From textual to structured Web API Descriptions

Knowledge Discovery and Data Mining Seminar 2017

Selcuk Aklanoglu, Han Che, Caroline Dieterich, Marcel Ruoff, Jan-Peter Schmidt,
Karlsruhe Institute of Technology
Sebastian Bader
Department Applied Informatics and Formal Description Methods (AIFB),
Karlsruhe Institute of Technology

***Abstract*—In this study, we investigate the possibility of using Knowledge Discovery and Data Mining approaches to transform human written textual Web Application Programming Interface (API) descriptions into structural descriptions, which can be consumed by machines. The central vision is to provide a solution, which ultimately allows machines to interchange APIs with similar purposes automatically. A use case would be to replace non-responding data providing API (e.g. weather, location, news) with another. Using descriptions provided by ProgrammableWeb, with over 15000 APIs one of the most prominent public API library, we conducted multiple steps of data cleaning and preprocessing followed by machine learning algorithms (K-Means, DBSCAN) to create clusters, which we validated manually. With an average of only 68,3 words per descriptions (45,6 after preprocessing), we found that the amount of clusters are between 300 and 700 unique clusters. After social-tagging activities on the prototype-website[1] for about 300 APIs, we receive a quality score of 60 %, which means, that more than half of the APIs are probably clustered correctly. But our results suggests, that the needed quantity for text mining and clustering was insufficiently met. Our clusters were thus also inconclusive. The complete code is available on github[2].**

I. Introduction

APPLICATION Programming Interfaces (API) are not just a way to give developers access to software or services anymore. A whole business economy has established around the concept of exposing APIs in order to provide access and services. Programmable Web currently lists 15839 public available API (State June 2017). While Programmable Web represent as set of manually collected APIs, APIHound claims to have crawled and indexed over 50000 APIs. According to 3scale CEO Steve Willmott, eBay, Flickr, and Salesforce were among the companies that helped popularize the API [1] leading towards entire business models evolving around API services and strategies. Along came new terminologies to define this industry such as API Economy and Business to Developer (B2D).

API Economy is defined as the positive affect APIs can have on an organizations profitability [2]. It can be loosely compared to the prestige of patents plaques. The more holistic a company provides APIs the more developers can make use of it. Well documented APIs attract developers to either actively consume APIs or even build their own business on top of it.

[1]http://webapi.bplaced.net/
[2]https://github.com/Cuky88/APICrawler

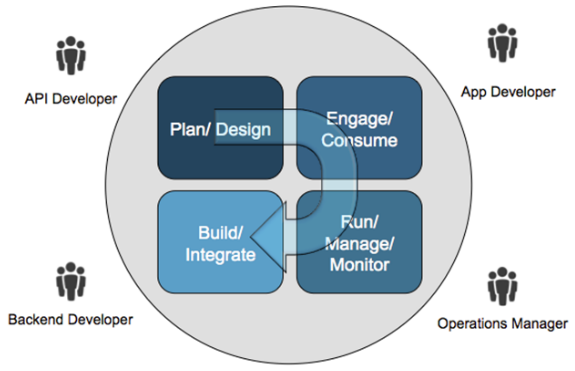

Fig. 1. MuleSoft on API Econom [2]

Business to Developer (B2D) (Fig. 1), implies a business model, which is targeting developers as customers. With the movement of open source and cloud technologies developers rarely have to rely on in-house resources anymore. Instead of building a new project form the ground up, developers can access numerous available 3rd party libraries to reduce developing time and effort. For certain services and libraries, they are even willing to pay a premium.

With the rise of API demand and supply, the API Economy calls for a product market similar ecosystem. While human customers have various sources to evaluate a product or service (or API), currently no solutions exist, which lets machines choose their own source of information or functionality. Most APIs are hardcoded into the application or software. The current digital market boasts of interconnectedness. Every product is in one way or the other connected to the internet and thus made smart, yet any or especially so called breaking changes to the API will require developers to update their code manually. For our research paper, we investigate on the basic requirements for machines to accomplish such breaking changes on their own. Our focus is on whether we can transform textual description to machine-usable structural descriptions, by using Programmable Web, one of the largest API aggregation open source available.

