# Information Retrieval Project Course (IE 691/681)

**Elina Tugaeva, Eliana Ruslanova, Marlena Kuc**

Faculty of Business Informatics and Mathematics
University of Mannheim
{etugaeva, eruslano, mkuc} @mail.uni-mannheim.de

## Abstract

This work is focused on the implementation of an information retrieval system based on the Vector Space Model and methods of its improvement. Three speed-up methods were implemented and discussed, namely, tiered index, pre-clustering and random projections. The tiered index was evaluated on short title queries whereas the pre-clustering and random projections - on long queries. None of the speed-ups showed the same performance as the basic retrieval, however, the random projections achieved the results close enough to the basic retrieve with much better retrieval efficiency.

## 1 Introduction

Against the backdrop of an increasing amount of unstructured data and growing demand for information need, the field of information retrieval became an area of intense research over the past few decades. The information retrieval problem can be summarised as a search of best-matching documents for a certain query. One way to approach it is via application of Vector Space Model (VSM). VSM represents a collection of documents as vectors that capture the relative importance of the terms in a document [1].

In our task, we focused on developing and evaluating basic version of VSM as well as its various upgrades, namely tiered index, pre-clustering and random projections, in Python. Our work is based on NFCorpus dataset[2], which contains medical documents and queries. Due to the limited computational power, the VSM implementation and evaluation were performed on the development subset which consisted of 3 files: 1) the file with 3193 documents; 2) the file with 325 queries; 3) the file with relevance links between documents and queries.

## 2 Implementation

### 2.1 Preprocessing

When building the information retrieval model, preprocessing is a step of high importance. We want to bring our textual data into the form that will be more digestible by model and that will increase the accuracy of our task. One of the standard NLP preprocessing techniques that we applied was stripping punctuation from the text. Numerical characters were also removed since they are not relevant for our analysis. In the next step, we converted uppercase characters to lowercase. Another crucial preprocessing method applied to our dataset was removing the stop words, as they do not have any meaning. It also helped us to reduce the size of the vocabulary. All the words were also cast to their roots via stemming. For that, we used the Porter stemming algorithm provided by `nltk` library. Since terms in queries which do not appear in the corpus are irrelevant, they were removed.

### 2.2 TF-IDF and cosine similarity

The basic version of VSM assumes calculating TF-IDF weights and all cosine similarity scores.

TF-IDF is used in the evaluation of how important a word is to a corpus in a document. In principle, it gives more weight to rare words and less weight to common words which are effectively treated like domain-specific stop words. Its first component, *term frequency* was calculated as:

$$tf(t_i, d_j) = 1 + \log(ft_d) \tag{1}$$

Second component of TF-IDF is *inversed document frequecy*, computed according to the following formula:

$$idf(t_i) = \log(\frac{N}{df_i}) \tag{2}$$

TF-IDF is then represented as multiplication of $tf(t_i, d_j)$ and $idf(t_i)$. The final implementation was done in function `tf_idf` which outputs dictionary of elements: $\{d_j : \{t_i : tfidf(t_i, d_j)\}\}$.

It is common that TF-IDF is combined with cosine similarity, denoted as:

$$\cos(d_i, d_j) = \frac{V(d_i) \cdot V(d_j)}{|V(d_i)| \cdot |V(d_j)|} \tag{3}$$

Cosine similarity compares vectors obtained in application of TF-IDF by measuring the cosine angle between them. Its main advantage is that it normalizes document lengths. In our case, it is particularly useful, as we applied raw term frequency which does not account for document length. Cosine similarity was implemented in function `cosine_similarity`. The function takes two vectors as arguments and computes vector magnitudes of each of them as well as their dot product. These sub results are then fed into the equation (3).

When performing information retrieval task on a single query, as a first step we represented both query and documents as vectors of TF-IDF weights. What is important to notice, the length of the query vector corresponds to the length of documents' vectors. Next, we computed cosine similarity scores for each of the query vector and each document vector. In the function `basic_retrieve` which contains described operations, user can specify the number of top *K* relevant documents that should be returned.

