Text Mining on Mailing Lists: Tagging

Florian Haimerl
Advisor: Daniel Raumer, Heiko Niedermayer
Seminar Innovative Internet Technologies and Mobile Communications WS2017/2018
Chair of Network Architectures and Services
Departments of Informatics, Technical University of Munich
Email: flohai@mytum.de

ABSTRACT
Keywords are an important metric for categorizing or clustering documents. If the number of documents in a collection of documents exceeds a certain amount, it becomes unfeasible to manually assign these keywords. One of these is the collection of mails in the IETF mailing lists.
For collections of that size, the only feasible way to assign keywords to all documents, is to automatically extract them. In this paper, we deal with this challenge in two different approaches. We implement both approaches to automatically extract keywords from the mails in the IETF mailing lists and compare their performance.

Keywords
Tagging, keyword extraction, IETF, mailing lists, RAKE, TF-IDF

1. INTRODUCTION
In the effort to standardize the Internet, the Internet Engineering Task Force (IETF) organizes itself through mailing lists [1]. Consisting of over 2 million mails, the amount of data these mailing lists represent is huge. To be able to bring order into these mails and to further analyze them, finding keywords that best describe the contents of a mail would be helpful. Since the number of mails is too vast to do this manually, this paper analyses and compares approaches for automatic keyword extraction.
For this purpose, we start with analyzing the problem in section 2. We describe the IETF mailing lists and the database, we were working on. Then, in section 3, we describe different approaches for automatically extracting keywords from text. We focus on RAKE, a ready to use Python library [9] and on a more elaborate approach that utilizes the NLTK [6] to compute TF-IDF values [12]. Afterwards, we describe our implementation of these approaches in section 4. In section 5, we compare the implemented approaches and analyze how well they were suited for the task. We finish by mentioning related work in section 6 and we sum up this paper with a conclusion in section 7.

2. PROBLEM ANALYSIS
In this section, we we will analyze the status quo and give an overview of the data we are working on.

2.1 IETF Mailing Lists
The Internet Engineering task force is an organization, that is tasked with creating documents standards and best current practices for the Internet [1]. The IETF is mainly organized through open mailing lists, anyone can contribute to. Since there are over 1000 lists [16], both active and inactive that contain a total of more than 2.1 million mails, organizing them or extracting scientific data from them is a challenge.

Table 1: Frequency of Special Characters

column	type	constraint
file	text	
key	integer	not null
date	timestamp w timezone	
date_local	timestamp	
sender_addr	text	
receiver	text	
subject	text	
messageid	text	not null
inreply	text	
spam	boolean	default false
spamscore	numeric	
sender_name	text	
person	bigint	
fast_person	bigint	

2.2 The Database
To work with and analyze the data contained in the IETF mailing lists, the Chair of Network Architectures and Services at the Technical University of Munich [14] has a postgreSQL database that contains the mails and the mailing lists of the IETF. For this paper, the relevant part of that database is the *mails* table (table 1). It contains a *file* and a *key* column. With those, it is possible to uniquely identify the file that contains the mail. Furthermore, we only made use of the *spam* column, so we could ignore spam mails.

2.3 Goal
Our goal is to find keywords that best describe the contents of the mails. Therefore, we need to find a metric to assign a score to every word that occurs in a mail and then select

doi: 10.2313/NET-2018-03-1_05

the top scoring words.
We are not interested in key phrases, since single words appear to be more useful for further analyzing or processing the mails. This way, one can for example cluster the mails by these keywords.

3. APPROACHES

There are many different approaches to automatically extract useful keywords from text [2]. Since implementing and comparing all of them would go beyond the intended scope of this paper, we chose to go in depth for two promising approaches.

