Neural Computing and Applications (2024) 36:13051–13085
https://doi.org/10.1007/s00521-024-09730-x

ORIGINAL ARTICLE

Natural language requirements testability measurement based on requirement smells

Morteza Zakeri-Nasrabadi[1] · Saeed Parsa[1]

Received: 2 March 2022 / Accepted: 25 March 2024 / Published online: 24 April 2024

Abstract
Requirements form the basis for defining software systems' obligations and tasks. Testable requirements help prevent failures, reduce maintenance costs, and make it easier to perform acceptance tests. However, despite the importance of measuring and quantifying requirements testability, no automatic approach for measuring requirements testability has been proposed based on the requirements smells, which are at odds with the requirements testability. This paper presents a mathematical model to evaluate and rank the natural language requirements testability based on an extensive set of nine requirements smells, detected automatically, and acceptance test efforts determined by requirement length and its application domain. Most of the smells stem from uncountable adjectives, context-sensitive, and ambiguous words. A comprehensive dictionary is required to detect such words. We offer a neural word embedding technique to generate such a dictionary automatically. Using the dictionary, we could automatically detect Polysemy smell (domain-specific ambiguity) for the first time in 10 application domains. Our empirical study on nearly 1000 software requirements from six well-known industrial and academic projects demonstrates that the proposed smell detection approach outperforms Smella, a state-of-the-art tool, in detecting requirements smells. The Precision and Recall of smell detection are improved with an average of 0.03 and 0.33, respectively, compared to the state-of-the-art. The proposed requirement testability model measures the testability of 985 requirements with a mean absolute error of 0.12 and a mean squared error of 0.03, demonstrating the model's potential for practical use.

Keywords Requirements engineering · Requirements testability · Natural language processing · Neural word embedding

1 Introduction

Requirements are the foundation upon which software systems are built. A minor deviation from requirements is strictly a failure that may cause fundamental damages to the ultimate software product. Many failures in software systems stem from poor requirements definition [1]. Correctly understanding what the system must do is critical to its success and validity [2]. Requirements should be tested in different stages of the software development life cycle (SDLC) to ensure they are understood and met appropriately.

Requirements testability is the degree to which requirements can be tested [3]. ISO/IEC/IEEE 24765 standard defines requirements testability as the "extent to which an objective and feasible test can be designed to determine whether a requirement is met" [4]. Unfortunately, few studies have focused on measuring and quantifying requirements testability [5].

The root cause of software failure is the misunderstanding of the requirements due to the requirements smell. However, it is observed that instead of requirements testability, researchers have focused on source code testability [6–10], which is itself affected by the testability of the requirements. In other words, instead of addressing the cause, they have focused on resolving the symptoms.

In functional and acceptance testing, requirements are considered obligatory artifacts, conducting the tests toward

✉ Saeed Parsa
parsa@iust.ac.ir

Morteza Zakeri-Nasrabadi
morteza_zakeri@comp.iust.ac.ir

[1] School of Computer Engineering, Iran University of Science and Technology, Hengam St., Resalat Square, Tehran, Tehran, Iran

the users' expectations of the software [11, 12]. Most of the proposed techniques for measuring requirements testability are domain specific [13], semi-automated [14], or based on a too small dataset for reliable machine learning training [3], making them inappropriate to use in most real-world applications.

Requirements testability is automatically measurable, provided that any smells can be detected and identified in the requirement definition. A *requirement smell* is a quality issue that can be improved by modifying the requirement definition via removing any ambiguity, misunderstandings, or difficulties with the validation and design of acceptance tests. Femmer et al. [15, 16] have introduced nine requirement smells and created a tool, Smella, to detect eight of the nine smells. Smella utilizes natural language processing (NLP) techniques to detect four out of the eight smells. The utilized NLP techniques are part of speech (POS) tagging [17, 18], lemmatization [17], and morphological analysis [19]. It recognizes the other four smells, including Subjective language, ambiguous adverbs and adjectives, loopholes, and Non-verifiable terms, by simply searching the requirement's words in a predefined dictionary of the smelly words. Their dictionary has been created manually by software engineering experts.

Most automated techniques for detecting ambiguity in requirements definitions rely on a predefined dictionary of ambiguous words [16, 20–22]. These dictionaries include words that are generally regarded as ambiguous [23, 24]. For instance, two requirements analysis tools, Quality Analyzer of Requirements Specifications (QuARS) [21] and Systemized Requirements Engineering Environment (SREE) [22], use domain-independent dictionaries, including vague words in every domain. Hence, the degree of detected smells by these tools is somewhat context independent. However, some words are not vague, but their definitions vary depending on how they are used. The ones most difficult to handle are the *polysemous* words having different meanings from computer terminology in domains other than computer science. Considering the impacts of context-sensitive and vague words on misunderstanding and, consequently, reducing the testability of requirements, a comprehensive dictionary including both ambiguous and context-sensitive words could be promising.

We found it challenging to prepare a suitable list of smelly words for all application domains, and there is no standard and comprehensive dictionary with such a purpose. Software requirements may be expressed in various languages other than English. Therefore, a systematic and language-agnostic approach is required to create a desirable dictionary of smelly words, along with their smell types. The only dictionary including context-sensitive nouns in five different domains is introduced in [25]. Ferrari et al. [25–27] have proposed a method based on *word embedding* to automatically detect domain-specific nouns by comparing their cosine similarities in five different domains. They have only focused on nouns, while other parts of speech such as verbs, adverbs, and adjectives may cause ambiguity. We extend their dictionary to include smelly words rather than solely nouns in ten different domains. Our automatically generated dictionary is of great use to determine the clarity and testability of textual requirement artifacts.

In this paper, we define requirements testability as a function of requirement smells and quantify the value of testability for any given requirement without imposing any template [28] for requirement definitions or restrictions on the application domain [13]. To this aim, a tool for automatically detecting nine requirement smells, boosted by a comprehensive dictionary of smelly words, is proposed. To create such a dictionary, inspired by [25], we identified the 1000 most frequently used words, common in the computer science domain and ten other application domains. Different meanings of each frequent word were extracted using word embedding and vector similarity on a large corpus of Wikipedia documents. In summary, the following contributions are made by this study:

i. To offer a new mathematical model, defining requirements testability in terms of the requirements smells, size of the requirements, and application domain. Our empirical studies, described in Sect. 4.5, demonstrate the correctness and applicability of the proposed requirements testability measurement equation.
ii. To automatically generate a dictionary of *smelly* or *context-sensitive* words having different meanings in different application domains and the word's smell types. A neural Word2Vec model is trained to calculate the cosine similarity between frequent words commonly used in computer science and ten other application domains.
iii. To introduce two new types of requirement smells, called *Polysemy* and *Uncertain Verbs*. The definitions are given in Sect. 3.2.
iv. To introduce the first publicly available requirements testability dataset, including requirements samples, their smells, and testability measures. The dataset has been manually annotated and validated by requirements engineer experts for natural language requirements in English. The annotated requirements dataset is available at https://doi.org/10.5281/zenodo.4266727.

The proposed approach is supported by a web-based requirements smell detection tool that automatically highlights smelly words and determines the testability of each input requirement. Our tool named Automatic

Requirements Testability Analyzer (ARTA) is a web application with a rich graphical user interface. The application receives a requirement as input, determines the smelly words visually, and reports its testability value. Researchers and practitioners can define a project, import all project requirements from an Excel file, and manually specify smelly words for each requirement. A web-based tool facilitates creating and managing large requirements datasets, while it can be used to service software developers, testers, and stakeholders to meet high-quality requirements. The ARTA tool is publicly available at https://m-zakeri.github.io/ARTA.

The remaining parts of this paper are organized as follows. In Sect. 2, the related works are discussed. Section 3 explains our proposed method for measuring software requirements testability. Section 4 evaluates and discusses the results of our experiments with the proposed method. Threats to validity are discussed in Sect. 5. Finally, Sect. 6 concludes the paper and outlines future works.

2 Related work

There have been significant advances in studying ambiguities in natural language requirements specifications as a basis for evaluating the quality of the requirements [29]. However, in total, we could find only a few studies explicitly on requirements testability [3, 13]. Despite several standards by IEEE, ISO, and IET [4, 30–32] defining software requirements testability, there has been no major intention to quantify the requirements testability.

Table 1 summarizes the related works in terms of their topics, techniques, evaluations, key advantages, and disadvantages. Considering the column labeled "Approach/Technique," it is observed that most of these approaches [14, 16, 20, 21, 23] evaluate a given requirement text by looking up the words in a dictionary of smelly words. Moreover, this column shows that dictionaries are mainly built manually by experts and researchers. The other observation, inspired by the last column, is that most dictionaries contain inherently ambiguous words [16, 20, 23]. They do not include domain-specific and context-sensitive ambiguous words. Column 2, labeled "Topic," shows that despite the strict emphasis on the potential impact of ambiguity on testability, defined by several standards, there are only a few approaches [3, 20] directly addressing the measurement of requirements testability. The column, labeled "Evaluation," shows that most approaches provide Precision and Recall. However, their datasets are not available.

The ambiguity detection method, called Automated Requirements Measurement (ARM) [20], had been applied to 46 requirement specification files for the software assurance technology center (STAC) projects in National Aeronautics and Space Administration (NASA). However, they have not provided any report on the Precision and Recall of their evaluation results. Their report is only a quantitative overview of their evaluation results on 46 specifications from NASA.

Another semi-automated approach, called Quality Analyzer of Requirements Specifications (QuARS) [21, 33], has been applied to requirement documents taken from four industrial projects. They have reported the number of selected defects in the requirements definitions. Their dictionaries only contain inherently ambiguous words in the English language, and they have not evaluated their system in terms of Precision and Recall. The QuARS tool has been recently applied to detect variability in natural language requirement documents [39].

Chantree et al. [40] have proposed a technique that automatically alerts authors of requirements to the presence of nocuous ambiguities based on four different heuristics and their combinations. Their heuristics includes coordination matches, distributional similarity, collocation frequency, and morphology. However, they have focused on detecting nocuous ambiguities, confirmed by multiple readers in multi-readings.

Gleich et al. [23] have automated the detection of six types of ambiguity using guidelines for ambiguity detection defined in the Ambiguity Handbook [24]. Their tool uses manually predefined dictionaries and light NLP techniques such as POS tagging to detect ambiguous words. It has been evaluated on 50 German and 50 English sentences with known ambiguities marked by experts. The reported Precision varies between 34% for the pure experts' opinion and 97% for a more guideline-based gold standard. The relation between ambiguity and requirement quality attributes, e.g., testability, has not been discussed.

Yang et al. [34] have studied anaphoric ambiguity, which occurs when readers may disagree on how pronouns should be interpreted. They have built some machine learning models to classify instances of ambiguity into nocuous or innocuous based on a set of 17 features made up of their heuristics scores. A set of 200 anaphoric instances manually labeled by experts have been used to evaluate their learned models' performance. An accuracy of 76.25% has been reported for their best model. Their dataset, containing only 200 instances, seems to be very small for detecting such ambiguities. A larger dataset may increase their model accuracy.

Systemized Requirements Engineering Environment (SREE) [22] is another ambiguity-finding tool (AFT) based on manually collected dictionaries of ambiguous words. SREE aims at the detection of ambiguities with a Recall of 100%. To achieve this, the authors have avoided using any NLP techniques, and instead, they have collected a large

Table 1 Summary and comparison of semi/fully automated requirement quality assurance approaches

Tool and publications	Topic	Approach/Technique	Evaluation	Key advantages	Key disadvantages
ARM [20]	Requirement quality attributes and indicators	- Manual dictionary - Size metrics - Readability metrics	Visualization and interpretation	Considering different quality aspects	- Limited dictionary of general ambiguous words - Weak empirical evaluation
QuARS [21, 33]	Quality model for requirements	- Manual dictionary - Syntactical analyzer	- Real examples - Quantification	Modular design for multi-lingual usage	- Limited dictionary of general ambiguous words - Weak empirical evaluation
[23]	Ambiguity detection	- Manual dictionary (Ambiguity Handbook) - POS tagging -Regular expressions	- Precision: 34–97% - Recall: 53–86%	- More comprehensive dictionary - Strong empirical evaluation	- Only detect general ambiguities - The tool is not available
[34]	Ambiguity detection (Anaphoric analysis)	Machine learning (classification)	- Accuracy: 76.25%	Strong feature engineering	Small training set
SREE [22]	Ambiguity detection	Manual dictionary (Ambiguity Indicator Corpus)	- Precision: 68% - Recall: ≈ 100%	Very high (robust) Recall	Relativity low Precision
[35]	Ambiguity detection	- SREE [22] - Regular expressions	- Precision: 83.16% - Recall: 85.39%	Strong empirical evaluation	Limited to requirements in the railway domain
RQA [14]	Requirement quality attributes and indicators	- Manual dictionary (Forbidden Terms) - Step function	-	Commercial tool supported	No empirical evaluation
[3]	Testability prediction	Machine learning (classification)	- Precision: 98% - Recall: 97% - ROC: 99%	A fully automated approach for testability measurement	A very small dataset with 89 requirement samples
Smella [16]	Ambiguity detection (Requirements Smells)	- Manual dictionary - Lemmatization - POS tagging	- Precision: 48% - Recall: 87%	- Follows ISO/IEC/IEEE 29148:2011 standard - Strong empirical evaluation	- No public tool, and datasets
[25–27]	Ambiguity detection (Cross-domain)	Word embedding	- Kendall's Tau: 88% - Real examples	Detecting cross -domain ambiguity fully automated	Only detects ambiguous nouns in five domains
REVV -Light [36]	Ambiguity detection (Near-synonyms)	Semantic Folding Theory [37]	- Precision: 51% - Recall: 25%	Open-source tool supported for analyzing and visualizing user-stories	- A semi-automated method required manual assessment - Only works on user stories written in a specific template
[38]	Ambiguity detection (Coordination ambiguity and prepositional -Phrase attachment ambiguity)	- POS tagging - Rule set - Keywords extraction	- Precision: 80% - Recall: 89%	Low false positive rate	- Manually defined patterns - Low coverage of patterns

dictionary of ambiguous words called Ambiguity Indicator Corpus (AIC). A careless increase in Recall has led to a pretty low Precision of 68%, as reported by SREE's creators, while they could not achieve a 100% Recall at all. In addition, SREE's AIC does not include any Polysemy words.

Ferarri et al. [35] have combined SREE [22] with a set of manually defined language structure patterns to find requirement defects in the railway domain. After several refinements of their patterns, they could achieve a Precision of 83.16% and a Recall of 85.39% on a dataset of 1866 railway requirements. Therefore, the obtained patterns are likely to depend on their dataset.

Génova et al. [14] have used a list of 11 desirable properties, three of which are understandability, unambiguity, and atomicity. They used their own metrics and threshold to quantify and evaluate these properties for textual requirements. However, manual determination of the thresholds to rank requirements definitions without any empirical evaluations is not generally reliable.

Hayes et al. [3] have used logistic regression to predict requirements testability based on several measures, including the number of predicates, number of words in each requirement, and number of complex words, by manually labeling requirements as testable and non-testable. They have used a dataset consisting of 89 requirements extracted from two projects to implement their approach. However, 89 samples cannot provide an accurate machine learning prediction model.

Femmer et al. [15, 16] have introduced the notion of requirement smell in the context of requirements engineering. They have offered a requirement evaluation tool, Smella, which can detect eight different smells in requirements written in the German language. Their handmade dictionary could detect four types of these smells. We have extended their lists of smells with two newly defined smells and have proposed a new approach to the automatic construction of a dictionary of smelly words.

Ferrari et al. [25–27] have proposed a method based on word embedding [41] to detect domain-specific ambiguities in software requirements. They determine the similarity of typical computer science nouns, e.g., "interface" and "database," when used in five different domains. They have only focused on nouns, while other parts of speech, such as verbs, adverbs, and adjectives, may cause ambiguity in the requirements definition. We extend their approach to include ten different application domains. We also include different parts of speech available in user requirements to create a comprehensive dictionary for detecting various smells and then measure requirements testability.

Dalpiaz et al. [36] have blended information visualization with NLP techniques, conceptual model extraction, and semantic similarity to find near-synonym terms in user stories. They specifically search for a type of terminological ambiguity, called Near-synonyms or Plesionyms, which are distinct terms referring to the same denotation. Their approach is limited to user stories with predefined templates. Their results reject the hypotheses that automatic ambiguity detection outperforms manual inspection in terms of Precision and Recall.

Ezzini et al. [38] have used a set of 39 manually defined patterns based on part-of-speech (POS) tags to detect coordination ambiguity (CA) and prepositional-phrase attachment ambiguity (PPA) [24, 42]. The co-occurrence frequency of ambiguous phrase candidates is then found in the application domain of the requirement by crawling the relevant Wikipedia documents to filter the false positive results. Manually defined patterns can be fit for multiple types of ambiguity simultaneously.

In general, it is observed that Precision and Recall of semi-automated approaches are highly dependent on the manual dictionaries provided by the researchers. The Precision could be improved by applying comprehensive dictionaries dictionaries not restricted to the expert's knowledge. Moreover, there is no public dataset of the natural language requirement with known smells to enable a fair comparison of different approaches.

We use NLP techniques, mainly lemmatization [17], POS tagging [18], and word embedding [41], to automatically detect requirement smells and measure software requirements testability, which none of the related works has used yet. Besides, we provide a publicly accessible web-based tool, ARTA, to measure natural language requirement clarity and testability. ARTA provides the user with a text editor to modify any detected requirement smell. Some commercial tools use lightweight NLP techniques to manage requirement quality issues [43, 44]. However, to the best of our knowledge, ARTA is the first open-source tool dedicated to requirement quality management that is not restricted to a specific domain or template. Other freely accessible web-based tools, such as Quality Analyser for Official Documents (QuOD) [45], identify natural language defects in business processes. Grammarly [46] detects issues in general English language documents that are not strictly relevant to requirements artifacts.

Our work is related to some recent papers that deal with different aspects of neural networks, such as state estimation, stability, control, fuzzing, and impulsive systems [47–51]. These papers use neural networks to model, analyze, or control complex systems, such as reaction–diffusion neural networks, interconnected nonlinear systems, fuzzing, or impulsive systems. They also deal with the challenges or uncertainties of neural networks, such as delays, impulses, attacks, or quantization. They evaluate

the performance or quality of neural networks, such as stability, robustness, or efficiency. Our work differs from these papers in that we use neural networks to generate a dictionary of smelly or context-sensitive words rather than to model or control systems. Furthermore, we evaluate the testability of requirements rather than the stability or performance of neural networks. However, our work also complements these papers in that we share some common objectives, such as understanding the complex interaction among inputs from different application domains.

3 Methodology

We propose a novel methodology for measuring and improving the testability of natural language requirements. Our methodology consists of four main contributions: (i) a mathematical model that defines requirements testability as a function of requirements smells, size, and application domain; (ii) an automated dictionary that identifies smelly or context-sensitive words for different application domains and calculates their semantic similarity using a neural Word2Vec model; (iii) two new types of requirement smells, Polysemy and Uncertain Verbs, that affect the testability of requirements; and (iv) a public dataset of requirements testability, with annotated and validated requirements samples, smells, and testability scores. We demonstrate the correctness and applicability of our methodology through empirical studies.