Retrieving information using solely TF-IDF and cosine similarity has certain shortcomings. For each query, we have to compute all cosine scores. Moreover, we do not make use of sparsity in term space [3]. In order to improve the efficiency of our model, we developed various upgrades which are discussed in further sections.

### 2.3 Tiered index

Tiered index addresses the problem of inefficiency in the computation of cosine scores. Instead of calculating cosine similarity for all $(query - document)$ pairs we can first filter out the most relevant documents. To achieve so, we build an inverted index for all terms in the corpus and then split it into multiple indexes according to $tf(t_i, d_j)$ scores. Documents with the highest scores are in the first index and subsequent indexes consist of progressively worse documents [4]. Then at the query time, we first match documents from the top tiers and then if not enough results are produced, we extend the list of candidates for match by documents from consecutive tiers.

The data structure needed for tiered index retrieval was implemented in function `tiered_index`. It first sorts the dictionary of inverted index by $tf(t_i, d_j)$ values, chunks each of its value into $K$ number of lists (tieres) specified by user. The tieres are of approximately same length. Finally the algorithm sorts documents by their integer $ID$ in an ascending order. The last step is crucial, as we still want to perform posting merges in linear time. The output of the function is dictionary with elements: $\{t_i : \{k_i : list(d_j)\}\}$, where $k_i$ denotes i-th tier. The example posting from achieved tiered index with 3 tieres is $\{0 : [1263, 1271, 1277],\ 1 : [1262, 1280],\ 2 : [0, 1270]\}$ for term *alkylphenol*.

When retrieving information from tiered index at each iteration we need to merge $n$ number of postings' fractions where $n$ in a number of terms in a query. In order to optimize the process of finding the intersection, the following algorithm has been proposed:

*1*) Tieres that need to be merged are sorted in an ascending order according to their length and stored in a list.

*2*) The intersection is performed on the first two elements and its output is attached to the list of tieres.

*3*) Merged tieres are removed from the list.

*4*) The list is sorted and repeats steps (2) and (3) until there is only one element left.

The final document ranking is performed in function `tiered_index_retrieve()`. If none or not enough relevant documents were found, the algorithm performs basic retrieval.

### 2.4 Pre-clustering

The ultimate goal of document clustering is to improve the efficiency of information retrieval through grouping similar documents together, assigning them to the same implicit topic based on lexical semantics: clustering of words by the co-occurrence patterns. The cluster hypothesis states the fundamental assumption we make when using clustering in information retrieval:

*Similar documents are usually relevant for the same retrieval task.*

To put it another way, the hypothesis states that if there is a document from a cluster that is relevant to a query, then it is likely that other documents from the same cluster are also relevant: we expect similar documents to relate similarly to the query.

### 2.4.1 Cluster pruning

In this section, we study cluster pruning, an extremely simple randomized technique. During preprocessing we randomly choose a subset of data points to be leaders the remaining data points are partitioned by which leader is the closest.

We choose leaders at random, as this procedure is relatively fast and randomly chosen leaders can reflect the true distribution of topics well. Intuitively, if the data set is nearly balanced, we expect the number of followers to each leader to be $N/\sqrt{N} = \sqrt{N}$.

Next, query processing is done by choosing the nearest leader to that query and retrieve $k$ nearest documents from the cluster the nearest leader is in.

This technique is characterized by relatively time-consuming offline stage, when leaders are chosen and $N \times \sqrt{N}$ similarities are calculated to assign each follower to its leader. The second, online stage, when the query is entered "on-the-fly" and the documents are to be retrieved is supposed to take noticeably less time as compared to the situation, when no preparatory step was done.

To begin with the manual implementation of the clustering algorithm one should choose the distance measure, which will be used for assigning followers to leaders and leaders to the queries. Since we are now in the settings of the Vector Space Model, it is natural to start with cosine similarity, which was explained in the preceding sections. With reference to the code, the function `allocate_docs_to_clusters()` performs the picking of random leaders and assignment followers to them. Since we proceed with cosine similarity, parameters should be set as follows: `cosine=False` and `Faiss=False`.

