The vector space model

Now we want to represent everything as vector: each term now is a vector of the canonical base of $\mathbb{R}^n$, with n the number of terms in the dictionary.

The representation of a set of documents as vectors in a common vector space is known as the **vector space model**. Also queries are seen as vectors in the same vector space of the documents collections.

The vector $\mathbf{v}(d)$ derived from the document d has one component for each dictionary term, the components are computed using the $tf - idf$ weighting scheme. The set of documents in a collection then may be viewed as a set of vectors in a vector space, in which there is an axis for each term. Note that this representation loses the relative ordering of the terms in each document.

A document then is simply a point in the n-dimensional space, where we use as components of the vector the $tf - idf$ of the terms inside the document. We can compute how similar tow documents are computing the **cosine similarity** between the two corresponding vectors: $sim(d_1, d_2) = \frac{\mathbf{v}(d1) \cdot \mathbf{v}(d2)}{|\mathbf{v}(d1)||\mathbf{v}(d2)|}$ (i.e. the cosine of the angle formed by the two vectors, with the denominator we length normalize the two vectors, so this is just the dot product of the normalized version of the two document vectors), higher when the two documents are collinear, i.e. point to the same direction. Remark that even if the document contain the same terms but with different frequencies they will point to different directions.

We can assume for simplicity that everything is a unit vector.

Also queries can be considered as unit vectors, with non-zero components corresponding to the query terms.

Now we can compute the cosine similarity between the queries and the documents, so that the answer to the query can be computed. Moreover we have now a natural way of scoring: $score(q, d) = sim(\mathbf{v}(q), \mathbf{v}(d1))$. Since all vectors are assumed to be unit vector, we can simply compute the inner product, i.e. $sim(q, d) = \mathbf{v}(q) \cdot \mathbf{v}(d1))$.

Notice now that each document has an associated score, we can retrieve the K most relevant documents (we are using the cosine similarity between the query vector and a document vector as a measure of the score of the document for that query). Notice also that opposite to the Boolean model, if not all term are present, then we can still have a positive score. So the matching in this case is not exact. We can use any measure (not only $tf - idf$) to define the document vectors.

Until now, we haven't considered the fact that computing the cosine similarity is not efficient (we need to go through all documents and then sort the results).

Computing Similarity Efficiently

For most documents the similarity score will be very low, so we avoid to compute it for all the documents. We can have an inverted index in which each term has an associated idf_t value. Each posting will have the term frequency $tf_{t,d}$ associated to it (since it depends on both the term and the document). So we can compute the score of each document while traversing the posting lists. Note that if a DocID does not appear in the posting list of any query term its score is score is zero.

Summarising: we walk through the postings in the inverted index for the terms in q, accumulating the total score for each document (we maintain also a idf value for each dictionary term and a tf value for each posting entry). This scheme computes a score for every document in the postings of any of the query terms; the total number of such documents may be considerably smaller than the collection size N.

When we compute similarity score (going across all the posting lists of the terms in the query), we can retrieve the K highest scoring documents using an *heap* structure.

Now we move for being correct but reasonably fast, to be possibly incorrect but fast. So what we want to retrieve is now the K documents that are *likely* to be the K highest scored, hoping to dramatically reduce the cost of computing the K documents we output, without materially altering the user's perceived relevance of the top K results.

The main idea behind all the techniques (heuristics) to find the inexact best K documents is to find a subset A of the documents that is both small and likely to contain documents with scores near to the K highest ranked, return the K highest ranked documents in A.

Note that many of these heuristics requires parameters to be tuned to the collection and application at hand.

Index elimination

Now we have the inverted index in which we associate to each term the $tf-idf$ score, and to each posting the term frequency in the document associated to the corresponding DocID. So we avoid compute the $tf-idf$ for terms with low idf.

In other words, we only consider documents containing terms whose idf exceeds a preset threshold (just doing this, the set of documents for which we compute cosines is greatly reduced). We can view this heuristic as: low-idf terms are treated as stop words and do not contribute to scoring.

The problem is that we have to decide where to cut the search. The cutoff value can be adapted according to the other terms present in the query.

Alternatively we can only consider documents in which most or all the query terms appears (this can be accomplished during the postings traversal), but a problem might be that we do not have at least K documents matching all query terms.

Another possibility is to use the **champion list** (also called *top docs*), so we have two lists for each list, containing the r highest-scoring documents (usually $r > K$) for a term t. For example for the $tf - idf$ weighting, these are the r documents with the highest tf values for term t. The value r is highly application dependent (but it should be much higher than K, especially if we use index elimination). By using champion list we can answer a query only by using the champion list, otherwise we use the normal inverted index.

So we compute the union of the champion lists of all terms in the query q, obtaining a set A of documents and we find the K highest ranked documents in A (computing cosine similarity).

The fact is that we might have too few documents if K is not known until the query is received. There is no reason to have the same value of r for all terms in the dictionary, it could for instance be set to be higher for rarer terms.

