Grant Agreement nº	732463
Project Acronym:	OpenReq
Project Title:	Intelligent Recommendation Decision Technologies for Community-Driven Requirements Engineering
Call identifier:	H2020-ICT-2016-1
Instrument:	RIA (Research and Innovation Action
Topic	ICT-10-16 Software Technologies
Start date of project	January 1st, 2017
Duration	36 months

D3.2 Recommender Engine - Version 1

Lead contractor:	UPC
Author(s):	HITEC, TUGraz, UPC
Submission date:	June 2018
Dissemination level:	PU

Project co-funded by the European Commission under the H2020 Programme.

Abstract: This deliverable presents version 1 of the stakeholders' recommendations engine in OpenReq. Specifically, it defines the architecture, design and future work (when applicable) for the tasks related to the screening and recommendation of relevant requirements (T3.2), the recommendations for improving requirements quality (T3.3) and the prediction of requirement properties (T3.4). It also includes the current state of deployment and integration of version 1 of the engine, together with the challenges and solutions faced during its development.

Table of Contents

1 INTRODUCTION

One of the stated objectives of the OpenReq project is to design an approach for assisting individual stakeholders in different requirements-related tasks such as defining, reusing, screening, understanding, evaluating, and ensuring quality. After identifying the tasks to be covered by the stakeholders' recommender engine in the deliverable *D3.1 - OpenReq Approach for Stakeholders' Recommendations* and in (Palomares et al, 2013), the following stage was to concretize the architecture and design of these tasks, followed by their subsequent development.

In this deliverable we show the architecture, design and current development status of the tasks that are part of version 1 of the stakeholders' recommender engine, those being:

- the screening and recommendation of relevant requirements (T3.2) (Section 2),
- the recommendations for improving requirements quality (T3.3) (Section 3), and
- the prediction of requirement properties (T3.4) (Section 4).

As stated in deliverable *D1.2 - Requirements analysis and design document*, the functionalities have been designed and implemented as microservices. Therefore, the design of these tasks covers, for each microservice, an explanation of the functionality behind the service, its sequence diagram and a usage example.

In addition, this deliverable covers also two further aspects of version 1 of this engine:

- the current state of its deployment and integration (Section 5), and
- the challenges faced during its design, implementation, development and integration, together with the solutions to overcome such challenges (Section 6).

Finally, Section 7 presents a summary of the work reported in this deliverable

2 SCREENING AND RECOMMENDATION OF RELEVANT REQUIREMENTS

The major aim of this task is to reduce the overall effort in a RE process. A typical RE project often contains requirements that are related or similar to requirements that appear in other RE projects. Related or similar requirements cross RE projects often occur due to language constraints, legal regulations, and company guidelines for the look and feel of software products or different domain definitions. Consequently, such similar or related requirements appear to be good candidates to be recommended to the requirements manager in order to be included as additional requirements. This way, the effort to manually define such requirements can be reduced as well as more completeness of requirements can be achieved, i.e., the probability that the requirements manager oversees relevant requirements can be decreased. Hence, this task focuses on the detection and recommendation of relevant requirements.

In order to find relevant requirements, mainly, content-based recommendation and classification techniques are exploited.

2.1 Architecture

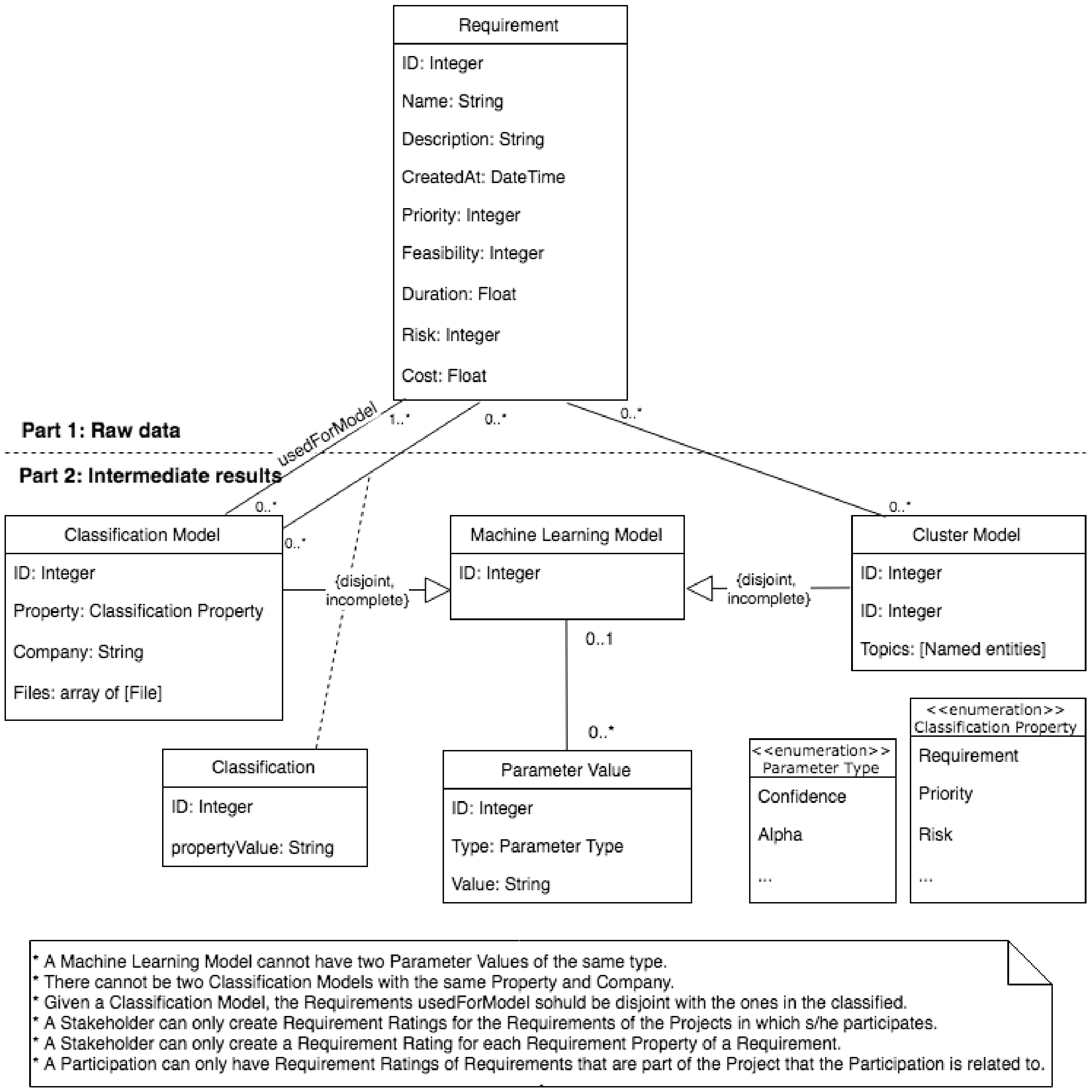
The architecture of the personal stakeholders' recommendation engine was already presented in deliverable D3.1 (Section 3.3.2). At that moment, we envisaged four microservices for task T3.2. However, at the end, the functionalities in this task have been covered using five microservices: a *recommendation* microservice and four microservices that compose a classifier component.

The requirements recommendation engine used by the official OpenReq prototype, which consists of an NLP pipeline and a clustering component, is delivered as two microservices. It covers the mentioned steps in deliverable D3.1 for task T3.2: A1 (screening), A3 (similarity check), and A4 (related requirements detection). This is simply due to the reason that for this microservice, the implementation of A1, A3 and A4 makes use of the same set of libraries in order to perform the aforementioned NLP tasks as well as the clustering approaches. Consequently, it turns out that a separation of these steps would not be reasonable and would introduce further implementation overhead and increase the implementation effort.

The classifier component (composed of 4 microservices) covers A2 (extraction of requirements). The design of this component also implied small changes in the "Intermediate results" layer of the Data entity relationship model (Figure 7 of D3.1), due to the classifier component provided by it (microservices corresponding to the *Classifier component* presented in the following Sections 2.2.3 to 2.2.6 of this deliverable). Figure 1 contains an excerpt of the data entity relationship model with the modifications. Specifically, the modifications consisted of:

- The name of the *Classification* and *Cluster* classes have been updated into *Classification Model* and *Cluster Model*, respectively, to represent more accurately their meaning.
- Due to an error detected in the original model, now *Classification Model* and *Cluster Model* inherit from *Machine Learning Model.*
- The cardinality of the associations between *Machine Learning Model* and *Classification Model*, and between *Machine Learning Model* and *Cluster Model* has been changed to 0..1 in both extremes of the associations.

- The *Classification Model* class has now two new attributes, *company* and *files*. *Company* represents the organization to which the requirement used to create the classification pertains (e.g., Qt), and *files* correspond to sequential files that actually contain the classification model.
- In the *Classification Model* class, the *value* attribute has been deleted and moved to the new association *classified*.
- A new association *classified* has been added between *Classification Model* and *Requirement*, which represents the requirements that have been classified with this classifier. An associative class *Classification* is derived from this association, which contains the *property value* that has been assigned by the classifier to this requirement.



* A Machine Learning Model cannot have two Parameter Values of the same type.
* There cannot be two Classification Models with the same Property and Company.
* Given a Classification Model, the Requirements usedForModel sohuld be disjoint with the ones in the classified.
* A Stakeholder can only create Requirement Ratings for the Requirements of the Projects in which s/he participates.
* A Stakeholder can only create a Requirement Rating for each Requirement Property of a Requirement.
* A Participation can only have Requirement Ratings of Requirements that are part of the Project that the Participation is related to.

