On- and Off-Topic Classification and Semantic Annotation of User-Generated Software Requirements

Markus Dollmann and **Michaela Geierhos**
Heinz Nixdorf Institute
University of Paderborn
Fürstenallee 11, 33102 Paderborn, Germany
{dollmann|geierhos}@hni.upb.de

Abstract

Users prefer natural language software requirements because of their usability and accessibility. When they describe their wishes for software development, they often provide off-topic information. We therefore present REaCT[1], an automated approach for identifying and semantically annotating the on-topic parts of requirement descriptions. It is designed to support requirement engineers in the elicitation process on detecting and analyzing requirements in user-generated content. Since no lexical resources with domain-specific information about requirements are available, we created a corpus of requirements written in controlled language by instructed users and uncontrolled language by uninstructed users. We annotated these requirements regarding predicate-argument structures, conditions, priorities, motivations and semantic roles and used this information to train classifiers for information extraction purposes. REaCT achieves an accuracy of 92% for the on- and off-topic classification task and an F_1-measure of 72% for the semantic annotation.

1 Introduction

"Requirements are what the software product, or hardware product, or service, or whatever you intend to build, is meant to do and to be" (Robertson and Robertson, 2012). This intuitive description of requirements has one disadvantage. It is as vague as a requirement that is written by an untrained user. More generally, functional requirements define what a product, system or process, or a part of it is meant to do (Robertson and Robertson, 2012; Vlas and Robinson, 2011). Due to its expressiveness, natural language (NL) became a popular medium of communication between users and developers during the requirement elicitation process (de Almeida Ferreira and da Silva, 2012; Mich et al., 2004). Especially in large ICT projects, requirements, wishes, and ideas of up to thousands of different users have to be grasped (Castro-Herrera et al., 2009). For this purpose, requirement engineers collect their data, look for project-relevant concepts and summarize the identified technical features. However, this hand-crafted aggregation and translation process from NL to formal specifications is error-prone (Goldin and Berry, 1994). Since people are getting tired and unfocused during this monotonous work, the risk of information loss increases. Hence, this process should be automated as far as possible to support requirement engineers.

In this paper, we introduce our approach to identify and annotate requirements in user-generated content. We acquired feature requests for open source software from SourceForge[2], specified by (potential) users of the software. We divided these requests into off-topic information and (on-topic) requirements to train a binary text classifier. This allows an automated identification of new requirements in user-generated content. In addition, we collected requirements in controlled language from the NFR corpus[3] and from web pages with user-story explanations. We annotated the semantically rele-

[1] Requirements Extraction and Classification Tool

[2] https://sourceforge.net

[3] http://openscience.us/repo/requirements/other-requirements/nfr

Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, pages 1807–1816,
Austin, Texas, November 1-5, 2016.

vant parts of the acquired requirements for information extraction purposes. This will support requirements engineers on requirement analysis and enables a further processing such as disambiguation or the resolution of incomplete expressions.

This paper is structured as follows: In Section 2, we discuss the notion of requirements. Then we provide an overview of previous work (Section 3), before we introduce lexical resources necessary for our method (Section 4). The approach itself is presented in Section 5 before it is evaluated in Section 6. Finally, we conclude this work in Section 7.

2 The Nature of Requirements

Requirement engineers and software developers have to meet users' wishes in order to create new software products. Such descriptions of software functionalities can be expressed in different ways: For example, by using controlled languages or formal methods, clarity and completeness can be achieved. But non-experts can hardly apply them and therefore do not belong to the user group. For this reason, users are encouraged to express their individual requirements for the desired software application in NL in order to improve user acceptance and satisfaction (Firesmith, 2005). In general, software requirements are expressed through active verbs such as *"to calculate"* or *"to publish"* (Robertson and Robertson, 2012). In this work, we distinguish requirements expressed in NL between controlled and uncontrolled language.