Another possibility that combines some kind of external scoring, is that for some documents we can compute a **static quality score** $g(d) \in [0, 1]$ that is independent for the query (e.g. the review score, the number of likes, the number of views), which helps us identify which documents are probably the most relevant. We can combine $g(d)$ with the scoring given by the query (e.g. with linear combination).

We can actually modify the order of the posting list in order to take into account the term frequency (instead of DocID), indeed union and intersection work as long as there is some ordering (is the order that makes them efficient). The idea is to order the documents by decreasing $tf_{t,d}$. In this way the documents which will obtain the highest scoring will be processed first. If the $tf_{t,d}$ value drops below a threshold, then we can stop. This is called **impact ordering**.

Cluster Pruning

In **cluster pruning** we have a preprocessing step during which we cluster the document vectors. Then, at query time, we consider only documents in a small number of clusters as candidates for which we compute cosine scores.

We select a subset of documents in which to compute the similarity score, offline we compute the similarity score between the leader and the other documents. A document is inside the cluster of the nearest leader.

More formally, we randomly select $M = \sqrt{N}$ documents as *leader*. Each leader identifies a cluster of documents. For each of the remaining documents (*followers*) we find the most similar among the M documents selected and we add it

to the corresponding cluster (the expected number of followers for each leader is $\approx \frac{N}{\sqrt{N}}$).

Then we compute the similarity between the query and the leaders, i.e. for a query q we find the document among the M leaders that is most similar to it. The candidate set A consists of L together with its followers, we compute the cosine scores for all documents in this candidate set. Finally the K highest ranked documents are selected among the ones in the cluster of the selected leader.

The fact is that the selection of $\sqrt{N}$ leaders randomly likely reflects the distribution of documents in the vector space: the most crowded regions will have the more leaders, and thus a finer partition into subregions.

A variant more likely to return the "real" K highest ranked document is the following: when creating clusters, each document is associated to b_1 leaders (i.e. it is part of more than one cluster. rather than a single closest leader), when a query is received, the clusters of the b_2 nearest leaders are considered (the basic scheme corresponds to the case $b_1 = b_2 = 1$).

We need as always a tradeoff between being correct and being fast: increasing b_1 or b_2 increases the likelihood of finding K documents that are more likely to be in the set of true top-scoring K documents, at the expense of more computation.

Tiered Indexes

Tiered indexes address the problem of not finding K documents in inexact top-K retrieval. They are a sort of generalisation of champion lists.

Indexes for different term frequencies, we can search in the index for Rank 1, that contains only the terms that contain a certain tf, if we find all K documents we can stop here. Most of the time we will stay in the highest ranked index.

The idea of **term proximity** is that we want to give higher importance to documents where query terms $t_1, t_2, \ldots, t_n$ appears close to each other (indeed this is evidence that the document has text focused on the query intent). We search for the length ω of the window (in term of number of words) in which $t_1, t_2, \ldots, t_k$ all appear. The lower the size of the window, the more relevant the document is.

We can use ω in two ways:

- hand-coding a scoring function;
- as an additional linear term whose weight we can learn from training samples.

Boolean queries

We still may want to answer boolean queries also in the vector space model, this is actually quite easy: we simply have to check what is the value of the term t

for a document d in the vector space model (i.e. we have to check if the entry t of the vector of d is positive), if it not 0 we have at least one term.

In general we cannot use the boolean model to answer queries formulated for the vector space model.

In the Boolean model the queries are written to *select documents* (through the presence of specific combinations of keywords, without introducing any relative ordering among them), in the vector space model (we have some ordering that is missing in the boolean model) queries are a form of *evidence accumulation* (where the presence of more query terms in a document adds tothe score of a document).

Wildcard queries

We want to answer wildcard queries in the vector space model. We can still construct the same separate index (permuterm or k-gram) and return the set of terms that satisfy the wildcards present in the query. Then we construct new queries. A document that contains more terms will have a more acute angle (so it is likely to be scored higher than another containing only one of them). So wildcards query are easy to add to the vector space model.

Phrase queries

In order to perform phrase queries, the problem is that the vector phrase model is incompatible to real phrase queries, since it is a *bag of words* without any ordering, while in phrase queries the ordering is important (the representation of documents as vectors is fundamentally lossy). Thus, an index built for vector space retrieval cannot, in general, be used for phrase queries. Moreover, there is no way of demanding a vector space score for a phrase query, we only know the relative weights of each term in a document.

A possibility is to combine the boolean and the vector space model:

- perform the phrase query and rank only the documents returned by the query;
- if less than K documents are returned, then reduce the phrase query and start again.

We are using the vector space model only for ranking, actually we are using a kind of boolean model to answer the query (we are combining two different ways of representing documents).

Evaluation of IR systems

To evaluate systems we usually want two components:

1. A set of standard benchmarks (collections to make results comparable, with a test suite of information needs, expressible as queries): the **Cranfield collection** (oldest one, too small nowadays) has both the collection of documents and the collection of queries, and which documents are relevant for each query (set of relevance judgement). The **TREC** (Text Retrieval Conference) has a range of text collections on different topics. Another one is the **Reuters** which is a large collection of newspaper articles.