Figure 1. Data entity relationship model excerpt of the Data layer (only the part that contains modifications with respect to the one presented in deliverable D3.1)

2.2 Design of current functionalities

2.2.1 similar-requirement-recommender

i. <u>Description</u>

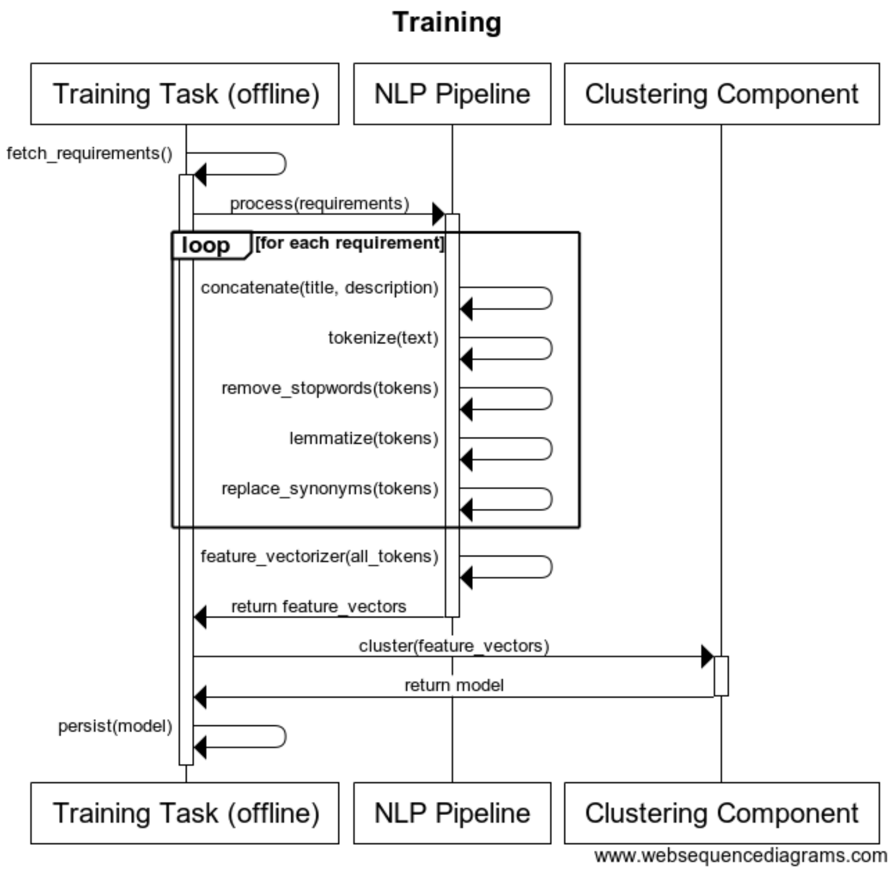
The mission of this task consists in the detection and recommendation of similar requirements by analysing requirements from other (past) RE projects. From an architectural point of view, the whole recommendation process includes several steps.

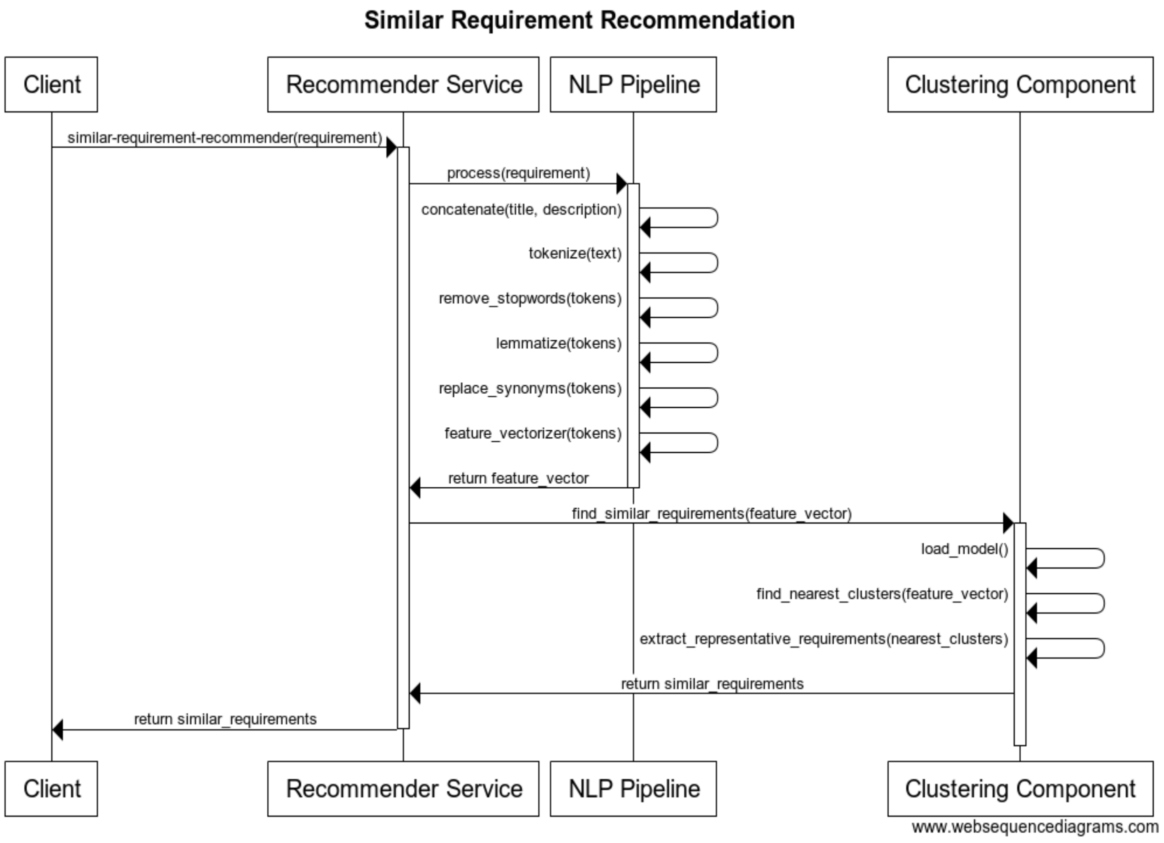
Besides some meta-data of requirements (such as explicitly defined skills required to solve the requirement), as main criteria to find relevant requirements, the title and description of requirements are taken into account. Given a detailed description and a title of every requirement, NLP techniques are applied to extract their relevant features. Thereby, the text is split into tokens and noisy data is removed (i.e., data cleaning). Next, stop words such as prepositions or articles, which do not represent valuable information of a requirement, are removed as well. After that, lemmatization is applied to the remaining tokens to reduce the number of tokens that share the same meaning. This way, undesired ambiguity-related issues caused by the same word appearing as plural and singular words, as verbs in different tenses, can be counteracted. Since each remaining token represents a feature, the number of features is reduced as well. This results in a less complex and more flexible recommendation model. Finally, the current NLP pipeline computes the TF-IDF (Term Frequency and Inverse Document Frequency) values of all remaining tokens, and these values are then used as features.

In the next phase, clustering is applied to group similar requirements based on the pre-processed tokens. Moreover, to better tackle the ambiguity of words (i.e., polysemy) appropriate (soft) clustering techniques such as Latent Semantic Analysis (LSA) are exploited. The recommendation system is based on hard-clustering in terms of hierarchical clustering as well as soft clustering in terms of LSA. Similar requirements that lie in the close proximity in the vector/latent space are then considered as candidates to be recommended to the requirements manager by the recommender system.

This microservice includes an internal offline training to fit the model (see next section for more details).

ii. <u>Sequence diagram</u>


Training
Training Task (offline)
NLP Pipeline
Clustering Component
fetch_requirements()
process(requirements)
loop [for each requirement]
concatenate(title, description)
tokenize(text)
remove_stopwords(tokens)
lemmatize(tokens)
replace_synonyms(tokens)
feature_vectorizer(all_tokens)
return feature_vectors
cluster(feature_vectors)
return model
persist(model)
Training Task (offline)
NLP Pipeline
Clustering Component
www.websequencediagrams.com


Similar Requirement Recommendation
Client
Recommender Service
NLP Pipeline
Clustering Component
similar-requirement-recommender(requirement)
process(requirement)
concatenate(title, description)
tokenize(text)
remove_stopwords(tokens)
lemmatize(tokens)
replace_synonyms(tokens)
feature_vectorizer(tokens)
return feature_vector
find_similar_requirements(feature_vector)
load_model()
find_nearest_clusters(feature_vector)
extract_representative_requirements(nearest_clusters)
return similar_requirements
return similar_requirements
Client
Recommender Service
NLP Pipeline
Clustering Component
www.websequencediagrams.com

iii. Example usage

URL	/tugraz/similar-requirement-recommender
Method	POST
URL params	None

Data params	{
	"id": 1,
	"title": "Speed Measurement",
	"description" : "As evaluation after a workout, the average speed must be shown. The following statistics should be displayed: average speed, maximum speed. This requires a time measurement, distance measurement, and a storage unit for storing the data."
	}

