# Document Embedding for Scientific Articles: A validation of word embeddings

H.J. Meijer[1,2][0000−1111−2222−3333] and R. Karimi[2][1111−2222−3333−4444]

[1] University of Amsterdam, Science park 904, 1012WX Amsterdam, The Netherlands
[2] Elsevier, Radarweg 29, 1043 NX Amsterdam, The Neterlands
meijerarjan@live.nl, r.karimi@elsevier.com

**Abstract.** Over the last few years, word embeddings have taken a dominant position in the Information Retrieval domain. Many studies have been done concerning the quality and application of word embeddings on general texts, such as the Wikipedia corpus and comments on review websites. Giving promising results, the word embeddings have been studied and improved over recent years. However, these studies have been focused on generic texts, which are not limited to the characteristics of in-domain texts such as rare domain-specific words or have been focussed on small sets of academic texts. This research focusses on the quality and application of word embeddings on domain-specific texts, concerning a large corpus of 1.391.543 scientific articles which have been published in 2017. We validate the word embeddings using a categorization task to match articles to journals. We furthermore create a 2-dimensional visualization of the word embeddings on journal level, to visualize journal relatedness.

**Keywords:** Word Embedding · Document Embedding · Article Embedding · Journal Embedding · Embedding Visualization · Embedding Validation.

## 1 Research Data

### 1.1 Corpus

The dataset we used for this research consists of articles published in 2017 which have been published in journals that have (in 2017) at least 150 publications. This selection leads to a dataset of 1.391.543 articles from 3.759 journals. Details about the corpus can be found in table 1.

| | Total count | Unique count | Average length[3] |
|---|---|---|---|
| Title words | 18.822.399 | 939.665 | 14 |
| Title tokens | 14.742.192 | 230.805 | 11 |
| Abstract words | 264.653.020 | 5.853.077 | 190 |
| Abstract tokens | 171.474.473 | 738.961 | 124 |
| Total words | 283.475.419 | 6.209.769 | 204 |
| Total tokens | 186.962.354 | 763.475 | 135 |

**Table 1.** Corpus size

**Text properties (author?)** [4] state that the Pareto distribution offers a good fit to the word occurrences in natural language. Their work shows that, due to the additional parameters in the Pareto-III[4], the tail of the data fits better with the model than the Zipf[5] model. The fit to the Pareto distribution shows the relation between the word occurrences rank and the actual occurrences. This kind of word-occurrences distribution holds for many texts, including the writings of William Shakespeare, scientific texts and novels[2]. The word occurrences in our corpus also follow the pattern of a Pareto distribution as described by **(author?)** [4]. Our word and token distribution is visualized in Figure 1, which displays the occurrences of the first 500 tokens of the corpus.

[3] Rounded

[4] The pareto distribution is a distribution in which recognizable by a long "tail" of low values and a quick and short ascend to the top values. More information: `https://en.wikipedia.org/wiki/Pareto_distribution`.

[5] Distribution similar to the Pareto distribution, see: `https://en.wikipedia.org/wiki/Zipf%27s_law`