II. Datasets and Crawling

Prior to our research and development, we looked for suitable datasets with good quality. There was not much research done in this field, therefore we had to look for different alternatives, which will be described in the following sections. Section II.A describes the Linked Web API dataset and its issues. Section II.B deals with the dataset crawled from Programmable Web [1] and section II.C shows an alternative way to get data from Google.

A. Linked Web API Dataset

Starting point was the Linked Web API dataset [3] with 9.850 APIs. Besides short descriptions of the API, it inherits semantic annotations and links between different attributes like title, tags and links to websites. For our purpose only the descriptions were relevant. The first inspection showed, that the descriptions were too short. Since we want to cluster the APIs on similarity based on their descriptions, those were crucial. First solution was to crawl the website of the API to get more textual descriptions. This seemed to be easily done, since the dataset also had a link to the API documentation website. When checking the links, we found out that 78 % (Fig. 2) of the links were missing. From the remaining links only 9 % were working. We tried to fix the rest of the links by deleting path and subdomain from the link. This slightly increased the amount of usable links. Overall the data quality which is achieved with this approach can be considered insufficient. A lot of the links do not point to the actual API description either because they are outdated or because the fixing approach just led us to a general page with often little connection to the API description.

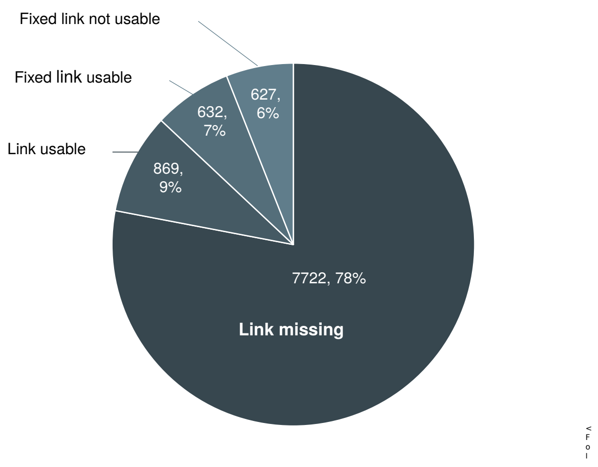

Fig. 2. Amount of working links in the Linked Web API Dataset

B. Programmable Web Dataset

Another approach to get usable data was to crawl Programmable Web with over 15.800 APIs. Therefore, we constructed a crawler with the Scrapy Framework [2] to crawl all the data. The API descriptions from Programmable Web have an averege length of about 70 words, which we assumed to be not enough. We examined the links to the API documentation sites, to see if we can crawl them, too. From the 15.800 APIs, about 32 % (Fig. 3) were not working. When we investigated the links deeper, we found out, that approximately 30 to 45 % of the links redirect to a subpages like main, authentication or payment pages. This was the same issue like with the Linked Web API dataset.

[1]https://www.programmableweb.com
[2]https://scrapy.org

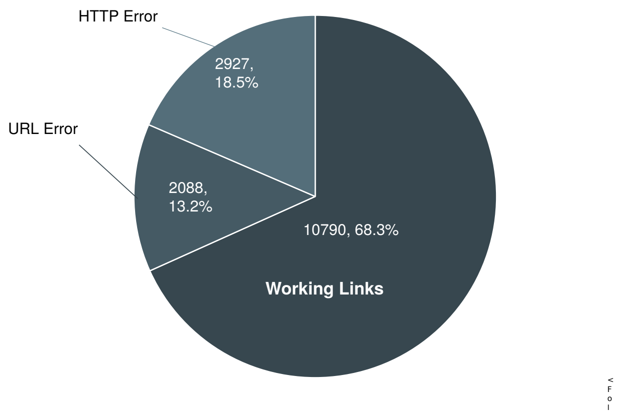

Fig. 3. Amount of working links in Programmable Web

C. Google Crawled Dataset

Since the links from the datasets in section II.A and II.B were not usefull, we decided to crawl a complete new dataset from Google. We constructed a search query out of the API names with additional filters, like:

> site:paypal.com ”api documentation” OR ”api reference” ”paypal payments api”

We saved the first five results with every result being in its own key-value pair, so that we can distinguish between the ranks of the results. Our assumption was, that Google will return the best result (link to the actual API documentation website) on rank 1. We investigated a couple of those links and came to the conclusion, that our assumption was false. Therefore we could not use the dataset from Google. We decided to use the Programmable Web dataset with the limited amount of API descriptions.

III. Preprocessing

The preprocessing was done in three steps. First the data was cleaned, afterwards the words of API descriptions were transformed into a matrix and finally the dimensions of the matrix were reduced.

A. Data Cleaning

The data cleaning included punctuation removal, number removal, transformation to lower case, stop word removal and lemmetization. The cleaning was done with Python, for lemmatization the lemmatizer from the Natural Language Toolkit (nltk) library was used [4]. The cleaned API descriptions were saved in JSON format.

B. Vectorization

After the data cleaning, we are left with API descriptions, referenced as documents, which consist of a varying number of words, referenced as terms. Before clustering algorithms can be applied on the data it has to be transformed. The goal here is to generate a matrix where every row represents a document and each column a term. The values of that matrix determine if a term is part of a description and which importance the term has. To archive this, we used the term frequency times inverse document frequency (TF-IDF) Vectorizer from the Python library scikitLearn [5]. The Vectorizer counts the frequency of a word in an API description and the frequency of that word in the whole data set. The importance of a word then is calculated by (1).

$$tfidf(d,t) = tf(t) * idf(d,t) \qquad (1)$$

tf(t) is the term frequency of a term in an API description. This term frequency is multiplied with the inverse document frequency which is calculated according to (2).

$$idf(d,t) = log\left(\frac{1+n}{1+df(d,t)}\right) \qquad (2)$$

n is the total number of API descriptions and $df(d,t)$ the document frequency, the number of documents in which a term t appears. With the TF-IDF Vectorizer, terms with a high term frequency but a low document frequency, have the highest tf-idf-value and can be considered more important than terms with a lower tf-idf-value. The result of the vectorization is a high dimensional sparse matrix.

C. Latent Semantic Analysis

A dimension reduction was not only necessary because of the size of the matrix, but is also suggested by [6]. They showed, that a dimension reduction with the Principal Component Analysis (PCA) before the use of K-Means leads to better clustering results. Having a huge sparse matrix makes it difficult to perform a PCA. Therefore, we used the transformer TruncatedSVD from the scikitLearn library to perform a dimension reduction [5]. The transformer uses a truncated Singular Value Decomposition (SVD) which is specially tailored to TF-IDF matrixes and therefore called ”Latent Semantic Analysis”. This makes a centering of the data unnecessary and is for that reason suitable for large sparse datasets [7].

IV. Manual Clustering

To be able to assess the quality of our trained clusters, we decided to manually create clusters and only evaluate (section V) on those manually clustered APIs. With that, we could test different clustering parameters and choose the best parameter combinations. We split the dataset according to the existing categories and took the five tags: Messaging, Payments, eCommerce, Mapping and Finance, since they contained most of the APIs. Based on the descriptions we clustered those APIs manually, so that we receive about 30 APIs per cluster. After 2-3 days of processing, we had 26 manually created clusters with 962 APIs.