Then the task is to construct the function, which uses query $q$ (already in the vector form) as input, required similarity of the doc to be retrieved - threshold, and a necessary number of documents to be retrieved - $K$ (5 most similar docs in the cluster by default) `ir_preclustering()`. The function requires either only threshold similarity (minimum similarity between the document and the query) or the required number of documents since it is rarely possible to satisfy both conditions.

Finally, let us introduce the connected function `retrieve_with_ preclustering()`, which now takes the string query as an input, vectorizes it and retrieves relevant documents in a form of dataframe.

### 2.4.2 FAISS

As an additional tool one could use is FAISS [**?** ] - a library for efficient similarity search. A typical operation in IndexFlatL2 is to exhaustively compare a set of query vectors and a set of database vectors in dimension $d$ (then select the $top - k$ smallest vectors). Most of the similarity search methods from FAISS use a compressed representation of the data in order to speed up the search process. While this can lead to less precise results, the resulting payoff in memory storage and time saved can greatly out-weigh a marginal loss in search precision.

The procedure is the same as with cosine similarity, so we have the same supporting functions: `ir_preclustering_faiss()`, which becomes a query vector as an input and retrieves dataframe of relevant documents and `retrieve_with_preclustering _faiss()`, which takes query text as an input and retrieves dataframe of relevant documents.

### 2.4.3 K-means clustering

The k-means is arguably the most common technique applied for document clustering. It is a non-parametric, distance-based clustering method. The use of K-means is purely initiative to use it as a comparative benchmark to the former approach. The main reasonable advantage of this model is that current k++ means algorithms are designed in such a way that they choose the initial centroids using a weighted method which makes it more likely that points further away will be chosen as the initial centroids. The idea is that while initialization is more complex and will take longer, the centroids will be more accurate and thus fewer iterations are needed, hence it will reduce overall time.

## 2.5 Random projections

As we already discussed, pre-clustering decreases the number of cosine computations by decreasing

the number of documents we calculate the similarity with. However, we might also reduce the cost of cosine computations by using one of the Locality Sensitive Hashing (LSH) techniques - random projections. The idea behind this method is the reduction of the length of vectors on which we perform cosine computations. To achieve this dimensionality reduction, we completed the following steps:

1. Generated and normalized $m$ random vectors with length $d$ where $d$ is the length of document vector (implemented in function `get_random_vectors`).

2. For each document $doc$ calculated the dot product with all random vectors $r_k$ and applied a hash function on each dot product which equals 1 if it is bigger than some threshold (implemented in function `compute_hash`).

3. Created a new vector that consisted of computed hash functions.

The random projections method is based on Johnson-Lindenstrauss lemma which states that a collection of points from one Euclidean space can be mapped to a smaller Euclidean space without distorting the distances between any pair of points by more than a factor of (1+ $\epsilon$) (where $\epsilon$ is an error level that is fixed).[5] Therefore, we can say that $d$-dimensional vectors can be projected to a $m$-dimensional sub-vectors ($m << d$) using a random matrix $m \times d$ and the distances between them will be preserved. Moreover, this method showed good results in the area of information retrieval in text documents comparing to other dimensionality reduction methods [6].

During our implementation, we needed to decide on the randomization function, threshold for hash function and size of new vectors with reduced dimensionality. Firstly, it was decided to use random positive numbers in the range $[0, 1]$ for creating random vectors. However, for this type of randomization, the threshold selection was complicated and the performance measures were extremely low (e.g., average precision on all queries was less than 1%). Therefore, it was decided to switch to the Gaussian distribution for creating random vectors because this distribution is used most often [6] and it allowed us to choose a zero threshold for our hash function (mean value of dot products is near zero). Secondly, we experimented with four different sizes of reduced dimensionality and performed a comparative analysis of performance measures dependent on these parameters. We assumed that performance metrics would improve as the size of new projected vectors increase. The results of this analysis are discussed in Section 3.5.

## 3 Evaluation

### 3.1 Evaluation metrics

There are several evaluation metrics we used to assess how well the search results satisfied the user's query intent. Since we have relevance feedback, we can compare retrieved results with ground truth. The most common measure is the precision - the fraction of the documents retrieved that are relevant to the user's information need. To be more precise, we evaluated precision at a given cut-off rank $k$ - the number of documents to be retrieved prespecified by the user, considering only the topmost results returned by the system. This measure is called precision at $k$ or $P@k$.