This section first formalizes requirements testability, $T(R)$, as a function of requirement smells and length of the requirements as the number of sentences in the requirement definition. Afterward, it is described how to detect requirements smells and eventually calculate the testability. We employ a well-known goal-question-metric (GQM) approach [52], illustrated in Table 2, to build and evaluate our proposed requirements testability measurement methodology.

The purpose is to quantify and measure the testability of the requirement artifacts (objects) defined in a natural language from the viewpoint of the requirement engineers. Quantifying requirements testability into an interpretable range of (0, 1] provides several benefits to requirement engineers:

i. To measure or estimate the cost and effort required to improve the requirement artifact before applying any validation activity, such as acceptance testing.
ii. To prioritize requirement artifacts based on their extent of testability and give focus on untestable parts first based on time and budget dedicated to the validation process.
iii. To reveal the existing gap between the domain experts and software developers' terminologies at the

Table 2 Goal-question-metric (GQM) approach used to measure requirements testability

Goal	Purpose issue object (products) viewpoint	To measure the testability of the requirement artifacts from the viewpoint of the requirement engineer
Question	Q1 Group 1	How many sentences and words are in the requirements?
Metrics	M1	Number of sentences in the requirement
	M2	Number of words in the requirement
Question	Q2 Group 1	What is the grammatical role of each word in the requirement?
Metrics	M3	Part Of Speech (POS) tags dictionary
Question	Q3 Group 2	To which extent is the requirement smelly?
Metrics	M4	Number of smelly words in the requirements
	M5	Type of smells
Question	Q4 Group 2	How clean is a given requirement in the natural language?
Metrics	M6	Requirement clarity, C(R), a function of M2, M3, M4, and M5.
Question	Q5 Group 3	What is the required test effort/cost of automatically or manually analyzing a single statement in the requirement from the viewpoint of the requirements engineer?
Metrics	M7	$\alpha \in [0, 1)$, a subjective factor determined/estimated by the requirement engineer based on the inherent criticality of the requirement, the application domain, and the importance of the validation process.
Question	Q6 Group 3	What is the testability value of a given requirement, defined in natural languages?
Metrics	M8	Requirement testability, T(R), a function of M1, M6, and M7.

beginning of the software development life cycle (SDLC) process.

iv. To distinguish the requirements that support the automatic acceptance test case generation algorithms such as the one recently proposed by Fischbach et al. [53].

In summary, our requirements testability formulation can be used as a concrete proxy to estimate the requirement debt often raised by the requirements engineering phase and help optimal decision-making from the viewpoint of project managers.

The underlying assumption is that requirement smells negatively affect the testability of requirements. Considering a requirement artifact as a standalone objective to be met by a part of the software product, the first group of questions, in Table 2, is concerned with the quality of the requirement artifacts in terms of the number of sentences and lexicon needed to describe the requirement. The second group of questions aims at characterizing the attributes of the requirement artifact that are relevant to its testability, e.g., requirement smells, and finally, the third group of questions concerns evaluating the testability of requirement artifacts.

3.1 Formalism

Requirements testability is of great concern for acceptance testing. Requirement smells are symptoms of the poor definition of software requirements, which leads to poor testability. The more smelly words, *i.e.*, words causing any kind of ambiguity in requirement definition, in a given requirement, R, the less testable the requirement will be:

$$T(R) \propto \frac{1}{n(w_R^S)} \tag{1}$$

where $n(w_R^S)$ is the number of smelly words in the requirement, R. Requirements smells and the smelly word dictionary are further described in Sects. 3.2 and 3.3.

As the length of a requirement statement increases, its testability will inversely decrease because, firstly, it is relatively more difficult to prepare test cases for a lengthy requirement definition rather than a short one. Secondly, there is a higher chance of having smells in a lengthy requirement.

The probability of untestability grows as the number of sentences in a requirement definition increases. Good scientific writing practice suggests writing short sentences. Short sentences are the crux of good scientific writing [54]. The reason is that short sentences are easier to understand. Similarly, describing a requirement in one short sentence will be much easier to understand and test rather than using several sentences. Multiple sentences of a requirement have been considered a negative factor in requirements quality by many researchers [21, 55–57]. Therefore, the testability, $T(R)$, of requirement, R, is also proportional to the inverse of the length of the requirement statement, $n(S_R)$:

$$T(R) \propto \frac{1}{n(S_R)} \tag{2}$$

We follow the metrics in Table 2 to formulate requirements testability according to relations 1 and 2. We begin by defining ideal definitions of natural language requirements, which form the cornerstone of our methodology.

Definition 1 **Clean requirement**: A software requirement that does not suffer from any known requirement smell is called a clean requirement.

Definition 2 **Fully testable requirement**: A software requirement that is clean and expressed in solely one sentence is called a fully or completely testable requirement.

These definitions indicate an upper bound (best-case) for two notions of *requirement clarity* and *requirement testability*, formulated as follows. Considering Definition 1, the *clarity*, $C(R)$, of the requirement, R, depends on the number of smelly words, $n(w_R^S)$, and the number, t, of different types of smells appearing in the requirement statement. Equation 3 determines requirement clarity, $C(R)$:

$$C(R) = \begin{cases} 1 & R \text{ is clean} \\ 1 - \left(\frac{n(w_R^S)}{n(w_R)}\right)^{\frac{1}{t}} & \text{otherwise} \end{cases} \tag{3}$$

where $n(w_R)$ is the total number of words in requirement R.

Considering Definition 2, the testability, $T(R)$, of a requirement, R, could be measured in terms of its clarity, $C(R)$, and its length, $n(S_R)$ as follows:

$$T(R) = \frac{C(R)}{(1+\alpha)^{n(S_R)-1}} \tag{4}$$

where $\alpha \in [0, 1)$, is a subjective factor, determined or estimated by the requirement engineer based on the inherent criticality of the requirement, the application domain, and the importance of the validation process.

According to Eq. (4), $T(R)$ depends on both the requirement clarity, $C(R)$, and the requirement length in terms of the number of sentences in the requirement, R.

The testability, $T(R)$, is reduced exponentially as the number of sentences, $n(S_R)$, increases based on our testability model. Moreover, the number of smelly words and the type of smells in the requirement statement negatively affect the requirement clarity, $C(R)$, which results in decreasing requirements testability. For example, if a requirement is ambiguous or vague, it may be challenging to design a test case verifying its fulfillment or non-fulfillment. Similarly, if a requirement is incomplete or inconsistent, it may lead to confusion or conflicts among stakeholders or developers and thus hampers the testing process [3, 16, 58].

The only hyperparameter in our testability model is α. It determines the impact of the size, in terms of the number of the sentences, $n(S_R)$, of the requirement, R, on the testability, $T(R)$, of the requirement definition. If $\alpha = 0$ then, $T(R) = C(R)$. In this case, the number of statements in a requirement definition does not impact its testability. It should be noted that Definition 2 specifies a *sufficient* and not *necessary* condition for a requirement to be fully testable. Indeed, a testable requirement may be expressed in more than one sentence. If each sentence in a requirement definition complements its former sentences, the value of α is 0. However, if each sentence in a requirement definition targets a different objective, the value of α approaches one. In such cases, the testability, $T(R)$, is reduced exponentially as the number of sentences, $n(S_R)$, increases based on our testability model.

The value of α also reflects the inherent difficulty of testing a system requirement, affected by the nature of its *domain*. For instance, understanding and validating a single requirement in a safety-critical system such as a self-driving car most probably need more effort than a business-critical system such as accounting. Moreover, it is relatively more challenging to test a business requirement than a simple function/non-functional requirement. If a business requirement is not defined appropriately, it does not matter how well the project is delivered; still, the business is not satisfied, and it is not right to blame the stakeholder. The computation of the α parameter is described in Sect. 3.4.

A sentence is the smallest meaningful unit of a language that can stand independently. At least one sentence is required to express a software requirement in any natural language regardless of the application domain, system criticality level, requirement type, and template. As a result, the first sentence should not impose any testing cost/effort for a given requirement. That is why our suggested formula to compute the requirement testability ignores the first sentence when computing the cost of tests.

According to Eq. (4), the requirements testability is also reduced as the number of smelly words, and the type of smells in the requirement statement increases, *i.e.*, requirement clarity decreases. More formally, our definition of requirements testability, given by $T(R)$, is based on Fuzzy logic instead of classical Binary logic, which brings a vast opportunity to use in the automated requirement analysis approaches, especially data-driven and learning-based analysis. The flexible definitions of Eqs. (3 and 4) allow emerging requirement smells and issues to be considered in calculating requirements clarity and testability. Requirement smells are the subject of the next section.

3.2 Requirements smells

Requirements smell cause misunderstandings in the interpretation and analysis of requirements. Such misunderstandings lead to flaws in design and costly acceptance tests. The catalog of eight requirement smells, along with their detection procedure, has been proposed by Femmer et al. [16]. We discovered that two requirement smells, *Loopholes* and *Open-ended*, are not distinguishable by their definitions [16]. Therefore, we merged them under the umbrella of *Non-verifiable terms*. Moreover, we coined two new requirements smells, named *Polysemy* and *Uncertain Verbs*. Table 3 summarizes the definitions of each requirement smell and its detection mechanism.

Smells related to the grammatical aspects of requirements (S4-S7) are detected via POS tagging techniques [18] in which the grammatical role of each word, e.g., noun, verb, adjective, adverb, or pronoun, is determined using linguistic and probabilistic models such as Hidden Markov Models [17]. Other smells (five out of nine smells) are identified using a dictionary. The *lemma* of each word [17] within the requirement is searched in a predefined dictionary of smelly words. If the word is found in that dictionary, the requirement is tagged with the corresponding smell name. Uncertain Verbs could be detected based on a small dictionary of English modals. However, the other four smells (S1, S2, S3, and S9) are more challenging to detect. Section 3.3 proposes an automated approach to make a comprehensive dictionary of *smelly words*.

3.3 Smelly words dictionary

Four out of nine types of smelly words, including Polysemy words, Subjective language, Ambiguous adverbs/adjectives, and Non-verifiable terms, could be looked up in

Table 3 Requirement smells used to measure requirements testability

Smell no.	Name	Definition	Example	Detection	References
S1	Subjective language	The words for which the semantic is not defined objectively	User friendly	Dictionary	[16]
S2	Ambiguous adverb/ adjective	Certain adverbs and adjectives that are unspecified by nature	Almost	Dictionary	[16]
S3	Non-verifiable terms	The terms that are difficult to verify as they offer a choice of possibilities or imply the fulfillment of a particular but imprecisely defined extent	Sufficient, as far as possible	Dictionary	This work, [16]
S4	Superlatives	Superlatives refer to adjectives in requirements that express a relation of the system to all other systems	Highest	POS tagging	[16]
S5	Comparative	Comparatives refer to adjectives in requirements that express a relation of the system to specific other systems or previous situations	More exact	POS tagging	[16]
S6	Negative statement	The statements that express a specific capability not to be provided by the system	Must not sign off	POS tagging	[16]
S7	Vague pronouns	The pronouns whose reference or relation is not clear to the reader base on the context	Which	POS tagging	[16]
S8	Uncertain Verbs	Verbs without reasonable certainty, less than 100%, are unclear and may cause doubt when evaluating a requirement	May, can	Small Dictionary	This work
S9	Polysemy	The coexistence of many possible meanings for a word or phrase base on the context	Call, return	Dictionary	This work

a dictionary, including the smelly word and its type. The dictionary proposed by Femmer et al. [16] is built manually for German lexicons. However, we could automatically build a dictionary of smelly words and their smell types using word embedding techniques. We applied a neural word embedding technique with the Continuous Bag of Words (CBOW) architecture [59] to build the dictionary on a large corpus of Wikipedia documents and extracted 1000 candidate words in 10 different domains of science and engineering. The CBOW architecture has been shown to be an effective and efficient model to find semantics relations between words in a textual corpus [59, 60].

The *Word2Vec* algorithm, applied by the word embedding technique, provides a vector representation for each selected word. For each frequent word in computer science, we computed the cosine similarity of its vector in the computer science contexts with its vectors in the other domains. A word is considered smelly, provided that its

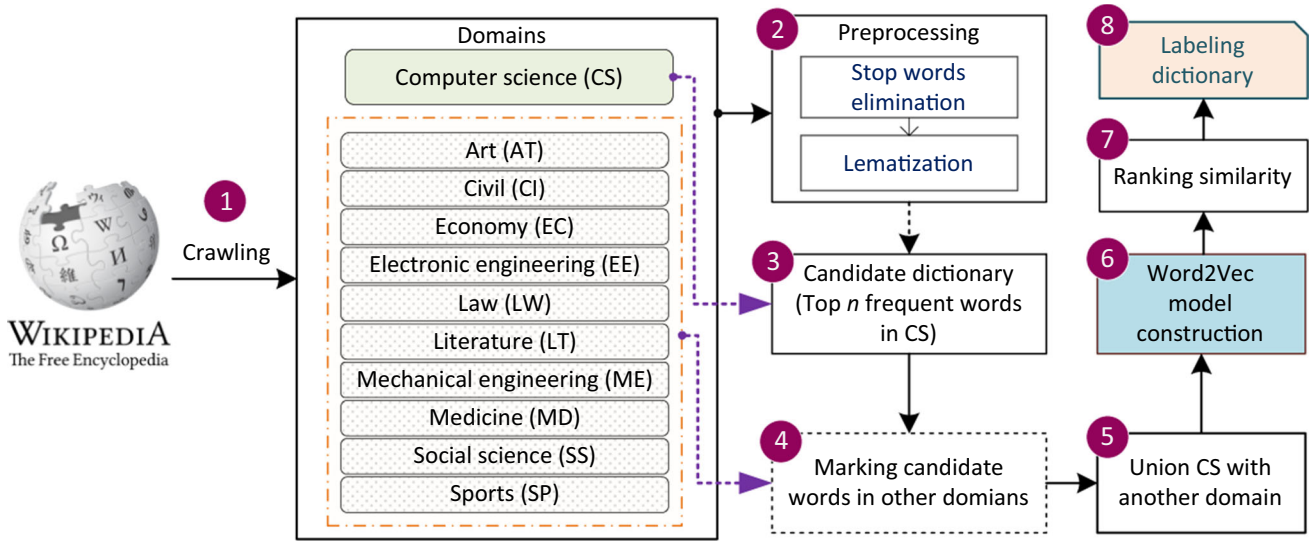


Fig. 1 Dictionary building process

vector similarity is lower than a given threshold. The selected words were then manually analyzed and labeled with the appropriate smell type. As a result, we could collect a dictionary of 700 smelly words and their types of smell. In this way, we could find frequent Computer Science (CS) words that appear in other contexts with different meanings. Figure 1 shows the steps for building a dictionary. The detail of each step is discussed in the following.

First, in addition to computer science, ten Wikipedia categories, for which software systems are often generated, were selected. Each Wikipedia category is structured as a tree in which nodes are subcategories and leaves are pages [25]. For each category, 500 subcategories were selected as different subjects, and then for each subcategory, the content of 20 pages (if any exist) with maximum length was retrieved. As a result, a considerable corpus of about 10K documents for each domain was selected (Step 1 in Fig. 1).

In Step 2, all the stop words were removed from the collected documents. Afterward, all the remaining words in the documents were lemmatized. Consequently, each word was replaced with its lemma [61]. We considered a list of 194 words, including articles, propositions, pronouns, abbreviations, numbers, and symbols, as stop words. Lemmatization is the process of transforming various inflected forms of a word (e.g., cats and cat's) to a single canonical word called the word's lemma (e.g., cat). The preprocessing results are shown in Table 4.

In the third step, we looked for and found the top 1000 most frequent words in the preprocessed documents of CS. Then in Step 4, we looked for these 1000 words in the documents collected for ten other application domains, shown in the first column of Table 4. All the words found in each of the documents were prefixed with an underline character. For instance, the word "return" was replaced with "_return" in all domains apart from the CS.

In Step 5, for each domain, apart from CS, in column one of Table 4, we created a new corpus including the documents of that domain, modified in Step 4, and the documents of CS. In Step 6, two word embedding vectors were trained for each prefixed word and its corresponding word in CS. As a result, we obtained 20 vectors for each of the 1000 frequent words in CS. Indeed, each pair of vectors belonged to a domain combined with CS. For instance, two vectors were created for the word "return" in the CS and MD domains.

In Step 7, the cosine similarities between the vector of each frequent word in CS and the other 10 domains were computed. The computed cosine similarities for each word were averaged. Afterward, we sorted the similarities ascendingly. The cosine similarity between two vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ is given by Eq. (5):

$$\mathrm{Sim}_{\cos}(v_1, v_2) = \frac{\vec{v}_1.\vec{v}_2}{|\vec{v}_1| \times |\vec{v}_2|} \tag{5}$$

In Step 8, starting with the highest-ranked word, we determine its type of smell from within the four predetermined smells of Polysemy words, Subjective language, Ambiguous adverbs/adjectives, and Non-verifiable terms. As a result, discussed in Sect. 4.3, 700 smelly words were selected, and together with their type, they established our dictionary of smelly words.

Algorithm 1 shows the pseudo-code of the dictionary-building process. It receives as input a list of application domain names, `domains`, number of frequent words, `n`, number of subcategories to crawl, `sub_cats`, and number of pages to retrieve for each domain, `pages`. The remaining three inputs are Word2Vec algorithm hyperparameters, including the vector's dimension used to represent each word in the corpus, `dim`, the minimum occurrence of the word to be considered in the learning process, `min`, and the number of neighbors of each word, `window`. The algorithm returns a dictionary of smelly words as its output.

Table 4 Selected domains statistics after preprocessing

Domain	Pages	Words (W)	Vocabulary (V)	V/W
Computer science (CS)	9541	13,575,325	323,500	0.0238
Art (AT)	9681	15,509,613	400,677	0.0258
Civil (CL)	9726	21,687,967	343,951	0.0159
Economy (EC)	9856	23,949,093	409,812	0.0171
Electronic engineering (EE)	9146	11,537,818	415,568	0.0360
Law (LW)	9639	16,558,509	350,407	0.0212
Literature (LT)	9960	17,442,460	551,146	0.0316
Mechanical engineering (ME)	9200	10,962,055	379,264	0.0346
Medicine (MD)	9674	13,260,338	322,465	0.0243
Social science (SS)	9610	15,520,799	357,477	0.0230
Sport (SP)	9653	13,438,100	375,577	0.0279

Algorithm 1 BuildDictionary

```
Input: List Domain, int n, sub_cats, pages, dim, min, window
Output: Dictionary ranked_words
/* 1. Add Computer Science as the first (No. 0) element to the list of domains defined by the user */
domains.insert("computer_science", 0)
/* 2. Create a corpus of cleaned (lemmatized, and without stop words) documents. */
DocumentList docs [ ] ← DocumentList [len (domains)]
foreach domain in domains do
    Document doc ← Wikipedia.crawl(domains, sub_cats, pages)
    doc.remove_stop_words()
    doc.lemmatize()
    docs.append(doc)
end
/* 3. Create a Word2Vec model for the words in each domain */
List frequent_words ← doc[0].get_frequent_words(n)
Word2VecList models [ ] ← Word2VecList [len (domains)]
for i ← 1 to len (docs) −1 do
    foreach word in docs[i].words do
        if word ∈ frequent_words then
            word ← "_" + word
        end
    end
    Word2Vec model ← Word2Vec (Merge (doc[0], doc[i]), dim, min, window)
    models.append(model)
end
/* 4. Create and fill smelly word dictionary */
Dictionary ranked_words = Dictionary ()
foreach word in frequent_words do
    sum ← 0
    foreach model in models do
        vec1 ← model.vectors(word)
        vec2 ← model.vectors("_" + word)
        sim ← CosineSimilarity (vec1, vec2)
        sum ← sum + sim;
    end
    sim_avg ← sum / len (domains)
    ranked_words.update(word, sim_avg)
end
/* 5. Sort the smelly word dictionary and return the result */
ranked_words.values().sort_ascending()
return ranked_words
```

3.4 Cost of multiple sentences

In the proposed requirements testability model, basically, we compute the clarity of a requirement as a factor directly affecting its testability. However, if a requirement has more than one sentence, besides the requirement clarity, the impact of the sentences on each other and the effort required to test the requirement should be considered. We use the α parameter to compute the extra cost/effort for multi-sentence requirement definitions.