V. Clustering

To suggest for a given API from a user similar APIs we tried two different approaches. In our first approach we used clustering algorithms which are mostly used for data mining and knowledge discovery, data compression and pattern recognition. Mostly the field of clustering algorithms for pattern recognition were capable to solve our given problem. In our case the algorithms assigned APIs to the same cluster if they had similar wording patterns and descriptions. Our other approach was to create a correlation matrix which declares for each API how similar it is to a given API. To achieve this the algorithm had to calculate the distance between each API and create a square matrix for the given data. To implement and test these approaches we used the algorithms described in the following subsections.

A. Normalized Google Distance

For the correlation matrix we tried to implement the algorithm introduced by J. Wu, L. Chen, Z. Zheng, M. R. Lyu and Z. Wu in their Paper Clustering Web services to facilitate service discovery [8]. The algorithm determines the similarity of each APIs word-vector with all the other word vectors by comparing the word in the word-vector with the normalized google distance. As a result, the algorithm produces a correlation-matrix where for any given API the distance to each other API can be withdrawn (Algorithm 1).

Algorithm 1 Normalized Google Distance

```
Input: word-vectors of the APIs
Output: Correlation matrix
   Procedure :
1: for each API $s_1$ do
2:    for each API $s_2$ do
3:       $Sim_{con}(s_1, s_2) = \frac{\sum_{\epsilon\ con_{s1}} \sum_{w_j \epsilon\ con_{s2}} 1 - NGD(w_i,\ w_j)}{|con_{s1}| \times |con_{s2}|}$
4:       Add $Sim_{con}\,(s_1, s_2)$ to the correlation matrix
5:    end for
6: end for
7: return correlation matrix
```

The main advantage is that the authors already tried to solve our use-case with their algorithm and demonstrated that it could work. They also used the same pre-processing techniques to determine the word-vectors. This means we had the same initial situation. To determine the normalized google distance, which is crucial for the determination of our results, we had to crawl google for each word combination in our word-vectors. Because we lacked the time we tried to use google ngram. Google ngram is a database with scanned books. It also provides command lines to build correlation-matrixes for given words, which would make it possible to implement the normalized google distance. Unfortunately, the current version is from 2012 which resulted in an insuperable problem. Words like instagram which were used in our word vectors and were crucial for interpretable results were not included in google ngrams. Because of this we did not use the implementation of this paper in our further project.

B. Density-Based Clustering

The first clustering algorithm we tried out was the density-based spatial clustering of applications with noise (DBSCAN). DBSCAN [9] is a density based clustering algorithm grouping together closely packed data-points with a given number of neighbours required for a point to be dense ($minPts$) and a maximum distance between two neighbours in a cluster (ϵ) (Algorithm 2).

Algorithm 2 Density-Based Clustering

```
Input: given number of neighbours required for a point to be
    dense,
    maximum distance between two neighbours in a cluster,
    word-vectors of the APIs
Output: Clusters with associated APIs
    Procedure :
for each unvisited API S in dataset do
    mark S as visited
    N = getNeighbors(S)
    if sizeOf(N) < minPts then
        mark S as NOISE
    else
        C = next Cluster
        expandCluster(S, N, C)
    end if
end for

expandClusters(S, N, eps, minPts) :
add P to cluster C
for each point in S' do
    if S' is not visited then
        mark S' as visited
        N' = getNeighbors(S')
        if sizeOf(N') ⩾ minPts then
            N = N joined with N'
        end if
    else if P' is not yet menmer of any cluster then
        add S' to cluster C
    end if
end for
```

The main advantage of DBSCAN is that APIs without any similar APIs in a given distance, also called noise, will be ignored and assigned to a separate cluster. This results in an undistorted result and as an information for an user that there is no similar API. Another advantage is that there is no fixed number of clusters because this will be optimized during the execution of the algorithm. The decisive disadvantage is that only the distance between two neighbours in a cluster are considered in the algorithm. Resulting in an unknown width of the clusters. This means that two completely different APIs could end up in the same cluster because there is a linking chain of APIs between them. With this problem there was no assurance that the algorithm produces a logical result for our use-case. Another problem occurred during the execution of our implementation. Given a set of 3000 APIs, number of neighbours required for a API to be consider dense of $minPts = 2$ and a $= 1,2$ our algorithm considered half of these APIs noise and the other half was put into clusters with an average number of seven APIs. The input can be interpreted that there have to be one other API in a distance of 1,2 around a given API. This means our input had to be very vague for any results. Because of this and the wrong assignment of APIs to the noise category we did not use DBSCAN for further implementation.

