$	SCH
AngularJS - CoreCLR	102	45	6	2	7	15 (33.33%)
AngularJS - PowerShell	102	39	2	3	7	12 (30.77%)
CoreCLR - PowerShell	45	39	3	2	1	6 (15.38%)

By observing Table 12, it is possible to notice that 15 out of 45 (33.33%) patterns extracted from the issues of the CoreCLR project have shared conditions with patterns mined from the project AngularJS, while 12 out of 39 (30.77%) of the patterns extracted for the PowerShell project have shared conditions with patterns extracted from the project AngularJS. Moreover, only 6 (out of 39, 15.38%) of the patterns identified for the PowerShell project present shared conditions with patterns extracted from the title of issues related to the CoreCLR and having the *"bug"* label assigned.

These percentages of similar patterns are slightly lower than the ones obtained for the *"enhancement"* issue type (as shown in Table 13). As a matter of fact, we observe that Guava generally share a high percentage of patterns with other projects. In particular, at least 10 out of 19 (52.63%) total patterns extracted from issue titles marked as *"enhancement"* and related to the Guava project, have shared conditions with patterns extracted from issue titles of the same type related to all other projects.

Although some of the identified recurrent patterns (*e.g.*, *"[something] returns [something]"*, *"[something] calls [something]"*) might not depend on the particular issue type, many of the extracted patterns are strictly connected with it. For instance, the patterns *"[something] fails"*, *"[something] do not work"*, and *"[something] throws [something]"* clearly indicate error descriptions and recurrently appear in the titles of the issues of the *"bug"* type related to all the inspected projects. Similarly, patterns such as *"[auxiliary] allow [something]"*, *"consider to use [something]"* and *"provide [something]"* often appear in the titles of issues related to different projects and having the *"enhancement"* label assigned.

On average, for the issue types we considered, about 35% of the patterns present in issue titles of a given project appear in titles of similar issues related to different projects. This means that around 65% of the recurrent linguistic patterns tend to be project-specific. However, these results vary depending on the topic and the project considered. An obvious consequence of our results is that automated solutions trained on project-specific data may achieve more accurate and reliable results for issue label prediction tasks. To this aim, approaches like NEON could be integrated into issue tracking systems, to silently collect project-specific recurrent patterns, with the aim of recommending the right label(s) to assign to a generic issue. On the one hand, a similar recommender system may stimulate developers to use issue labels more frequently. On the other hand, NEON can also be useful to easily mark unlabeled issues, in order to support bug prioritization processes [28], [38].

TABLE 13
Patterns with shared conditions extracted from different projects for issues of type "enhancement".

P_1 - P_2	PH_1	PH_2	$H_i \subset H_j$	$H_j \subset H_i$	$H_i \equiv H_j$	SCH
CoreCLR - Guava	185	19	1	5	4	10 (52.63%)
CoreCLR - PowerShell	185	105	3	8	18	29 (27.62%)
CoreCLR - Spring Boot	185	117	20	3	12	35 (29.91%)
Guava - PowerShell	19	105	4	2	5	11 (57.89%)
Guava - Spring Boot	19	117	10	0	0	10 (52.63%)
PowerShell - Spring Boot	105	117	20	5	4	29 (27.62%)

RQ4 summary: *About 35% of the linguistic patterns found in issue titles of a given project appear in titles of the same type related to different projects. Such a percentage can be considered quite relevant and indicate that, indeed, recurring patterns occur when describing — even across different projects — issues of the same type. This is especially true given that the considered issue types ("bug" and "enhancement") are quite broad in terms of problems developers can report.*

4.3.2 Threats to validity

Construct validity. In this study, the pairs of *patterns with shared conditions* (*i.e.,* the patterns that involve the same specific syntactic structures) are considered semantically similar. However, there could be pairs of linguistic patterns involving different grammatical structures that may be semantically related. This represents a threat to construct validity since semantically-related linguistic patterns with different sets of conditions could appear in issue titles of different projects with a non-negligible frequency. For alleviating this issue, not only pairs of *equivalent* patterns, but also pairs of patterns with inclusion relationships (*i.e., contained in*) have been considered in our computation.

Internal validity. In a specific project, a reduced set of items belonging to a particular issue type could result in the low variability of the related issue titles and, consequently, this could lead to erroneous estimations. To mitigate this threat, for each considered issue type (*e.g.,"bug", "enhancement"*, etc.), only projects having at least 100 issues of that type have been selected for extracting language patterns and computing our results.