3.1 RAKE

RAKE [9] is a Python implementation of the Rapid Automatic Keyword Extraction algorithm, proposed by Rose, et al. [10]. The algorithm first identifies candidate keywords, before scoring those keywords based on their frequency in the document. This approach is completely corpus independent, which makes it efficient for a dynamic corpus like our database, that grows with every mail that is sent to one of the mailing lists.
Even though RAKE is also capable of identifying key phrases, as mentioned in section 2.3, we will focus on identifying single keywords. This is possible, since there are multiple parameters, that can be used to configure RAKE. This way, besides the maximum number of words in a key phrase, one can configure a stop word list, the minimum number of chars in a keyword and the minimum number of occurrences of the key phrase in in the document. Furthermore, there are some parameters that are only important for key phrases with more than one word.
We chose RAKE as one of our candidates because in contrast to our second approach, which we will explain in the next section, it is an out of the box approach, that is completely corpus independent.

3.2 TF-IDF with NLTK

Our second approach, is primarily based on the TF-IDF metric. Leskovec, et al. call it a "measure of word importance"[4]. It is the product of the document frequency tf and the inverse document frequency idf. Here, tf is the number of times, a word occurs in the document under inspection, whereas idf is the proportion between the number of documents in the corpus and how many of those documents contain said word. In short, idf measures the importance of a word in the corpus. An in depth explanation on the TF-IDF metric is provided by Leskovec, et al. [4].
With this approach, we are more flexible to make modifications to suit our goal. Thus, we can use the Natural Language Toolkit (NLTK) to improve the keywords, we receive with TF-IDF [13]. The NLTK "is a leading platform for building Python programs to work with human language data" [6]. This way, we can use a tokenizer and a lemmatizer, provided with the NLTK. The lemmatizer makes sure, words with the same stem can be treated as the same. For example, words like *write* and *writing* will be seen as equal and *writing* will be replaced by *write*. Also, using the NLTK offers the possibility to further narrow down the selected keywords to only use a certain type of words, like nouns, or to remove some type of words, like human names.
We chose this approach, since it is a good contrast to RAKE in some major points. While RAKE only works on a single document, the TF-IDF is computed on the whole document corpus. This way, the keywords promise to be more relevant in context of the corpus. Also, since this approach is less out of the box, it is possible to further refine it.

4. IMPLEMENTATION

We implemented the keyword extraction in Python, since the libraries we used were written in Python. Also, the scripts, the chair provided for working with the database, were also written in this language.
For us, the most important of these scripts are in the *mail* package. After getting the identifiers of the mails we want to analyze from the postgreSQL database, we get the actual text from the mail with the *mail.get_message* function. Furthermore, we can use *mail.strip_html* to remove HTML tags from HTML mails.
To store the keywords we extract from the mails, we introduce two new tables. The table *keyword* will contain each keyword the script finds, exactly once. Furthermore, the table *mails_to_keyword* (table 2) will contain relations between these keywords and the mails in the *mails* table. For each one of these relations, we also store the algorithm, that was used and the score the keyword reached. By storing the keywords this way, instead of storing a comma separated list in the *mails* table, processing the keywords will be easier. For example, clustering the mails by keywords can easily be done by grouping them with SQL.

After starting the Python command line tool, the user enters a short configuration dialog. Here, the user can make decisions like which approach (section 3) to use or if the script should be run in *debug* mode. In debug mode, the keywords will be printed to the command line instead of being written to the database.
For the actual keyword extraction, the implementation of the two approaches has some major differences, even though we made them fit to the same structure. First, an *init_analyzer* function initializes an analyzer object. Then a *handle_mail_cursor* function loops over the mails and extracts the keywords with the analyzer.
Next, we will describe how those functions work in the different approaches.

Table 2: mails_to_keyword

column	type
messageid	TEXT REFERENCES mails
keywordid	INTEGER REFERENCES keyword
score	INTEGER
algorithm	VARCHAR(6)

4.1 RAKE

Since RAKE is an out of the box solution, in this case the *init_analyzer* function is fairly straightforward.

```
return rake.Rake("Rake/SmartStoplist.txt",
                 3, 1, 2)
```

This creates a *Rake* object, that uses *Rake/SmartStoplist.txt* as a stopword list. As candidate words, it considers all words with at least three characters that occur at least twice in

doi: 10.2313/NET-2018-03-1_05

the document. The third argument can be used to define the number of words in a keyphrase. In our case, we set it to one, since we are only looking for keywords.