A controlled language is a subset of NL, which is characterized by a restricted grammar and/or limited vocabulary (Yue et al., 2010). Requirements in controlled language do not suffer from ambiguity, redundancy and complexity (Yue et al., 2010). That is why these recommendations lead to a desirable input for text processing. Robertson and Robertson (2012) therefore recommend specifying each requirement in a single sentence with one verb. Furthermore, they suggest the following start of record *"The [system/product/process] shall ..."*, which focuses on the functionality and keeps the active form of a sentence. An example therefore is *"The system shall display the Events in a graph by time."* Another type of controlled requirements are user stories. They follow the form *"As a [role], I want [something] so that [benefit]"* and describe software functionalities from the user's perspective (Cohn, 2004). Compared to the previous ones, they do not focus on the technical implementation but concentrate on the goals and resulting benefits. An example therefore is *"As a Creator, I want to upload a video from my local machine so that any users can view it."*

We also consider uncontrolled language in this work because requirements are usually specified by users that have not been instructed for any type of formulation. Requirements in uncontrolled language do not stick to grammar and/or orthographic rules and may contain abbreviations, acronyms or emoticons. There is no restriction how to express oneself. An example therefore is *"Hello, I would like to suggest the implementation of expiration date for the master password :)"*.

In the following, the word "requirement" is used for a described functionality. We assume that its textualization is written within a single English sentence. Requirements are specified in documents like the *Software Requirements Specification* (SRS). We refer to SRS and other forms (e.g. e-mails, memos from workshops, transcripts of interviews or entries in bug-trackers) as requirement documentations.

3 Previous Work

It is quite common that requirement engineers elicit requirements together with users in interviews, group meetings, or by using questionnaires (Mich, 1996). Researchers developed (semi-) automated and collaborative approaches to support requirement engineers in this process (Ankori and Ankori, 2005; Castro-Herrera et al., 2009). Besides the elicitation in interaction with the users, an identification of requirements from existing sources is possible. For example, John and Dörr (2003) used documentations from related products to derive requirements for a new product. Vlas and Robinson (2011) used unstructured, informal, NL feature requests from the platform SourceForge to collect requirements for open source software. They presented a rule-based method to identify and classify requirements according to the quality criteria of the *McCall's Quality Model* (McCall, 1977). Analogous to their work, we want to automatically detect requirements in user-generated content. While they applied a rule-based

method, we plan to identify requirements in user-generated content with a machine learning approach. Since those approaches automatically identify patterns for this classification task, we expect a higher recall and more reliable results.

Goldin and Berry (1994) identified so-called abstractions (i.e. relevant terms and concepts related to a product) of elicited requirements for a better comprehension of the domain and its restrictions. Their tool *AbstFinder* is based on the idea that the significance of terms and concepts is related to the number of mentions in the text. However, in some cases, there is only a weak correlation between the term frequencies and their relevance in documents. This problem can be reduced by a statistical corpus analysis, when the actual term frequency is similar to the expected (Sawyer et al., 2002; Gacitua et al., 2011). This approach eliminates corpus specific stopwords and misleading frequent terms. In our work, we intent to perform a content analysis of the previously detected requirements. However, instead of only identifying significant terms and concepts, we capture the semantically relevant parts of requirements such as conditions, motivations, roles or actions (cf. Figure 1).

In addition to the identification of abstractions, there are methods to transform NL requirements into graphical models (e.g. in *Unified Modeling Language*) (Harmain and Gaizauskas, 2003; Ambriola and Gervasi, 2006; Körner and Gelhausen, 2008). A systematic literature review, done by Yue et al. (2010), aims at the modeling of requirements by comparing transformation techniques in such models. Unlike those techniques, we aim to keep the expressive aspect of the original textual requirements and semantically annotate them for filtering purposes. These results can be further used for different NLP tasks such as disambiguation, resolution of vagueness or the compensation of underspecification.

The semantic annotation task of this work is similar to semantic role labeling (SLR). According to Jurafsky and Martin (2015), the goal of SLR is understanding events and their participants, especially being able to answer the question *who did what to whom* (and perhaps also *when and where*). In this work, we seek to adapt this goal to the requirements domain, where we want to answer the question *what actions should be done by which component* (and perhaps also *who wants to perform that action*, *are there any conditions*, *what is the motivation for performing this action* and *is there a priority assigned to the requirement*).

4 Gathering and Annotation of Controlled and Uncontrolled Requirements

There are benchmarks comparing automated methods for requirement engineering (Tichy et al., 2015). However, none of the published datasets is sufficient to train a text classifier, since annotated information is missing. For our purposes, we need a data set with annotated predicate-argument structures, conditions, priorities, motivations and semantic roles. We therefore created a semantically annotated corpus by using the categories shown in Figure 1, which represent all information bits of a requirement. Since the approach should be able to distinguish between (on-topic) requirements and off-topic comments, we acquired software domain-specific off-topic sentences, too.