External validity. Two threats could weaken the external validity of this study. The first one concerns the selection of the projects that could not be representative of all the communities of developers. To ensure a good representativeness and limit the impact of this threat on the generalization of the results, for each programming language we selected the top starred projects, *i.e.,* the most popular ones. This allowed us to collect a large volume of data that foster an acceptable inner variability, increasing the external validity of the findings. At the same time, the relatively limited number of projects considered implies that we analyzed a limited set of application domains. Therefore, we did not evaluate the impact of a specific domain on the NEON's effectiveness. Mitigating this threat would have meant to substantially enlarge the number of projects, which would have lead to more dependable findings. Considered that, to the best of authors' knowledge, this is the first study that investigates such a topic, the findings could be considered as a first result, which could be better analyzed with a larger study, to evaluate further factors that may influence the patterns developers use for defining the issues.

5 Related Work

Previous research has defined various approaches based on natural language grammatical/linguistic patterns, with the aim of supporting software engineering activities. More specifically, in a previous work [19], we identified a set of 231 linguistic patterns for automatically classifying informative paragraphs in development mailing lists. Pandita *et al.* [29] make use of natural language templates to infer formal method specifications from natural language API descriptions. Chaparro *et al.* [11] defined 154 recurrent patterns to describe observed behavior (OB), expected behavior (EB) and step to reproduce (S2R) in bug descriptions with the aim of detecting the presence (or absence) of these pieces of information in such kind of artifacts (*i.e.,* bug descriptions). Zhou *et al.* [45], [46] employed specific linguistic patterns for automatically detecting inconsistencies between API documents and source code. Frequent grammatical patterns (i.e., dependencies in which either the governor or the dependent is a code-like term) along with structural features were also used by Petrosyan *et al.* [36] to discover tutorial sections that explain a given API type.

In the context of requirements engineering, linguistic rules have been defined by researchers to mine informative data (*e.g.,* feature requests and bugs reported by users) from app reviews [13], [24], [25], [34]. Similar rules have also proven to be useful to summarize feedback from users with the purpose of recommending developers the software changes that need to be applied [17], [18], [31].

In all these studies, linguistic patterns were identified through manual processes. In most cases, the manual analysis of a large corpus of documents has been required to define a reliable set of rules able to achieve reasonably good performances in classifying/extracting target data.

To the best of authors' knowledge, this is the first work proposing an algorithm for automatically mining recurrent grammatical frames from a corpus of similar documents. The most similar work to our research has been proposed by Vu *et al.* [41], who developed PUMA, an automated phrase-based approach to extract user opinions in app reviews. Their approach automatically mines phrases templates (*i.e.,* sequences of POS tags) from a corpus of user reviews. App reviews matching frequent templates are then extracted, to be clustered. Differently from PUMA, NEON is able to discover common grammatical structures (*i.e.,* typed dependencies paths) existing between phrase elements (*i.e.,* words). Thus, while the PUMA phrase templates rely on the order in which the grammatical information appears (*i.e.,* sequences of POS tags), our approach is more aimed at analyzing the intrinsic dependencies existing between the sentences' elements. For this reason, our approach is less sensitive to tags misalignments that may occur in similar text fragments, since two natural language sentences may share a common grammatical structure, without matching the same sequence of POS tags (*e.g.,* the two sentences illustrated in Figure 3 present different sequences of POS tags).

6 Conclusions and Future Work

Informal documentation (*e.g.,* mailing lists, chats, bug reports, user feedback) is becoming increasingly important

in software development, and automated approaches for better managing information contained in such documentation are required. To this aim, the literature reports several approaches that treat textual information as a bag of words. While this strategy is simple and effective in some circumstances, it has been shown not to be adequate for identifying the *intention* behind a natural language text fragment [19]. Since the automated recognition of such *intentions* could help developers in detecting text content useful to accomplish different and specific maintenance and evolution tasks, in a previous work we proposed DECA [19], an approach leveraging language syntactical patterns to classify sentences' intent. Similar approaches have been proposed for several other purposes (*e.g.*, apps' reviews classification [23], [34], bug reports quality assessment [11], inconsistencies detection in API documents [45], [46]). However, the main disadvantage of such approaches is that they require the manual identification of recurrent language patterns that would be exploited for automated classification. To better comprehend the effort required to identify such patterns from the taggers' side, in a study with Master's and Ph.D. students we first found that the identification effort increases proportionally to the number and types of structures involved. To reduce such effort, we proposed NEON, an approach to automatically mine recurrent syntactical patterns from software informal documents. Two studies performed on different types of software informal documents (*i.e.*, app reviews, development emails) allowed us to assess NEON's time-saving capability, as well as the classification effectiveness of heuristics automatically identified by our tool.