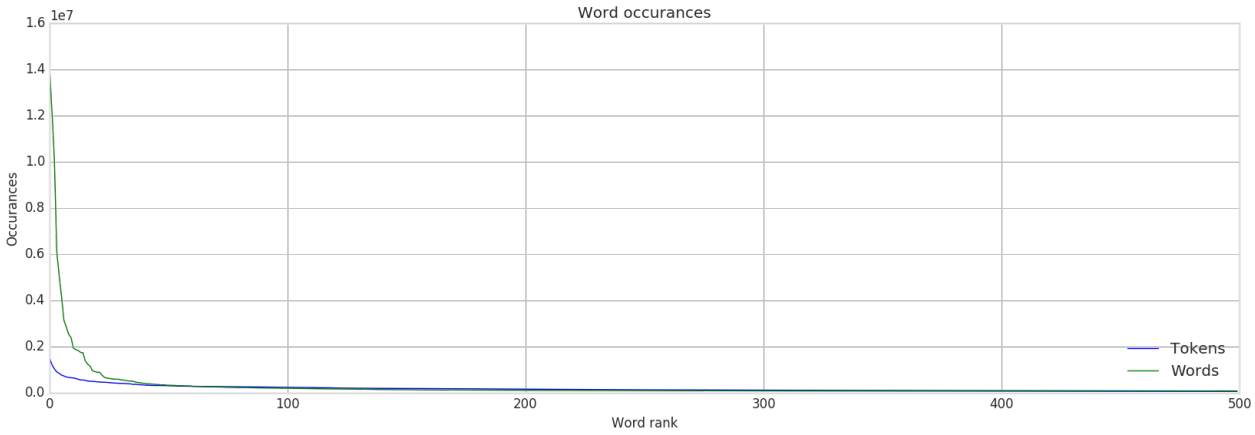


**Fig. 1.** Word and token occurrences. The word counts pattern resembles a Pareto distribution; the tokenization reduced the number of high-frequency words. The y-axis shows the number of occurances for each word, the x-axis shows the occurence rank of the word, meaning the position of the word in a list sorted on word occurance.

### 1.2 Datasets

For this research we used a (pre-made) tokenized dataset, created from the earlier described corpus, which reduces the total amount of words by 34% (see table 1). From this tokenized set, we created the embeddings and the TF-IDF feature vectors.

**Tokenization** The following steps have been applied to the words from the text to create a tokenized set:

1. Removed punctuation
2. Removed all non-ASCII characters
3. Transformed all characters to lower-case
4. Removed stop-words, as provided by the NLTK[6] library
5. Removed numbers
6. Stemmed all words, using the stemmer provided by the NLTK library

These transformations reduced our dataset by 34%, resulting in a tokenized set of 186.962.354 tokens. ]

**Embedding** For this research, we reused the word embeddings created by **(author?)** [3]. These embeddings have a vector dimension of 300, which is an industry default. They have been trained on the entire Elsevier corpus (titles and abstracts respectively), not limited to the subset we used for this research. To

[6] Natural Language ToolKit, `https://www.nltk.org/`

create higher-level embeddings, we average all component embeddings. Thus, to create article embeddings (higher level), we take the average of all normalized word embeddings (component) for that article. Journal embeddings are created by taking the average of all normalized article embeddings. We use the average to combine multiple embeddings into one because we determine the meaning of a larger text as the "average meaning" of all components. Since this meaning is represented as an embedding, we can average the embeddings to create the average meaning. We have used multiple embedding optimizations for this research.

**Default embedding** The default embedding is created from the pre-trained word embeddings; no modifications have been applied to this set. We refer to this set in the figures as **embedding**.

**TF-IDF weighted embedding** The TF-IDF weighted embedding sets, referred to as TF-IDF embedding, are the default word embeddings weighted with a TF-IDF score per word.
The TF-IDF is calculated with a raw token count, and a smoothed inverted document frequency, calculated as follows:

$$IDF = \log_{10}(\frac{|A|}{|A_t|}) \tag{1}$$

Where $|A|$ is the total count of articles and $|A_t|$ is the count of articles containing term t. The article embeddings are a normalized summation of each word vector multiplied by it TF-IDF value. Since we take a sum of all words, the Term Frequency is embedded as the raw count of each word. We will refer to this embedding in the figures as **tfidf_embedding**.

**10K TF-IDF embedding** The 10K embedding set is generated similarity to the TF-IDF embedding, this version only uses the 10.000 most common tokens, reducing the number of tokens it uses. This set was created to see if the limitation to 10.000 tokens reduces the amount of noise[7], increasing the performance. We will refer to this embedding in the figures as the **10k_embedding**.

**5K TF-IDF embedding** The 5K TF-IDF embedding is the TF-IDF embedding set, limited to the 5.000 most common words. This set was created to limit the number of tokens more aggressively, and with that, cancel out more noise. We will refer to this embedding in the figures as **5k_embedding**.