We pinpoint four different aspects of the parameter alpha, summarized in Figure 2, as mentioned in Sect. 3.1. These factors include application domain, system criticality, requirements type, and requirements document template. Each of these factors participates equally in determining the value of alpha. Equation (6) is used to compute alpha:

$$\alpha = \frac{1}{4}(diss_{normalized}(D) + CriticalityLevel + RequirementsType + DocumentTemplate) \quad (6)$$

The dissimilarities of various domains with computer science are measured by applying a word embedding technique described in Sect. 3.3. The range of values for the other three parameters is determined by equally partitioning the interval [0, 1) into the number of possible options for each parameter. After selecting the interval, the exact value for each of these parameters is computed by applying a severity policy. Such policies are commonly applied to compute code smells [62]. Two kinds of *hardened* and *softened* policies may be applied. The hardened policy considers the *maximum* allowed value for each parameter option. In contrast, the softened policy uses the *minimum* possible value in the range of the possible values for the option.

The *application domain* parameter measures the distance between the computer science and the requirement's domain. For instance, the civil engineering domain has fewer Polysemy words in common with computer science than electrical engineering. The distance between computer science and the other domains depends on the vector similarity of the words most frequently used in computer science with the same words used in the other domains. We applied Algorithm 1 to compute vector similarities, described in Sect. 3.3. The dissimilarities between the computer science domain, *CS*, and each application domain, *D*, are computed as follows:

$$dissim\ (D) = (1 - avg_{sim}(D,\ CS)) \times \frac{V(D)}{W(D)} \quad (7)$$

In Eq. (7), $dissim\ (D)$ indicates the dissimilarity between the application domain, D, and computer science; $avg_{sim}(D, CS)$ represents the average cosine similarity between the most frequent words in computer science and domain D; $V(D)$ is the vocabulary size of the collected documents for D; $W(D)$ is the number of words in the collected documents for the domain D. The dissimilarity between an application domain, D, and the CS domain can be measured as the average dissimilarity of their common

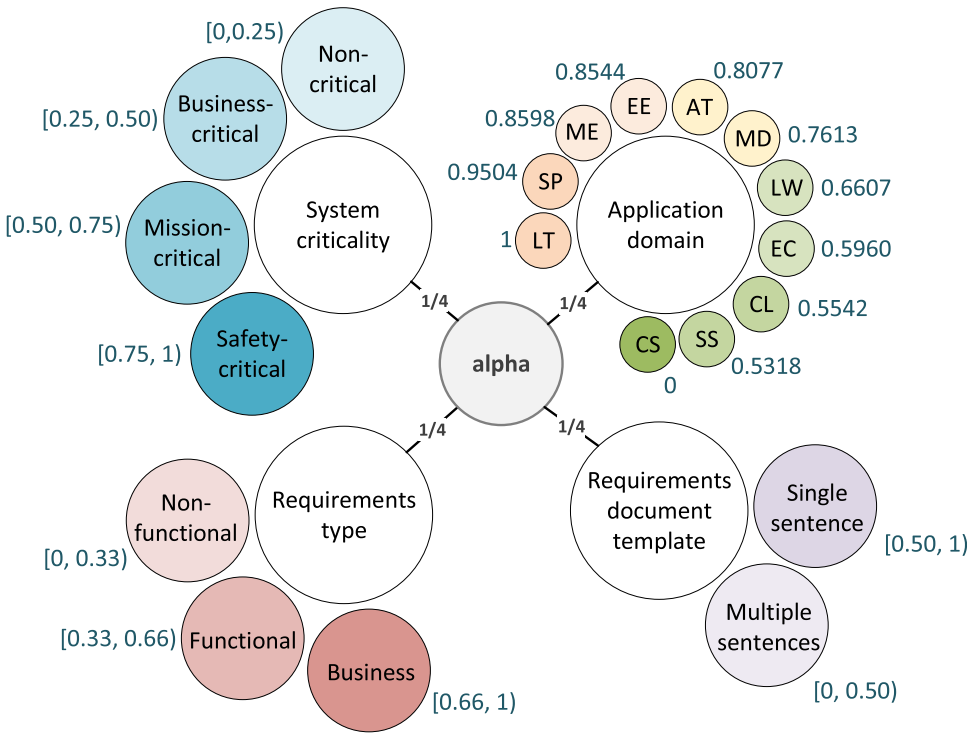


Fig. 2 Guidelines to determine the alpha as the base cost of extra sentences in a requirement

frequent words in the two domains. The average dissimilarity can be defined as '1 − average similarity'.

In Eq. (7), the average dissimilarity is also multiplied by the ratio of vocabulary size, $|V(D)|$, to all words, $|W(D)|$, in a collected corpus for the domain, D, to consider a difficulty factor of words in the domain. Therefore, whenever the average dissimilarity for two application domains with the CS domain is equal, the domain whose vocabulary is larger or whose total of words is smaller is considered more dissimilar to CS. The reason is that a domain with many distinct words (large vocabulary size) or a domain for which fewer documents can be found is more difficult to understand than a domain with a small vocabulary or many documents. To make this idea more concrete, let's consider two hypothetical domains: A and B. Suppose that both domains have the same average dissimilarity with CS, but domain A has a vocabulary size of 1000 words and a total of 10,000 words in its corpus, while domain B has a vocabulary size of 500 words and a total of 20,000 words in its corpus. According to Eq. (7), domain A would be more dissimilar to CS than domain B because it has a larger ratio of vocabulary size to total words: $\frac{|V(A)|}{|W(A)|} = 0.1$, while $\frac{|V(B)|}{|W(B)|} = 0.025$. This means that domain A has more unique or rare words than B that are not common in CS or that domain A has fewer available documents to learn from than domain B. Therefore, domain A is more difficult to understand than domain B from the perspective of CS.

Table 5 shows the $dissim\ (D)$ value computed by Eq. (7) for each domain D. The last row represents the normalized value of $dissim\ (D)$, computed by Eq. (8).

$$dissim_{\ normalized}(D) = \frac{dissim\ (D) - \min\limits_{d \in Domains} dissim\ (d)}{\max\limits_{d \in Domains} dissim\ (d) - \min\limits_{d \in Domains} dissim\ (d)} \quad (8)$$

where $Domains = \{CS,\ SS,\ CL,\ EC,\ LW,\ MD,\ AT,\ EE,\ ME,\ SP,\ LT\}$.

Equation (8) normalizes dissimilarities to the range [0, 1]. According to the last row of Table 5, the dis-similarities of the ten application domains with computer science in the descending order of their magnitude are Literature, Sport, Mechanical Engineering, Electrical Engineering, Art, Medicine, Law, Economy, Civil, Social Science, and Computer Science. It is observed that Literature, Sport, and Mechanical Engineering are more difficult to understand than Social Science, Civil Engineering, and Economic domains. If the requirements belong to more than one domain, then $dissim_{normalized}(D)$ shall be computed by averaging all the relevant domains.

System criticality, shown in Figure 2, is a second parameter used in the computation of alpha. Regarding system criticality, we consider four levels of non-critical, business-critical, mission-critical, and safety-critical systems [63]. As discussed in the manuscript, understanding and validating a single requirement in a safety-critical system, such as a self-driving car, presumably, need more effort than a business-critical system, such as accounting. In this respect, as shown in Figure 2, we assign a criticality value in the intervals of [0, 0.25), [0.25, 0.5), [0.5, 0.75), and finally [0.75, 1], to the non-critical, business-critical, mission-critical, and finally safety-critical systems, respectively.

Requirements types, typically, are non-functional, functional, and business [64, 65]. Testing a business requirement is relatively more difficult than a simple, functional, or non-functional requirement. If a business requirement is not defined appropriately, it does not matter how well the project is delivered; the business still will not be satisfied, and it is not right to blame the stakeholder. We equally divided the interval of [0, 1) into three parts. Each part reveals the range for extra sentences' cost according to the importance of the requirement types, shown in Figure 2. It should be noted that extra sentences make the non-functional requirements, which are inherently subjective, more testable due to better descriptions of these requirements.

Finally, the *requirements document template* parameter determines whether the requirements definition follows a specific predefined format, e.g., "*System shall do X.*" We consider a relatively higher cost for violation of or non-conformance to the chosen template than the cost of having no templates in cases of multiple sentences requirements [28].

3.5 Requirements testability measurement algorithm

Our smell detection and requirements testability measurement algorithm is shown in Algorithm 2. It receives as

Table 5 Application domain dissimilarities, computed for ten different domains and used in our study

Domain (D)	SS	LW	EC	CL	AT	LT	EE	ME	SP	MD
$avg_{sim}(D,\ CS)$	0.6288	0.4997	0.4405	0.4404	0.4974	0.4920	0.6190	0.6011	0.4531	0.4970
$1 - avg_{sim}(D,\ CS)$	0.3712	0.5003	0.5595	0.5596	0.5026	0.5080	0.3810	0.3989	0.5469	0.5030
$dissim\ (D)$	0.0085	0.0106	0.0096	0.0089	0.0130	0.0161	0.0137	0.0138	0.0153	0.0122
$dissim_{normalized}(D)$	0.5318	0.6607	0.5960	0.5542	0.8077	1	0.8544	0.8598	0.9504	0.7613

input a requirement, R, a smelly words dictionary, SD, described in Sect. 3.3; the POS dictionary, PD, containing POS tags and their corresponding smells, shown in Table 3; and the cost, α, of testing each sentence in the requirement, R. It returns the testability value, T, and a list of smelly words, LS, and their smell types observed in the given requirement, R.

At first, the algorithm splits the input requirement, R, into a list of sentences, and the POS tag for each word in each of the sentences is determined using a POSTagger function. Then, for each word in the pos_tags list, the grammatical role of the word is searched in the POS dictionary, PD, to find possible smells with the POS tagging strategy. If the word does not have any smell, it is searched in the smelly words dictionary, SD, to find dictionary-based smells. In both conditions, if a smell is detected, the smelly word and its type of smell will be added to LS. Finally, the clarity score and testability for the requirement, R, are computed according to Eqs. (3 and 4), respectively.

3.6 Design and implementation

Our proposed method for requirements testability consists of four components, illustrated in Figure 3. The Data Model component contains project identities, their requirements, and smells labeled by experts. The Smell Detector component detects the smells in a requirement definition. If a smell is found that does not exist in our smell list, then we can easily add its detection strategy to this component. The Testability Measurement component calculates the testability value for each requirement in the project, and the fourth component, User Interface, provides data visualization and create-read-update-delete (CRUD) operations for the projects, requirements, and smells.

The proposed system is implemented in Python. We use TextBlob [66], a library for typical natural language processing tasks, for requirement preprocessing and POS

Algorithm 2 FindRequirmentSmellsAndTestability

Input: Requirement R, SmellyWordsDictionary SD, POSDictionary PD, float α
Output: Testability T, ListOfSmells LS

```
float clarity, T;
list LS;
sentences ← Split (R.Text)                                   /* List of sentences in the requirement text */
pos_tags ← POSTagger (sentences)     /* List of tokens in the requirement sentences, containing (word, tag) tuples */
foreach token in pos_tags do
    if PD[token.Key] is not null then    /* If the word tag is found in PD, i.e., its grammatical role, is reckoned as a smell */
        LS.add(token.Word, PD[token.Tag])
    else if SD[token.Word] is not null then  /* If word found in SD, i.e.  it is detected by the automated dictionary */
        LS.add(token.Word, SD[token.Word])
    end
end
if len (LS) == 0 then                                        /* If no smell has been found in the requirement */
    clarity ← 1
else
    float smellness ← len (LS) / len (pos_tags)
    clarity ← 1 - power (smellness, 1 / len (set (LS[:, 1])) )
end
T ← clarity / power (1 + α, len (sentences)-1)
return (T, LS)
```

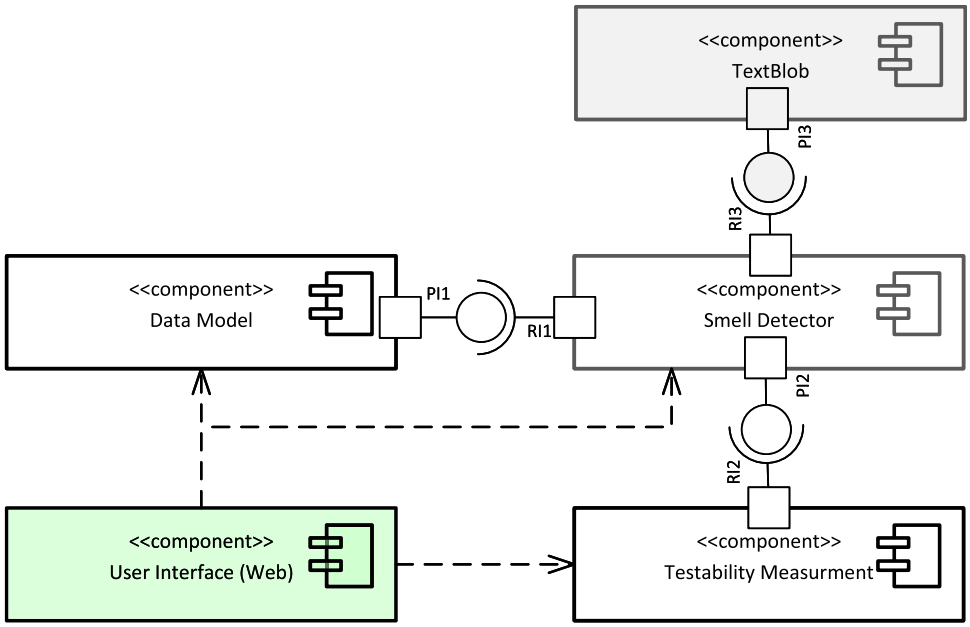


Fig. 3 ARTA component diagram

tagging. Word embedding learning is performed by Gensim [67], which is a Python topic modeling and semantic analysis library. Finally, our web-based user interfaces are implemented by Django [68]. Django is a high-level Python web framework that encourages rapid development. The source code of ARTA is available on the public GitHub repository https://github.com/m-zakeri/ARTA.

3.6.1 ARTA web interface

As described in Sect. 1, our system has been implemented as a standalone web application. Once a requirement is added to a project, the system automatically detects requirement smells and computes different quality metrics, including requirements testability and clarity, shown in Figure 4a. ARTA also highlights smells detected in a requirement definition and notifies the requirement engineer to refactor the requirement, considering its smell type. In addition to automatically detecting the smells, our Automatic Requirements Testability Analyzer, ARTA, allows specifying the requirements smells manually.

The second important module in ARTA is the Manual Labeling module. It helps researchers specify smells found in a requirement definition and provides a review flag that the reviewers can use to ensure the correctness of the smell type assigned to the requirement. Figure 4b shows the ARTA manual smell labeling module. For the time being, since a large dataset of textual requirements with known smells is not available, machine learning-based techniques for smell detection are not applicable. The labeling module provided by ARTA accelerates the process of creating the desired dataset to develop and evaluate smell detection algorithms.

4 Evaluation

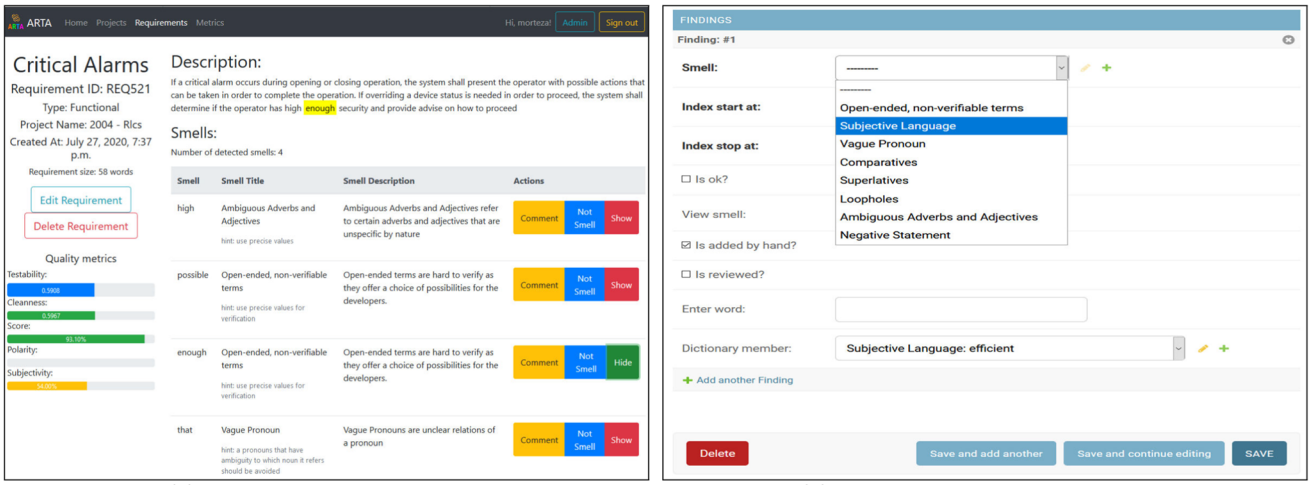
This section reports the setup and the result of our proposed methodology evaluation.

4.1 Experimental setup

4.1.1 Research questions

We design our experiments to answer the following research questions about the requirement smells and requirements testability:

- **RQ_1 (Requirement smells prevalence)** What is the prevalence of requirement smells in real-world software projects? Which smells are more frequent than others? Is there a relation between requirement smells and the size and number of requirements in projects?
- **RQ_2 (Smelly words dictionary)** Which frequently used words in computer science are context-sensitive and thereby cause ambiguity?
- **RQ_3 (Requirement smells detection performance)** How effective is the automated dictionary mechanism in spotting smells in software requirements? Which smells are more difficult to detect?
- **RQ_4 (Requirements testability measurement performance)** How effective is the proposed method in measuring the testability of the requirements? Which



(a) Requirement analyzer module (b) Requirement smells labeling module

Fig. 4 ARTA web interface

smells affect requirements testability more than the others?