Hence, we can conclude that NEON is a valid support for researchers interested in identifying recurrent linguistic patterns that are used by developers in software informal documentation: NEON-generated heuristics could represent a good initial set of heuristics that humans can further refine to achieve better classification performance. In addition, a further study on the issue descriptions reported in issue tracking systems indicated that NEON may be useful in collecting project-specific language patterns that can be leveraged to predict the label to assign to an issue.

As future work, to improve the patterns' identification capabilities of NEON, we plan to provide our approach with the ability to also detect semantically similar patterns, and not only the syntactically similar ones. Finally, we plan to test alternative and less rigid ways to represent linguistic patterns, which are not based on XML-based rules to facilitate the adoption of NEON in practical contexts.

ACKNOWLEDGMENT

We gratefully thank all the participants involved in our studies. S. Panichella and H. Gall acknowledge the Swiss National Science foundation's support for the project "*SURF-MobileAppsData*" (SNF Project No. 200021-166275).

REFERENCES

[1] 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 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*, 2008, p. 23.

[2] 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*, 2006, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1134336

[3] 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*, 2012, pp. 375–385. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227177

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

[5] G. Bavota, "Mining unstructured data in software repositories: Current and future trends," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, 2016, pp. 1–12.

[6] 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*, 2008, pp. 195–199. [Online]. Available: https://doi.org/10.1109/ICGSE.2008.17

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

[8] 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*, 2012, p. 44. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393647

[9] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella, "Improving IR-based traceability recovery via noun-based indexing of software artifacts," *Journal of Software: Evolution and Process*, vol. 25, no. 7, pp. 743–762, 2013. [Online]. Available: https://doi.org/10.1002/smr.1564

[10] L. Cerulo, M. Ceccarelli, M. Di Penta, and G. Canfora, "A hidden markov model to detect coded information islands in free text," in *13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, September 22-23, 2013*, 2013, pp. 157–166. [Online]. Available: https://doi.org/10.1109/SCAM.2013.6648197

[11] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di 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*, 2017, pp. 396–407. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106285

[12] P. Chatterjee, M. A. Nishi, K. Damevski, V. Augustine, L. Pollock, and N. A. Kraft, "What information about code snippets is available in different software-related documents? an exploratory study," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 382–386.

[13] A. Ciurumelea, S. Panichella, and H. C. Gall, "Automated user reviews analyser," in *ICSE (Companion Volume)*. ACM, 2018, pp. 317–318.

[14] J. Cohen, "Statistical power analysis for the behavioral sciences second edition," *Lawrence Erlbaum Associates, Publishers*, 1988.

[15] M.-C. de Marneffe and C. D. Manning, "The stanford typed dependencies representation," in *Coling 2008: Proceedings of the Workshop on Cross-Framework and Cross-Domain Parser Evaluation*, ser. CrossParser '08. Stroudsburg, PA, USA: Association for Computational Linguistics, 2008, pp. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1608858.1608859

[16] 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*, vol. 41, no. 6, pp. 391–407, 1990.

[17] 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*, 2016, pp. 499–510. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950299

[18] A. Di Sorbo, S. Panichella, C. V. Alexandru, C. A. Visaggio, and G. Canfora, "SURF: summarizer of user reviews feedback," in *ICSE (Companion Volume)*. IEEE Computer Society, 2017, pp. 55–58.

[19] 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*, 2015, pp. 12–23. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.12

[20] 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*, 2016, pp. 641–644. [Online]. Available: http://doi.acm.org/10.1145/2889160.2889170

[21] D. G. Feitelson, "Using students as experimental subjects in software engineering research - A review and discussion of the evidence," *CoRR*, vol. abs/1512.08409, 2015. [Online]. Available: http://arxiv.org/abs/1512.08409

[22] G. Grano, A. Ciurumelea, S. Panichella, F. Palomba, and H. C. Gall, "Exploring the integration of user feedback in automated testing of android applications," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 72–83.