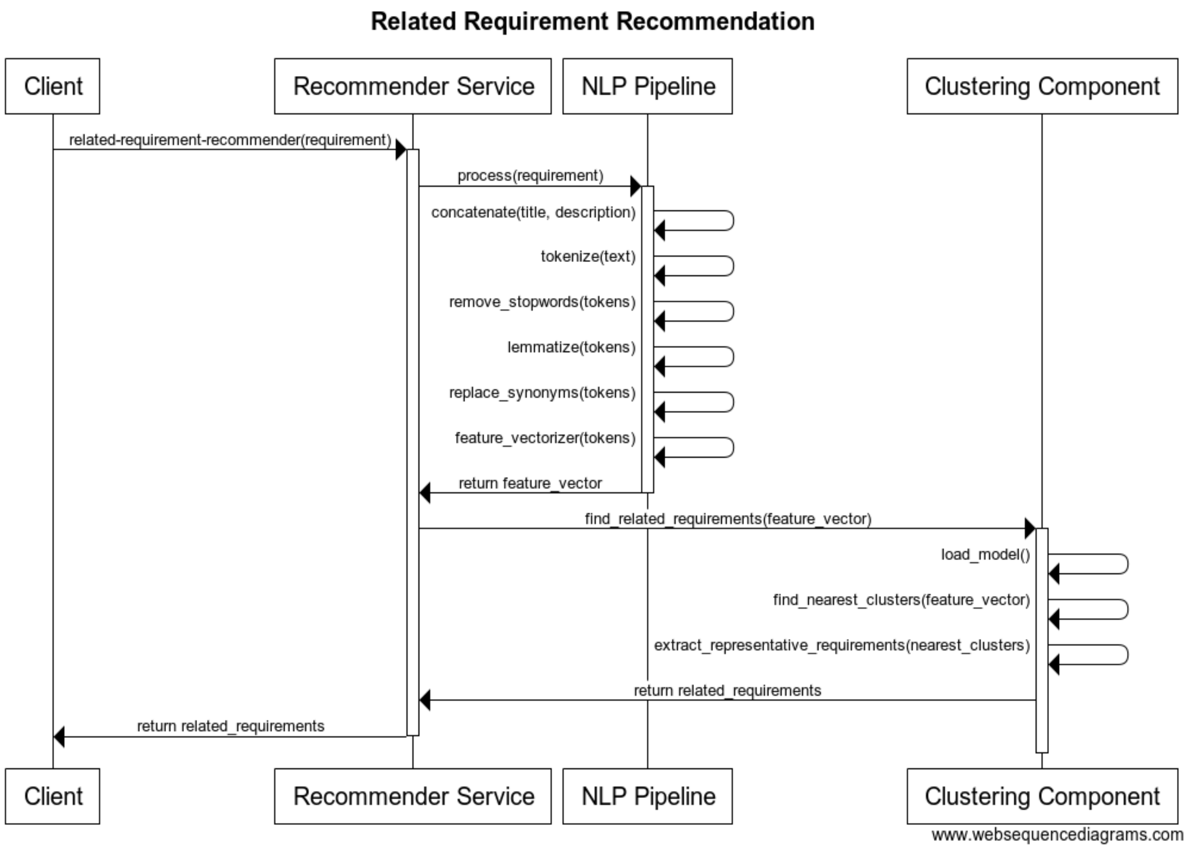
Returns	[{ "id": "28", “title”: “Distance Measurement” "description" : "For statistical purposes, a distance measurement is necessary which requires data from a GPS sensor. This data is needed for the evaluation software and therefore stored in memory." }, ...]
Return explanation	This service returns a list of similar requirements to the given requirement (input requirement). The requirements in the list follow the same format as the input requirement.

2.2.2 *related-requirement-recommender*

i. Description

The microservice for detecting related requirements follows more or less the same structure as the microservice for detecting similar requirements (see Section 2.2.1). The underlying clustering approach differs slightly from the microservice that is about detecting similar requirements. In contrast to the microservice which detects similar requirements, where the focus lies on detecting requirements which have a bigger overlap (duplicate), the related requirements recommendation approach focuses on finding related requirements based on the same context information.

ii. Sequence diagram


Related Requirement Recommendation
Client
Recommender Service
NLP Pipeline
Clustering Component
related-requirement-recommender(requirement)
process(requirement)
concatenate(title, description)
tokenize(text)
remove_stopwords(tokens)
lemmatize(tokens)
replace_synonyms(tokens)
feature_vectorizer(tokens)
return feature_vector
find_related_requirements(feature_vector)
load_model()
find_nearest_clusters(feature_vector)
extract_representative_requirements(nearest_clusters)
return related_requirements
return related_requirements
www.websequencediagrams.com

iii. Example usage

URL	/tugraz/related-requirement-recommender
Method	POST
URL params	None
Data params	{ "id": 1, "title": "Speed Measurement", "description" : "As evaluation after a workout, the average speed must be shown. The following statistics should be displayed: average speed, maximum speed. This requires a time measurement, distance measurement, and a storage unit for storing the data." }
Returns	[{ "id": "13", “title”: “GPS” "description" : "To capture position data, a GPS sensor should be used. Through the measured position and time information, the speed and the distance can be measured." }, ...]

Return explanation	This service returns a list of related requirements to the given requirement (input requirement). The requirements in the list follow the same format as the input requirement.

2.2.3 *classifier-component/train*

i. Description

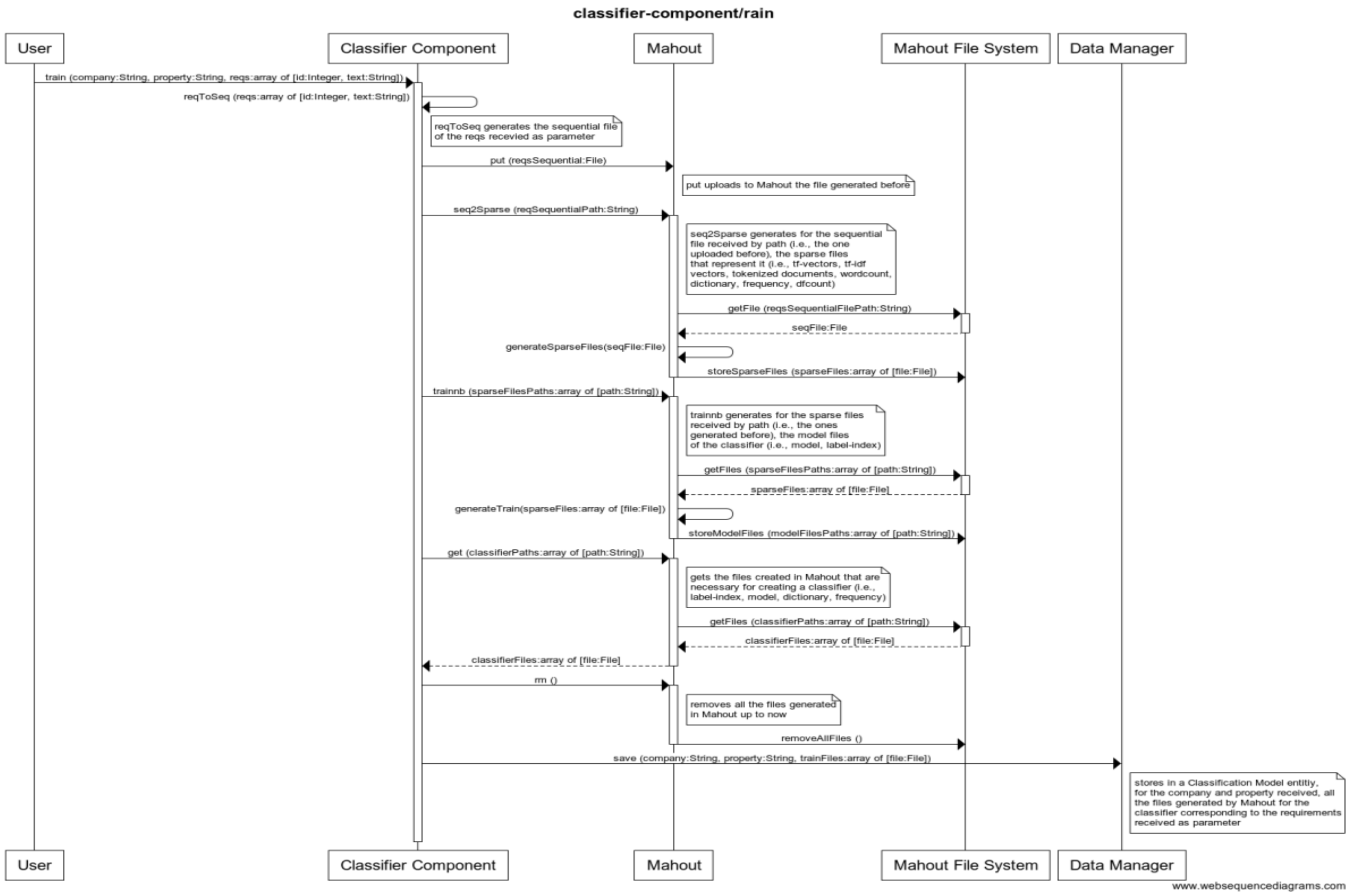
The goal of the *train* microservice is to train a classifier to assign values to a property. In the context of the requirements recommendation engine, the property is *isReq*, and it represents whether a piece of text is an actual requirement or other information (not representative for the requirements engineering stage of a software project) (similar to identifying whether a piece of text is a bug or not in (Antoniol et al., 2008)).