Therefore, we acquired requirements in controlled language from the system's and the user's perspective. While requirements from the system's perspective are describing technical software functionalities, the requirements from the user's perspective express wishes for software, to fulfill user needs. For instance, the NFR corpus[4] covers the system's perspective of controlled requirements specifications. It consists of 255 functional and 370 non-functional requirements whereof we used the functional subset to cover the system's perspective. Since we could not identify any requirement corpus that describes a software at user's request, we acquired 304 user stories from websites and books that describe how to write user stories.

However, these requirements in controlled language have not the same characteristics as uncontrolled requirements descriptions. For the acquisition of uncontrolled requirements, we adapted the idea of Vlas and Robinson (2011) that is based on feature requests gathered from the open-source software platform SourceForge[5]. These feature requests

[4] https://terapromise.csc.ncsu.edu/repo/requirements/nfr/nfr.arff

[5] https://sourceforge.net

are created by users that have not been instructed for any type of formulation. Since these requests do not only contain requirements, we split them into sentences and manually classified them in requirements and off-topic information. Here, we consider social communication, descriptions of workflows, descriptions of existing software features, feedback, salutations, or greetings as off-topic information. In total, we gathered 200 uncontrolled on-topic sentences (i.e. requirements) and 492 off-topic ones.

Then we analyzed the acquired requirements in order to identify the different possible semantic categories to annotate their relevant content in our requirements corpus (cf. Figure 1):

- **component**
  - refinement of component
- **action**
  - argument of action
- condition
- priority
- motivation
- role
- **object**
  - refinement of object
- **sub-action**
  - argument of sub-action
- sub-priority
- sub-role
- **sub-object**
  - refinement of sub-object

Figure 1: Semantic categories in our software requirements corpus used for annotation purposes

The categories *component* or *role*, *action* and *object* are usually represented by subject, predicate and object of a sentence. In general, a description refers to a *component*, either to a product or system itself or to a part of the product/system. *Actions* describe what a component should accomplish and affect. Actions have an effect on *Objects*. The authors of the requirements can refine the description of components and objects, which is covered by the categories *refinement of component* and *refinement of object*. For each action, users can set a certain *priority*, describe their *motivation* for a specific functionality, state *conditions*, and/or even define some semantic *roles*. Apart from the component and the object, additional arguments of the action (predicate of a sentence) are annotated with *argument of action*. In some cases, requirements contain sub-requirements in subordinate clauses. The annotators took this into account when using the predefined sub-categories. An example of an annotated requirement is shown in Figure 2.

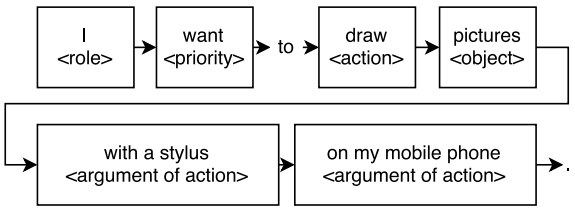

Figure 2: Annotated requirement sample

Two annotators independently labeled the categories in the requirements. We define one of the annotation set as gold standard and the other as candidate set. We will use the gold standard for training and testing purposes in Section 5 and 6 and the candidate set for calculating an inter-annotator agreement. In total, our gold standard consists of 3,996 labeled elements (i.e. clauses, phrases, and even modality). The frequency distribution is shown in Table 1.

Semantic Category	CR	UR	Total
component	241	84	325
refinement of component	6	16	22
action	526	204	730
argument of action	180	104	284
condition	94	39	133
priority	488	209	697
motivation	33	19	52
role	406	42	448
object	540	195	735
refinement of object	155	48	203
sub-action	76	40	116
argument of sub-action	27	14	41
sub-priority	22	16	38
sub-role	22	11	33
sub-object	78	37	115
refinement of sub-object	16	8	24
Total	2,910	1,086	3,996

Table 1: Number of annotated elements per category in our gold standard (CR=controlled requirements, UR=uncontrolled requirements)

The inter-annotator agreement in multi-token annotations is commonly evaluated by using F_1-score (Chinchor, 1998). The two annotators achieve an agreement of 80%, whereby the comparison was invoked from the gold standard.

