1 Corpus

The dataset we used for this research consists of articles 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. Details about the corpus can be found in table 1.

	Total count	Unique count	Average length
Title words	18.822.399	939.665	13,53
Title tokens	14.742.192	230.805	10,64
Abstract words	264.653.020	5.853.077	190,19
Abstract tokens	171.474.473	738.961	123,71
Total words	283.475.419	6.209.769	203,71
Total tokens	186.962.354	763.475	134,36

Table 1: Corpus size

1.1 Text properties

?] state that the Pareto distribution offers a good fit to the word occurrences in natural language. Their models show that, due to the additional parameters in the Pareto-III, the tail of the data fits better with the model than the Zipf model. This 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 and scientific texts and novels[**?**]. The word occurrences in our corpus also follow the pattern of a Pareto distribution as described by **?**]. Our distribution is visualized in figure 1, which displays the occurrences of the first 500 tokens of the corpus.

Figure 1: Word and token occurrences

2 Datasets

For this researched 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.

2.1 Tokenization

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[1] 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.

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

2.2 Embedding

For this research, we reused the word embeddings created by **?**]. These embeddings have a vector dimension 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 create higher-level embeddings, we average all component embeddings. Thus, to create article embeddings, we take the average of all normalized word embeddings 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 form 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 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:

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

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. 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, 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 is that common words will 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.

2.3 TF-IDF

To create the TF-IDF feature vectors, we used the TF-IDF model and a hasher from PySparks 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 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".

Figure 2: TF-IDF performance on title and abstract

Figure 3: TF-IDF performance on title and abstract

Figure 2 shows the performance of 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 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 feature vectors to create a journal feature vector, just as we did with the embeddings.

3 Research environment

The research will be done in a python-databricks environment, which uses spark, a library that offers tools to work with big-data. **?**] 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 between 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.

4 Pipeline

We procesessed the TF-IDF sets and the embedding sets via the same pipeline, using their common vector properties. This 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

4.1 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.

4.2 Create journal embeddings

From our training set we create the journal embeddings, which are created for most sets[2] by averaging the article embeddings or feature vectors.

4.3 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 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

[2] see paragraph datasets

- Title similarity between the actual journal and the article

Distance metrics
To calculate the distance between to vectors, cosine similarity is commonly used. 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. Table 2 shows the results of this validation.
These results show high similarity between cosine-based metrics (Cosine & Correlation) and euclidean based

Metric[3]	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

metrics (Euclidean, Seuclidean & Sqeuclidean). This similarity is expected, since the cosine and euclidean distances should yield the same results on normalized sets. The results show that some enhancement on the euclidean algorithm result in slightly improved results, although not significant. Also the Cityblock metric yields results close to the Cosine metric, it has 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 better comparable with other work.

4.4 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
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 at 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 the 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.

5 Two-dimensional plot

To create a plot, we transformed the 300-dimensional journal vectors into 2-dimensional vectors using TSNE based on pca[4]. These 2-dimensional vectors, representing the x & y coordinate, can then be drawn 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.

[4]Principal component analysis