The microservice creates, for the requirements contained in the array received as parameter, the files that represent a classifier that uses these requirements for training and stores these files inside a classification model for the company and property received as parameter. Specifically, these files are: dictionary, frequency, label-index and model. The classifier corresponds to a Naive Bayes classifier. The file creation is done using Mahout[1].

Right now, the name of the company and the property are used as primary key on the database. Therefore, it is not possible to have more than one classifier for the same company and property.

ii. Sequence diagram

[1] https://mahout.apache.org/


classifier-component/rain
User
Classifier Component
Mahout
Mahout File System
Data Manager
train (company:String, property:String, reqs:array of [id:Integer, text:String])
reqToSeq (reqs:array of [id:Integer, text:String])
reqToSeq generates the sequential file of the reqs recevied as parameter
put (reqsSequential:File)
put uploads to Mahout the file generated before
seq2Sparse (reqSequentialPath:String)
seq2Sparse generates for the sequential file received by path (i.e., the one uploaded before), the sparse files that represent it (i.e., tf-vectors, tf-idf vectors, tokenized documents, wordcount, dictionary, frequency, dfcount)
getFile (reqsSequentialFilePath:String)
seqFile:File
generateSparseFiles(seqFile:File)
storeSparseFiles (sparseFiles:array of [file:File])
trainnb (sparseFilesPaths:array of [path:String])
trainnb generates for the sparse files received by path (i.e., the ones generated before), the model files of the classifier (i.e., model, label-index)
getFiles (sparseFilesPaths:array of [path:String])
sparseFiles:array of [file:File]
generateTrain(sparseFiles:array of [file:File])
storeModelFiles (modelFilesPaths:array of [path:String])
get (classifierPaths:array of [path:String])
gets the files created in Mahout that are necessary for creating a classifier (i.e., label-index, model, dictionary, frequency)
getFiles (classifierPaths:array of [path:String])
classifierFiles:array of [file:File]
classifierFiles:array of [file:File]
rm ()
removes all the files generated in Mahout up to now
removeAllFiles ()
save (company:String, property:String, trainFiles:array of [file:File])
stores in a Classification Model entitiy, for the company and property received, all the files generated by Mahout for the classifier corresponding to the requirements received as parameter
User
Classifier Component
Mahout
Mahout File System
Data Manager
www.websequencediagrams.com

iii. Example usage

URL	/upc/classifier-component/train
Method	POST
URL params	None
Data params	{ "requirements" :[{ "id": "4523", "property_value": "Prose"[2], "text" : "Reports and correspondence" }, { "id": "1039", "property_value": "DEF"[3], "text" : "Sufficient number of parking spaces must be provided close to the office." }], "company": "Siemens", "property": "isReq" }
Returns	None

2.2.4 *classifier-component/classify*

i. Description

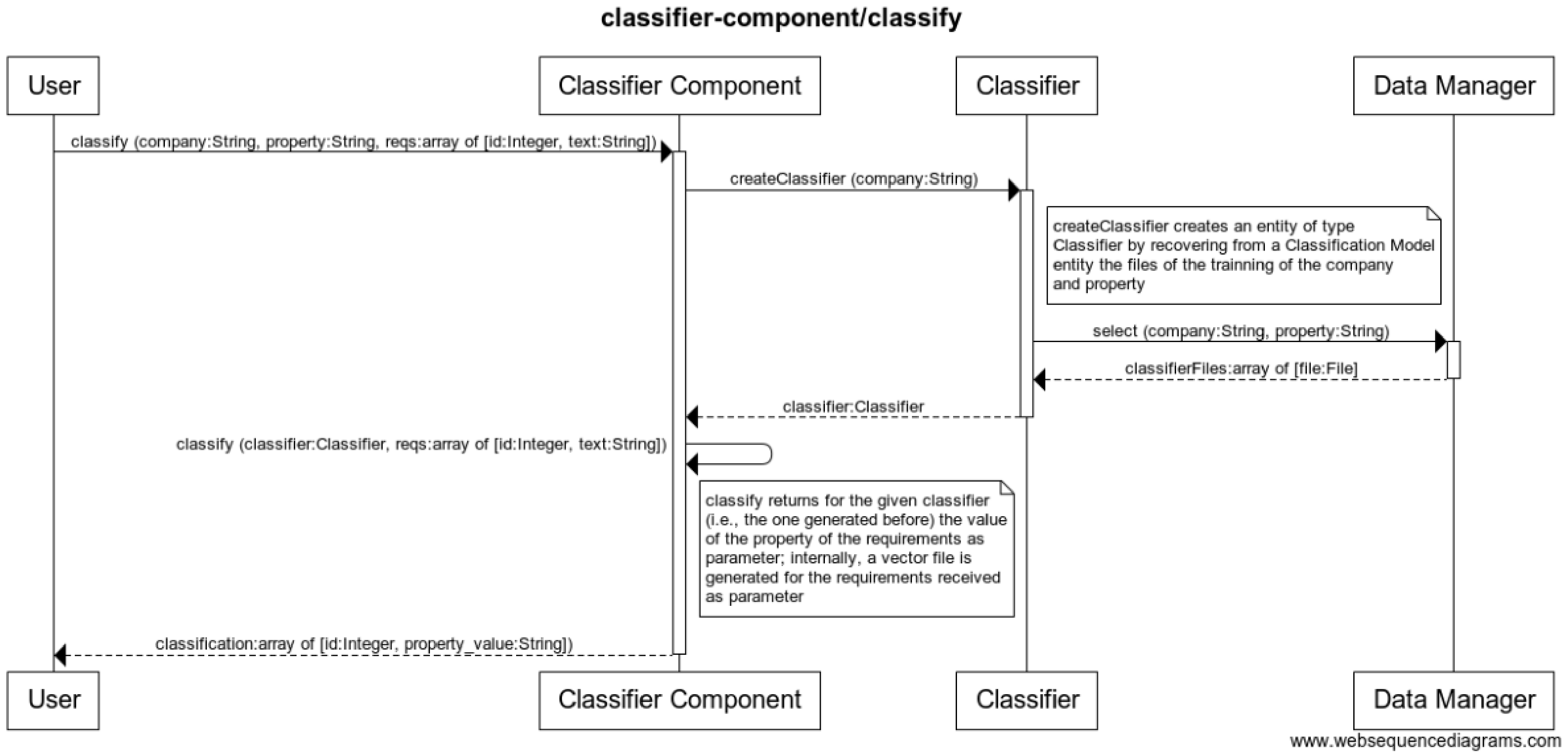
The *classify* microservice assigns values to a requirement property given the case that a classifier was already trained. The information about the classifier is stored in a database in a Classification Model entity (using as primary keys the name of the company to which pertains the requirement used to train the classifier and the property it tries to predict). In this case, the property is *isReq* (see section 2.2.3 for its definition).

The microservice creates first an entity of type Classifier by recovering from the database connected to the data layer the training files for the specific company and property (which are received as parameter). Afterwards, it uses this classifier to classify (i.e., assign a property value to) the requirements received as parameter.

ii. Sequence diagram

[2] *Prose* is the label that represents other information different from an actual requirement (i.e., information not representative for the requirements engineering stage of a software project)

[3] *DEF* is the label that represents a piece of text is an actual requirement


classifier-component/classify
User
Classifier Component
Classifier
Data Manager
classify (company:String, property:String, reqs:array of [id:Integer, text:String])
createClassifier (company:String)
createClassifier creates an entity of type Classifier by recovering from a Classification Model entity the files of the trainning of the company and property
select (company:String, property:String)
classifierFiles:array of [file:File]
classifier:Classifier
classify (classifier:Classifier, reqs:array of [id:Integer, text:String])
classify returns for the given classifier (i.e., the one generated before) the value of the property of the requirements as parameter; internally, a vector file is generated for the requirements received as parameter
classification:array of [id:Integer, property_value:String])
User
Classifier Component
Classifier
Data Manager
www.websequencediagrams.com

iii. Example usage

URL	/upc/classifier-component/classify
Method	POST
URL params	None
Data params	{ "requirements" :[{ "id": "5432", "text" : "Construction Site" }, { "id": "10030", "text" : "Contractor must arrange the Construction Site in accordance with Articles 133 and 134 of the Building Act and Commission Regulation (EC)" }], "company": "Siemens", "property": "isReq" }
Returns	"recommendations": [{ "requirement": “5432”, "property_value": "Prose", "confidence": 93.44250563767345 }, { "requirement": “10030”, "property_value": "DEF", "confidence": 93.79632918450244 }] }
Return explanation	The output of this microservice indicates for each requirement received in the input and the indicated property and company, the value predicted for the property. In the example above, two requirements with id’s *5432* and *10030* are received in the input, the property to be predicted is *isReq* (i.e., the piece of text is an actual requirement or not) and the name of the company is *Siemens*. The microservice returns that the requirement with id *5432* is supposed to be *prose* (which makes sense because its text seems to be kind of the head of a section), and that the requirement *10030* is supposed to be a requirement (i.e., *DEF*) (which makes sense because the piece of text is establishing a constraint of the regulation to be used). In addition, the confidence value of the prediction of each property is returned (i.e., the closer to 100 the confidence is, the more certain is the classifier on the prediction done).