Many information extraction tasks use the IOB encoding[6] for annotation purposes. When using the IOB encoding, the first token of an element is split into its head (first token) and its tail (rest of the element). That way, its boundaries are labeled with B (begin) and I (inside). This allows separating successive elements of the same category. Thus, we use the IOB encoding during the annotation step. However, we want to discuss a drawback of this notation: When applying text classification approaches in information extraction tasks with IOB encoding, the number of classes reduplicates and this reduces the amount of training data per class. During our annotation process, successive elements of the same semantic category only occurred in the case of *argument of the action* and *argument of the sub-action*. When we disregard the IOB encoding, we can easily split up (sub-)actions by keywords such as *"in"*, *"by"*, *"from"*, *"as"*, *"on"*, *"to"*, *"into"*, *"for"*, and *"through"*. So if we use IO encoding, it can be easily transformed to the IOB encoding. The only difference between IOB and IO encoding is that it does not distinguish between the head and tail of an element and therefore does not double the number of classes.

5 REaCT – A Two-Stage Approach

Requirement documentations are the input of our system. Figure 3 illustrates the two-stage approach divided in two separate classification tasks. First, we apply an on-/off-topic classification to decide whether a sentence is a requirement or irrelevant for the further processing (cf. Section 5.1). Then, the previously identified requirements were automatically annotated (Section 5.2). As a result, we get filtered and semantically annotated requirements in XML or JSON.

The models for on-/off-topic classification and semantic annotation are trained on the gathered requirements (cf. Section 4). We split up the gold standard on sentence level in a ratio of 4:1 in a training set of 607 requirements and test set of 152 requirements. Furthermore, we used 10-fold cross validation on the training set for algorithm configuration and feature engineering (cf. Section 5.1 and Section 5.2). Finally, our approach is evaluated on the test set (cf. Section 6).

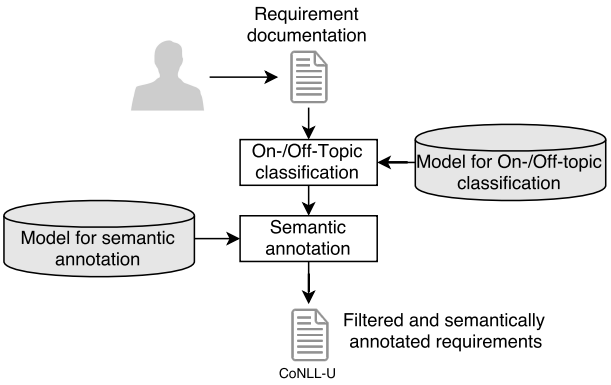

Figure 3: Processing pipeline of the two-stage approach

5.1 On-/Off-Topic Classification Task

User requirement documentations often contain off-topic information. Therefore, we present a binary text classification approach that distinguishes between requirements and off-topic content. Thus, we trained different classification algorithms and tested them using various features and parameter settings. We compared the results to select the best algorithm together with its best-suited parameter values and features.

5.1.1 Features

To differentiate requirements from off-topic content, the sentences will be transformed in numerical feature vectors using a bag-of-words approach with different settings[7]. The features for the transformation are listed along with their possible parameter settings in Table 2. We can choose whether the feature should be taken from word or character n-grams (a.1). For both versions, the unit can range between $[n, m]$ (a.2), which can be specified by parameters. Here, we consider all combinations of $n = [1, 5]$ and $m = [1, 5]$ (where $m \geq n$). If the feature should be build from word n-grams, stopword detection is possible (a.3). Additionally, terms can be ignored that reach a document frequency below or above a given

[6] I (inside), O (outside) or B (begin) of an element

[7] Parameters; to be chosen during algorithm configuration

threshold (e.g. domain-specific stopwords) (a.4 and a.5). Another threshold can be specified to only consider the top features ordered by term frequency (a.6). Besides, it is possible to re-weight the units in the bag-of-words model in relation to the inverse document frequency (IDF) (a.7). Moreover, the frequency vector can be reduced to binary values (a.8), so that the bag-of-words model only contains information about the term occurrence but not about its calculated frequency. We also consider the length of a sentence as feature (b). Furthermore, the frequency of the part-of-speech (POS) tags (c) and the dependencies between the tokens (d) can be added to the feature vector[8]. These two features are optional (c.1 and d.1). This set of features covers the domain-specific characteristics and should enable the identification of the requirements.