C. K-Means Clustering

The clustering algorithm we used for our final implementation is K-Means, one of the most widely used clustering algorithm. For a given set of data-points in a n-dimensional space R^n and an integer k the algorithm determines k points, called centres, minimizing the sum of the mean square distance from the data points to their nearest centre. We are using the K-Means adaptation of the Lloyed-algorithm described in Algorithm 3 [10].

Algorithm 3 K-Means Algorithm

```
Input: Integer k,
    word-vectors of the APIs $W = w_1, w_1, ...w_n$
Output: k centres, affiliation of the APIs
    Procedure :
Arbitrarily choose an initial k centres $C = c_1, c_2, ..., c_k$.
Repeat
    Assign each item $w_i$ to the cluster which has
    the closest cetroid;
    Calculate new mean for each cluster;
Until convergence criteria is met.
return centres, affiliation of the APIs
```

The main advantage is the simplicity of the algorithm and performance. Another advantage is that because of the minimization of the mean square distance of the APIs to their centres we also minimize the distance between two APIs in the same cluster. The main challenge of the K-Means algorithm is that the integer k is fixed and cant be optimized with the algorithm itself. To counteract this problem we split the possible number of clusters into intervals. For each interval we executed the K-Means algorithm for the boarder points of the interval and compared the result of these outputs in our evaluation. A problem that we couldnt overcome was the inclusion of noise in our results. APIs without any similar APIs were still included in a cluster and this could distort our results. Since we used K-Means for our final implementation the results of this algorithm will be evaluated in the Evaluation in section VI.

VI. Evaluation

K-Means in combination with LSA leaves us with three parameters that have to be determined, the cluster size, the distance metric, and the number of dimensions the input data was reduced to. To determine the best values for those parameters we run the clustering with different parameter combinations. The clusters resulting from the different combinations were then compared to the manually formed clusters. We used two different methods to determine the quality of the machine created clusters, the Jaccard-Coefficient together with the Rand Statistic and Precision and Recall. As a last step we selected one of the best parameter combinations to manually evaluate the correctness of the clustering. The following paragraph introduces the two different quality measurements and the process of the manual evaluation.

A. Jaccard Coefficient

As [11] suggests we used Jaccard-Coefficient for quality measurement. Jaccard measures the intersection of two quantities. In our case one quantity are the manual created clusters, lets name it C, the other one are the cluster sets that K-Means provides, lets name it K. Formally described in equation (3) and (4):

$$manual\ Cluster\ C := \{c_1, c_2, ..., c_n\} \tag{3}$$

$$k - means\ Cluster\ K := \{k_1, k_2, ..., k_n\} \tag{4}$$

We used the approach for calculating Jaccard as described in [11]. Therefore, we build every possible pair Y in C (5).

$$Y = (y_1, y_2) : y_1, y_2 \in C \tag{5}$$

Furthermore, we search for Y in K (6).

$$\Rightarrow\ try\ to\ find: \\ X = (x_1, x_2) : x_1 = y_1, x_2 = y_2 \in K \tag{6}$$

Now we can derive for every possible X and Y whether y_1 and y_2 are in the same cluster and if x_1 and x_2 are in the same cluster. We can get the following combinations:

Same-Same (SS)	$i.e.\ y_1, y_2 \in c_l; x_1, x_2 \in k_h$
Same-Different (SD)	$i.e.\ y_1, y_2 \in c_l; x_1 \in k_h; x_2 \notin k_h$
Different-Same (DS)	$i.e.\ y_1 \in c_l; y_2 \notin c_l; x_1, x_2 \in k_h$
Different-Diffr. (DD)	$i.e.\ y_1 \in c_l; y_2 \notin c_l; x_1 \in k_h; x_2 \notin k_h$

We count how many SS, SD, DS and DD pairs exists. With these numbers, we can calculate the Jaccard-Coefficient (7). We also provide Rand Statistics (8), which can be easily calculated from the same numbers.

$$Jaccard\ Coefficient:\ j := \frac{|SS|}{|SS| + |DS| + |SD|} \tag{7}$$

$$Rand\ statistics:\ rs := \frac{|SS| + |DD|}{|SS| + |DS| + |SD| + |DD|} \tag{8}$$

A bigger Jaccard-Coefficient indicates a higher number of APIs which are in the right cluster with each other and is therefore better.

But this approach quickly showed some problems. Due to a high number of APIs in our dataset we have a huge number of operations for calculating the Jaccard-Coefficient. It showed that evaluation took much more time than creating the clusters with K-Means. The following paragraph shows what the problem was and how we solved it. Because of the pair-pair comparison we had to run very often through our API list. According to [11] we have to build M pairs over our API list (9).

$$M = \frac{N * (N-1)}{2} \approx \frac{10,000 * 9,999}{2} = 49,995,000 \tag{9}$$

These nearly 50 million pairs have to be checked against the pairs build from our manual clustering list which contained nearly 1000 APIs. That means there are nearly half a million pairs. Finally, we have to do a bit less than $50 * 10^6 * 0,5 * 10^6 = 25 * 10^{12}$ comparisons. This took much more time than building our clusters with K-Means and was not useable due to lack of time for our work. We tried to fix that problem with two different ideas. One is described in the next section and is about using other parameters that are not calculated by comparing pairs which decreased complexity. The other idea is easier and also fixed our problem. We were able to decrease the bigger pairs list dramatically by only building pairs that are also in our manual clustered list. Because if they are not clustered manually they cannot be used for evaluation and thus, are not necessary to be build. This is achieved by a reduction of the API list at start time of the comparison algorithm. This decreased runtime from hours to a few minutes which is quite acceptable.

B. Precision and Recall

The other idea of fixing the runtime problem was using precision and recall as coefficient for similarity. These two are also often used for cluster similarity measurement. Basically, they use the same concept with classifying if an API is true positive, false positive, true negative or false negative sorted into a cluster according to a certain set of clusters that is assumed to be correct. The advantage is that we do not need to build pairs for calculation and can decrease runtime by that [12].

Let assume we have two sets of APIs:

$$manual\ Cluster\ C := \{c_1, c_2, ..., c_n\}$$

$$k - means\ Cluster\ K := \{k_1, k_2, ..., k_n\}$$

The algorithm initially breaks down the list of APIs to the amount of manually clustered API list to increase performance. Afterwards it takes one API from the manual clustered list and identifies its cluster. If that cluster was not evaluated already, we are looking for all other APIs that are contained in this cluster in the manual clustered list. Having a selected cluster (c_i) with all its APIs we are looking for the most similar cluster in the machine cluster set. Therefore, we are iterating through K and define a new list of clusters named T. If we find an API that is in c_i we store the cluster id from the machine clustered list in T and count how many APIs that are contained in c_i are there in. If we have done this for all APIs is in K we can derive from T which cluster in K is most similar to c_i (the selected cluster from C, our manual clustered list). This most similar cluster is called t_i. From that we can easily calculate recall and precision between these two clusters by checking the four following possibilities:

True positive (tp)	$x_n \in c_i, x_n \in t_i$
False positive (fp)	$x_n \notin c_i, x_n \in t_i$
True negative (tn)	$x_n \notin c_i, x_n \notin t_i$
False negative (fn)	$x_n \in c_i, x_n \notin t_i$