4.1.2 Dataset

We evaluated our proposed methodology on a set of real-world software requirements obtained from the PURE dataset [69]. It consists of requirements documentation of 79 industrial and academic projects. In this repository, the description of all the included projects and their requirements is available in PDF, DOC, HTML, and XML formats. The collector of the PURE dataset has already extracted the requirement of 18 projects into XML files, and we selected these projects as the initial set of our case studies.

Manual determination of the requirements smell is time-consuming because each requirement should be studied and analyzed carefully several times by at least two experts to achieve reliable results. For the scope of this paper, we chose six real-life projects with different domains and a different number of requirements. We manually extracted the requirements definitions of these projects into a Microsoft Excel file and labeled each requirement with its smell types. Table 6 shows the projects and the number of requirements for each project. In total, 985 requirements were obtained from six projects. It is worth noting that our primary dataset consists of 5000 requirements from 15 projects. Each of the requirements was labeled by four groups of IUST software engineering students. However, based on the consensus, we chose a subset of requirements with the most reliable labels after the second and third revisions.

Table 6 Selected projects and their requirements

Project	Topic	Scope/domain	Number of requirements	Number of words
EIRENE	Digital radio standard for railway	Industry/EE	564	14,707
ERTMS/ETCS	Train control system	Industry/EE, ME	199	3566
CCTNS	Crime investigation management system	Industry/LW	115	3635
Gamma-J	Web store	Academic/EC, CS	51	683
KeePass	Password management system	Industry/CS	32	456
Peering	Internetworking of content delivery network through Peering	Academic/CS	24	195
Sum			985	23,242

4.1.3 Dataset creation procedure

We employed a cross-labeling process in which each requirement was labeled by one student and then reviewed by two other students. Four groups of IUST postgraduate (2nd-semester M.Sc.) students in the field of software engineering were chosen during the Advanced Software Engineering (ASE) course presented by the corresponding author of this paper. At least one of the students in each group had relevant industrial experiences. All the postgraduate students who participated in the labeling process were asked to carefully study the ISO/IEC/IEEE 29148:2011 [70], titled "*Systems and software engineering — Life cycle processes — Requirements engineering*" to get insights into the standard definitions of requirement smells and dictionaries. Students were also asked to look at similar standards [71] and study recent papers in the field of requirements engineering [16, 27, 29]. Students were then trained for two 1.5-hour sessions. The training was about the concept of requirement smells and smell detection mechanisms and the process of labeling.

The selected projects were randomly assigned to each group of students, and we asked each group to study the project documents to get ideas about the scope and domain of the projects. Figure 5 shows the cross-labeling process used in our dataset creation phase. The requirements of all projects in each group were divided into three parts. Each part was labeled by a postgraduate in software engineering. Each student also reviewed the result of labeling by two other teammates. In cases of disagreement, the results were resolved by discussion and interlocution. Using this cross-labeling mechanism, we could ensure that smells were determined carefully, and that our results were as reliable as possible. Finally, all requirements were reviewed by a Ph.D. student in software engineering (first author of the paper). In cases where it is complicated to decide if the requirement has a specific smell, we simply removed the requirement from our final dataset due to the lack of expert knowledge or different labels.

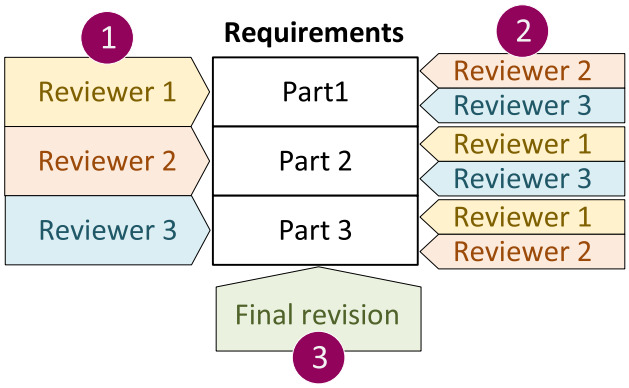


Fig. 5 Cross-labeling process

The overall labeling process took about three months. Unfortunately, due to different students' skills, different nature of projects, and lack of time, we could not confirm the label of all requirements in all the projects. We discarded projects in which the reviewers could not evaluate more than 10% of their requirements. In addition, one group could not complete the labeling process.

Finally, we ended up with 985 requirements that both the experts in software industries and our teams agreed on their type of smells. We observed an average Fleiss' Kappa of 0.9 for the six confirmed projects, which shows almost perfect agreement and confirms our dataset set is based on a reliable consensus. This is the first public requirement dataset labeled by the known smells on its own.

As described in Sect. 4.1.2, all the requirement definitions were labeled and stored in an Excel file. This Excel file contains 11 columns, including the requirement text, project file, and one column for each smell. Figure 6 shows the Excel datasheet schema and the five sample requirements. The smelly word for each requirement is written in the cell (i, j), where i is the row requirement, and j is the column of smell. Therefore, our dataset not only points out the smell but also locates the position of the smell within the text of the requirements. If a requirement has more than one word with the same smell type, e.g., two subjective terms, these words are separated by an asterisk (*) symbol in the cell (i, j). Empty cells are marked by a '–' character. This template facilitates quick and easy retrieval of requirements to evaluate requirement smells and testability.

4.1.4 Analysis procedure

Algorithm 1 is applied to create our smelly word dictionary using application domains listed in Table 4. We set `n` = 1000, `sub_cats` = 500, `pages` = 20, `dim` = 50, `min` = 5, `window` = 10, for our experiments. Afterward, we ran Algorithm 2 to find smells in reach requirement and compute testability. The results of both algorithms are saved in Excel files. Our experiments are performed by analyzing the data of these Excel files.

We used the Google Colaboratory platform [72] to crawl Wikipedia and build Word2Vec models, which is a free cloud-based computational resource. All the experiments were performed on a Windows 10 (x64) machine with a 2.6 GHz Intel® Core™ i7 6700HQ CPU and 16 GB of RAM.

Neither the implementation nor the dataset of Smella [16] is publicly available. Therefore, we reimplemented Smella [16]. We did our best to follow all instructions mentioned by Femmer et al. [16] to make a free implementation of Smella, to be used for evaluation of the

Requirement_text	File	Subjective_lang.	Ambiguous_adv._adj.	Nonverifiable_term	Superlative	Comparative	Negative	Vague_pron.	Uncertain_verb	Polysemy
The System user interface must be suitable for users with special needs; that is, compatible with specialist software that may be used and with appropriate interface guidelines.	0000 - cctns.pdf	suitable	-	-	-	-	-	that	may	-
The System must provide End User and Administrator functions which are easy to use and intuitive throughout.	0000 - cctns.pdf	easy	-	-	-	-	-	-	-	-
The System must allow persistent defaults for data entry where desirable.	0000 - cctns.xml	desirable*defaults	desirable	-	-	-	-	-	must	-
Visual and acoustic warnings to the driver about possible intervention from ETCS shall be given to enable the driver to react and avoid intervention.	2007-ertms.xml	-	-	possible	-	-	-	-	-	-
The driver shall have the possibility to select the language, this does not concern non pre-defined texts sent from the trackside.	2007-ertms.xml	concern*static*dynamic	-	-	-	-	doesnot	this	-	-
A train shall be supervised to its static and dynamic train speed profiles.	2007-ertms.xml	-	-	-	-	-	-	its	-	static*dynamic

Fig. 6 Requirements dataset schema with samples

requirements defined in English. The original Smella tool has been evaluated on four projects for which the requirements are written in the German language. The first three cases contain requirements produced in different industrial contexts, and the fourth one is a student project.

We followed the very same process described by Femmer et al. in Section 4.2 of their article [16] to reimplement their tool. They have used hand-made dictionaries based on the ISO/IEC/IEEE 29148:2011 standard [70] to detect requirements smells. The ISO/IEC/IEEE 29148:2011 standard defines the construct of a good requirement and provides attributes and characteristics of requirements. This standard also provides guidelines to characterize a requirement definition and offer some examples for each language criterion. Requirements language criteria are described in Section 5.2.7 of the ISO/IEC/IEEE 29148:2011 standard document [70]. We used the examples provided by the standard and searched the WordNet dictionary for finding their synsets to manually build dictionaries similar to the ones in reference [16]. We compared the performance of their hand-made dictionary with our automatically built dictionary on the prepared dataset.

4.1.5 Evaluation metrics

To measure the performance of the proposed method in finding requirement smells, we use F1 score, Precision, and Recall given by Eqs. (9) to (11). These are standard and widely used metrics for evaluating the performance of binary classification models, such as our neural word embedding technique for detecting smelly or context-sensitive words. Accuracy measures the overall correctness of the model, Precision measures the proportion of true positives among the predicted positives, Recall measures the proportion of true positives among the actual positives, and F1 score measures the harmonic mean of Precision and Recall. These metrics allow us to assess the effectiveness and efficiency of our smell detection technique and compare it with the state-of-the-art tool, Smella [16]. As mentioned, the F1 score is defined as the harmonic mean of the Precision and Recall:

$$F_1 = 2 \times \frac{PPV \times TPR}{PPV + TPR} \tag{9}$$

where PPV is the positive predictive rate known as Precision, determining the fraction of relevant instances among all retrieved instances. TPR is the true positive rate, known as Recall or sensitivity. TPR determines the fraction of all relevant retrieved instances. Precision and Recall are given by the following equations:

$$PPV = \frac{TP}{TP + FP} \tag{10}$$

$$TPR = \frac{TP}{TP + FN} \tag{11}$$

In the above relations, TP indicates the number of detected smelly words; FP indicates the number of words that do not have any smells but are selected incorrectly as smelly. Finally, FN is the number of smelly words that are not detected as smelly.

To compute requirements testability based on the number and type of the requirements smells, we use Eq. (4), which computes testability in the range (0, 1]. Errors in our automatic evaluation of testability are calculated as the difference between the testability values obtained automatically and values computed manually (ground truth values). To evaluate errors in the automatic computation of requirements testability, we report standard error metrics, including mean absolute error (MAE), mean squared error (MSE), root means square error (RMSE), mean squared logarithmic error (MSLnE), and median absolute error (MdAE). Mean absolute error measures the average magnitude of the errors in the predictions, and mean square error measures the average squared magnitude

of the errors in the predictions. These metrics allow us to evaluate the accuracy and robustness of our testability prediction model and demonstrate its potential for practical use. Error metrics are computed by Eqs. (12) to (15). In these equations, $\hat{y}_i$ indicates the value computed automatically for the ith requirement sample; y_i is the true value for $\hat{y}_i$, computed manually; $y = <y_1, \ldots, y_n>$, and n is the number of the samples.

$$\mathrm{MAE}(y, \hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1} |y_i - \widehat{y_i}| \tag{12}$$

$$MSE((y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n-1} (y_i - \hat{y}_i)^2 \tag{13}$$

$$\mathrm{MSLnE}(y, \hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1} (\log_e (1 + y_i) - \log_e (1 + \hat{y}_i))^2 \tag{14}$$

$$\mathrm{MdAE}(y, \hat{y}) = \mathrm{median}(|y_1 - \widehat{y}_1|, \ldots, |y_n - \widehat{y}_n|) \tag{15}$$

4.2 Requirement smell prevalence

To answer RQ_1, we measure the frequency of smells in our labeled dataset. Figure 7 shows the box plot of existing smells in each project's requirements. Datapoints denote the number of smells in each requirement. It is observed that CCTNS has the most smells, while the number of the requirement of this project is significantly lower than the EIRENE project. Therefore, it is observed that the frequency of smells does not depend on the number of requirements in the project. Figure 8 shows the frequency of the requirements smells types. For each project, the mean of smells' frequency along with the confidence interval of 95% for mean value, μ, around bootstrapped values [73], is shown by error bars ($\updownarrow$) in Figure 8.

One can observe that Polysemy and Subjective language are the two most prevalent smells in requirements. Non-verifiable terms are also frequent; nevertheless, other smells occur occasionally. On average, the fraction of smelly words to the total requirement's words in our dataset is about 4.93%, which indicates that the frequency of requirements smells are *similar* to the code smells [74]. However, the percentage of requirements with at least one smell is 72.39%, denoting that most of the textual requirements suffer from smells and need to modify or refactor before use. Therefore, automated mechanisms to find requirement smells are necessary and beneficial to the SDLC.

Figure 9 shows the number of smells in each requirement against the requirement's length, *i.e.*, the number of words. The continuous line illustrates the regression line between the two variables, and the bulleted numbers denote the requirements clarity degree. It can be observed that as the number of words in a requirement increases, the number of smells increases. The Spearman rank-order