#	Feature/Parameter	Possible Values
a	Bag of words	
a.1	analyzer	word, char
a.2	ngram_range	(1,1),(1,2), ...,(5,5)
a.3	stop_words	True, False
a.4	max_df	[0,8,1,0]
a.5	min_df	[0.0,0.5]
a.6	max_features	int or None
a.7	use_idf	True, False
a.8	binary	True, False
b	Length of the sentence	
c	Dependencies	
c.1	use_dep	True, False
d	Part of speech	
d.1	use_pos	True, False

Table 2: Features for on-/off-topic classification together with their corresponding parameters

5.1.2 Selected Algorithms

We selected the following algorithms from the *scikit-learn* library[9] for binary classification: decision tree (`DecisionTreeClassifier`), Naive Bayes (`BernoulliNB` and `MultionmialNB`), support vector machine (`SVC` and `NuSVC`) as well as ensemble methods (`BaggingClassifier`, `RandomForestClassifier`, `ExtraTreeClassifier` and `AdaBoostClassifier`). Finally, after evaluating these algorithms, we chose the best one for the classification task (cf. Section 6).

5.2 Semantic Annotation Task

For each identified requirement, the approach should annotate the semantic components (cf. Figure 1). Here, we use text classification techniques on token level for information extraction purposes. The benefit is that these techniques can automatically learn rules to classify data from the annotated elements (cf. Section 4). Each token will be assigned to one of the semantic categories presented in Figure 1 or the additional class *O* (outside according IOB notation).

We decided in favor of IO encoding during classification to reduce the drawback described in Section 4. We finally convert the classification results into the IOB encoding by labeling the head of each element as begin and the tail as inside. By using the keywords listed in Section 4 as separators, we further distinguish the beginning and the inner parts of arguments.

5.2.1 Features

In the second classification step, we had to adapt the features to token level. The goal of feature engineering is to capture the characteristics of the tokens embedded in their surrounding context. We divided the features in four groups: orthographic and semantic features of the token, contextual features, and traceable classification results.

Orthographic features of a token are its graphematic representation (a) and additional flags that decide if a token contains a number (b), is capitalized (c), or is somehow uppercased (d) (cf. Table 3). For the graphematic representation, we can choose between the token or the lemma (a.1). Another orthographic feature provides information about the length of the token (e). Furthermore, we can use the pre- and suffix characters of the token as features (f and g). Their lengths are configurable (f.1 and g.1).

[8] We use spaCy (`https://spacy.io`) for POS tagging and dependency parsing

[9] `http://scikit-learn.org`

#	Feature/Parameter	Possible Values
a	Graphematic representation	
a.1	use_lemma	True, False
b	Token contains a number	
c	Token is capitalized	
d	Token is somehow uppercased	
e	Length of the token	
f	Prefix of the token	
f.1	length_prefix	[0,5]
g	Suffix of the token	
g.1	length_suffix	[0,5]

Table 3: Orthographic features for semantic annotation

Furthermore, we consider the relevance (h), the POS tag (i) and the WordNet ID of a token (j) as its semantic features (cf. Table 4). By checking the stopword status of a token, we can decide about its relevance. Besides, the POS tag of each token is used as feature. When applying the POS information, we can choose between the *Universal Tag Set*[10] (consisting of 17 POS tags) and the *Penn Treebank Tag Set*[11] (including of 36 POS tags) (i.1). Another boolean feature tells us whether the token appears in *WordNet*[12]. We use this feature as indicator for component or object identification.

#	Feature/Parameter	Possible Values
h	Relevance	
i	Part-of-speech tag	
i.1	extended_tagset	True, False
j	WordNet ID	

Table 4: Semantic features for semantic annotation

As contextual features, we use sentence length (k), index of the token in the sentence (l), as well as the tagging and dependency parsing information of the surrounding tokens (m, n and o) (cf. Table 5). Thus, the POS tags sequences of the n previous and the next m token are considered, where n and m are defined during algorithm configuration (l.1 and n.1). Moreover, it can be specified if each POS tag should be stored as a single feature or should be concatenated (e.g. NOUN+VERB+NOUN) (l.2 and n.2).