We are counting how these four case occurred and can calculate from that precision (10) and recall (11).

$$Precision = \frac{|tp|}{|tp| + |fp|} \tag{10}$$

$$Recall = \frac{|tp|}{|tp| + |fn|} \tag{11}$$

With that we are able to calculate precision and recall for cluster t_i. To determine the quality of K we calculate precision and recall for each cluster and print the average for each at the end of calculation. The clustering with the overall highest precision and an overall highest recall can be considered the best clustering. For an easier comparison the F-measure (12) is calculated.

$$F = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{12}$$

The F-measure is the harmonic mean between precision and recall. The higher the value the better the quality of the clusters.

C. *Manual Evaluation*

We created a simple website[3] to do social-tagging. It enables an easy manual evaluation and access to anyone to use our results. The data the site is based on is a JSON file. It contains the result of the K-Means clustering with a parameter combination that had one of the best quality measures.

The user can enter a name of an API to find similar APIs. The site provides the data of our machine clustered data. It looks up similar APIs from the JSON file and provides them with a small description. The user has then the possibility to give a positive vote if a suggested API matches the one he put in or give a negative vote if not. With the user input an overall quality measure can be calculated as the percentage of correctly clustered APIs (13).

$$Quality = \frac{positive\ votes}{all\ votes} \tag{13}$$

VII. Results

We run the evaluation with all possible combinations of the predefined values for the parameters. For the number of

[3]http://webapi.bplaced.net

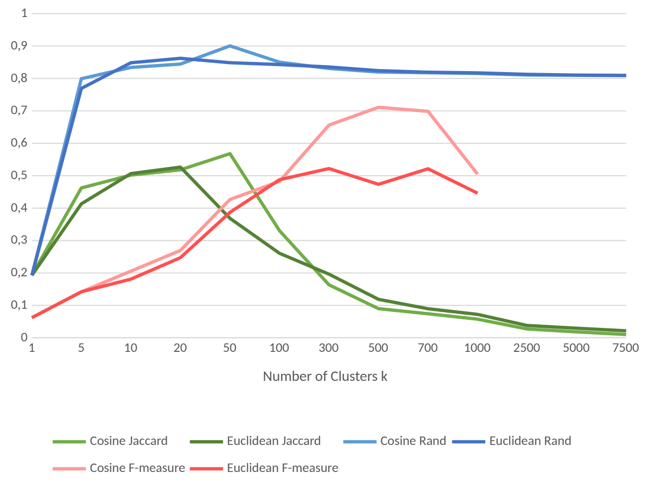

Fig. 4. Difference between Jaccard coefficient and F-messure

clusters k, 13 values between $k = 1$ to $k = 7500$ were chosen. With a k of one, all APIs are in one cluster and with a k of 7500 the average size of a cluster is 2,4. We used two different metrics for distance measurements, the Euclidian distance and the Cosine distance and did a dimension reduction to $dim = 10$, $dim = 100$, $dim = 250$ and $dim = 500$ dimensions. This leaves us with $13 * 2 * 4 = 104$ clustering cycles. For every cycle the quality measures were calculated.

The comparison of the quality measures revealed, that they suggest different optimal k, regardless of the dim and the distance metric. The Jaccard-Coefficient as well as the Rand Statistic have their maximum between $k = 20$ and $k = 50$ for all dim and distance metric combinations. Whereas the F-measure indicates an optimal number of clusters between $k = 300$ and $k = 700$. The low k suggested by the Jaccard-Coefficient would mean that we have an average cluster size of around 52,6, which is too high. The optimal k of the Jaccard-Coefficient is also very close to the number of manually created clusters. It is to investigate to what extend the Jaccard-Coefficient is influenced by the manual clustering. To avoid any bias which might be caused in this case we put the focus of the evaluation on the precision and recall measure and therefore on the F-measure (Fig. 4).