[23] G. Grano, A. Di Sorbo, F. Mercaldo, C. A. Visaggio, G. Canfora, and S. Panichella, "Android apps and user feedback: a dataset for software evolution and quality improvement," in *WAMA@ESEC/SIGSOFT FSE*. ACM, 2017, pp. 8–11.

[24] C. Iacob and R. Harrison, "Retrieving and analyzing mobile apps feature requests from online reviews," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 41–44. [Online]. Available: https://doi.org/10.1109/MSR.2013.6624001

[25] C. Iacob, R. Harrison, and S. Faily, "Online reviews as first class artifacts in mobile app development," in *Mobile Computing, Applications, and Services - 5th International Conference, MobiCASE 2013, Paris, France, November 7-8, 2013, Revised Selected Papers*, 2013, pp. 47–53. [Online]. Available: https://doi.org/10.1007/978-3-319-05452-0_4

[26] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, June 2008, pp. 52–61.

[27] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.

[28] A. J. Ko, B. A. Myers, and D. H. Chau, "A linguistic analysis of how people describe software problems," in *Proceedings of the Visual Languages and Human-Centric Computing*, ser. VLHCC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 127–134. [Online]. Available: https://doi.org/10.1109/VLHCC.2006.3

[29] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. M. Paradkar, "Inferring method specifications from natural language API descriptions," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 815–825. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227137

[30] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. D. Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 522–531. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606598

[31] S. Panichella, "Summarization techniques for code, change, testing, and user feedback (invited paper)," in *VST@SANER*. IEEE, 2018, pp. 1–5.

[32] S. Panichella, G. Bavota, M. Di 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*, 2014, pp. 251–260. [Online]. Available: https://doi.org/10.1109/ICSME.2014.47

[33] S. Panichella, G. Canfora, M. Di Penta, and R. Oliveto, "How the evolution of emerging collaborations relates to code changes: an empirical study," in *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, 2014, pp. 177–188. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597145

[34] 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 *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, 2015, pp. 281–290. [Online]. Available: https://doi.org/10.1109/ICSM.2015.7332474

[35] L. Pelloni, G. Grano, A. Ciurumelea, S. Panichella, F. Palomba, and H. C. Gall, "Becloma: Augmenting stack traces with user review information," in *SANER*. IEEE Computer Society, 2018, pp. 522–526.

[36] G. Petrosyan, M. P. Robillard, and R. De Mori, "Discovering information explaining api types using text classification," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 869–879. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818859

[37] M. A. Pett, *Nonparametric statistics for health care research: Statistics for small samples and unusual distributions*. Sage Publications, 2015.

[38] A. Sureka and K. V. Indukuri, "Linguistic analysis of bug report titles with respect to the dimension of bug importance," in *Proceedings of the Third Annual ACM Bangalore Conference*, ser. COMPUTE '10. New York, NY, USA: ACM, 2010, pp. 9:1–9:6. [Online]. Available: http://doi.acm.org/10.1145/1754288.1754297

[39] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," in *Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics, HLT-NAACL 2003, Edmonton, Canada, May 27 - June 1, 2003*, 2003. [Online]. Available: http://aclweb.org/anthology/N/N03/N03-1033.pdf

[40] P. M. Vu, T. T. Nguyen, H. V. Pham, and T. T. Nguyen, "Mining user opinions in mobile app reviews: A keyword-based approach (t)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 749–759. [Online]. Available: https://doi.org/10.1109/ASE.2015.85

[41] P. M. Vu, H. V. Pham, T. T. Nguyen, and T. T. Nguyen, "Phrase-based extraction of user opinions in mobile app reviews," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 726–731. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970365

[42] K. Wang, Z. Ming, and T.-S. Chua, "A syntactic tree matching approach to finding similar questions in community-based qa services," in *Proceedings of the 32Nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '09. New York, NY, USA: ACM, 2009, pp. 187–194. [Online]. Available: http://doi.acm.org/10.1145/1571941.1571975

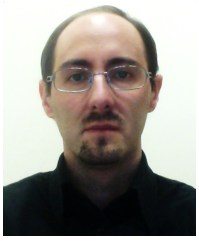
[43] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 562–567.