#	Feature/Parameter	Possible Values
k	Sentence length	
l	Index of the token	
m	Previous part-of-speech tags	
l.1	n_pos_prev	[0,15]
l.2	conc_prev_pos	True, False
n	Subsequent part-of-speech tags	
n.1	n_pos_succ	[0,15]
n.2	conc_succ_pos	True, False
o	Dependencies	

Table 5: Contextual features for semantic annotation

The classification task is carried out from left to right in the sentence. This enables the consideration of previous classification results (cf. Table 6). We implemented two slightly different variants that can be combined on demand: Firstly, we can define a fixed number of previous classification results as independent or concatenated features (i.e. a sliding window (p)). Secondly, the number of token already assigned to a particular class may be a valuable information (q). This is especially of interest for the hierarchical structure of the categories: For instance, a *sub-object* should only occur if an *object* has already been identified. These two features are optional (p.1 and q.1). The size of the sliding window will be specified during algorithm configuration (p.2).

#	Feature/Parameter	Possible Values
p	Sliding windows	
p.1	conc_prev_labels	True, False
p.2	window_size	[0,10]
q	Number of previous labels per category	
q.1	use_prev_labels	True, False

Table 6: Traceable classification results for semantic annotation

[10] http://universaldependencies.org/u/pos/

[11] http://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

[12] https://wordnet.princeton.edu

5.2.2 Selected Algorithms

In addition to the classifiers we already used in the on-/off-topic classification task, we considered three sequential learning algorithms: conditional random fields (`FrankWolfeSSVM`) from the *PyStruct*-library[13], multinomial hidden markov model (`MultinomialHMM`) as well as structured perceptron from the *seqlearn*-library[14]. We could not estimate feasible parameter settings for the `NuSVC` classifier, so that this classifier was ignored. We chose the algorithm with the best results on the test set for annotating the requirements (cf. Section 6).

6 Evaluation

As mentioned in Section 5, the data was separated in a ratio of 4:1 in a training and a test set. We trained all classifiers on the training set with their defined settings from the automated algorithm configuration. Subsequently, we evaluated these classifiers on the test set. Our results are shown in Table 7 that lists the accuracy for the best classifier per algorithm family of the on-/off-topic classification task. The `ExtraTreeClassifier` performs best on the test data with an accuracy of 92%. The accuracy was calculated with the following formula:

$$accuracy = \frac{\#true\ positives + \#true\ negatives}{\#classified\ requirements}$$

The `ExtraTreeClassifier` is an implementation of *Extremely Randomized Trees* (Geurts et al., 2006). We achieved the best result when using character n-grams as features in the model with a fixed length of 4. Thereby, we considered term occurrence instead of term frequency and IDF. Before creating the bag-of-words model, the approach removes stopwords. Furthermore, the frequency of the POS tags and their dependencies are used as features. In total, the `ExtraTreeClassifier` used 167 estimators based on entropy in the ensemble (algorithm-specific parameters).

[13] `https://pystruct.github.io`

[14] `https://github.com/larsmans/seqlearn`

Classifier	Accuracy
AdaBoostClassifier	0.87
ExtraTreeClassifier	0.92
MultinomialNB	0.89
NuSVC	0.90

Table 7: Accuracy of best classifiers per algorithm family in the on-/off-topic classification task after algorithm configuration

Table 8 shows the values for precision, recall, and F_1 of the `ExtraTreeClassifier`. In brief, the introduced approach detects requirements in user-generated content with an average F_1-score of 91%.

Class	Precision	Recall	F_1
off-topic info	0.94	0.85	0.89
requirements	0.89	0.96	0.93
Avg.	0.92	0.91	0.91

Table 8: Evaluation results for the on-/off-topic classification with the `ExtraTreeClassifier`

Table 9 provides an overview of the results of the semantic annotation task. To determine the F_1-score, the agreement of the predicted and the a priori given annotations is necessary to count an element as true positive.