**1K-6K TF-IDF embedding** The 1K-6K TF-IDF embedding is the TF-IDF embedding limited to the top 6.000 most common words, without the top 1.000 most common words. The rationale for this cutting is that common words will

[7] Noise in this context are rare word

occur in many articles, creating noise, by cutting off the top 1.000 and cutting off everything below 6.000 we tried to reduce the noise by filtering common words. This cut results in a set of 5.000 tokens, which allows us to compare it to the 5K TF-IDF set. We will refer to this embedding in the figures as **1k_6k_embedding**. We refer to the **10k_embedding**, **5k_embedding** and the **1k_6k_embedding** as the **limited TF-IDF embeddings**, because these embedding use TF-IDF weighing, and have been limited in their vocabulary due to the cut-off's.

**TF-IDF** To create the TF-IDF feature vectors, we used the TF-IDF model and a hasher from PySparks MlLib library[8]. The TF-IDF feature vectors are created by hashing the tokens with the hasher, which has a set hash bucket size. These hashed values are passed on to the TF-IDF model, resulting in a feature vector. The vector dimensions of the feature vector equal the number of hash buckets. To limit the computational and storage expenses and to reduce noise by rare words, we limit our vocabulary size. We label the TF-IDF configurations as follows: *vocabularysize/hashbucketsize*. Furthermore, we denote 1.000 as 1K, since we deal with chosen values which can be exactly noted given this notation. This means that the set with a 10.000 vocabulary size and a 10.000 hash bucket size will be denoted as "10K/10K". We will refer to the TF-IDF configurations in our figures as "tfidf_vocabulary size_hash bucket size".

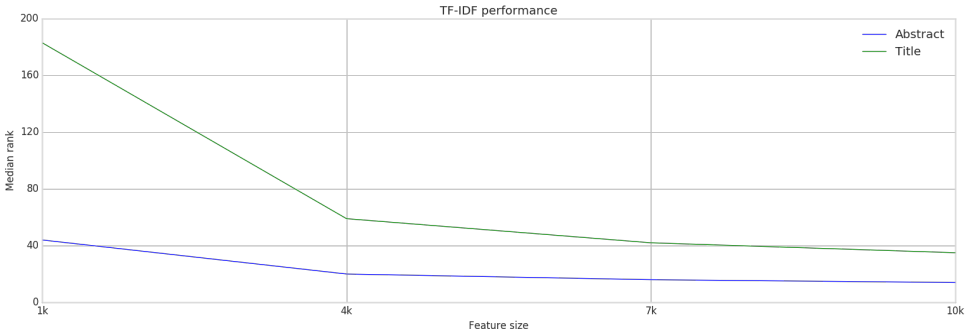


**Fig. 2.** TF-IDF performance on title and abstract.

[8] `http://spark.apache.org/docs/2.0.0/api/python/pyspark.mllib.html`.

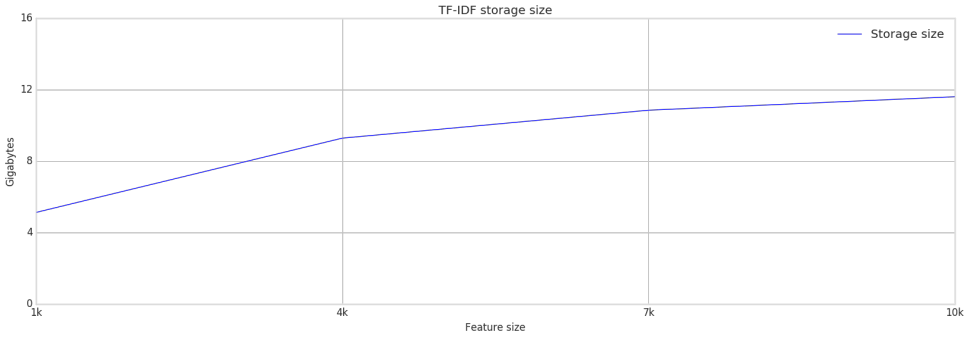


**Fig. 3.** TF-IDF memory usage for title and abstract combined.

Figure 2 shows the performance of TF-IDF on title and abstract as median rank, and Figure 3 shows the storage size in gigabyte, of the 1k/1K, 4K/4K, 7K/7K and 10K/10K TF-IDF configurations. This plot shows that, while the required storage size stagnates, the performance on title also quickly stagnates; the performance of the abstracts show similar behaviour. The stagnation of the storage size is likely due to the fact that the terms which are added to the larger vocabulary sets occur less often than the other words (the cut-off is based on word occurrence, descending). Given this information, we have chosen to use the 10K/10K, 10K/5K and 5K/5K configurations to compare our embedding results to because these configurations can be kept in memory (storage size) while it is not likely that larger sets will give much improvement. The TF-IDF features are created on article level. We average the set of article feature vectors to create a journal feature vector, just as we did with the embeddings.