Thanks to the way RAKE works, *handle_mail_cursor* is similarly simple. We only have to call the *Rake* object's run method, handing over the text we want to extract the keywords from as the only argument.

```
tags = analyzer.run(text)
```

This returns a list of tuples, containing each word in the document and it's score. The higher this score is, the better the word is suited as a keyword.

4.2 TF-IDF with NLTK

This approach is more complicated. We use Python's machine learning library sklearn [11] to compute the TF-IDF values of the documents.
Therefore, in *init_analyzer* we have to train the analyzer on a dataset. We use a subset of the mails in the database (if possible, all mails) as the training set.

```
vectorizer=TfidfVectorizer(tokenizer=tokenize,
                           decode_error='ignore')
vectorizer.fit_transform(mails)
return vectorizer
```

We initialize the tfidfVectorizer with a custom tokenizer, so we can use the NLTKs stopword list and lemmatizer. Then, we call $vectorizer.fit\_transform()$ with our training set. This method computes *idf* values for the vocabulary in the training set.

Once the tfidfVectorizer is initialized, we can extract keywords from each mail:

```
tags = analyzer.transform([text])
feature_names = analyzer.get_feature_names()
for i in values.nonzero()[1]:
    return_dict[feature_names[i]]=values[0,i]
return return_dict
```

$analyzer.transform$ computes the TF-IDF values of all terms in the mail. $analyzer.get\_feature\_names$ returns a mapping from the terms' ids to the actual terms, since the matrix returned by $analyzer.transform$ only contains ids and not the actual terms. We then create a dictionary, that maps the TF-IDF values to the terms. Finally, the best suited keywords for each mail, are the ones with the highest TF-IDF values.

5. EVALUATION

For Evaluation, we consider two different metrics. Under *runtime*, we compare the time the approaches need for the different parts of the algorithm. Under *Results* we compare the keywords, the two algorithms extracted, by looking at false positives and false negatives.

5.1 Runtime

When analyzing the runtime of the two approaches, we have to differentiate between the *init_analyzer* and the *handle_mail_cursor* phases, we described in section 4. In this section, we will evaluate those two phases separately and then give a short combined verdict.

5.1.1 initAnalyzer

As we already described in section 4, the two approaches differ greatly in this part of the algorithm. While in case of RAKE the initialization phase only consists of setting a few configuration values, for TF-IDF we have to do a lot more. The initialization phase of TF-IDF is a training phase, where the analyzer has to analyze all mails in the training set. This factor has a huge effect on the runtimes of the initialization phase.
In table 3, we show the runtime for different sizes of training sets.

Table 3: runtime initAnalyzer in seconds

Algorithm	10 entries		100 entries		1000 entries	
	total	entry	total	entry	total	entry
RAKE	0.001	0	0.001	0	0.001	0
TF-IDF	2.834	0.283	8.250	0.082	33.096	0.033

For RAKE, the runtime of the initialization phase is negligible. It takes under one millisecond and does not depend on the size of the training set. This is exactly the result one would expect, since RAKE only does a few configurations in this phase.
Also as expected, TF-IDF is much slower in this phase, since it has to train its model and compute all the IDF values for the vocabulary of the training set. As one can see in table 3, the time the initialization phase takes for one entry decreases, the more mails we have in the training set. This shows, that even though the TF-IDF implementation takes a lot of time, especially compared to RAKE, it scales really well for a bigger training set.

5.1.2 handleMailCursor

In the main phase of the algorithm, the actual keyword extraction, we expect the timings to behave similarly in both algorithms. In table 4, we see that this is roughly the case.

Table 4: runtime handleMailCursor in seconds

Algorithm	10 entries		100 entries		1000 entries	
	total	entry	total	entry	total	entry
RAKE	0.065	0.007	0.876	0.009	6.194	0.006
TF-IDF	0.354	0.035	7.556	0.076	82.790	0.083

Both algorithms do not have big changes in the amount of time it takes to extract keywords from one mail. TF-IDF get's slightly slower, the more mails we process. We assume that this is due to the fact, that more mails lead to a bigger vocabulary. This seems to slightly slow down the process of calculating TF-IDF scores.
The general difference in performance between the two approaches is in a big part because of the tokenizer, that uses the NLTK. Here, lemmatizing the documents takes up a noticeable amount of time.