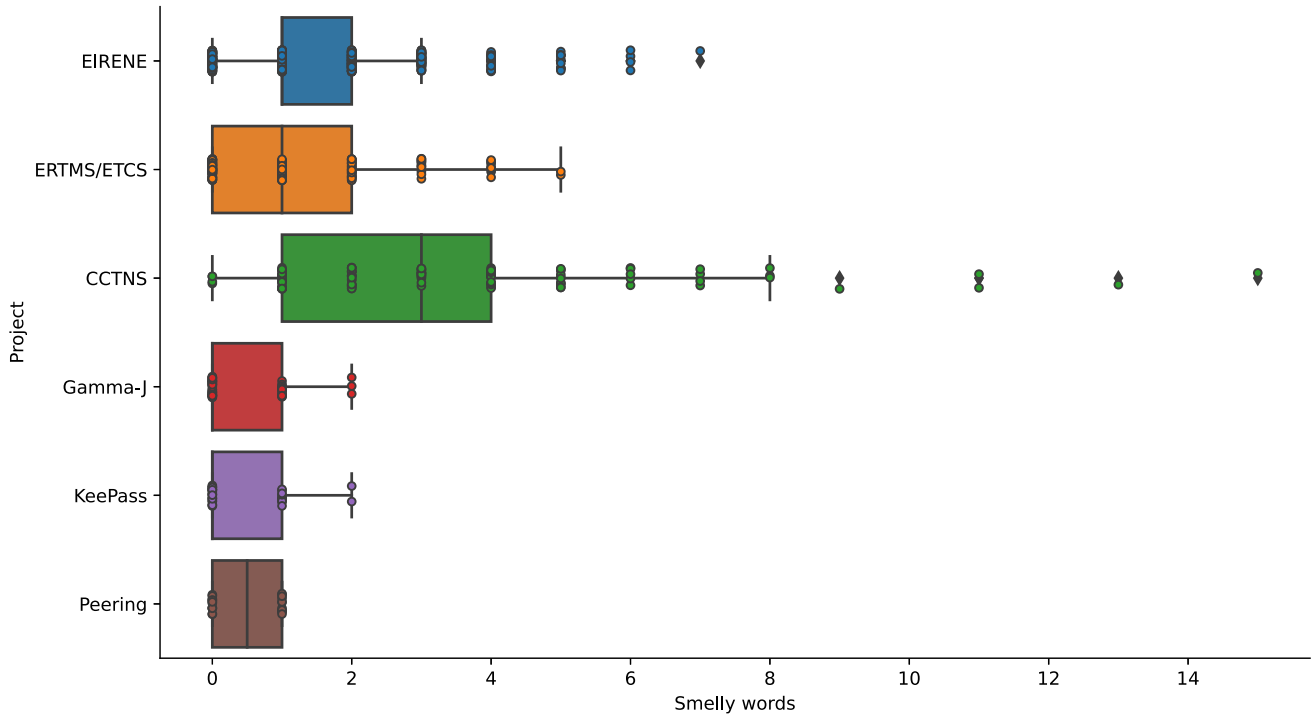


Fig. 7 Frequency of the requirement smells

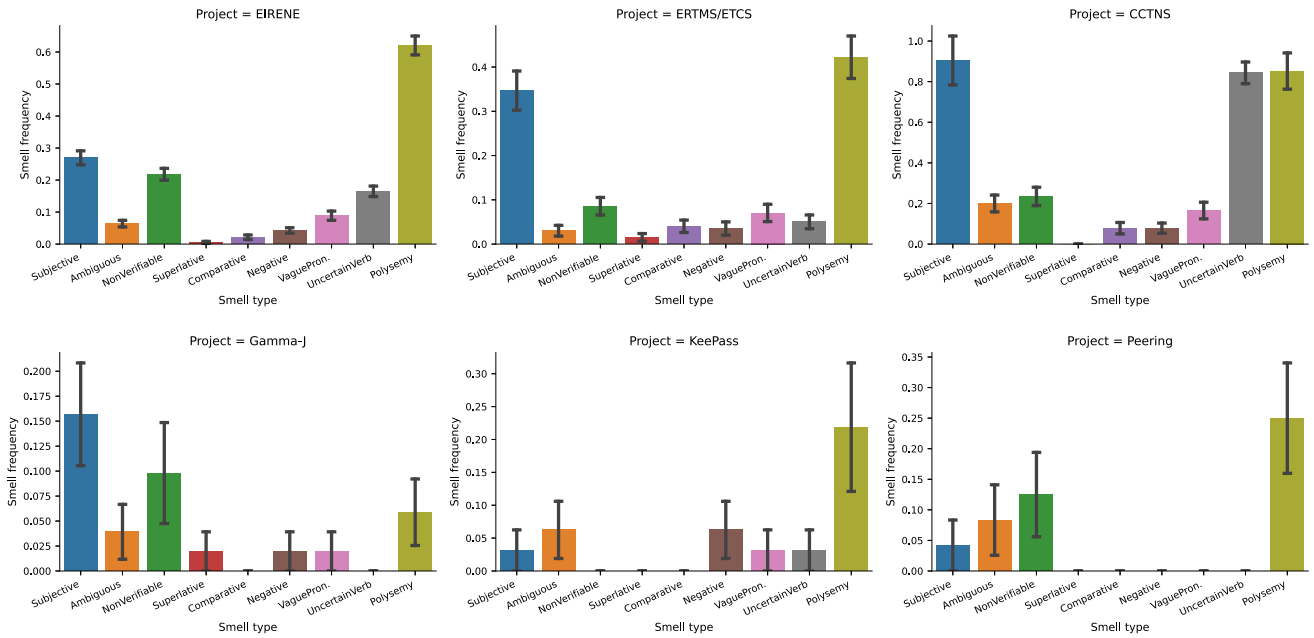


Fig. 8 Variation in smells within the requirements of each project

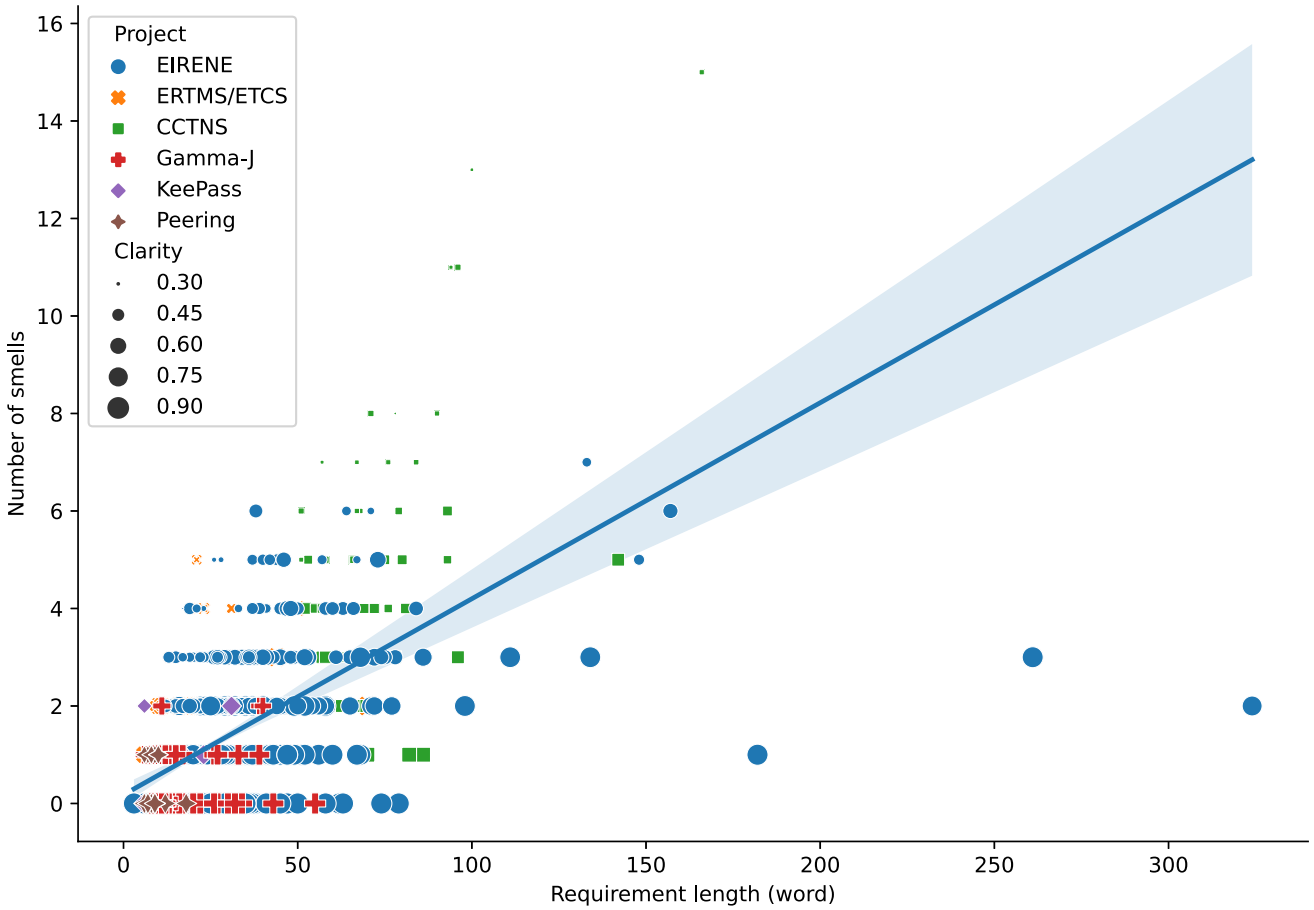


Fig. 9 Number of smells against the requirement size along with a linear regression line

correlation coefficient between the number of smells and the length of the requirements for our dataset is 0.5253 with a p-value of 5.9998×10^{-71}, which means that they are positively correlated. As we expected, the requirements with smaller lengths are cleaner and more testable than requirements with many words.

A relatively similar result has been proposed by Femmer et al. [16], where a Spearman correlation of 0.9 between the number of smells and the length of the requirements has been reported. Their result shows that the number of detected smells strongly correlates with the size of the requirement artifact, while for our study, the correlation is not very strong. One primary reason is the language used to express the requirements. It seems that English is more suitable to express requirements than German. Another reason is that Femmer et al. [16] use their findings to compute the correlation coefficient, which is prone to be a wrong result due to the low Precision of automatic smell detection for most types of smells. Hence, we believe that our results are more realistic than the results reported by Femmer et al. [16].

Answer to RQ_1: *Only 4.93% of words are recognized as smelly words in the software requirements, while 72.39% of requirements contain at least one kind of smell. Polysemy, Subjective language, and Non-verifiable terms are the most prevalent requirement smells. The number of smells increases as the requirement length grows, with a correlation coefficient of* 0.5253.

4.3 Smelly words dictionary analysis

The descriptive analysis of the requirements dataset in the previous section shows that the most prevalent requirement smells, *i.e.*, Polysemy, Subjective language, and Non-verifiable terms are smells whose current detection mechanism is based on a dictionary. Therefore, it is essential to build an efficient and comprehensive dictionary to harness these smells. The first 30 words and the last five words of the automated dictionary with corresponding similarity values and smells are shown in Table 7. The dictionary contains the top 1000 frequent words in computer science sorted in descending order by their similarity in 10 domains. The "mean" similarity column shows the average cosine similarity between each word in computer science and other domains.

Words with different meanings in different domains were less similar. For example, the word "call" has the least similarity since it has various meanings, such as "telephone call," "voice," and "entitle." At the same time, in computer science, the word "call" usually means "invoking a function," "method," or "subroutine" of the program, which is different from the other domains. In such cases, according to the context, the requirement engineer can replace the smelly word with non-smelly and clear ones. For instance, the word "call" may be replaced with "phone call" or "function call" in a requirement definition. In the last rows, the words that have a single and exact meaning are observed. These words are mostly specific to the computer science domain. For example, "object oriented," "c++," and "boolean" are dedicated words in computer science, even when used in other domains. Therefore, they showed high similarity for all the predefined domains.

After ranking each word based on its similarity measure, a thorough investigation was made to label each word with its exact type of smell. Sorting the list of the ranked words, on the similarity measure, we noticed that the words with a mean cosine similarity of greater than or equal to 0.5943 do not expose any smell. Specifically, the word "association" with a mean cosine similarity of 0.5943 is the last smelly word in our sorted list, shown in Table 7.

We observed that words with a similarity above 0.5943 are specific to the computer science domain or represent only one concept. Therefore, these words (about 250 words of 1000 preliminary words) do not indicate any smell. It is worth noting that there are words with a low similarity value that have no smells. For example, the word "add" is a verb identified as a clean word by our experts. Nevertheless, the proposed mechanism for selecting candidate words and computing words' similarity with the Word2Vec algorithm sounds efficient and applicable to any language.

We performed a sensitivity analysis [75] to show that the threshold identification is robust and domain-independent. We repeated the calculation of mean similarity for the top 1000 frequent words in computer science ten times. For each experiment, one application domain was removed, and the mean similarity was calculated to measure the impact of the removed domain on the obtained threshold. Figure 10 shows a scatter plot of the cumulative number of smelly words at each similarity point. The total number of smelly words reaches a constant value as the similarity exceeds 0.5943 in each experiment. Indeed, eliminating one application domain does not change the number of smelly words in our experiments.

Table 8, for each experiment, shows the changes in the mean similarity of the last smelly word in the list of the words, sorted on the order of their similarity with the 1000 most frequent computer science words. We observe small changes (less than 0.0057 or 5.7% on average) in the similarity threshold compared to the experiment that takes all domains into account. Therefore, it is concluded that the obtained threshold is not sensitive to any specific application domain. It is worth mentioning that frequent words in the computer whose similarity measure is greater than the

Table 7 Smelly words dictionary with labels and similarity values

Frequent CS word	SS	Lw	Ec	Ci	At	Li	EE	ME	Sp	Me	Mean	Smell
Call	0.2556	0.0993	− 0.0076	0.1854	0.0513	0.1037	0.1099	0.2191	0.0216	0.0393	**0.1078**	Polysemy
Part	0.2612	0.1506	0.0837	0.1368	0.0708	0.2082	0.1674	0.1280	0.1409	0.1551	**0.1503**	Subjective language
Present	0.2415	0.1947	0.2045	0.1286	0.1191	0.1147	0.1536	0.1917	0.0799	0.2485	**0.1677**	Polysemy
Another	0.2770	0.1948	0.1912	0.0790	0.1932	0.2597	0.1489	0.1612	0.0397	0.1397	**0.1684**	Non-verifiable terms
Include	0.1852	0.1677	0.0598	0.1489	0.1586	0.2206	0.2656	0.2754	0.1140	0.1091	**0.1705**	Non-verifiable terms
Along	0.1732	0.1975	0.1285	0.2035	0.1185	0.2308	0.2435	0.2244	0.0241	0.1743	**0.1718**	Ambiguous Adv./ Adj.
Take	0.3570	0.1006	0.1204	0.1772	0.2616	0.2125	0.1474	0.2900	0.0813	0.1972	**0.1945**	Polysemy
Already	0.2950	0.1905	0.1267	0.1750	0.2094	0.1838	0.2782	0.2716	0.1543	0.1264	**0.2011**	Ambiguous Adv./ Adj.
Entire	0.4104	0.2050	0.1224	0.2752	0.2008	0.2056	0.2246	0.2423	0.0481	0.0913	**0.2026**	Subjective language
Full	0.2906	0.0909	0.0786	0.2157	0.1810	0.2097	0.3246	0.3175	0.1624	0.1849	**0.2056**	Non-verifiable terms
Add	0.2838	0.1889	0.0781	0.0831	0.2910	0.1506	0.3537	0.2867	0.1702	0.1844	**0.2070**	–
Originally	0.2812	0.1603	0.0463	0.2590	0.1833	0.1929	0.3361	0.3191	0.1930	0.2083	**0.2180**	Ambiguous Adv./ Adj.
Previous	0.2815	0.2184	0.2743	0.2052	0.2230	0.2479	0.2319	0.2607	0.1370	0.2157	**0.2296**	Ambiguous Adv./ Adj.
Main	0.3090	0.1158	0.1475	0.2320	0.2468	0.2231	0.2672	0.3294	0.1871	0.2586	**0.2317**	Polysemy
Similar	0.3339	0.1896	0.1181	0.2023	0.1978	0.2594	0.3135	0.3294	0.1580	0.2424	**0.2344**	Subjective language
Aso	0.3155	0.1537	0.2034	0.2222	0.2968	0.2697	0.2685	0.2713	0.1552	0.2152	**0.2372**	–
Made	0.3313	0.2491	0.1927	0.1541	0.2650	0.3075	0.1972	0.2983	0.1816	0.2025	**0.2379**	Polysemy
Third	0.2974	0.2196	0.1359	0.2283	0.2091	0.2262	0.2541	0.2974	0.2024	0.3638	**0.2434**	–
Whose	0.3652	0.2408	0.1707	0.2232	0.2675	0.2951	0.2259	0.3819	0.2005	0.1109	**0.2482**	Ambiguous Adv./ Adj.
Start	0.2834	0.2368	0.1147	0.2395	0.2582	0.3080	0.3353	0.3685	0.1701	0.1957	**0.2510**	Polysemy
Come	0.4019	0.2010	0.1851	0.1940	0.2852	0.2377	0.2728	0.2858	0.2359	0.2168	**0.2516**	–
Section	0.3110	0.3493	0.1927	0.2092	0.0943	0.2678	0.2992	0.3845	0.1160	0.2971	**0.2521**	Polysemy
Base	0.4224	0.2937	0.2077	0.2849	0.1788	0.2559	0.3755	0.3005	0.0944	0.1362	**0.2550**	Polysemy
Together	0.3719	0.2903	0.2291	0.1992	0.2872	0.1722	0.2503	0.3338	0.2420	0.1949	**0.2571**	Subjective language
Long	0.2112	0.1932	0.1269	0.2155	0.3180	0.4016	0.2853	0.3117	0.3016	0.2152	**0.2580**	Non-verifiable terms
Initial	0.3660	0.3135	0.2138	0.1846	0.1986	0.2404	0.3449	0.3607	0.1596	0.2050	**0.2587**	Polysemy
Following	0.2980	0.2050	0.2668	0.2162	0.2681	0.3331	0.2942	0.3092	0.2107	0.2070	**0.2608**	Non-verifiable terms
Known	0.3065	0.2525	0.2288	0.2375	0.2039	0.3220	0.2796	0.3481	0.1955	0.2803	**0.2655**	Subjective language
Left	0.3595	0.1969	0.1169	0.1753	0.2929	0.2991	0.3343	0.3560	0.2588	0.2750	**0.2665**	Polysemy
…	…	…	…	…	…	…	…	…	…	…	…	…
Association	0.6754	0.5938	0.5140	0.5576	0.4982	0.5582	0.6628	0.7138	0.5486	0.6205	**0.5943**	Polysemy
…	…	…	…	…	…	…	…	…	…	…	…	…
Mathematician	0.8688	0.7527	0.7337	0.7666	0.7550	0.7832	0.8694	0.8943	0.8276	0.8265	**0.8078**	–
Campus	0.8890	0.8208	0.8110	0.5987	0.8450	0.8634	0.8481	0.8758	0.7585	0.8185	**0.8129**	–
Psychology	0.9075	0.8486	0.8005	0.7675	0.7624	0.7466	0.8784	0.8613	0.7558	0.8884	**0.8217**	–
Transistor	0.9181	0.8378	0.8487	0.7249	0.8515	0.8016	0.9454	0.9250	0.7137	0.8169	**0.8384**	–
Bachelor	0.9451	0.8951	0.7462	0.8573	0.7746	0.7050	0.9391	0.9319	0.7910	0.8417	**0.8427**	–
Nobel	0.9466	0.8532	0.8438	0.7646	0.7467	0.7315	0.9549	0.9516	0.7962	0.8593	**0.8448**	–
Boolean	0.7968	0.7875	1.0000	1.0000	0.7515	0.5891	0.8090	0.7170	1.0000	1.0000	**0.8451**	–

Table 7 (continued)

Frequent CS word	SS	Lw	Ec	Ci	At	Li	EE	ME	Sp	Me	Mean	Smell
Undergraduate	0.9233	0.8025	0.8521	0.7699	0.7883	0.8693	0.8950	0.8924	0.8442	0.8787	**0.8516**	–
c++	0.8431	0.7483	1.0000	1.0000	0.6741	0.7058	0.8694	0.7278	1.0000	1.0000	**0.8568**	–
Object oriented	0.8209	0.6667	1.0000	1.0000	0.7013	1.0000	0.8233	0.7220	1.0000	1.0000	**0.8734**	–
Mean	**0.6288**	**0.4997**	**0.4404**	**0.4404**	**0.4974**	**0.4920**	**0.6190**	**0.6011**	**0.4531**	**0.4970**	0.5169	–

The average similarity values corresponding to each word and each domain have been bolded

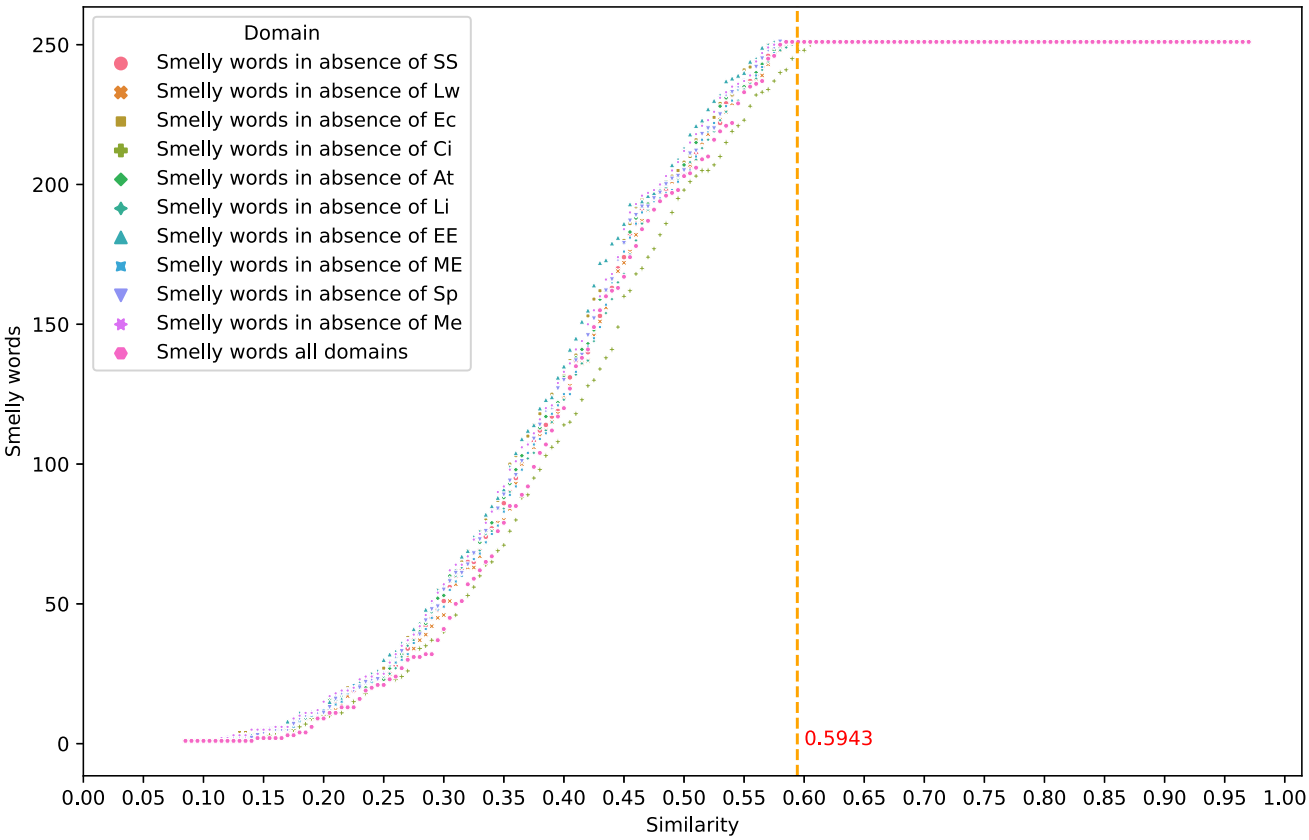


Fig. 10 Number of smelly words for different similarity values

observed threshold are not smelly, while those with a similarity measure less than the threshold are mostly smelly.