In our setting, when the system returns a ranked sequence of documents, it is desirable to also consider the order in which the returned documents are presented. This can be done by the use of Average Precision of the single query and Mean Average Precision across all the queries.

In our disposal, there are graded relevance annotations, which depict the level of relevance of the document to the given query. This information was also used in performance evaluation since the user will always prefer to have highly relevant documents on the top. (Normalized Discounted) Cumulative Gain (nDCG) takes into account the graded relevances of documents when evaluating the ranking produced by IR systems.

As for the evaluation of retrieval efficiency, we are going to measure the time it took some algorithm to perform the evaluation on all queries. Except for tiered index, the conversion of documents and queries into vectors was not counted.

### 3.2 Basic retrieval

For further discussion on the effectiveness of the selected speed-up methods, it is necessary to first present the results of the basic VSM model which represented on table 2. This and all further models were evaluated on retrieval of top 5 documents

which have the highest cosine similarity with the given query.

| $\overline{\text{P@k}}$ | MAP | $\overline{\text{nDCG}}$ |
|---|---|---|
| *Basic retrieve on queries* | | |
| 0.394 | 0.260 | 0.336 |
| *Basic retrieve on titles* | | |
| 0.328 | 0.238 | 0.286 |

Table 1: Performance measures of basic retrieval

The evaluation of basic VSM on all queries was running for 8 min 15 sec and for query titles for 10 min 3 sec. In the next sections, we are going to compare the results of implemented speed-ups with these metrics.

### 3.3 Tiered index

Tiered index search is reasonable only for short queries, as it aims at finding the intersection between all query terms. If queries are long, which is the case of our dataset, very few or none documents can be found and as a result algorithm is forced to perform basic retrieval instead. For the reason stated above, the evaluation of tiered index was done on query titles.

| $\overline{\text{P@k}}$ | MAP | $\overline{\text{nDCG}}$ |
|---|---|---|
| *Tiered index retrieve (on titles)* | | |
| 0.165 | 0.11 | 0.141 |

Table 2: Performance measures of tiered index retrieval

The retrieval process with help of tiered index with number of tieres = 4, lasted 16min 1s. However it should be noted that queries were vectorized inside the algorithm. Comparing to the baseline model (basic retrieval) tiered index did not bring any improvement in terms of accuracy.

### 3.4 Pre-clustering

The pre-clustering algorithm was run and subsequently assessed with respect to different random state (referred as RS in the table). Average performances of clustering with cosine similarity, FAISS and K-Means are summarized in the table 3.

Performance metrics of both pre-clustering with cosine similarity and with K-means algorithm are close to the basic method, where similarities with all documents are calculated. Low performance showed the FAISS method. A possible explanation may be due to inherited trade-off calculation between the speed and the accuracy.

| RS | $\overline{\text{P@k}}$ | MAP | $\overline{\text{nDCG}}$ |
|---|---|---|---|
| *Pre-clustering with cosine similarity* | | | |
| 11 | 0.164 | 0.110 | 0.139 |
| 110 | 0.176 | 0.110 | 0.145 |
| 1100 | 0.221 | 0.149 | 0.188 |
| *Pre-clustering with FAISS* | | | |
| 11 | 0.014 | 0.009 | 0.012 |
| 110 | 0.025 | 0.019 | 0.020 |
| 1100 | 0.015 | 0.008 | 0.013 |
| *Pre-clustering with K-means* | | | |
| 11 | 0.291 | 0.189 | 0.245 |
| 110 | 0.279 | 0.170 | 0.227 |
| 1100 | 0.329 | 0.233 | 0.284 |

Table 3: Perfomance measures on pre-clustering

Among pre-clustering algorithms, the best retrieval speed exhibited FAISS (1min 25s for retrieving 5 documents for all 325 queries) followed by cosine similarity (1min 47s), and the last goes K-means (10min 5s). It is vital to mention that the time was measured only for 'on-the-fly' calculations, assuming all offline preparations are already done before the user enters a query. So the time spent for retrieval by former two algorithms improved significantly as compared to the basic method, whereas K-means took more time as compared to the basic one even despite the fact that it is supposed to perform fewer calculations. Combining performance measured in terms of metrics discussed above and the time performance - one could recommend using pre-clustering with leaders using cosine similarity. We are staying sceptical about the usage of FAISS due to its low retrieval performance.