## 2 Pipeline

### 2.1 Research environment

The research has been done in a python-databricks environment, which used Spark SQL, a library that offers tools to work with big-data. **(author?)** [1] state that Spark SQL lets programmers leverage the benefits of relational processing and lets SQL users call complex analytic libraries in Spark. This allows for much tighter integration of relational and procedural processing. The paper further states that Spark SQL makes it significantly simpler and more efficient to write data pipelines that mix relational and procedural processing while offering substantial speedups over previous SQL-on-Spark engines.

### 2.2 Data processing

We processed the TF-IDF sets and the embedding sets [10] via the same pipeline, using their common vector properties. Using the same pipeline ensures comparable results. The pipeline is set-up as follows:

1. Create training and validation set
2. Create journal embeddings
3. Categorize validation articles
4. Calculate performance metrics

### 2.3 Create training and validation set

We split our initial set 80% - 20%. We use the 80% set as the training set for the journal representations, and the 20% set as the validation set for the journal representations. This split is based on a random number given to each article, ensuring that all datasets (i.e. tfidf_10k_10k, embedding) have the same (random) training and validation set.

### 2.4 Create journal embeddings

From our training set we create the journal embeddings, which are created for most sets[11] by averaging the article embeddings or feature vectors.

### 2.5 Categorize validation articles

To categorize the articles, we calculate the distance between the title of the article, and the title of each journal from the validation set. We also do this for the abstract of the article and the abstract of each journal. During this process we record of:

- Title-based-rank of the actual journal
- Abstract-based-rank of the actual journal
- Best scored journal on the abstract similarity
- Best scored journal on the title similarity
- Abstract similarity between the actual journal and the article
- Title similarity between the actual journal and the article

**Distance metrics** To calculate the distance between two vectors, cosine similarity is commonly used. We validated the quality of cosine similarity as a distance metric by comparing it to other similarity metrics available in the SciPy library[12]. We calculate the similarities based on the normalized embeddings, and compared the distance metrics based on the default embedding set. Table 2 shows the results of this validation.

| Metric[13] | Median title rank | Average title rank | Median abstract rank | Average abstract rank |
|---|---|---|---|---|
| Braycurtis | 28 | 130 | 23 | 124 |
| Canberra | 33 | 148 | 26 | 133 |
| Chebyshev | 57 | 256 | 41 | 191 |
| *Cityblock* | *28* | *130* | *23* | *124* |
| *Correlation* | *27* | *127* | *23* | *122* |
| **Cosine** | **27** | **127** | **23** | **122** |
| Dice | 1995 | 1929 | 1995 | 1929 |
| *Euclidean* | *27* | *127* | *23* | *122* |
| Hamming | 1995 | 1929 | 1995 | 1929 |
| Jaccard | 1995 | 1929 | 1995 | 1929 |
| Kulsinski | 1995 | 1929 | 1995 | 1929 |
| Mahalanobis | 136 | 544 | 75 | 449 |
| Matching | 1995 | 1929 | 1995 | 1929 |
| Rogerstanimoto | 1995 | 1929 | 1995 | 1929 |
| Russellrao | 1995 | 1929 | 1995 | 1929 |
| *Seuclidean* | *27* | *124* | *22* | *115* |
| Sokalmichener | 1995 | 1929 | 1995 | 1929 |
| Sokalsneath | 1995 | 1929 | 1995 | 1929 |
| *Sqeuclidean* | *27* | *127* | *23* | *122* |
| Yule | 1995 | 1929 | 1995 | 1929 |