Again, the `ExtraTreeClassifier` achieves the best F_1-score of 72%. We gained the best results by using 171 randomized decision trees based on entropy (algorithm-specific parameters). As features, we took the POS tags from *Universal Tag Set* for the twelve previous and the three following tokens. Traceable classification results are taken into account by a sliding window of size 1. Besides, we validate if a class label has already been assigned. For each considered token, the four prefix and the two suffix characters as well as the graphematic representation of the token are applied as features.

The sequential learning algorithms (`FrankWolfeSSVM`, `MultinomialHMM` and `StructuredPerceptron`) perform worse than the other classifiers. We assume that this is due to the small amount of available training data. However, the methods depending on decision trees, especially the ensemble methods (`RandomForestClassifier`, `Bagging-`

Classifier and ExtraTreeClassifier), perform significantly better.

Classifier	F_1
AdaBoostClassifier	0.33
ExtraTreeClassifier	0.72
FrankWolfeSSVM	0.50
MultinomialNB	0.64
SVC	0.70

Table 9: F_1-scores of best classifiers per algorithm family in the semantic annotation task after algorithm configuration

In Table 10, we provide detailed results achieved with the ExtraTreeClassifier for the different semantic categories. The recognition of main aspects (component, action and object) reached F_1-scores of 73%, 80% and 68%. The semantic categories, that have only a few training examples, are more error-prone (e.g. sub-action or sub-object).

Semantic Category	Precision	Recall	F_1
component	0.71	0.75	0.73
ref. of component	0.17	0.14	0.15
action	0.78	0.82	0.80
arg. of action	0.49	0.62	0.54
condition	0.88	0.61	0.72
priority	0.96	0.96	0.96
motivation	0.67	0.29	0.40
role	0.93	0.86	0.89
object	0.63	0.74	0.68
ref. of object	0.69	0.51	0.59
sub-action	0.46	0.44	0.45
arg. of sub-action	0.33	0.29	0.31
sub-priority	0.44	0.57	0.50
sub-role	0.40	0.80	0.53
sub-object	0.35	0.33	0.34
ref. of sub-object	0.67	0.33	0.44
Avg.	0.72	0.73	0.73

Table 10: Evaluation results for the semantic annotation with the ExtraTreeClassifier

7 Conclusion and Future Work

Requirement engineers and software developers have to meet users' wishes to create new software products. The goal of this work was to develop a system that can identify and analyze requirements expressed in natural language. These are written by users unlimited in their way of expression. Our system REaCT achieves an accuracy of 92% in distinguishing between on- and off-topic information in the user-generated requirement descriptions. The text classification approach for semantic annotation reaches an F_1-score of 72% – a satisfying result compared to the inter-annotator agreement of 80%. One possibility to improve the quality of the semantic annotation is to expand the training set. Especially the sequential learning techniques need more training data. Besides, this would have a positive impact on those semantic categories that only contain a small number of annotated elements.

Developers and requirement engineers can facilely identify requirements written by users for products in different scenarios by applying our approach. Moreover, the semantic annotations are useful for further NLP tasks. User-generated software requirements adhere to the same quality standards as software requirements that are collected and revised by experts: They should be complete, unambiguous and consistent (Hsia et al., 1993). Since there was no assistant system to check the quality for many years (Hussain et al., 2007) we plan to extend the provided system in order to provide some quality analysis of the extracted information. We have already developed concepts to generate suggestions for non-experts, how to complete or clarify their requirement descriptions (Geierhos et al., 2015). Based on these insights, we want to implement a system for the resolution of vagueness and incompleteness of NL requirements.

Acknowledgments

Special thanks to our colleagues Frederik S. Bäumer and David Kopecki for their support during the semantic annotation of the requirements. This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre „On-The-Fly Computing“ (SFB 901).

References

Vincenzo Ambriola and Vincenzo Gervasi. 2006. On the Systematic Analysis of Natural Language Requirements with CIRCE. *Automated Software Engineering*, 13(1):107–167.

Ronit Ankori and Ronit Ankori. 2005. Automatic requirements elicitation in agile processes. In *Proceedings of the 2005 IEEE International Conference on Software - Science, Technology and Engineering*, pages 101–109. IEEE.

Carlos Castro-Herrera, Chuan Duan, Jane Cleland-Huang, and Bamshad Mobasher. 2009. A recommender system for requirements elicitation in large-scale software projects. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 1419–1426. ACM.