5.1.3 Verdict

It is pretty obvious, that RAKE is much faster than our TF-IDF implementation, due to it scoring words independently of other documents. But the TF-IDF approach also does not explode with a bigger data set. In this approach, we can win a lot of time, if we save the analyzer (for example with

doi: 10.2313/NET-2018-03-1_05

Python's *Pickle* module [8]) after it has been initialized, so we can reuse it in the next run.

5.2 Results

Both approaches are able to yield reasonably good results. In appendix A, we attached an arbitrarily selected example mail from the database. This mail is a more or less typical example for many of the mails we find in the database.

Table 5: 10 best scoring Keywords for appendix A

RAKE		TF-IDF	
sip	1.25	presence	0.476
refresh	1.0	sip	0.431
upload	1.0	upload	0.259
proposed	1.0	document	0.203
expiration	1.0	register	0.199
		method	0.161
		expiration	0.137
		generalization	0.137
		refresh	0.130
		use	0.127

As one can see in table 5, both approaches yield a useful selection of keywords. In the next sections, we will go a bit more into detail, by looking at false positives and false negatives the algorithms produce. We will also delve into a few further observations.

5.2.1 False Positives

In this case, RAKE yields really good results. The only keyword that subjectively seems like a false positive, is "proposed". On the other hand, TF-IDF produced a few keywords that seem like bad choices (i.e. "refresh" or "use") , but those do have low scores.

5.2.2 False Negatives

Here, RAKE fails to produce the top scoring keyword of the TF-IDF approach. This is most likely due to the fact, that RAKE only returns keywords that reach a certain threshold.

5.2.3 Further observations

We made a few observations with other documents, that are not visible in our example document. Since RAKE does not lemmatize the documents or use some other kind of natural language processing like the NLTK, we sometimes receive useless keywords like parts of URIs or email addresses.
Also, the lack of lemmatizing leads to RAKE sometimes returning the same keyword twice, with only slight differences like one being the plural of the other one.
One phenomenon we observe in both approaches is that often, human names are extracted as keywords. This is something that could be prevented in the TF-IDF approach, by optimizing the tokenizer.

5.3 Overall verdict

In general, we see RAKE yields a reasonably good ratio between false positives and false negatives and also outperforms the TF-IDF approach by far, when it comes to runtime. But, as we explained in section 5.2.3, RAKE also produces a lot of useless keywords. This is a big trade off, considering the fact that TF-IDF produces comparably good or even better results, especially when it comes to not missing any important keywords.
RAKE's big trade off could be fixed by modifying the RAKE implementation [9] and adding the option to use a custom tokenizer.

6. RELATED WORK

As we already mentioned in section 3, there are a lot of different approaches to extracting keywords from text. One famous approach is the textRank algorithm [5] by Rada Mihalcea and Paul Tarau. This is a graph based algorithm, that has similarities to Google's pageRank algorithm [7]. TextRank is also able to extract the most important sentences from a document, which can be used for automatic summary generation.
Another noteworthy approach is the KEA algorithm [15]. For this algorithm, a Java library is available. What makes this approach different to the ones we implemented, is that it requires a training set, that contains documents with manually assigned keywords.
Finally, we want to have a short look at another RAKE implementation by Sujit Pal [3]. He used the NLTK to streamline the standard RAKE implementation. For example, he used the NLTK tokenizer instead of RAKE tokenizing the text by itself.

7. CONCLUSION

In this paper, we did not introduce new approaches to keyword extraction. Instead, the aim was to find a good way of extracting keywords from the mails in the IETF mailing lists, by utilizing some of the existing approaches for keyword extraction. Therefore, we selected the two approaches we deemed most suitable for this task: The Rapid Automatic Keyword Extraction algorithm (RAKE) [10] and an approach, that computes TF-IDF scores [12] and optimizes this by employing the Natural Language Toolkit [6] for lemmatizing, tokenizing and stop word removal.
We described an implementation of these approaches and compared the results of both approaches based on those implementations. This comparison shows, that in the best case, RAKE would be the better choice, most of all due to its much shorter runtime. But, since in some cases RAKE produces unusable results, we reach the conclusion that in the current implementation, the TF-IDF approach is to be preferred.
In section 6, we mentioned an approach that modifies RAKE, by using the NLTK for tokenizing [3]. This approach can serve as an example for how to modify RAKE to better suit our needs.