> **Answer to RQ_2**: *We observed that words, such as "call," "part," and "present," with an average cosine similarity of less than* 0.5943 *in all domains, are most likely to have different meanings in different domains of use. On the other hand, words such as "object oriented," "c++," and "cpu" with an average cosine similarity close to one, are most likely clear and unambiguous, without any smells.*

4.4 Requirement smell detection analysis

To evaluate the proposed method's effectiveness in detecting requirement smells, we ran Algorithm 2 on our dataset and compared the results with manually labeled data as the ground truth. As discussed earlier, for Uncertain Verbs, we use a small hand-made dictionary consisting of common English modals: "may," "might," " can," " could," and" should." We made an exception for the word "shall" as it is often used to express functional requirements [76]. For other smells, identified by a dictionary mechanism, we used the dictionary of smelly words that

Table 8 Variations in the similarity of the last smelly words in the automatically created dictionary

Domains	The similarity of the last smelly word	Absolute change in similarity value
All domains	0.5943	0
All domains except SS	0.5943	0
All domains except Lw	0.5943	0
All domains except Ec	0.6032	0.0089
All domains except Ci	0.5984	0.0041
All domains except At	0.6050	0.0107
All domains except Li	0.5983	0.0040
All domains except EE	0.5867	0.0076
All domains except ME	0.5810	0.0133
All domains except Sp	0.5994	0.0051
All domains except Me	0.5913	0.0030
Average	–	0.0057

Table 9 Smell detection performance

Requirement smell	Precision	Recall	F1
Subjective language	0.3421	0.4885	0.4024
Ambiguous adv./adj.	0.4898	0.3380	0.4000
Non-verifiable term	0.2598	0.2994	0.2782
Superlative	0.3500	0.8750	0.5000
Comparative	0.4746	0.8750	0.6154
Negative	0.2897	0.9333	0.4421
Vague pron.	0.1363	0.6966	0.2279
Uncertain-verb	0.7766	0.9815	0.8671
Polysemy	0.6688	0.8322	0.7416
Average	0.4209	0.7022	0.4972

was prepared automatically. We also implemented, Smella [16] to compare its performance with our smell detection tool.

Table 9 shows the Precision, Recall, and F1 for detecting different types of smells for all the requirements in our dataset. It is observed that for the Polysemy smell, as the most prevalent requirement smell, both the Precision and Recall are quite remarkable and demonstrate the applicability of our smell detection method in practice. We demonstrate the applicability of our approach on several examples of real-world software requirements in Sect. 4.5.4.

The gap between Precision and Recall is very high for most smell types. Indeed, the Recall is higher than the Precision, which means that our algorithm is good enough to identify all smelly words, but it also selects words that do not contain any smell and hence leads to a high *false-positive rate*. The primary reason is that the words are only detected based on their grammatical role or their existence in a predefined dictionary, which may be misleading for a particular context. It can be realized that reducing the false positive rate in requirement smell detection tools needs more advanced techniques, considering the overall context of the requirement statement.

The smells with lower F1 scores are, apparently, more difficult to detect. For example, the F1 score of vague pronouns is very low since determining the references for all the pronouns in a sentence is a complicated and error-prone NLP task. However, as shown in Figure 11, our

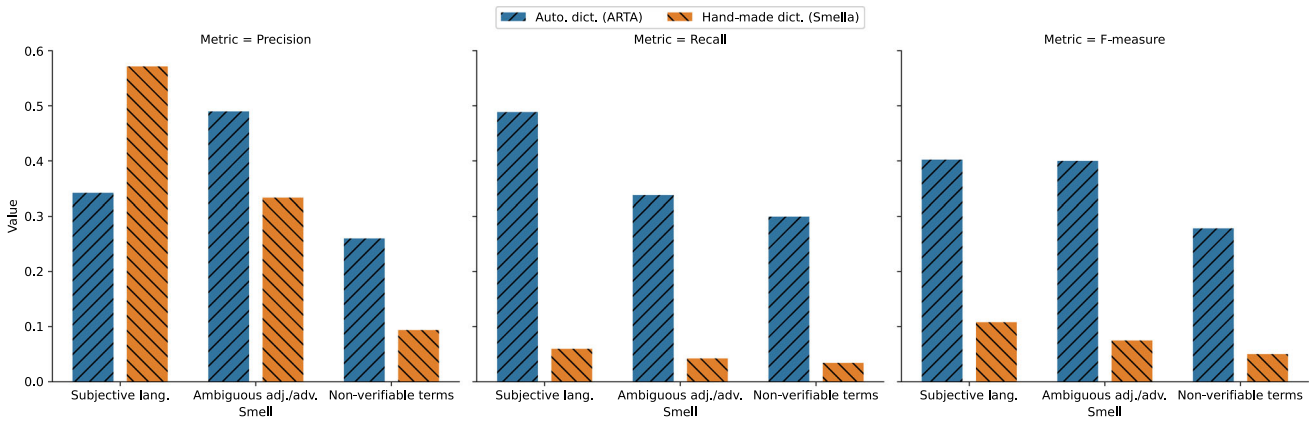


Fig. 11 Performance of ARTA and Smella [16] in detecting dictionary-based requirement smells

approach outperforms the state-of-the-art requirements smell detection tool, *i.e.*, Smella [16]. The figure shows the Precision, Recall, and F1 score of ARTA and Smella for three types of requirement smells detected with the dictionary. Our approaches have a better performance than Smella in all three smells. The Smella tool could not recognize Polysemy smells. Therefore, we have not illustrated the performance of this smell in Figure 11. ARTA improves the Precision, Recall, and F1 score of dictionary-based smell detection, respectively, by an average of 0.0311, 0.3299, 0.2824.

Answer to RQ_3: *On average automated dictionary improves the F-measure of requirement smell detection by 28.24% on our dataset. Polysemy and Uncertain verbs can be detected with high Precision and Recall. On the other hand, Vague pronouns and Non-verifiable terms are more difficult to detect even when using an automated dictionary mechanism.*

4.5 Requirements testability analysis

In our last experiment, we compute the testability value of each requirement in our dataset based on Eq. (4). Three computations are performed for each requirement: the first computation is based on the ground-truth dataset, the second computation is based on our approach obtained by ARTA web application, and the third computation is based on smells finding by Smella [16]. We reckoned the smells detected by each approach separately and calculated requirements clarity and testability values. Then, error metrics are computed by comparing the estimated value with corresponding ground-truth values.

4.5.1 Computing and comparing alpha values

Algorithm 2 measures requirements testability based on our testability model. It needs parameter alpha, the base cost of analyzing an extra sentence in a given requirement definition, discussed in Sect. 3.4. Table 10 shows the computed values of alpha for all projects in our dataset. For each project, we have selected the most appropriate values for the four different aspects of alpha, shown in Figure 2. The lower bound and upper bound of alpha have been determined using softened and hardened policies. From Table 10, we can observe that the test effort, addressed by alpha, for a safety-critical project such as a train system controller, ERTMS/ETCS, is more than a password management system, KeePass.

Figure 12 shows the impact of the parameter α on the requirements testability of different projects in our dataset. The points show the value of the testability of individual requirements. The average testability of the requirements for both the softened and hardened policies of alpha is shown with different markers. The requirements testability in the absence of alpha ($\alpha = 0$) is the same requirement clarity. Therefore, Figure 12 also denotes the upper bound and lower bound for requirements testability values of each project in our dataset. It is observed that the testability of the requirements in the EIRENE and CCTNS projects is seriously affected by parameter α, while for other projects, testability differences are negligible. It means that most of the requirements in EIRENE and CCTNS projects, which are critical systems, have been expressed in more than one sentence.

4.5.2 Quantitative analysis

Figure 13 depicts the mean of testability along with the standard deviation error for each project in our dataset when the softened policy is used for alpha. Figure 14 shows the same diagram when the hardened policy is chosen for the alpha. The testability of the individual requirements also has been shown with data points on both figures.

It can be observed that our proposed tool, ARTA, underestimates requirements testability in most (five of six) projects in our dataset, while the hand-made dictionary used in Smella often overestimates the testability. ARTA produces testability values that are closer to the ground-truth than the ones produced by Smella [16] in four of six projects, including EIRENE, ERTMS/ETCS, CCTNS, and Peering. About 92% of the requirements in our datasets

Table 10 Boundary values of alpha for projects used in our study

Project	Domain	Criticality	Type	Template	Alpha	
					Softened	Hardened
EIRENE	EE	Safety-critical	Functional	Multiple sentences	0.4836	0.7535
ERTMS/ETCS	EE + ME	Safety-critical	Functional	Single sentence	0.6093	0.8792
CCTNS	LW	Business-critical	Functional	Multiple sentences	0.3102	0.5801
Gamma-J	EC + CS	Business-critical	Functional	Single sentence	0.3445	0.6144
KeePass	CS	Non-critical	Functional	Single sentence	0.2075	0.4150
Peering	CS	Business-critical	Functional	Single sentence	0.2700	0.5399

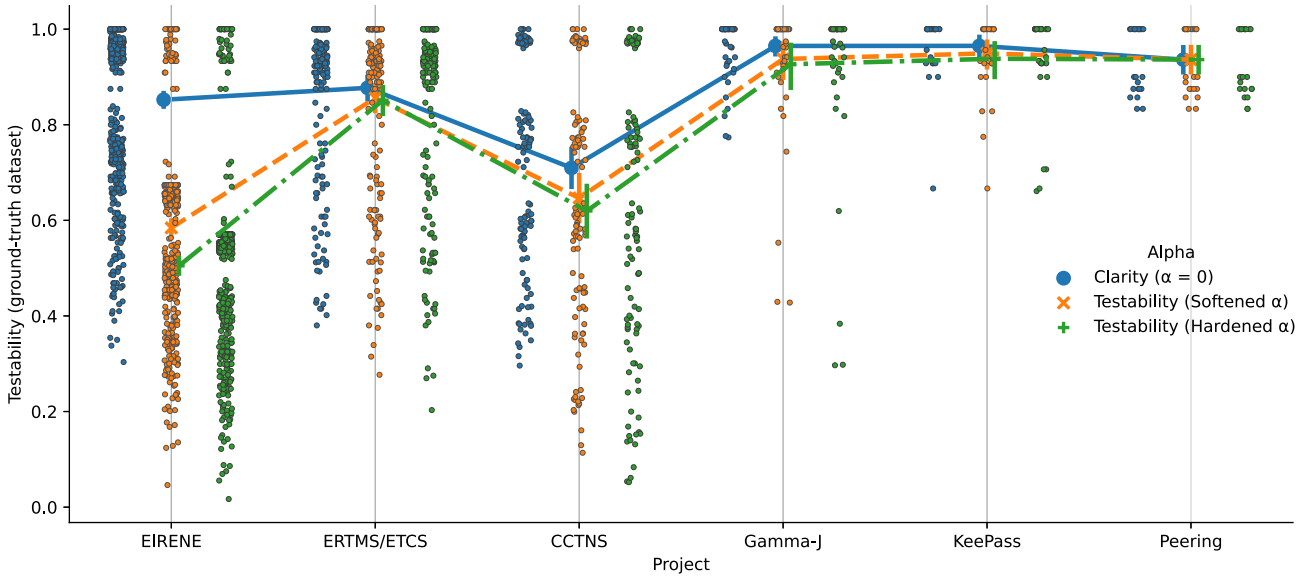


Fig. 12 Requirements testability with different values of α for each project

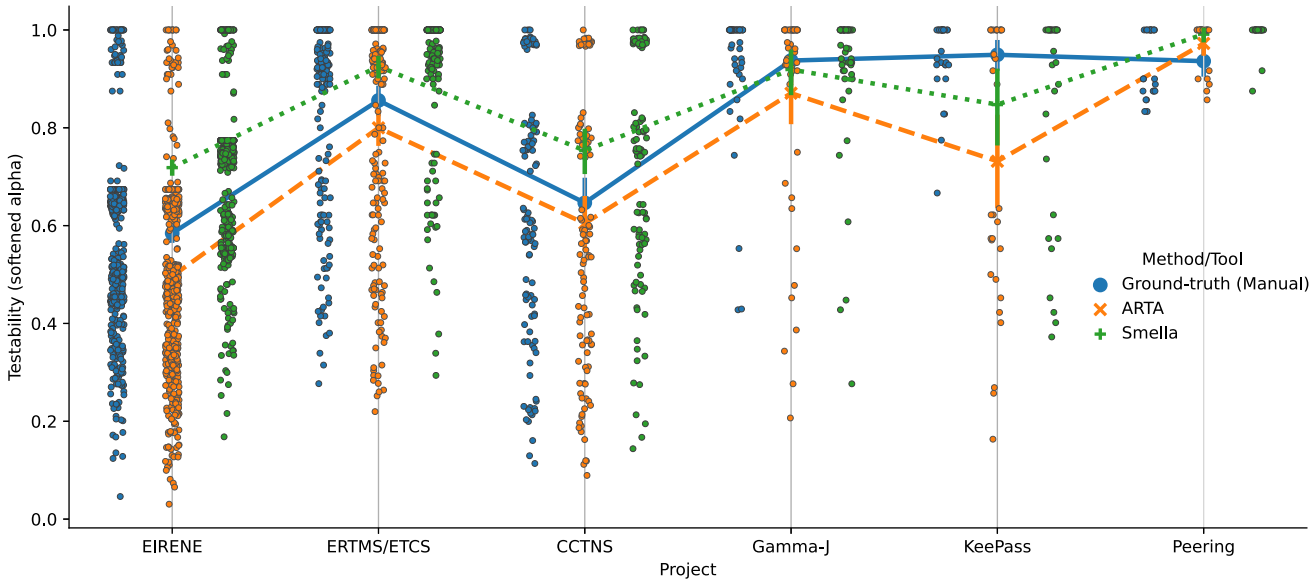


Fig. 13 Comparison of requirements testability results with the ground-truth dataset (softened alpha)

belong to these projects. Figures 13 and 14 also reveal that most requirements of the evaluated systems are testable since the data points are denser in the top area of the charts.

Figure 15 gives detailed information about the requirements testability status in the prepared dataset and proposed approach. It illustrates the distribution of testability values for the requirements on each project with histogram plots. Figure 15a and c shows the distribution of ground-truth values with softened alpha. Figure 15b and d shows the distribution of estimated values via our proposed method for measuring requirements testability based on smells. Histogram plots indicate that our tool, ARTA, preserved the distribution of ground-truth values. Indeed, the distribution of requirements testability values computed by our proposed model is highly similar to the ground-truth testability values in the prepared dataset, indicating that the proposed testability model is accurate at all testability levels (histogram bins). Moreover, requirements testability values have peaks at [0.55, 0.75] and [0.90, 1.0], revealing moderate to high testability for most requirements.

Table 11 shows the value of error metrics for estimating testability by our method with two softened and hardened policies for alpha. The total mean squared error is between 0.0293 and 0.0339 (respectively, for hardened and softened alpha), which means that our approach can measure the

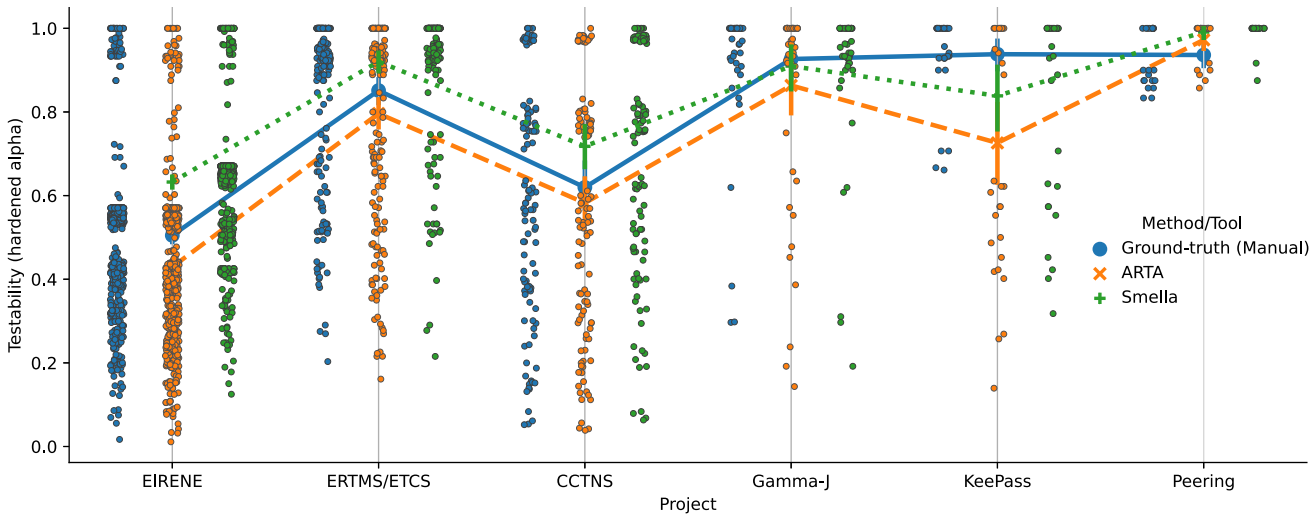
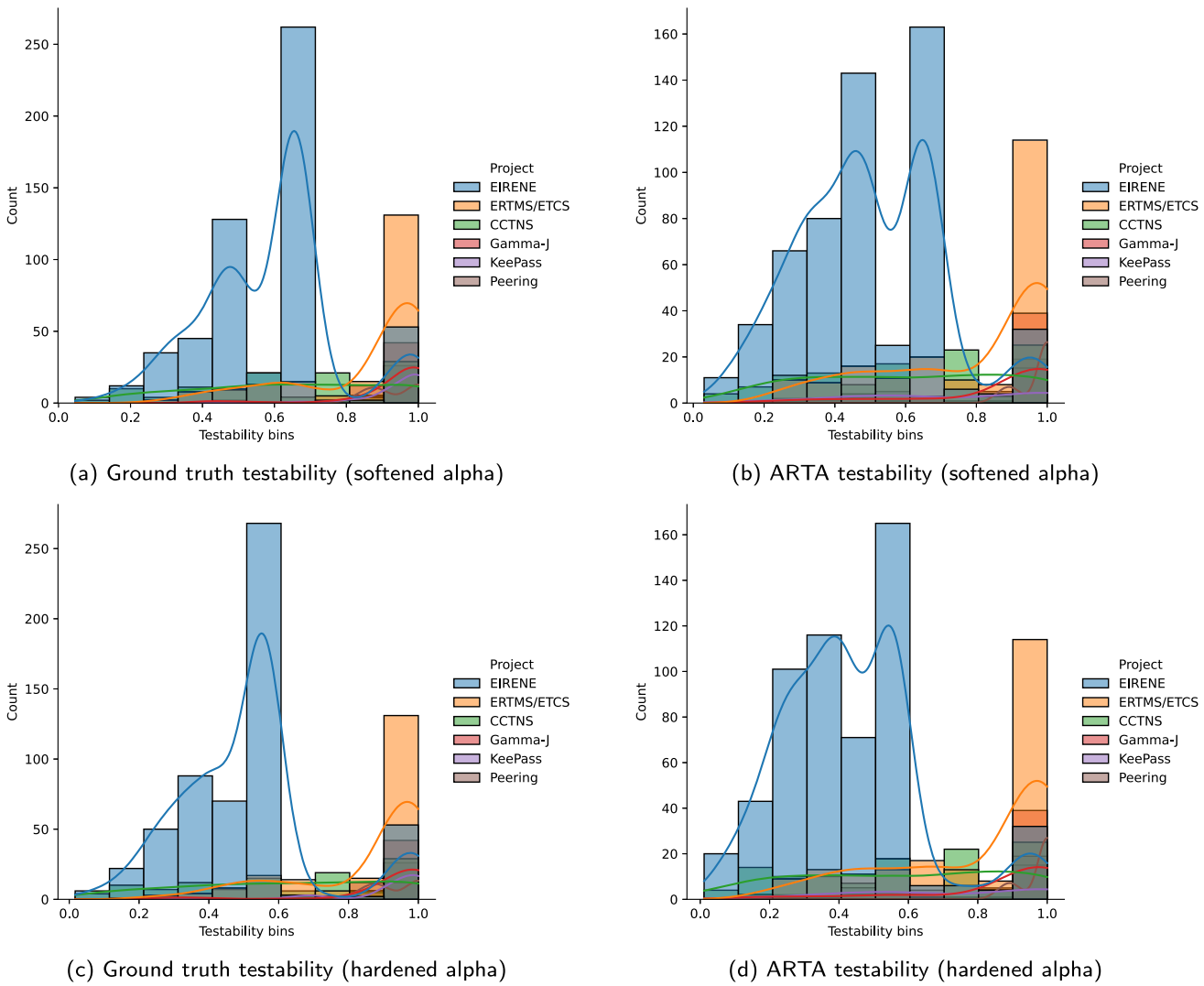