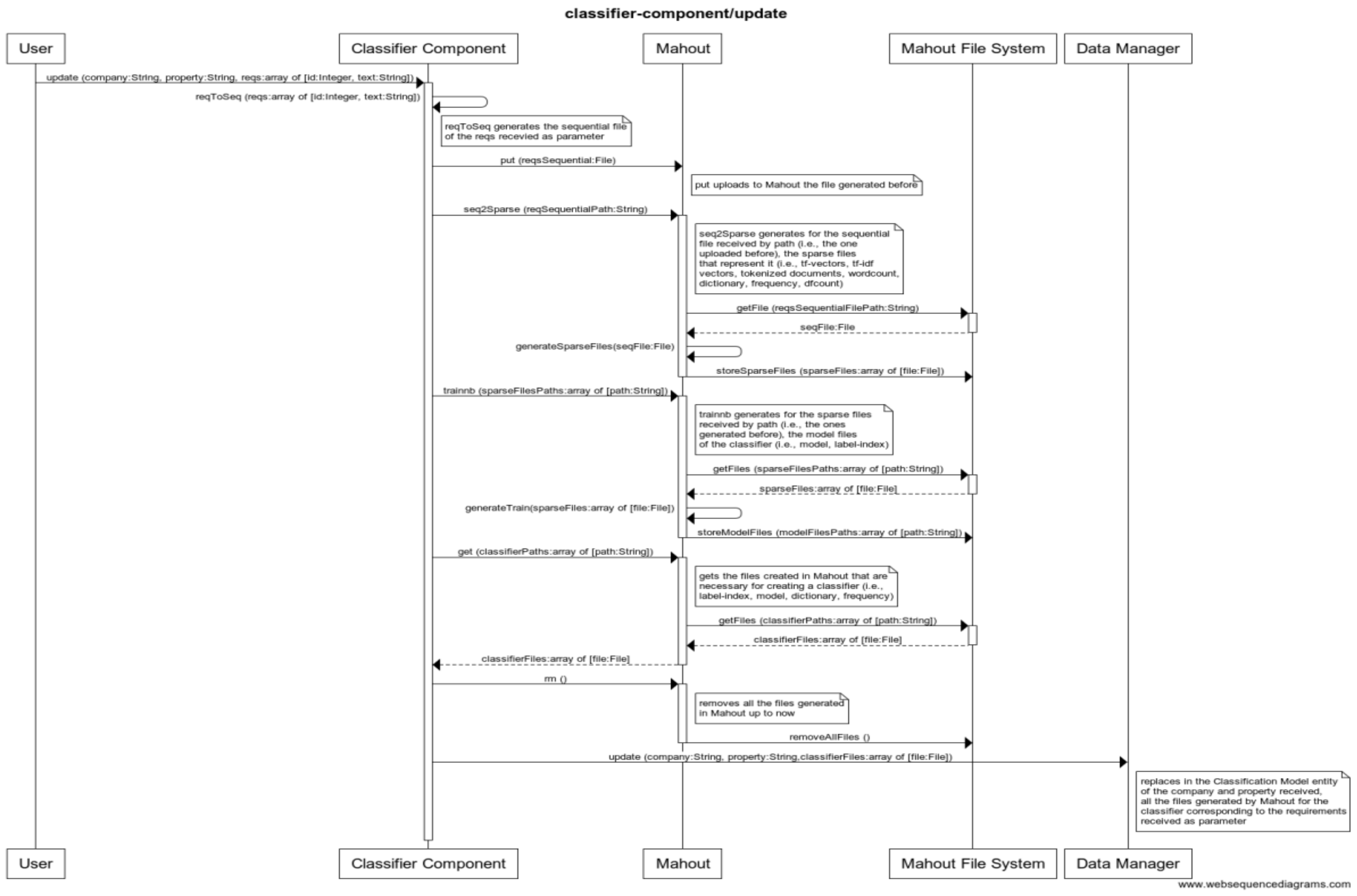
2.2.5 classifier-component/update

i. Description

The *update* microservice updates an existing classifier, which is used to assign values to a requirement property. The information about the classifier is stored in a Classification Model entity in a database (using as primary keys the name of the company to which pertains the requirement used to train the classifier and the property it tries to predict). In this case, the property is *isReq* (see section 2.2.3 for its definition).

For doing so, the microservice creates, for the requirements contained in the array received as parameter, the files that represent a classifier that uses these requirements for training. Specifically, these files are: dictionary, frequency, label-index and model. Afterwards, it replaces in the database connected to the data layer the training files for the specific company and property (which are received as parameter). The files creation is done using Mahout[1].

ii. Sequence diagram


classifier-component/update
User
Classifier Component
Mahout
Mahout File System
Data Manager
update (company:String, property:String, reqs:array of [id:Integer, text:String])
reqToSeq (reqs:array of [id:Integer, text:String])
reqToSeq generates the sequential file of the reqs recevied as parameter
put (reqsSequential:File)
put uploads to Mahout the file generated before
seq2Sparse (reqSequentialPath:String)
seq2Sparse generates for the sequential file received by path (i.e., the one uploaded before), the sparse files that represent it (i.e., tf-vectors, tf-idf vectors, tokenized documents, wordcount, dictionary, frequency, dfcount)
getFile (reqsSequentialFilePath:String)
seqFile:File
generateSparseFiles(seqFile:File)
storeSparseFiles (sparseFiles:array of [file:File])
trainnb (sparseFilesPaths:array of [path:String])
trainnb generates for the sparse files received by path (i.e., the ones generated before), the model files of the classifier (i.e., model, label-index)
getFiles (sparseFilesPaths:array of [path:String])
sparseFiles:array of [file:File]
generateTrain(sparseFiles:array of [file:File])
storeModelFiles (modelFilesPaths:array of [path:String])
get (classifierPaths:array of [path:String])
gets the files created in Mahout that are necessary for creating a classifier (i.e., label-index, model, dictionary, frequency)
getFiles (classifierPaths:array of [path:String])
classifierFiles:array of [file:File]
classifierFiles:array of [file:File]
rm ()
removes all the files generated in Mahout up to now
removeAllFiles ()
update (company:String, property:String,classifierFiles:array of [file:File])
replaces in the Classification Model entity of the company and property received, all the files generated by Mahout for the classifier corresponding to the requirements received as parameter
User
Classifier Component
Mahout
Mahout File System
Data Manager
www.websequencediagrams.com

iii. Example usage

URL	/upc/classifier-component/update
Method	POST
URL params	None
Data params	{ "requirements" :[{ "id": "4523", "property_value": "Prose", "text" : "Reports and correspondence" }, { "id": "1039", "property_value": "DEF", "text" : "Sufficient number of parking spaces must be provided close to the office." }], "company": "Siemens", "property": “isReq” }
Returns	None

2.2.6 *classifier-component/train&test*

i. Description

The *train&test* microservice returns the result of the k-cross-validation[4] (*k* received as parameter) using as set of requirements the ones received as parameter.

The microservice splits the set of requirements received as parameter in *k* groups. Then, for each group, trains a Naive Bayes classifier with all the requirements received as parameters except the ones in the group, and next, it tests the classifier with the requirements that are in the group. This test returns several statistical measures. After all the test for the *k* groups have been executed, the microservice returns the average of all the statistical measures. Specifically, these measures are:

- *Accuracy*. Statistic that measures the rate of correctly classified instances (i.e., true positives and true negatives) with respect to the number of classified instances.

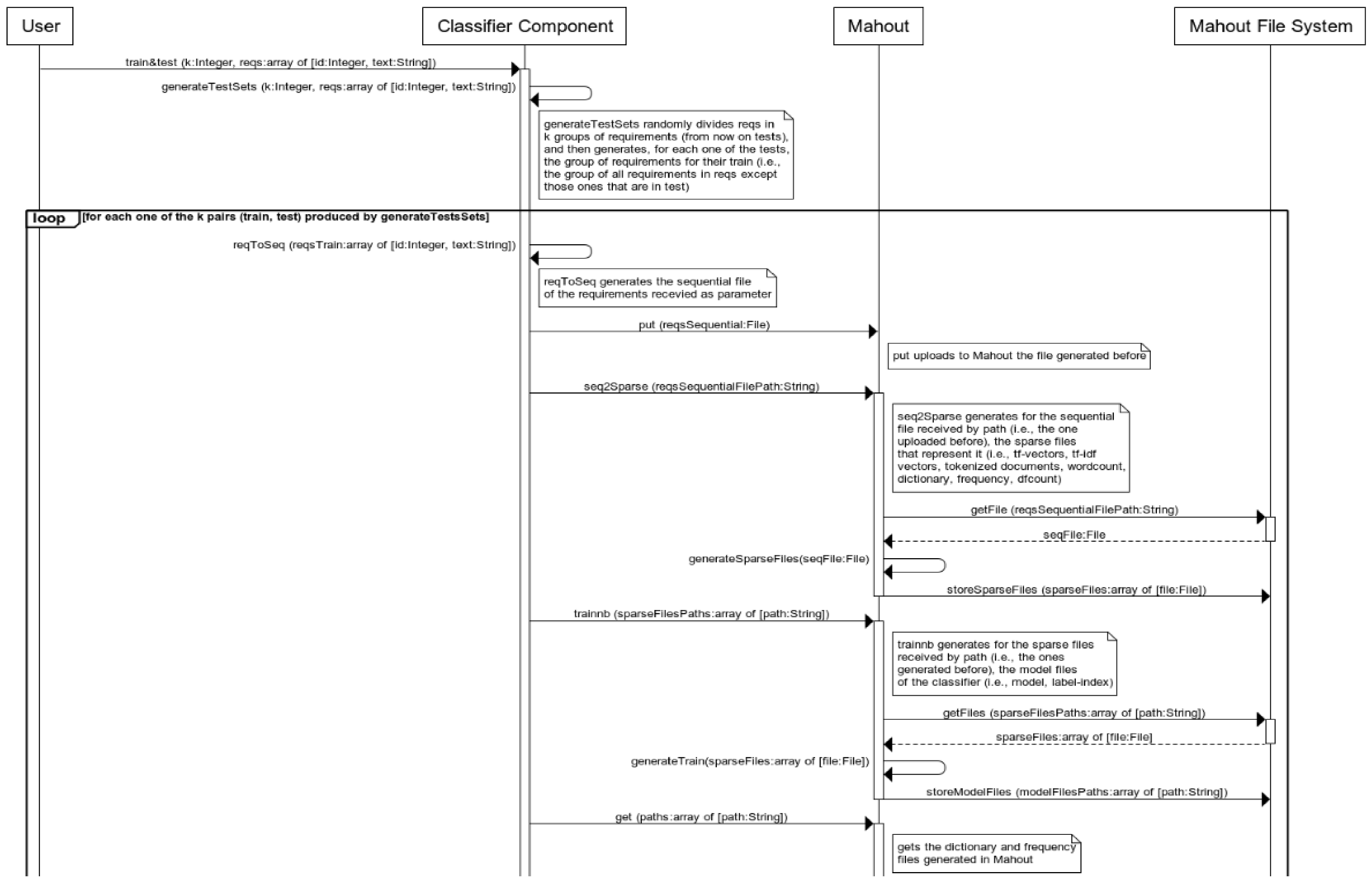
[4] *Cross-validation* is a technique to evaluate predictive models by partitioning the original sample into a training set to train the model, and a test set to evaluate it. In *k-cross-validation*, the original sample is randomly partitioned into *k* equal size subsamples.