The F-measure shows a similar height for a dimension reduction to $dim = 100$, $dim = 250$ or $dim = 500$ dimensions. The F-measure of a clustering with a $dim = 10$ is overall lower than one of a clustering with $dim = 100$, $dim = 250$ or $dim = 500$. It is not possible to make a general assumption which distance metric produces better cluster quality. It changes depending on the parameter combination. The Euclidian metric gives a higher F-measure with $dim = 100$ and the Cosine metric with $dim = 250$ or $dim = 500$.

Of all the parameter combinations we tested, the parameter combinations $k = 500$, $dim = 500$, cosine and $k = 700$, $dim = 100$, euclidian had the best F-measures with 0,71 and 0,74. The results of the K-Means clustering with the parameters $k = 500$, $dim = 500$, cosine were then integrated into the website. 303 APIs were evaluated through the website.

The resulting quality is 59 %.

VIII. Conclusion

Starting with 3 different datasets, we came down to use only one dataset, due to data quality issues. Attempts to solve those issues where unsuccessful. We used therefore state-of-the-art text preprocessing techniques on a quite small text corpus and evaluated three clustering algorithms. At the end, we decided for K-Means clustering and a sort of grid-search to find the best parameter combinations. We used two metrics, the Jaccard-Coefficient and precision and recall, to assess the quality of the machine generated clusters and to compare them with our manually created clusters. At the end, we created a prototype-website, where we could apply social-tagging and assess the overall quality, with about 60 %.

This shows clearly, that our approach works and enables an useful new functionality. Nevertheless, there is a huge potential for improvements. First of all, more descriptions of an API need to be collected. We think, that it will be very difficult, even impossible to automate this step. Our efforts to do so, were unfruitful. Another point, which can be optimized, is the manual clustering. It would be very interesting to see, which effect more manual clustered APIs as well more social-tagged APIs would have on the quality metrics. We assume that it will influence the Jaccard-Coefficient highly as stated in section VII. And of course the search for the optimal parameter combination can be optimized. It is crucial, which interval is chosen for k. We chose a high interval and bigger steps between the k. Even running on a GPU, it took about an hour to run all the combinations in K-Means. To have better results, the steps need to be reduced, preferably to 1. But this would cause high computational costs. So, other approaches, for example like heuristic search could be evaluted.

We prepared a github-wiki[4] with every needed information and a guide, how to setup and run the code.

References

[1] "tmcnet," http://www.tmcnet.com/voip/features/articles/411284-apis-have-landed.htm, accessed: 2017-08-29.

[2] "Mulesoft," https://www.mulesoft.com/resources/api/what-is-an-api-economy, accessed: 2017-08-29.

[3] M. Dojchinovski and T. Vitvar, "Linked web apis dataset," *Semantic Web*, no. Preprint, pp. 1–11.

[4] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[6] C. Ding and X. He, "K-means clustering via principal component analysis," in *Proceedings of the twenty-first international conference on Machine learning.* ACM, 2004, p. 29.

[7] R. Albright, "Taming text with the svd," *SAS Institute Inc., Cary, NC*, 2004.

[8] J. Wu, L. Chen, Z. Zheng, M. R. Lyu, and Z. Wu, "Clustering web services to facilitate service discovery," *Knowledge and Information Systems*, vol. 38, no. 1, pp. 207–229, 2014.

[9] "Scikit learn," http://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html, accessed: 2017-08-30.

[10] K. A. Nazeer and M. Sebastian, "Improving the accuracy and efficiency of the k-means clustering algorithm," in *Proceedings of the World congress on Engineering*, vol. 1, 2009, pp. 1–3.

[11] M. Halkidi, Y. Batistakis, and M. Vazirgiannis, "Cluster validity methods: part i," *ACM Sigmod Record*, vol. 31, no. 2, pp. 40–45, 2002.

[12] D. M. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," 2011.

[4]https://github.com/Cuky88/APICrawler/wiki