Fig. 14 Comparison of requirements testability results with the ground-truth dataset (hardened alpha)



(a) Ground truth testability (softened alpha)

(b) ARTA testability (softened alpha)

(c) Ground truth testability (hardened alpha)

(d) ARTA testability (hardened alpha)

Fig. 15 Distribution of the requirements testability values

Table 11 Testability measurement errors

Project	Alpha	MAE	MSE	RMSE	MSLnE	MdAE
EIRENE	Softened	0.1230	0.3001	0.1735	0.0135	0.1076
	Hardened	0.1050	0.0229	0.1514	0.0109	0.0899
ERTMS/ETCS	Softened	0.1246	0.0431	0.2076	0.0151	0.0588
	Hardened	0.1236	0.0428	0.2068	0.0150	0.0588
CCTNS	Softened	0.1012	0.0233	0.1525	0.0088	0.0449
	Hardened	0.0952	0.0218	0.1475	0.0082	0.0421
Gamma-J	Softened	0.1081	0.0307	0.1753	0.0106	0.0667
	Hardened	0.1044	0.0295	0.1718	0.0100	0.0667
KeePass	Softened	0.2326	0.1061	0.3257	0.0392	0.1577
	Hardened	0.2270	0.1010	0.3178	0.0376	0.1427
Peering	Softened	0.0524	0.0069	0.0828	0.0018	0.0
	Hardened	0.0524	0.0069	0.0828	0.0018	0.0
All requirements	Softened	0.1218	0.0339	0.1840	0.0137	0.0795
	Hardened	0.1102	0.0293	0.1711	0.0120	0.0714

testability of the requirements based on requirement smells with a relatively low error ($<5\%$). At the same time, error values are relatively different for each project due to various types and frequencies of their smells. For instance, in the KeePass project, we got the worst results. Figure 12 shows that 32 requirements in the KeePass project are almost clean and contain only a short sentence. Hence, the KeePass requirements are highly testable. Underestimating the testability of the requirements for the KeePass project originates from a relatively high false-positive rate of smells, such as vague pronouns, which improvement in them can be the subject of future works. Another important observation is that the prediction results are a bit more accurate with a hardened policy for the alpha parameter. The main reason is that the impact of test effort is increased while the impact of test quality remains fixed in terms of requirement clarity. Nevertheless, the difference between the upper and lower bound of testability measurement error is negligible, which means that our approach is stable regarding the alpha parameter.

4.5.3 Evaluation by experts

To evaluate the accuracy and usefulness of the proposed testability measurement formula, we randomly selected 21 requirements from each project and ranked them in the order of their testability magnitude, computed by Eq. (4). Thereafter, we asked three independent experts, highly experienced in requirements engineering, to manually score the selected requirements based on the extent to which objectives and feasible tests can be designed to determine whether each requirements is met or not [4]. Our experts were asked to score the requirements testability between 1 and 10 for each selected requirement. Then, we averaged the testability scores assigned by the experts to each requirement and computed the rank of manual evaluation for each requirement. Finally, we computed the Spearman rank-order correlation between the requirements' ranks provided by our approach and the ranks by the manual evaluation scores.

Table 12 shows the correlation analysis between our ranking results and the experts' opinions about the

Table 12 Spearman rank-order correlation coefficient with the associated p-value between expert testability scores and ARTA

Project	Softened value for testability		Hardened value for testability	
	Spearman correlation	*P*-value	Spearman correlation	*P*-value
EIRENE	0.9547	6.3744×10^{-11}	0.9349	1.5555×10^{-9}
ERTMS/ETCS	0.8590	1.2358×10^{-6}	0.8590	1.2358×10^{-6}
CCTNS	0.9324	2.1803×10^{-9}	0.9423	5.4530×10^{-10}
Gamma-J	0.8749	4.4837×10^{-7}	0.8749	4.4837×10^{-7}
KeePass	0.8900	1.4816×10^{-7}	0.8973	8.2228×10^{-8}
Peering	0.5399	1.3910×10^{-2}	0.5399	1.3990×10^{-2}
Total	0.9310	1.7530×10^{-53}	0.9321	6.5681×10^{-8}

Table 13 Examples of requirements with smells and their clarity and testability values

Requirement	Project	Requirement clarity	Requirement testability	
			Softened alpha	Hardened alpha
R1: "For **calls** between a controller and the lead cab, it shall be possible to add the controller to the multi-driver **call** • Either the lead driver **calls** the controller or the controller **calls** the lead driver • In the latter **case**, the controller is automatically added into the multi-driver **call** • Functional identity of the controller shall be displayed in the leading cab •"	EIRENE	0.69	0.21 (0.21)	0.13 (0.13)
R2: "The composition of **call** groups shall be able to be modified within the network • A single-user shall be able to be a member of one or **more call** groups•"	EIRENE	0.68	0.46 (0.46)	0.39 (0.39)
R3: "The maximum length of a message **segment** shall be 96 characters • A message can include **several segments** •"	EIRENE	0.58	0.39 (0.44)	0.33 (0.37)
R4: "If a document is either **too long**, dispersed over **several pages** or in a **specific** layout **that** is not **suitable** for online reading, a printer-friendly version of the document **should** be provided that prints the content in a form acceptable to the user (e.g. in the expected layout, paper format, or orientation) •"	CCTNS	0.27	0.27 (0.27)	0.27 (0.27)
R5: "When the train is stationary or after a **certain** time (e.g. the time for "route releasing" of the overlap, the **release** speed calculation shall be based on the distance to the **danger point** (if calculated on-board) • The condition for this change shall be defined for each target as infrastructure data •"	ERTMS/ ETCS	0.72	0.45 (0.35)	0.38 (0.30)
R6: "All data related to the **word** must be shown • For example, if user **types** "abc" and abc is part of a password and a username, both entries must be shown •"	KeePass	0.93	0.77 (0.57)	0.66 (0.48)
R7: "The system will employ on demand asynchronous loading for **faster** execution of **pages** •"	Gamma-J	0.61	0.61 (0.61)	0.61 (0.61)
R8: "Primary CDN has already acquired **sufficient external** resources •"	Peering	0.50	0.50 (0.88)	0.50 (0.88)

requirements. The last row of the table shows the results for all requirements. A correlation coefficient of 0.93 is observed, for all evaluated requirements in this experiment in the last row. The null hypothesis that the result occurred by chance is rejected with a p-value less than 0.01.

The correlation coefficient for all evaluated requirements is close to one (≈0.93), indicating that our tool can rank the requirement statements based on their testability values such that the ranks are in high agreement with the ones obtained by experienced requirement engineers. This way, our tool can measure the requirement testability automatically. The ideal value for the correlation coefficient is one, which means there is no difference between manual and automated ranking. It is worth noting that the p-value for the 0.93 correlation coefficient is less than 0.01, indicating a confidence level of 99% for the obtained correlation coefficient. Therefore, the correlation coefficient of 0.93 confirms the accuracy and usefulness of our requirements testability measurement approach.

4.5.4 Qualitative analysis

To make sense of the concept of requirements testability based on our proposed method, several concrete examples selected from our requirement dataset, along with their smells, clarity, and testability, are shown in Table 13. Actual smells have been **bolded** throughout the text, and words identified by ARTA as smells have been marked with an underline. Sentences within a requirement are separated by • symbol. The upper bound and lower bound of testability have been reported considering softened and hardened policies for alpha. Testability has also been computed based on our automatic method and shown inside parentheses underneath the ground-truth values.

Focusing on Table 13, we observe some exciting results supporting the applicability of our proposed testability measurement mathematical model. For example, the clarity of both the requirements R1 and R2 is almost equal, while it is evident that testing and validating R1, presumably, requires more effort than R2. The effort is taken into account by both the alpha parameter and the number of sentences in the requirement. Our testability quantification model can correctly rank these two requirements. On the other hand, the number of sentences of both the requirements R2 and R3 are equal. However, the clarity of R2 is more than R3. Therefore, the testability of R3 is less than R2.

The model has correctly detected all the smells in R1 and R2 definitions that imply no testability measurement error. However, in requirement R3, our model has detected the word "can" as a requirement smell incorrectly. The reason is that sometimes, and not all the time, "can" is used in the sense of "may" to give permission. It needs a large corpus of texts labeled with the definition of "can" as "can" or "may" to use Word2Vec analysis for learning the contexts in which "can" can be replaced with "may."

Requirement R4 shows another desired aspect of our testability model. This requirement contains eight smells of six different types, making it too smelly. However, all smells have occurred in one sentence, *i.e.*, the minimum possible unit for declaring a requirement in natural language. Therefore, there is no cost for extra sentences when testing the requirement. Indeed, it only expresses one requirement. It is observed that both the requirement smells and the requirement's length are indicators of the requirements testability. Table 13 also denotes that there are various requirements with different levels of testability in our labeled dataset. Therefore, it can be used in future works on requirement quality analysis.

4.5.5 Important smells affecting requirements testability

To understand which smells are more important in determining requirements testability, we fit a decision tree regressor [77] on our dataset using the number of each smell as independent variables or features and the value of testability as dependent variables. The decision tree regressor, at each step, finds the best feature and divides samples based on the most appropriate value of this feature, such that the regression error, e.g., mean squared error (MSE), is minimized.

Figure 16 shows three top levels of the decision tree regressor after training on the dataset containing the lower bounds of testability values obtained by our proposed method. The selected feature, its cut-off value, *i.e.*, the value of the feature in which data are divided into the separated sub-trees, the mean squared error (MSE) of division, the portion of samples, and the Gini index as impurity value is shown on each node. The root of the decision tree regressor indicates that the most important feature to estimate testability is the Polysemy smell with a cut-off point of 0.5, MSE of 0.0706, and impurity of 0.6346. Uncertain verbs and Subjective language constitute the second level of decision tree regressor. It is observed that frequent requirement smells, e.g., Polysemy, have more negative impacts on testability than other smells. However, a dataset with many labeled samples is still required to build a more reliable machine-learning model to predict testability. It is worth noting that distinguishing testable requirements from non-testable ones is not a trivial machine learning task since most requirements are testable, and therefore, the dataset is imbalanced.

Answer to RQ_4: *Requirement testability could be determined based on requirement smell and length of the requirement with a mean absolute error of* 0.12 *and mean squared error of* 0.03. *The most important smells affecting requirements testability are Polysemy, Uncertain Verbs, and Subjective language. Prevalent requirement smells affect requirements testability more than other smells.*

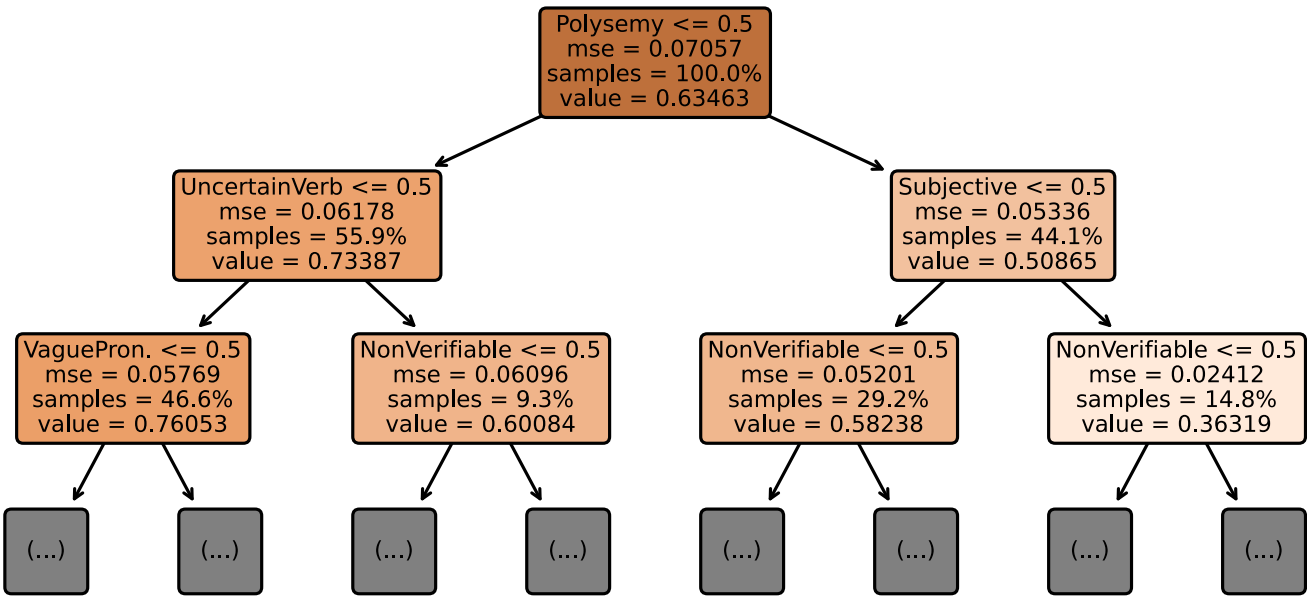


Fig. 16 Most important smells affecting requirements testability

5 Threats to validity

We argue the most important threats that may affect the validation of our proposed method and its evaluation and discuss how we mitigate them.

5.1 Threats to construct validity

The main threat to the construct validity of our approach is the proposed testability formula. Our requirements testability formula aims to provide a means to *compare and rank* requirements based on their testability measures. It is based on simple heuristics to measure requirements testability from the viewpoint of the requirement engineer obtained by applying the goal-question-metric (GQM) approach [52]. The dissimilarities of various domains with computer science have been measured empirically by applying a word embedding technique on a considerable corpus of Wikipedia documents. Other aspects affecting the parameter alpha are ordinal according to the importance of different levels in each aspect. For instance, the cost of acceptance testing of a requirement in safety-critical systems is more than a non-critical system [63]. Further empirical validations must be performed to tune the numerical parameters, requiring large annotated requirement smell datasets.

5.2 Threats to internal validity

Threats to internal validity are related to the correctness of the experiment outcomes. A major threat to the internal validity of our approach results is that the experience of our postgraduate students, as well as the experts, could play a role in their judgments on the detection of smells in our requirements dataset. As described in Sect. 4.1.2, initially, 5000 requirements were labeled by the students. The requirements were discussed and labeled by four groups of three postgraduate students. Initially, we used these 5000 labeled requirements to evaluate our smell detection approach. When comparing the manually detected smells with the ones detected by our approach, we came across relatively high false negative and false positive rates for some types of smells.

In some cases, the difficulty was with our approach, and we corrected the approach as much as we could. However, we noticed that in most cases, the requirements were not labeled appropriately. Thereafter, the manually labeled dataset provided by our students was revised several times by experts in the software industry. We mitigated the threat by discarding those requirements for which there was not a total agreement on the type of smells. Finally, we ended up with 985 requirements that both the experts in software industries and our teams agreed on their types of smells. There are still false negatives and false positives when comparing the smells detected by our approach with the ones detected manually and included in our dataset.

Another internal threat is the choice of word embedding algorithm and hyperparameters used to configure the algorithm. Different word embedding algorithms can be used to train Word2Vec models [59, 78]. We used the Continuous Bag of Words (CBOW) algorithm [59], one of the state-of-the-art neural word embedding algorithms, with default hyperparameters available in the Gensim library [67]. According to Mikolov et al. [59] the CBOW Word2Vec algorithm is several times faster to train than the well-known Skip-gram Word2Vec algorithm. The computational complexity of CBOW depends on the vocabulary size (V), the context window size (C), the embedding dimension (D), and the hidden layer size (H). The number of operations per training iteration is $O(N \times (D \times C + D \times H + H \times V))$, where N is the number of training examples. The complexity can be reduced by using techniques that decrease V or N, such as hierarchical softmax, negative sampling, or subsampling. These techniques make CBOW [59] more efficient and scalable without affecting the word embedding quality. In addition, CBOW is slightly more accurate than other Word2Vec techniques, such as Skip-gram for the frequent words [59, 60], making it more suitable for building our smelly words dictionary. Finally, the CBOW architecture works better than the neural net language models (NLMs) on the semantic-syntactic relationship task [59]. For these reasons, we chose CBOW to find the most frequent words in computer science, which may contain different meanings in other domains. Nevertheless, exploring other word embedding models and their suit hyperparameters for building the smelly word dictionary is part of our ongoing work.

5.3 Threats to external validity

Threats to external validity are related to the generalization of our outcomes. The outcomes might be affected by three factors. Firstly, our dictionary of smelly words is limited to only ten different application domains. Therefore, it could be of no use for the other domains. However, the dictionary can be extended automatically with the smelly words in any domain. Secondly, we have considered only nine requirement smells to compute our proposed requirement evaluation metric which is called requirement clarity. Newly introduced smells should be integrated with our tools.

The third external threat is that our experiments have been performed on natural language requirements in English. Therefore, the Precision and Recall value may differ

for requirements written in other languages. However, our dictionary-building approach can be applied to application domains in any language since Wikipedia articles are available for a wide range of human languages.

6 Conclusion

Software requirements testability can be expressed and measured as a function of requirement smells and the requirement's length. In this paper, we show that a newly defined requirement smell, Polysemy, along with Subjective language and Non-verifiable terms, are the most prevalent smells in the software requirements, resulting in non-testable requirements. These smells could be detected using a comprehensive dictionary provided by training a Word2Vec model on a massive textual corpus in different domains and ranking ambiguous words and terms with many possible meanings. The proposed dictionary is extended to include smelly words rather than solely nouns in ten different application domains. The experiments with nearly 1000 software requirements from six well-known industrial and academic projects demonstrate that the proposed method supported by our web-based tool, ARTA, outperforms Smella, a state-of-the-art tool, to detect requirement smells. ARTA improves the Precision, Recall, and F1 score of dictionary-based smell detection, respectively, with an average of 0.0311, 0.3299, and 0.2824. As a result, ARTA measures the requirement testability with a mean absolute error of 0.12 and a mean squared error of 0.03. The correlation coefficient between the requirements testability ranks computed by ARTA and the ranks obtained by experts' opinions is close to one indicating the applicability of the proposed approach for use in practice.

Our results show that while only 4.93% of words are recognized as smelly words in the software requirements, 72.39% of requirements contain at least one kind of smell. The frequency of requirements smells in each requirement is similar to the code smells. Subjective languages and Polysemy are among the most crucial smells affecting requirements testability, and the number of smells in the requirements increases as the requirement length grows. Therefore, both the requirements smells and the length of the requirements must be considered when measuring the testability of requirements definitions. Quantifying requirements testability in terms of requirement smells allows requirement engineers and domain experts to know about the issues in requirements with the exact location of the issue and refactor the smelly parts as soon as possible. It directly affects the performance of functional and acceptance testing in the software development lifecycle.