[44] Z. Wu and M. Palmer, "Verbs semantics and lexical selection," in *Proceedings of the 32Nd Annual Meeting on Association for Computational Linguistics*, ser. ACL '94. Stroudsburg, PA, USA: Association for Computational Linguistics, 1994, pp. 133–138. [Online]. Available: http://dx.doi.org/10.3115/981732.981751

[45] Y. Zhou, C. Wang, X. Yan, T. Chen, S. Panichella, and H. C. Gall, "Automatic detection and repair recommendation of directive defects in java api documentation," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[46] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing apis documentation and code to detect directive defects," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 27–37. [Online]. Available: https://doi.org/10.1109/ICSE.2017.11

Andrea Di Sorbo Andrea Di Sorbo is a postdoctoral researcher at the University of Sannio, Italy. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, text analysis and software security and privacy. He is an author of several papers appeared in flagship international conferences (ICSE, FSE, ASE) . He is Editorial Board member of Frontiers in Big Data. He serves and has served as program committee member of various international conferences (ARES, SEAA) and as reviewer for several journals in the field of software engineering, as Journal of Software: Evolution and Processes edited by Wiley, the Empirical Software Engineering journal edited by Springer, and IEEE Transactions on Software Engineering.



Massimiliano Di Penta Massimiliano Di Penta is an associate professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, search-based software engineering, and service-centric software engineering. He is an author of over 250 papers appeared in international journals, conferences, and workshops. He serves and has served in the organizing and program committees of more than 100 conferences, including ICSE, FSE, ASE, ICSME. He is in the editorial board of the Empirical Software Engineering Journal edited by Springer, and of the Journal of Software: Evolution and Processes edited by Wiley, and has served the editorial board of the IEEE Transactions on Software Engineering.

Sebastiano Panichella Sebastiano Panichella is a passionate Computer Science Researcher at Zurich University of Applied Science (ZHAW). His research interests are in the domain of Software Engineering (SE) and cloud computing (CC).He is author of over forty papers appeared in International Conferences (ICSE, ASE, FSE, ICSME, etc.) and Journals (EMSE. IST, etc.). These research work involved studies with industrial companies and open source projects and received best paper awards. He serves and has served as program committee member of various international conference (e.g., ICSE, ASE, ICPC, ICSME, SANER, MSR). Dr. Panichella was selected as one of the top-20 (second in Switzerland) Most Active Early Stage Researchers (Results reported by the JSS journal) in Software Engineering (SE). He is a member of IEEE. He is Editorial Board Member of Journal of Software: evolution and process (JSEP). He is also Review Board member of the EMSE journal.

Gerardo Canfora Gerardo Canfora is a professor of computer science at the School of Engineering of the University of Sannio, Italy. He serves on the program and organizing committees of a number of international conferences. He was general chair of WCRE06 and CSMR03, and program co-chair of ICSE15, WETSoM12 and 10, ICSM01 and 07, IWPSE05, CSMR04 and IWPC97. He is co-editor of the Journal of Software: Evolution and Processes. Canfora authored 200 research papers; his research interests include software maintenance and evolution, security and privacy, empirical software engineering, and service-oriented computing.

Corrado A. Visaggio Corrado Aaron Visaggio is associate professor of CyberSecurity at the Department of Engineering of University of Sannio. He is chair of the node of University of Sannio for the CINI National Cyber Security Lab. He is the scientific coordinator of several projects funded by firms operating in CyberSecurity, concerning malware analysis, vulnerability assessment, and data protection. He serves in the Editorial Board of the International Journal of Computer Virology and Hacking techniques (Springer), as associate editor in Frontiers in Big Data, and in several Program Committees (MALWARE, ARES, SECRYPT, SEKE, ITASEC, FoRSE, DATA, Hufo, MobiSys, WETSOM, ISSRE); he was also the workshop chair of WETSOM and WMA. His main research interests are: malware analysis, data privacy and protection, software security, empirical software engineering.

Harald C. Gall Harald Gall is Dean of the Faculty of Business, Economics, and Informatics at the University of Zurich, Switzerland (UZH). He is professor of Software Engineering in the Department of Informatics at UZH. He studied at the Technical University in Vienna, Austria, and holds a PhD (Dr. techn.) and master's degree (Dipl.-Ing.) in Informatics. His research interests are in evidence-based software engineering with focus on quality in software products and processes. This focuses on long-term software evolution, software architectures, software quality analysis, data mining of software repositories, cloud-based software development, and empirical software engineering. He is probably best known for his work on software evolution analysis and mining software archives. Since 1997 he has worked on devising ways in which mining these repositories can help to better understand software development, to devise predictions about quality attributes, and to exploit this knowledge in software analysis tools such as Evolizer or ChangeDistiller.