Nancy A. Chinchor, editor. 1998. *Proceedings of the Seventh Message Understanding Conference (MUC-7) Named Entity Task Definition*, Fairfax, VA.

Mike Cohn. 2004. *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co., Redwood City, CA, USA.

David de Almeida Ferreira and Alberto Rodrigues da Silva. 2012. RSLingo: An information extraction approach toward formal requirements specifications. In *Model-Driven Requirements Engineering Workshop*, pages 39–48. IEEE.

Donald G. Firesmith. 2005. Are Your Requirements Complete? *Journal of Object Technology*, 4(2):27–43, February.

Ricardo Gacitua, Pete Sawyer, and Vincenzo Gervasi. 2011. Relevance-based abstraction identification: technique and evaluation. *Requirements Engineering*, 16(3):251–265.

Michaela Geierhos, Sabine Schulze, and Frederik Simon Bäumer. 2015. What did you mean? Facing the Challenges of User-generated Software Requirements. In *Proceedings of the 7th International Conference on Agents and Artificial Intelligence*, pages 277–283, 10 - 12 January. Lisbon. ISBN: 978-989-758-073-4.

Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely randomized trees. *Machine Learning*, 63(1):3–42.

Leah Goldin and Daniel M. Berry. 1994. AbstFinder, A Prototype Abstraction Finder for Natural Language Text for Use in Requirements Elicitation: Design, Methodology, and Evaluation. *Automated Software Engineering*, 4(4):375–412.

H.M. Harmain and R. Gaizauskas. 2003. CM-Builder: A Natural Language-Based CASE Tool for Object-Oriented Analysis. *IEEE International Conference on Software - Science, Technology & Engineering*, 10(2):157–181.

Pei Hsia, Alan Davis, and David Kung. 1993. Status Report: Requirements Engineering. *IEEE Software*, 10(6):75–79, November.

Ishrar Hussain, Olga Ormandjieva, and Leila Kosseim. 2007. Automatic Quality Assessment of SRS Text by Means of a Decision-Tree-Based Text Classifier. In *Proceedings of the 7th International Conference on Quality Software*, QSIC '07, pages 209–218. IEEE.

Isabel John and Jörg Dörr. 2003. Elicitation of Requirements from User Documentation. In *Proceedings of the 9th International Workshop on Requirements Engineering: Foundation of Software Quality*, pages 17–26. Springer.

Daniel Jurafsky and James H Martin. 2015. Semantic role labeling. In *Speech and Language Processing*. 3rd ed. draft edition.

Sven J. Körner and Tom Gelhausen. 2008. Improving Automatic Model Creation using Ontologies. In *Proceedings of the 20th International Conference on Software Engineering & Knowledge Engineering*, pages 691–696. Knowledge Systems Institute.

Jim McCall. 1977. McCall's Quality Model. `http://www.sqa.net/softwarequalityattributes.html`.

Luisa Mich, Mariangela Franch, and Pier Luigi Novi Inverardi. 2004. Market research for requirements analysis using linguistic tools. *Requirements Engineering*, 9(2):151–151.

Luisa Mich. 1996. NL-OOPS: from natural language to object oriented requirements using the natural language processing system LOLITA. *Natural Language Engineering*, 2:161–187.

James Robertson and Suzanne Robertson. 2012. *Mastering the Requirements Process*. Getting Requirements Right. Addison-Wesley Publishing, New York, NY, USA.

Peter Sawyer, Paul Rayson, and Roger Garside. 2002. REVERE: support for requirements synthesis from documents. *Information Systems Frontiers*, 4(3):343–353.

Walter F. Tichy, Mathias Landhäußer, and Sven J. Körner. 2015. nlrpBENCH: A Benchmark for Natural Language Requirements Processing. In *Multikonferenz Software Engineering & Management 2015*. GI.

Radu Vlas and William N. Robinson. 2011. A Rule-Based Natural Language Technique for Requirements Discovery and Classification in Open-Source Software Development Projects. In *Proceedings of the 44th Hawaii International Conference on System Sciences*, pages 1–10. IEEE.

Tao Yue, Lionel C. Briand, and Yvan Labiche. 2010. A systematic review of transformation approaches between user requirements and analysis models. *Requirements Engineering*, 16(2):75–99.