Future works can focus on utilizing the ARTA platform to enlarge the dataset for more requirements and then use machine learning predictive models to predict requirements smells as well as requirements testability based on the requirement statement. Most smells are identified by analyzing the whole sentence rather than a single word. We plan to improve smell detection performance, specifically reducing the false-positive rate by applying Recurrent Neural Networks (RNNs) [79] and Transformer models [80]. RNNs and Transformers are new deep learning models that can be used to consider the entire requirement text for identifying smells without any dictionary or word-level analysis. Another opportunity for the study is to propose a method for automatically refactoring the requirement smells after detection. Similar methods in NLP tasks, such as anaphora resolution, which most commonly appears as pronoun resolution, can be exploited as primary solutions for recommending a proper word to use instead of smelly words.

Funding This study has received no funding from any organization.

Data availability The datasets generated and analyzed during the current study are available in Zenodo, https://doi.org/10.5281/zenodo.4266727.

Declarations

Conflict of interest All of the authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. dos Santos EC, Vilain P (2018) Automated acceptance tests as software requirements: an experiment to compare the applicability of fit tables and Gherkin language. Springer, Cham, pp 104–119
2. Femmer H, Vogelsang A (2019) Requirements quality is quality in use. IEEE Softw 36(3):83–91. https://doi.org/10.1109/MS.2018.110161823
3. Hayes JH, Li W, Yu T, Han X, Hays M, Woodson C (2015) Measuring requirement quality to predict testability. In: 2015 IEEE second international workshop on artificial intelligence for requirements engineering (AIRE). IEEE, pp 1–8. https://doi.org/10.1109/AIRE.2015.7337622. URL: http://ieeexplore.ieee.org/document/7337622/
4. ISO/IEC/IEEE (2017a) ISO/IEC/IEEE 24765:2017 Systems and software engineering-vocabulary. [Online]. Available: https://www.iso.org/standard/71952.html
5. Garousi V, Felderer M, KıLıçaslan FN (2019) A survey on software testability. Inf Softw Technol 108:35–64. https://doi.org/10.1016/j.infsof.2018.12.003. URL:https://linkinghub.elsevier.com/retrieve/pii/S0950584918302490
6. Khan RA, Mustafa K (2009) Metric based testability model for object oriented design (MTMOOD). ACM SIGSOFT Softw Eng

Notes 34(2):1. https://doi.org/10.1145/1507195.1507204. URL:http://portal.acm.org/citation.cfm?doid=1507195.1507204
7. Shaheen MR, Du Bousquet L (2014) Survey of source code metrics for evaluating testability of object oriented systems. Technical Report, Inria France. URL: https://hal.inria.fr/hal-00953403
8. Terragni V, Salza P, Pezzè M (2020) Measuring software testability modulo test quality. In: Proceedings of the 28th international conference on program comprehension, pp 241–251, New York, NY, USA, ACM. ISBN 9781450379588. https://doi.org/10.1145/3387904.3389273. URL: https://dl.acm.org/doi/10.1145/3387904.3389273
9. Zakeri-Nasrabadi M, Parsa S (2021) Learning to predict software testability. pp 1–5. IEEE, 3. ISBN 978-1-6654-1241-4. https://doi.org/10.1109/CSICC52343.2021.9420548
10. Morteza Z-N, Saeed P (2022) An ensemble meta-estimator to predict source code testability. Appl Soft Comput 129:109562. https://doi.org/10.1016/j.asoc.2022.109562. https://linkinghub.elsevier.com/retrieve/pii/S1568494622006263
11. Ammann P, Offutt J (2016) Introduction to software testing. Cambridge University Press, Cambridge. https://doi.org/10.1017/9781316771273
12. Beer A, Junker M, Femmer H, Felderer M (2017) Initial investigations on the influence of requirement smells on test-case design. In: 2017 IEEE 25th international requirements engineering conference workshops (REW), pp 323–326. IEEE. ISBN 978-1-5386-3488-2. https://doi.org/10.1109/REW.2017.43. URL: http://ieeexplore.ieee.org/document/8054872/
13. Izosimov V, Ingelsson U, Wallin A (2012) Requirement decomposition and testability in development of safety-critical automotive components. pp 74–86. https://doi.org/10.1007/978-3-642-33678-2_7
14. Gonzalo G, Fuentes José M, Juan L, Omar H, Valentín M (2013) A framework to measure and improve the quality of textual requirements. Requir Eng 18(1):25–41. https://doi.org/10.1007/s00766-011-0134-z
15. Femmer H, Fernández DM, Juergens E, Klose M, Zimmer I, Zimmer J (2014) Rapid requirements checks with requirements smells: two case studies. In: Proceedings of the 1st international workshop on rapid continuous software engineering-RCoSE 2014, pp 10–19, New York, New York, USA, ACM Press. ISBN 9781450328562. https://doi.org/10.1145/2593812.2593817
16. Henning F, Méndez FD, Stefan W, Sebastian E (2017) Rapid quality assurance with requirements smells. J Syst Softw 123:190–213. https://doi.org/10.1016/j.jss.2016.02.047
17. Jurafsky D, Martin JH (2009) Speech and language processing, 2nd edn. Prentice-Hall Inc., Upper Saddle River
18. Petrov S, Das D, McDonald R (2011) A universal part-of-speech tagset. Computing Research Repository-CORR
19. Christopher D (1999) Manning; and Hinrich Schütze. Foundations of statistical natural language processing. MIT Press, Cambridge
20. Wilson WM, Rosenberg LH, Hyatt LE (1997) Automated analysis of requirement specifications. In: Proceedings of the 19th international conference on Software engineering-ICSE '97, pp 161–171, New York, New York, USA, ACM Press. ISBN 0897919149. https://doi.org/10.1145/253228.253258
21. Fabbrini F, Fusani M, Gnesi S, Lami G (2001) The linguistic approach to the natural language requirements quality: benefit of the use of an automatic tool. In: Proceedings 26th annual NASA goddard software engineering workshop, pp 97–105, IEEE Comput. Soc, 2001. ISBN 0-7695-1456-1. https://doi.org/10.1109/SEW.2001.992662. URL: http://ieeexplore.ieee.org/document/992662/
22. Tjong SF, Berry DM (2013) The design of SREE-a prototype potential ambiguity finder for requirements specifications and lessons learned. pp 80–95. https://doi.org/10.1007/978-3-642-37422-7_6
23. Gleich B, Creighton O, Kof L (2010) Ambiguity detection: towards a tool explaining ambiguity sources. pp 218–232. https://doi.org/10.1007/978-3-642-14192-8_20
24. Berry DM, Kamsties E, Krieger MM (2003) From contract drafting to software specification: linguistic sources of ambiguity. https://cs.uwaterloo.ca/~dberry/handbook/ambiguityHandbook.pdf
25. Ferrari A, Donati B, Gnesi S (2017) Detecting domain-specific ambiguities: an NLP approach based on Wikipedia crawling and word embeddings. In: 2017 IEEE 25th international requirements engineering conference workshops (REW). pp 393–399. IEEE. ISBN 978-1-5386-3488-2. https://doi.org/10.1109/REW.2017.20. URL: http://ieeexplore.ieee.org/document/8054883/
26. Ferrari A, Esuli A, Gnesi S (2018) Identification of cross-domain ambiguity with language models. In: 2018 5th international workshop on artificial intelligence for requirements engineering (AIRE), pp 31–38. IEEE. ISBN 978-1-5386-8404-7. https://doi.org/10.1109/AIRE.2018.00011. URL: https://ieeexplore.ieee.org/document/8501308/
27. Alessio F, Andrea E (2019) An NLP approach for cross-domain ambiguity detection in requirements engineering. Autom Softw Eng 26(3):559–598
28. Chetan A, Mehrdad S, Lionel B, Frank Z (2015) Automated checking of conformance to requirements templates using natural language processing. IEEE Trans Softw Eng 41(10):944–968. https://doi.org/10.1109/TSE.2015.2428709
29. Liping Z, Waad A, Alessio F, Letsholo Keletso J, Ajagbe Muideen A, Erol-Valeriu C, Batista-Navarro Riza T (2021) Natural language processing for requirements engineering: a systematic mapping study. ACM Comput Surv. https://doi.org/10.1145/3444689
30. IEEE (1990) IEEE 610.12-1990-IEEE Standard Glossary of Software Engineering Terminology. URL: https://standards.ieee.org/standard/610_12-1990.html
31. ISO/IEC/IEEE (2017b) ISO/IEC/IEEE 12207:2017(en) Systems and software engineering-software life cycle processes. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:12207:ed-1:v1:en
32. ISO/IEC/IEEE (2018) ISO/IEC/IEEE 29148:2018(en) Systems and software engineering-life cycle processes-requirements engineering. URL: https://www.iso.org/obp/ui#iso:std:iso-iec-ieee:29148:ed-2:v1:en
33. Lami G, Gnesi S, Fabbrini F, Fusani M, Trentanni G (2004) An automatic tool for the analysis of natural language requirements. Informe técnico, CNR Information Science and Technology Institute, Pisa, Italia, Setiembre
34. Hui Y, de Roeck A, Vincenzo G, Alistai W, Bashar N (2011) Analysing anaphoric ambiguity in natural language requirements. Requir Eng 16(3):163–189. https://doi.org/10.1007/s00766-011-0119-y
35. Alessio F, Gloria G, Benedetta R, Iacopo T, Stefano B, Alessandro F, Stefania G (2018) Detecting requirements defects with NLP patterns: an industrial experience in the railway domain. Empir Softw Eng 23(6):3684–3733. https://doi.org/10.1007/s10664-018-9596-7
36. Dalpiaz F, van der Schalk I, Brinkkemper S, Aydemir FB, Lucassen G (2019) Detecting terminological ambiguity in user stories: tool and experimentation. Inf Softw Technol 110:3–16. https://doi.org/10.1016/j.infsof.2018.12.007
37. Webber FDS (2015) Semantic folding theory and its application in semantic fingerprinting. arXiv:1511.08855
38. Ezzini S, Abualhaija S, Arora C, Sabetzadeh M, Briand LC (2021) Using domain-specific corpora for improved handling of ambiguity in requirements. In: 2021 IEEE/ACM 43rd

international conference on software engineering (ICSE). pp 1485–1497. IEEE. ISBN 978-1-6654-0296-5. https://doi.org/10.1109/ICSE43902.2021.00133. URL: https://ieeexplore.ieee.org/document/9402055/
39. Fantechi A, Gnesi S, Semini L (2019) Applying the QuARS tool to detect variability. In: Proceedings of the 23rd international systems and software product line conference volume B - SPLC '19, pp 1–4, New York, New York, USA, ACM Press. ISBN 9781450366687. https://doi.org/10.1145/3307630.3342388. URL: http://dl.acm.org/citation.cfm?doid=3307630.3342388
40. Chantree F, Nuseibeh B, de Roeck A, Willis A (2006) Identifying nocuous ambiguities in natural language requirements. In: 14th IEEE international requirements engineering conference (RE'06), pp 59–68. IEEE. ISBN 0-7695-2555-5. https://doi.org/10.1109/RE.2006.31. URL: http://ieeexplore.ieee.org/document/1704049/
41. Tom Y, Devamanyu H, Soujanya P, Erik C (2018) Recent trends in deep learning based natural language processing. IEEE Comput Intell Mag 13(3):55–75. https://doi.org/10.1109/MCI.2018.2840738
42. de Bruijn F, Dekkers HL (2010) Ambiguity in natural language software requirements: a case study. In: Roel W, Anne P (eds) Requirements engineering: foundation for software quality. Springer, Berlin, pp 233–247
43. QRACorp. (2021) QVscribe. [Online]. Available: https://qracorp.com
44. Visure Solutions Inc. (2020) Visure Quality Analyzer-requirements quality metrics. [Online]. Available: https://visuresolutions.com/visure-quality-analyzer-write-requirements
45. Ferrari A, Spagnolo GO, Fiscella A, Parente G (2019) QuOD: an NLP tool to improve the quality of business process descriptions. pp 267–281. https://doi.org/10.1007/978-3-030-30985-5_17. URL: http://link.springer.com/10.1007/978-3-030-30985-5_17
46. Shevchenko A, Lytvyn M, Lider D (2009) Grammarly. [Online]. Available: https://www.grammarly.com/grammar-check
47. Song X, Wu N, Song S, Stojanovic V (2013) Switching-like event-triggered state estimation for reaction-diffusion neural networks against dos attacks. Neural Proc Lett 55:8997–9018. https://doi.org/10.1007/s11063-023-11189-1
48. Tengda W, Xiaodi L, Vladimir S (2021) Input-to-state stability of impulsive reaction-diffusion neural networks with infinite distributed delays. Nonlinear Dyn 103:1733–1755. https://doi.org/10.1007/s11071-021-06208-6
49. Xiaona S, Peng S, Shuai S, Vladimir S (2023) Quantized neural adaptive finite-time preassigned performance control for interconnected nonlinear systems. Neural Comput Appl 35:15429–15446. https://doi.org/10.1007/s00521-023-08361-y
50. Zakeri NM, Parsa S, Kalaee A (2021) Format-aware learn & fuzz: deep test data generation for efficient fuzzing. Neural Comput Appl. https://doi.org/10.1007/s00521-020-05039-7
51. Zhilu X, Xiaodi L, Vladimir S (2021) Exponential stability of nonlinear state-dependent delayed impulsive systems with applications. Nonlinear Anal Hybrid Syst 42:101088. https://doi.org/10.1016/j.nahs.2021.101088
52. Basili VR, Caldiera G, Rombach HD (1994) The goal question metric approach. Encycl Softw Eng, 528–532
53. Fischbach J, Vogelsang A, Spies D, Wehrle A, Junker M, Freudenstein D (2020) SPECMATE: automated creation of test cases from acceptance criteria. In: 2020 IEEE 13th international conference on software testing, validation and verification (ICST), pp 321–331. IEEE, ISBN 978-1-7281-5778-8. https://doi.org/10.1109/ICST46399.2020.00040. URL: https://ieeexplore.ieee.org/document/9159056/
54. Robertson S, Robertson J (2006) Mastering the requirements process. ACM Press books, Addison-Wesley, Boston
55. Fabbrini F, Fusani M, Gnesi S, Lami G (2000) Quality evaluation of software requirement specifications. In: Proceedings of the software and internet quality week 2000 conference, pp 1–18
56. Ian F (2002) Alexander and Richard Stevens. Addison-Wesley Professional, Writing better requirements. 0321131630
57. Huertas C, Juárez-Ramírez R (2013) Towards assessing the quality of functional requirements using english/spanish controlled languages and context free grammar. In: Proc. third international conference on digital information and communication technology and its applications (DICTAP 2013), Ostrava, Czech Republic on, pp 234–241. Citeseer
58. Beer A, Felderer M (2018) Measuring and improving testability of system requirements in an industrial context by applying the goal question metric approach. In: Proceedings of the 5th international workshop on requirements engineering and testing, RET '18, pp 25–32, New York, NY, USA. Association for Computing Machinery. ISBN 9781450357494. https://doi.org/10.1145/3195538.3195542
59. Tomas M, Kai C, Greg C, Jeffrey D (2013) Efficient estimation of word representations in vector space
60. Giatsoglou M, Vozalis MG, Diamantaras K, Vakali A, Sarigiannidis G, Chatzisavvas KC (2017) Sentiment analysis leveraging emotions and word embeddings. Expert Syst Appl 69:214–224. https://doi.org/10.1016/j.eswa.2016.10.043
61. Juergens E, Deissenboeck F, Feilkas M, Hummel B, Schaetz B, Wagner S, Domann C, Streit J (2010) Can clone detection support quality assessments of requirements specifications? In: Proceedings of the 32nd ACM/IEEE international conference on software engineering-ICSE '10, volume 2, pp 79, New York, New York, USA. ACM Press. ISBN 9781605587196. https://doi.org/10.1145/1810295.1810308. URL: http://portal.acm.org/citation.cfm?doid=1810295.1810308
62. Fontana FA, Ferme V, Zanoni M, Roveda R (2015) Towards a prioritization of code debt: a code smell Intensity Index. In: 2015 IEEE 7th international workshop on managing technical debt (MTD), pp 16–24. IEEE, oct 2015. ISBN 978-1-4673-7378-4. https://doi.org/10.1109/MTD.2015.7332620. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7332620
63. Dubrova E (2013) Fault-tolerant design. Springer, Berlin
64. Ian S (2016) Software engineering, 10th edn. Pearson Education Limited, Harlow
65. Goldsmith RF (2004) Discovering real business requirements for software project success. Artech House Inc., Boston
66. Loria S (2020) TextBlob: simplified text processing. [Online]. Available: https://textblob.readthedocs.io/en/dev/
67. Rehurek R (2020) Gensim. [Online]. Available: https://radimrehurek.com/gensim/
68. Adrian H, Simon W (2020) Django. [Online]. Available: https://www.djangoproject.com/
69. Ferrari A, Spagnolo GO, Gnesi S (2017) PURE: a dataset of public requirements documents. In: 2017 IEEE 25th international requirements engineering conference (RE). pp 502–505. IEEE. ISBN 978-1-5386-3191-1. https://doi.org/10.1109/RE.2017.29. URL: http://ieeexplore.ieee.org/document/8049173/
70. ISO/IEC/IEEE (2011) IEEE/ISO/IEC 29148-2011 - ISO/IEC/IEEE International Standard-Systems and software engineering–Life cycle processes–Requirements engineering. URL: https://standards.ieee.org/standard/29148-2011.html
71. Florian S, Brian B (2013) A literature survey on international standards for systems requirements engineering. Proc Comput Sci 16:796–805. https://doi.org/10.1016/j.procs.2013.01.083
72. Google (2020) Google colab. [Online]. Available: https://colab.research.google.com
73. DiCiccio TJ, Efron B (1996) Bootstrap confidence intervals. Statist Sci 11(3):189–228. https://doi.org/10.1214/ss/1032280214

74. Fabiano P, Di ND, De RC, De LA (2020) A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. J Syst Softw 169:110693. https://doi.org/10.1016/j.jss.2020.110693
75. Christopher Frey H, Patil SR (2002) Identification and review of sensitivity analysis methods. Risk Anal 22:553–578. https://doi.org/10.1111/0272-4332.00039
76. Wiegers Karl E, Joy B (2013) Software requirements 3. Microsoft Press, Redmond
77. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: machine learning in python. J Mach Learn Res 12:2825–2830
78. Pennington J, Socher R, Manning CD (2014) GloVe: global vectors for word representation. In: Empirical methods in natural language processing (EMNLP), pp 1532–1543, 2014. URL: http://www.aclweb.org/anthology/D14-1162
79. Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MIT Press, Cambridge
80. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L (2017) Attention is all you need. In: Guyon I, Von Luxburg U, Bengio S, Wallach H, Fergus R, Vishwanathan S, Garnett R (eds) Advances in neural information processing systems, vol 30. Curran Associates Inc, Glasgow