- *Weighted precision.* The precision of a label is the rate of true positives with respect to the sum of true positives and false positives. The weighted precision measures the precision of all the labels taking into account their appearance (e.g., if 60% of the instances have label A and 40% have label B, the weighted precision is 0.4*precision_A + 0.6*precision_B).
- *Weighted recall.* The recall of a label is the rate of true positives with respect to the sum of true positives and false negatives. The weighted recall is calculated following the same principle explained in the weighted precision.
- *Weighted f1 score.* The f1 score of a label is the is the harmonic mean of precision and recall. The weighted f1 score is calculated following the same principle explained in the weighted precision.
- *Kappa.* Statistic that measures the relationship between "accuracy" v.s. "random classification".
- *Reliability.* Statistics that measures the overall consistency, i.e., having a high reliability means that the produced results are similar under consistent conditions.
- *Reliability (std deviation).* Standard deviation of the reliability across the k tests.

The classifier corresponds to a Naive Bayes classifier. The classifier creation and test is done using Mahout[1].

ii. <u>Sequence diagram</u>

classifier-component/train&test


User
Classifier Component
Mahout
Mahout File System
train&test (k:Integer, reqs:array of [id:Integer, text:String])
generateTestSets (k:Integer, reqs:array of [id:Integer, text:String])
loop
[for each one of the k pairs (train, test) produced by generateTestsSets]
reqToSeq (reqsTrain:array of [id:Integer, text:String])
put (reqsSequential:File)
seq2Sparse (reqsSequentialFilePath:String)
getFile (reqsSequentialFilePath:String)
seqFile:File
generateSparseFiles(seqFile:File)
storeSparseFiles (sparseFiles:array of [file:File])
trainnb (sparseFilesPaths:array of [path:String])
getFiles (sparseFilesPaths:array of [path:String])
sparseFiles:array of [file:File]
generateTrain(sparseFiles:array of [file:File])
storeModelFiles (modelFilesPaths:array of [path:String])
get (paths:array of [path:String])

generateTestSets randomly divides reqs in k groups of requirements (from now on tests), and then generates, for each one of the tests, the group of requirements for their train (i.e., the group of all requirements in reqs except those ones that are in test)

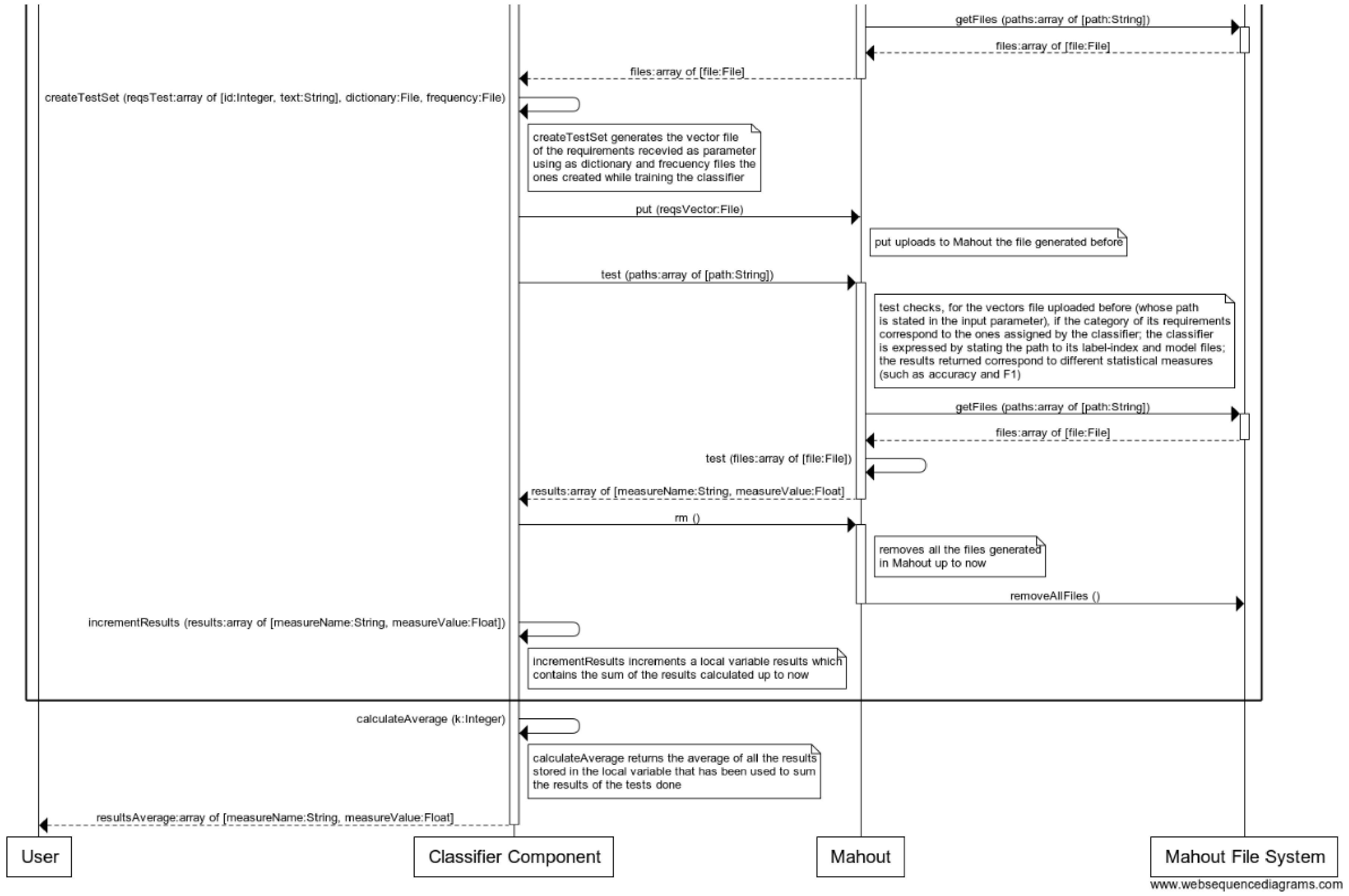
reqToSeq generates the sequential file of the requirements recevied as parameter

put uploads to Mahout the file generated before

seq2Sparse generates for the sequential file received by path (i.e., the one uploaded before), the sparse files that represent it (i.e., tf-vectors, tf-idf vectors, tokenized documents, wordcount, dictionary, frequency, dfcount)

trainnb generates for the sparse files received by path (i.e., the ones generated before), the model files of the classifier (i.e., model, label-index)

gets the dictionary and frequency files generated in Mahout


getFiles (paths:array of [path:String])
files:array of [file:File]
files:array of [file:File]
createTestSet (reqsTest:array of [id:Integer, text:String], dictionary:File, frequency:File)
createTestSet generates the vector file of the requirements recevied as parameter using as dictionary and frecuency files the ones created while training the classifier
put (reqsVector:File)
put uploads to Mahout the file generated before
test (paths:array of [path:String])
test checks, for the vectors file uploaded before (whose path is stated in the input parameter), if the category of its requirements correspond to the ones assigned by the classifier; the classifier is expressed by stating the path to its label-index and model files; the results returned correspond to different statistical measures (such as accuracy and F1)
getFiles (paths:array of [path:String])
files:array of [file:File]
test (files:array of [file:File])
results:array of [measureName:String, measureValue:Float]
rm ()
removes all the files generated in Mahout up to now
removeAllFiles ()
incrementResults (results:array of [measureName:String, measureValue:Float])
incrementResults increments a local variable results which contains the sum of the results calculated up to now
calculateAverage (k:Integer)
calculateAverage returns the average of all the results stored in the local variable that has been used to sum the results of the tests done
resultsAverage:array of [measureName:String, measureValue:Float]
User
Classifier Component
Mahout
Mahout File System
www.websequencediagrams.com

iii. Example usage

URL	/upc/classifier-component/train&test
Method	POST
URL params	None
Data params	{ “requirements" :[{ "id": "4523", "property_value": "Prose", "text" : "Reports and correspondence" }, { "id": "100147", "property_value": "DEF", "text" : " Contractor shall submit Progress Reports" }, { "id": "1004", "property_value": "Prose", "text" : "STATIC SWITCHING" }, { "id": "1039", "property_value": "DEF", "text" : "Sufficient number of parking spaces must be provided close to the office." }], "k": "2" }
Returns	{ "kappa": "0,3238", "accuracy": "78,4608", "reliability": "52,8879", "reliability_std_deviation": "0,4581", "weighted_precision": "0,907", "weighted_recall": "0,7846", "weighted_f1_score": "0,8243" }
Return explanation	Returns the statistic measures of the k-cross-validation done with the information received in the input (basically, the id of the text, the text, and the property value (i.e., label) assigned to the text.