**Table 2.** Distance metric performance for word embeddings on the categorization of academic texts

These results show high similarity between cosine-based metrics (Cosine & Correlation) and euclidean based metrics (Euclidean, Seuclidean & Sqeuclidean). This similarity is expected, since the cosine and euclidean distances should yield the same results on normalized sets[14]. The results show that some enhancement on the euclidean algorithm (Seuclidean) result in slightly improved results, although not significant. Also the Cityblock metric yields results close to the Cosine metric, only having a slightly worse performance, which is also not significant. Because of this, we will use the cosine-similarity as the distance metric, which will make our results easier to compare with other work, since the cosine-similarity is a commonly used similarity measure.

### 2.6 Performance measurement

We use multiple metrics to validate the performance of the embedding sets and TF-IDF sets on the categorization task. These metrics are:

1. F1-score

[10] See Chapter 3.2 Datasets.
[11] See chapter 3.2 Datasets.
[12] `https://www.scipy.org/`
[13] As defined and provided by the SciPy library
[14] See: `https://en.wikipedia.org/wiki/Cosine_similarity#Properties`.

2. Median & average rank
3. Rank distribution

**F1-score** For our matching algorithm, we classify the results as follows:
$TruePositive$ = Articles that are correctly matched to the current journal
$FalsePositive$ = Articles that are incorrectly matched to other journals
$FalseNegative$ = Articles that are incorrectly matched to the current journal

With these definitions, we calculate the Recall, Precision & F1 as follows:

$$Recall = \frac{|TruePositive|}{|TruePositive| + |FalseNegative|}$$

$$Precision = \frac{|TruePositive|}{|TruePositive| + |FalsePositive|}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall|}$$

here we denote the amount of articles in a class as $|class|$.

**Median & average rank** A dataset rank is calculated using the ranks of all article in that dataset. The median rank of a dataset indicates the point where 50% of all articles are to the left of a virtual line and 50% are to the right of this line. The median rank therefore indicates the position of the "normal/typical" article. This median rank differs from the average, since the median is not influenced by a single value. If the median is 20, it does not matter if an article rank is 25 or 250, it does matter however if an article rank is 19 or 21. For the average rank this is not the case, the average rank is influenced by single articles. Therefore, it does not represents the "normal/typical" article but it represents the exact average rank of all articles in the validation set, which is changed when an article rank is changed from 25 to 250.
We use the median rank to indicate at which rank the typical article would be ranked, based on its title and abstract. We also calculate the average score for the title and abstract of all articles. This gives us four scores: **median title rank, median abstract rank, average title rank & average abstract rank.**

**Rank distribution** To further analyse the ranking results, we plot the article rank distribution to get an indication of the ranking-landscape.

**Two-dimensional plot** To create a plot, we transformed the 300-dimensional journal vectors into 2-dimensional vectors using TSNE based on PCA[15]. These 2-dimensional vectors, representing the x & y coordinate, can then be drawn

[15] Principal component analysis

in a plot. To visualize the preservation of journal-relatedness while converting the 300 dimensions to 2 dimensions, we create groupings using k-means. These groupings are created on the 300-dimensional vectors and are visualized in the plot using colors. We use the k-means groupings due to the lack of subject-based grouping values in our dataset, for this research we used 2, 5, 8, 10, 20, 25, 30, 35 and 40 as the amount of groups. We included only the plots of 8 groups, because we find this plot the most informative. All group amounts can be seen in the digital version of the plots.

[3]

# Bibliography


[1] Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., et al.: Spark sql: Relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 1383–1394. ACM (2015)

[2] Thurner, S., Hanel, R., Liu, B., Corominas-Murtra, B.: Understanding zipf's law of word frequencies through sample-space collapse in sentence formation. Journal of the Royal Society Interface **12**(108), 20150330 (2015)

[3] Truong, J.: An evaluation of the word mover's distance and the centroid method in the problem of document clustering (2017)

[4] Wiegand, M., Nadarajah, S., Si, Y.: Word frequencies: A comparison of pareto type distributions. Physics Letters A (2018)