### 3.5 Random projections

As it was already mentioned in section 2.5, we analyzed the dependence of performance metrics on the size of reduced dimensionality (the size of new projected vectors $m = 1000, 5000, 10000, 15000$). The results are presented on tables 4 and 5.

On the one hand, as we suggested, the best performance is shown by the model with dimensionality $m = 15000$. Closer we get to our original dimensionality, the less the error we have, however, at the same time, the slower our performance improves.

| m | $\overline{\text{P@k}}$ | MAP | $\overline{\text{nDCG}}$ |
|---|---|---|---|
| *Random projections* | | | |
| 1000 | 0.178 | 0.092 | 0.148 |
| 5000 | 0.316 | 0.198 | 0.270 |
| 10000 | 0.355 | 0.237 | 0.309 |
| **15000** | **0.370** | **0.244** | **0.318** |

Table 4: Performance measures of random projections

| m | Time, sec |
|---|---|
| **1000** | **10.6** |
| 5000 | 12.9 |
| 10000 | 15.6 |
| 15000 | 19.8 |

Table 5: Evaluation times of random projections

On the other hand, the retrieval efficiency of random projections is better when we have less dimensionality. This result can be explained by the fact that we have fewer dot product calculations on smaller vectors. The best result was exhibited by the model with $m = 1000$ dimensionality. The run time execution of the evaluation on all queries was 10.6 seconds. However, due to the fact that this model had the lowest performance measure values, we could argue its efficiency. It took 19.8 seconds to execute the evaluation of the model with $m = 15000$ dimensionality but the values of evaluation metrics have doubled. Therefore, for the random projection method, we should make a trade-off between performance and efficiency. In our case, the difference in retrieval speed was not so drastic as in retrieval performance, therefore we stated the model with bigger ($m = 15000$) dimensionality as the best model in this speed-up method.

## 4 Summary

In this work, we investigated the information retrieval system based on Vector Space Model and different methods for its improvement. First, we described the core VSM model based on tf-idf scores and how it was implemented. Then we focused on the theoretical background behind the chosen speed-up approaches, namely, tiered index, pre-clustering and random projections, and briefly mentioned how they were implemented. Next, we introduced the evaluation metrics we used to assess how well our models performed. And, finally, we evaluated their retrieval performance and efficiency and compared to the basic retrieval results.

To sum up our discussion, we are going to compare the results of implemented speed-ups. (some sentence about tired index) Pre-clustering with K-Means exhibited good performance, however, its efficiency is arguable since it took more time to evaluate all full queries comparing to the basic retrieval. Another clustering method, simple pre-clustering with cosine similarities, performed worse but achieved better time efficiency. In the meantime, Faiss showed the best retrieval speed among all pre-clustering algorithms but exhibited the worst retrieval performance. And, although none of the speed-up methods achieved the same performance as the basic retrieval, random projections showed comparably good results with much less time than any other speed-ups.

## References


[1] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.

[2] Vera Boteva, Demian Gholipour, Artem Sokolov, and Stefan Riezler. A full-text learning to rank dataset for medical information retrieval. 2016.

[3] Hedvig Kjellström. Lecture 5: Scoring, weighting, vector space model. https://www.kth.se/social/files/56c4d886f2765449d27d5407/lecture05.pdf. Online; accessed 15 May 2020.

[4] Jesse Anderton. Tiered indexes indexing, session 12. https://course.ccs.neu.edu/cs6200sp15/slides/m04.s12%20-%20tiered%20indexes.pdf. Online; accessed 15 May 2020.

[5] William B Johnson, Joram Lindenstrauss, and Gideon Schechtman. Extensions of lipschitz maps into banach spaces. *Israel Journal of Mathematics*, 54(2):129–138, 1986.

[6] Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 245–250, 2001.