2.3 Future work

Future work regarding the requirements recommendation service (see Section 2.2.1 and 2.2.2) will go beyond the limits of LSA and include the consideration of word2vec and doc2vec representations of words. That way, a more descriptive characteristic representation of the words can be used. This representation can be exploited in combination with clustering to improve the prediction performance of the generated model (see Lau et al. 2016).

Furthermore, the existing content-based recommender can be extended such that the aggregated/estimated utility of the requirements evaluated by different stakeholders is taken into account.

Future extensions of the requirements recommendation services will also be able to identify bug reports based on an entered comment of a user. This can be achieved by using name-entity recognition to look for similar bugs. In addition, it will also be able to identify individual requirements, i.e., it will be able to separate a text (e.g., a sentence containing more than one requirement) into individual requirements. The classification service could also take into account the context in which the requirement is (for instance, the placement of the requirement in a specific document and section).

Other extensions of the microservices will also present a list of prioritized recommendations to the users as well as allow the users (humans) to give feedback on the presented recommendations (i.e., retrain the models based on the human feedback).

Finally, at the implementation level, the microservices need to be adapted to the new standard input JSON defined inside OpenReq, which was delivered too close to M18 as to update the microservices.

3 RECOMMENDATION FOR IMPROVING REQUIREMENTS QUALITY

All of the "recommendation for improving requirements quality" APIs are designed to give quick suggestions on how to improve the quality of natural language requirements using different NLP techniques.

3.1 Architecture

The APIs for recommendation for improving requirements quality are nested under two microservices, accessible at "/hitec/prs-improving-requirements-quality" and "/upc/reqquality/check-conformance-to-templates". Each API has its own endpoint, listed below with the respective description. The APIs do not talk to each other, and they do not interact with any other microservices (OpenReq or otherwise). When a call is made to one of the prs-improving-requirements-quality endpoints, a Python script is run against the given requirements passed in the JSON body to the API, and the Python script returns a simple JSON object with the recommendations for improving requirements quality.

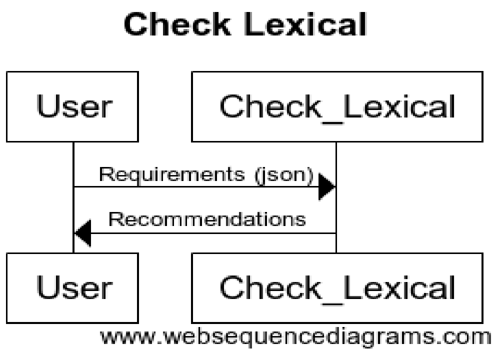
3.2 Design of current functionalities

3.2.1 prs-improving-requirements-quality/check-lexical

i. Description

The check-lexical microservice checks individual words and simple word combinations against a lexicon built from previous research (Tjong and Berry 2013), to test for ambiguities. These ambiguities include dangerous plurals, imprecise words, weak words, pronoun misuse, and more.

ii. Sequence diagram



iii. Example usage

URL	/hitec/prs-improving-requirements-quality/check-lexical
Method	POST
URL params	None
Data params	{ "requirements": [{

	"id": 1,
	"elements": [
	{
	"id": 1,
	"name": "description",
	"text": "This is actually a good requirement.",
	"created_at": 1526385600
	}
	],
	"status": "new",
	"created_at": 1526385600
	}
	]
	}

Returns	[
	{
	"1": {
	"1": [
	{
	"text": "good",
	"title": "Vague",
	"description": "These vague words and symbols are likely to introduce ambiguity.",
	"index_start": 19,
	"index_end": 23
	}
	]
	}
	}
	]

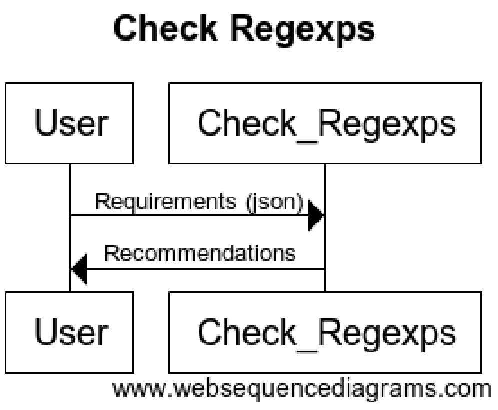
Return explanation	This endpoint returns a list of ambiguities for each element under each requirement. Each ambiguity contains the *text* that was identified as ambiguous, the *title* of the ambiguity, the *description* of the ambiguity, and the *index_start* and *index_end* that can be used to programmatically identify where the ambiguity exists within the original element (which is necessary since a single sentence may have multiple duplicate words, and not all of them may be ambiguous).

3.2.2 *prs-improving-requirements-quality/check-regexps*

i. Description

The check-regexps API checks word combinations and phrases using a lexicon of regular expressions built from previous research (Gleich et al. 2010), to test for ambiguities. These ambiguities include unclear inclusion, ambiguous plurals, and unclear associativity, among others.

ii. Sequence diagram



iii. Example usage

URL	/hitec/prs-improving-requirements-quality/check-regexps
Method	POST
URL params	None

Data params	
	```
	{
	  "requirements": [
	    {
	      "id": 1,
	      "elements": [
	        {
	          "id": 1,
	          "name": "description",
	          "text": "The system shall read HTML and PDF or DOC files.",
	          "created_at": 1526385600
	        }
	      ],
	      "status": "new",
	      "created_at": 1526385600
	    }
	  ]
	}
	```

Returns	
	```
	[
	  {
	    "1": {
	      "1": [
	        {
	          "text": "and PDF or",
	          "title": "Unclear Associativity",
	          "description": "The combination of “and” and “or” leads to unclear
	associativity.",
	          "index_start": 27,
	          "index_end": 37
	        }
	      ]
	```

	`    }`
	`  }`
	`]`

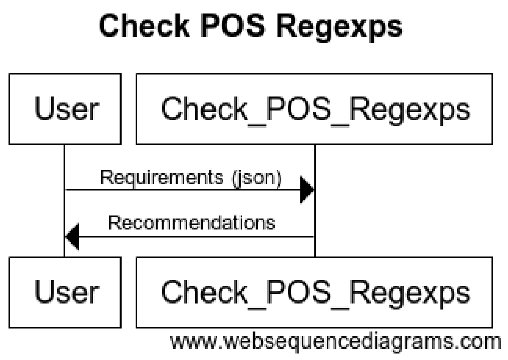
Return explanation	This endpoint returns a list of ambiguities for each element under each requirement. Each ambiguity contains the *text* that was identified as ambiguous, the *title* of the ambiguity, the *description* of the ambiguity, and the *index_start* and *index_end* that can be used to programmatically identify where the ambiguity exists within the original element (which is necessary since a single sentence may have multiple duplicate words, and not all of them may be ambiguous).

3.2.3 *prs-improving-requirements-quality/check-pos-regexps*

i. Description

The check-pos-regexps API checks word combinations and phrases using a lexicon of regular expressions designed to be run against natural language texts that have been converted to include their part-of-speech (POS) tags (Gleich et al. 2010). The combination of both POS tagging and regular expressions allows much more complicated ambiguities to be identified such as adjectives, adverbs, and passive ambiguity.

ii. Sequence diagram



iii. Example usage

URL	/hitec/prs-improving-requirements-quality/check-pos-regexps
Method	POST
URL params	None

Data params	`{`
	`  "requirements": [`
	`    {`
	`      "id": 1,`
	`      "elements": [`
	`        {`
	`          "id": 1,`
	`          "name": "description",`

	"text": "The system will be tested.", "created_at": 1526385600 }], "status": "new", "created_at": 1526385600 }] }
Returns	[{ "1": { "1": [{ "text": "be tested", "title": "Passive Ambiguity", "description": "Authors should state requirements in active form, as passive conceals who is responsible for the action.", "index_start": 25, "index_end": 16 }] } }]
Return explanation	This endpoint returns a list of ambiguities for each element under each requirement. Each ambiguity contains the *text* that was identified as ambiguous, the *title* of the ambiguity, the *description* of the ambiguity, and the *index_start* and *index_end* that can be used to programmatically identify where the ambiguity exists within the original element (which is necessary since a single sentence may have multiple duplicate words, and not all of them may be ambiguous).

