Data

Corpus
The dataset we used for this research consists of all articles[1] published in 2017 which have been published in a journal that has, in 2017, atleast 150 publications. This results in a total dataset of 1.391.543 articles from 3.759 journals.

	Total word count	Unique word count	Total token count	Unique token count
Title	18.822.399	939.665	14.742.192	230.805
Abstract	264.653.020	5.853.077	171.474.473	738.961
Total	283.475.419	6.209.769	186.962.354	763.475

Table 1: Corpus size

The word occurrences follow the pattern of a pareto distribution as described by **?**]. This distribution is visualized in XXXXXX, which displays the occurrences of the first 500 tokens of the corpus.

Datasets
For this researched we used a (pre-made) tokenized dataset, which reduces the total amount of words by 34%, from this tokenized set, we created the embeddings and the TF-IDF feature vectors.

<u>Tokenization</u>
The following steps have been applied to the words 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 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.

<u>Embedding</u>
For this research we reused the word embeddings created by **?**]. These embeddings have a vector size of 300, which is an industry default. They have been trained on the entire Elsevier corpus, not limited to the subset we used for this research. To construct article embeddings we take the average of all normalized word embeddings for that article. Journal embeddings are constructed in the same way, we averaged all the normalized article embeddings to create the journal embeddings. We have used multiple embedding configurations for this research

<u>- Default embedding</u>
The default embedding is created form the pre-trained word embeddings, no modifications have been applied to this set.

<u>- TF-IDF embedding</u>
The TF-IDF weighted embedding set, 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:

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

[1]From Elseviers corpus

Where $|A|$ is the total count of articles and $|A_t|$ is the count of articles containing term t. The articles 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.

- 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 amount of tokens it uses. This set was created to see if the limitation to 10.000 tokens reduces the amount of noise, and with that increasing the performance.

- 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 more aggressively limit the amount of tokens, and with that, cancel out more noise.

- 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 is that common words will occur in many articles, creating noise, by cutting of the top 1.000 and cutting of 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.

TF-IDF

To create the TF-IDF feature vectors, we used the TF-IDF model and a hasher from pysparks the MlLib library. 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 which vector dimensions equals the amount of hash buckets. To limit the computational and storage expenses and to reduce noise by rare words, we limit our vocabulary size. We denote 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.

The graph XXX shows the performance and storage size[2] of the 1k/1K, 4K/4K, 7K/7K and 10K/10K TF-IDF configurations. This plot shows that, while the required storage size keeps rising, the performance on title quickly stagnates, and the performance on abstract follows too. Given this information, we have chosen to use the 10K/10K, 10K/5K and 5K/5K configurations to compare our embedding results to. The TF-IDF features are created on article level. We average the set of article embeddings to create a journal embedding.

Pipeline

We processessed the TF-IDF sets an the embedding sets via the same pipeline, based on vector calculations. This ensures comparable results. The pipeline is setup as follows:

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

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 article each record, ensuring that all set have the same (random) training and validation set.

Create journal embeddings

From our training set we create the journal embeddings, which are created for most sets[3] by averaging the article embeddings or feature vectors.

Categorize validation articles

To categorize the articles, we calculate the distance between the title- and abstract embedding of each article,

[2]in gigabyte, 1024 based

[3]see XXXXX

from the validation set, to the title- and abstract embedding of each journal, during this process we keep track 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 to vectors, cosine similarity is commonly used CITATIONS NEEDED. We validated the quality of cosine similarity as a distance metrics by comparing it to all other similarity matrices available in the SciPy library, which we used to calculate the distances. We calculate the similarities based on the normalized embeddings, and compared the distance metrics based on the default embedding set. XXXX shows the results of this validation.

Metric[a]	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
minkowski	27	127	23	122
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

[a]As defined and provided by the SciPy library

These values are used in the Performance measurements.

Performance measurement

We use multiple metrics to indicate the performance of the embeddings on a categorization task. These metrics are:

1. F1-score
2. Median & average rank
3. Rank distribution

- F1-score

We define the positive & negative metrics 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

We used these metrics to calculate the Recall, Precision & F1 as follows:

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

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

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

- Median & average rank

We use the median rank to indicate around which rank the 'standard' article would be ranked, based on its title or abstract. We do this by taking the median of the respective rank from each article. This gives us an indication of the behaviour of most articles in our validation set. This median rank (mostly) ignores the outliers, we therefore also use the average rank, which gives a more global indication, although this rank may be over-influenced by some outliers.

- Rank distribution

To further analyse the ranking results, we plot the rank distribution to get an indication of the ranking-landscape. We limit ourselves to the following categories: 1 (absolute hits), top-10, top-20, top-30, top-40, top-50, top-100 and 100+.