8. REFERENCES

[1] H. Alvestrand. A mission statement for the ietf. BCP 95, RFC Editor, October 2004.

[2] Getting Started with Keyword Extraction. `http://textminingonline.com/getting-started-with-keyword-extraction`.

[3] Implementing the RAKE Algorithm with NLTK. `http://sujitpal.blogspot.de/2013/03/implementing-rake-algorithm-with-nltk.html`.

[4] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive datasets*. Cambridge university press, 2014.

doi: 10.2313/NET-2018-03-1_05

[5] R. Mihalcea and P. Tarau. Textrank: Bringing order into text. In *Proceedings of the 2004 conference on empirical methods in natural language processing*, 2004.

[6] Natural Language Toolkit. `http://www.nltk.org/`.

[7] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[8] pickle âĂŤ Python object serialization. `https://docs.python.org/3/library/pickle.html`.

[9] RAKE - A python implementation of the Rapid Automatic Keyword Extraction. `https://github.com/zelandiya/RAKE-tutorial`.

[10] S. Rose, D. Engel, N. Cramer, and W. Cowley. Automatic keyword extraction from individual documents. *Text Mining: Applications and Theory*, pages 1–20, 2010.

[11] scikit-learn: Machine Learning in Python. `http://scikit-learn.org/stable/`.

[12] Tf-idf: A Single-Page Tutorial. `http://www.tfidf.com/`.

[13] TF-IDF with scikit-learn. `http://www.bogotobogo.com/python/NLTK/tf_idf_with_scikit-learn_NLTK.php`.

[14] TUM Informatics - Chair of Network Architectures and Services. `https://www.net.in.tum.de/homepage/`.

[15] I. H. Witten, G. W. Paynter, E. Frank, C. Gutwin, and C. G. Nevill-Manning. Kea: Practical automatic keyphrase extraction. In *Proceedings of the fourth ACM conference on Digital libraries*, pages 254–255. ACM, 1999.

[16] www.ietf.org Mailing Lists. `https://www.ietf.org/mailman/listinfo/`.

doi: 10.2313/NET-2018-03-1_05

APPENDIX

A. EXAMPLE EMAIL

```
hello, my name is ****,
i sent this mail in the simple wg...

which is your opinion ?


> i think that this question: "upload presence document" is a
> critical issue.
>
> there isn't a standard procedure to upload and to modify the
> presence document, but it is the source of the presence status ...
> if the UA... if the PUA change its media capability, or
> change its status, it MUST refresh the presence document.
>
>
> the upload must be made through the protocol sip!!!
> and not push the presence doc via another protocol, such as HTTP POST.
>
>
> I agree with ******* when he says in the draft
> "Requirement for Pubblication of SIP related service data"
> : "... it is felt that the SIP REGISTER request is NOT the
> appropriate mechanisme for handing this loading od SIP
> service information..." but i'm not sure that a
> generalization of the REGISTER function, as proposed in the
> draft, is a good idea
>
>
> The use of the method REGISTER, for the upload,introduces one
> series of problems, just one example: the expiration of the
> Presence document and the expiration of the current
> communication addresses (i.e. Contact Addres) are different...
>
> I have instead a proposed other:
> the use of method NOTIFY...
> this method is already used for send the presence state to
> a particular subscriber...
> the its generalization for the upload or refresh of Presence
> Document is very simple and has the advantage of not
> modify/generalization a very important message for sip as REGISTER...
>
>

_____________________________________________
Sipping mailing list  http://www1.ietf.org/mailman/listinfo/sipping
This list is for NEW development of the application of SIP
Use sip-implementors@cs.columbia.edu for questions on current sip
Use sip@ietf.org for new developments of core SIP
```


doi: 10.2313/NET-2018-03-1_05