3.3 Future work

The microservice check-conformance-to-templates remains to be developed. It will check that the text of a requirement follow one of the requirement templates defined inside the microservice. For doing so, the requirements will be first converted to include their part-of-speech (POS) tags (Gleich et al. 2010).

In addition, we will develop a microservice that will measure the quality of the requirements, in the sense that the microservice will turn a quality score between 0 and 1 (being 1 the representation of "perfect quality"). Implicit in what we have developed so far, quality can be measured by the amount of suggestions returned by the current microservices (for instance, if 100 out of 400 words in a requirement document are ambiguous, the document would be assigned a quality score of 0.75). In the future, we are looking to deliver a microservice that does this assignment explicitly, with a complete picture of all the quality metrics.

We are also conducting a systematic mapping study of requirement improvement algorithms, which will lead to additional algorithms that can be developed and integrated into this set of APIs. Additionally, with a complete picture of the state-of-the-art in requirement improvement algorithms, we are looking to make advances in this area to contribute to both OpenReq as an implementation, and the research community as published research. Therefore, some services might be enhanced afterwards. For instance, the check-conformance-to-templates service could be enlarged with new identified templates.

Additionally, efforts towards improving requirements quality are being implemented in the form of stakeholder engagement and topic extraction. These projects mark extensions to the original, intended functionality offered for recommendations for improving requirements quality, and will be implemented in the coming months.

4 PREDICTING REQUIREMENT PROPERTIES

The goal of all the "predicting requirements properties" microservices is to give recommendations on the possible value of different properties of requirements. For each one of the properties, different techniques will be tested to see which one provides better results.

Some of the properties are well-known in the requirements literature (e.g., risk and priority), while others are specific for the trials cases. Specifically, the properties that are particular for the trials are:

- Component (for the Qt trial), which is a part of a software project. In the case of Qt, there is only one software project, so it is the part of the software to which the requirement is related (e.g., Bluetooth, Wifi, and NFC).
- Environment (for the Qt trial), which is the operating system in which the requirement should work on (e.g., Windows, Linux or Symbian).
- Compliant (for the Siemens trial), which represents if the requirement can be fulfilled with an existing solution, if an existing solution needs modifications/adaptations to fulfil the requirement, or if a new technical solution is needed.
- Domain (for the Siemens trial), which represents the department that should evaluate[5] the requirement (similar to the maintenance teams used in (Di Luca et al., 2002).
- Technical solution (for the Siemens trial), which represents the existing solution that might fulfil the requirement.

4.1 Architecture

At the moment of handing out this deliverable, the architecture of this task only needs the modifications presented in Section 2.1; no extra modifications are needed with respect to the architecture presented in deliverable D3.1.

4.2 Design of current functionalities

The first approach we are using for the prediction of properties is based on a Naive-Bayes multi-valued classifier (i.e., the groups of the classifier are more than two). Therefore, we are using the same microservices of the *Classifier component* defined in subsections 2.2.3 to 2.2.6 to train, classify, update and train&test the classifier, respectively. The only difference is that for each property, the name of the property is passed to the train, classify and update services in the parameter *property*. For instance, in the case of the Priority property, the parameter property would receive the string *priority*.

4.3 Future work

This task finishes in M24, so this is just a first version of the functionalities. Up to now, the recommendation of each requirement property has been approached as a classification task with *n* classes. This task can also be seen as a similarity and relatedness problem, where the

[5] In the case of the Siemens trial, they work with bid projects. For the requirements engineering stage, these means that the requirements are already stated, and they have to evaluate if the requirements (already stated in the bid project) can be fulfilled by the company, how much effort would it take, and so on.

requirement at hand (i.e., the requirement for which we want to predict the property) is matched with similar or related requirements that have already been defined in past or on-going projects. Therefore, we aim to test other approaches that reduce this task to a similarity and relatedness detection case:

- one based on a hybrid recommender combining collaborative and content-based techniques as well as clustering learning techniques (possibly improved with similarity NLP techniques) (similar to the one used to identify similar and related requirements in Section 2.2.1 and 2.2.2), and
- one based on topic modelling (Chien 2016), which can be used to associate a label / tag (i.e., a topic) to a requirement or a subset of them, and then cluster the requirements in groups of related ones. The clustering can be done at different level of granularities (e.g., a hierarchy of topics), achieving different levels of relatedness.

5 DEPLOYMENT AND INTEGRATION

The requirements recommendation microservices (see Section 2) are currently in the final testing phase and will thereafter be deployed on the ENG infrastructure. Some of the microservices of the requirements recommender (the ones in Section 2.2.1 and 2.2.2) are already called by the official OpenReq prototype as well as the outcomes presented in the user-interface.

The prs-improving-requirements-quality microservices (see Sections 3.2.1, 3.2.2, and 3.2.3) are all deployed on the ENG infrastructure. They can be accessed adding http://openreq.esl.eng.it/ to the URL shown in the explanation of the microservices. The integration of the functionality offered by this microservice is in progress, currently on hold due to delays in the final stage of deployment. We expect the functionalities will be integrated soon into the OpenReq UI.

6 CHALLENGES AND SOLUTIONS

In this section we describe the challenges we encountered and the solutions selected during the development and testing phase.

6.1 Deployment difficulties

Challenge: Deployment on the ENG infrastructure was slower than expected.

Solution: Constant contact with ENG personnel to address the issues.

6.2 Lack of experience with Mahout

Challenge: During the development of the classifier component (Sections 2.2.3 to 2.2.6), several problems related to its implementation with Mahout appeared (some of them due to the lack of experience of the development team with Mahout). For instance, the train&test microservice was giving pretty bad results, due to the fact that the classifier was not using the proper files for dictionary and frequency. In addition, there was a bug caused by using the categories (i.e., property_values) randomly.

Solution: A lot of time spent looking at documentation; similar problems and solutions investigated in public forums.

6.3 Mahout integration

Challenge: Mahout usually is installed in a server. However, we wanted to integrate the Mahout component inside the classifier component. That way, no external installation of Mahout was needed in the deployment server (easing the deployment stage).

Solution: Similar problems and solutions investigated in public forums; integration of Mahout as much as possible inside the classifier component, creating a small installation guide for the remaining steps that could not be automatized.

7 Summary

In this deliverable we have presented the architecture, design and future work of the current functionalities of the version 1 of the stakeholders' recommender engine. Such version covers a part of Task 3.2 (Screening and recommendation of relevant requirements), T3.3 (Recommendation for improving requirements quality) and T3.4 (Predicting requirements properties).

For each one of the tasks covered for the version 1 of the stakeholders' recommender engine, we have described: 1) its architecture (i.e., whether modifications have been needed with respect to the architecture presented in deliverable D3.1); 2) its design (i.e., for each one of the microservices, we presented a description its sequence diagram(s), and an example of usage); and, 3) the future work. The only task that has needed changes in the architecture is T3.2, where small changes have been necessary in the data entity model to cover unforeseen needs of the task. The future work of these tasks mainly encompasses the implementation of some nice-to-have functionalities and also improvements on the already developed functionalities.

Afterwards, we summarized the current state of deployment and integration of the current functionalities. In a nutshell, three of the microservices have been deployed (those ones presented in Sections 3.2.1, 3.2.2, and 3.2.3), and two microservices have been integrated in the OpenReq UI (the ones presented in Section 2.2.1 and 2.2.2). Problems arose in the ENG deployment infrastructure that have slowed the deployment and integration of the services, but mitigation actions have been taken place which will speed up the deployment process.

Finally, we also briefly described the challenges faced, and the solutions applied to solve them. They are mainly related to deployment issues, to the use of Mahout in the classifier component (the one presented from Section 2.2.3 to 2.2.6), and to the difficulty to achieve further extensions of the current functionalities.

8 References

G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, Y. G. Guéhéneuc, *Is it a bug or an enhancement?: a text-based approach to classify change requests*, Proceedings of the IBM Centre for Advanced Studies Conference on Collaborative Research, 2008.

J. T. Chien, *Hierarchical Theme and Topic Modeling*, IEEE Transactions on Neural Networks and Learning Systems, 27, pp. 565–578, 2016.

G. A. Di Lucca, M. Di Penta, S. Gradara, *An Approach to Classify Software Maintenance Requests*, IEEE International Conference on Software Maintenance, pp. 93-102, 2002.

B. Gleich, O. Creighton, L. Kof, *Ambiguity detection: Towards a tool explaining ambiguity sources*, in International Working Conference on Requirements Engineering: Foundation for Software Quality, 2010.

J. H. Lau, T. Baldwin, *An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation*, CoRR, http://arxiv.org/abs/1607.05368, 2016.

A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, *Easy approach to requirements syntax (EARS)*, in Proc. 17th IEEE Int. Requirements Eng. Conf., 2009.

C. Palomares, X. Franch, D. Fucci, *Personal Recommendations in Requirements Engineering: The OpenReq Approach*, in 24th Int. Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'18), 2018

K. Pohl and C. Rupp, *Requirements Engineering Fundamentals*, 1st ed, Rocky Nook, Santa Barbara, CA 93103, 2011.

S.F. Tjong, D.M. Berry, *The design of SREE—a prototype potential ambiguity finder for requirements specifications and lessons learned*, in International Working Conference on Requirements Engineering: Foundation for Software Quality, 2013.