Remark: a document is relevant if it addresses the stated information need, not because it just happens to contain all the words in the query.

2. With ranked retireval the ranking is extremely important: if we retrieve all documents but the first are not relevant it is not good. We must find a way to evaluate ranked retrieval. We need to extend precision and recall. We can use the **precision-recall** curve, the **eleven point** interpolated average precision, the **Mean Average Precision**, **Precision at k** and **R-precision**.

Precision-Recall curve

We take the documents one at a time and compute precision and recall (which are set-based measures: computed using unordered sets of documents) for the first $1, 2, 3, 4$ etc. retrieved documents. The curve has a sawtooth shape, so *interpolated precision* is also used.

Consider the $k+1$-th retrieved document: if it is non-relevant then recall is the same as for the top k documents, but precision has dropped; if it is relevant then both precision and recall increase, and the curve jags up to the right.

We evaluate the interpolated precision for a recall level r: the maximum precision found for all recall levels $r' \geq r$. The justification is that almost anyone would be prepared to look at a few more documents if it would increase the percentage of the viewed set that were relevant (i.e. if the precision of the larger set is higher).

Examining the precision-curve is informative, but there is often a desire to boil this information down to a few number, or even to a single number. The traditional way of doing this is to look at the **eleven-point interpolated precision**.

Eleven point interpolated precision

We can look at the level of precision that we have for different (eleven) measure of recall. We expect when the recall increase the precision to decrease, so the precision decrease when we want more and more documents. Still we don't have a single value (unless we take the average of those values).

Mean Average Precision

MAP provides a single-figure measure of quality across recall levels.

We have a collection of queries $Q = \{q_1, \dots, q_n\}$, for each query we have of collection of documents $\{d_1, \dots, d_{m_j}\}$ that are relevant.

We consider R_{jk} the number of ranked documents retrieved for the j^{th} query before finding the k relevant documents.

We take the average precision across all the queries, computed on the set of all the documents:

$$MAP(Q) = \frac{1}{n} \sum_{j=1}^{n} \left(\frac{1}{m_j} \sum_{k=1}^{m_j} Precision(R_{jk}) \right)$$

When a relevant document is not retrieved at all, the precision value in the above equation is taken to be 0. If always retrieve all the relevant documents the precision will be 1.

For a single information need, the average precision approximates the area under the uninterpolated precision-recall curve, so the MAP is roughly the average area under the precision-recall curve for a set of queries.

The MAP value for a test collection is the arithmetic mean of average precision values for individual information needs, this has the effect of weighting each information need equally in the final reported number, even if many documents are relevant to some queries and very few are relevant to other queries.

Precision at K and R-precision

Precision at k means that we record the precision of the first k retrieved documents (i.e. at fixed low levels of retrieved results). If there are less than k relevant documents than precision cannot be one. Its value is highly dependent on the number of relevant documents that exists.

It has the advantage of not requiring any estimate of the size of the set of relevant documents. The problem is that it does not average well because the total number of relevant documents for a query has a strong influence on precision at k.

To alleviate this problem we use usually the R-precision: if there are R relevant documents for a query, the R-precision is the precision of the top R ranked documents returned by the query (then it adjusts for the size of the set of relevant documents).

For all these measures, we usually evaluate on a collection of queries, not on a single query, then average the results.

Relevance Feedback

The idea is that the user has a set of results (after performing a query) and decides which results are relevant and which are not (the user is so involved in the IR process so as to improve the final result set). The system computes a new set of results based on the feedback from the user. If necessary, the process is repeated. The system is adaptive and everytime receives feedbacks modify the result.

This can help in refining the query without having the user to reformulate the query. This process exploits the idea that it may be difficult to formulate a good query when you don't know the collection well, but it's easy to judge particular documents, and so it makes sense to engage an iterative query refinement of this sort.

If we consider the query as a point in space, we are only performing *local search*, it isn't a *global method* (such as spelling correction, which can bring to a completely different part of the space).

Relevant feedback does not work in case of mispelling (since it is a local method), doesn't help when searching documents in another language, it is also ineffective in case of vocabulary mismatch between the user and the collection (they call the same thing with different terms).

The Rocchio Algorithm

The **Rocchio algorithm** is the classic algorithm for implementing relevance feedback.

The idea is that if we see the documents as points in space, we have the query and the results evaluated by the user. We use this information to compute a new vector to sum to the query and modify the query. Formally: we have a set of relevant documents C_r and a set of non-relevant documents C_{nr}, we want to maximise the similarity of the query with the set of relevant documents while minimising it with respect to the set of non-relevant documents.

The optimal query is that that maximises the difference between the similarity between the query and the relevant document and the similarity between the query and the non relevant documents:

$$\bar{q}_{opt} = argmax_{\bar{q}}[sim(\bar{q}, C_r) - sim(\bar{q}, C_{nr})]$$

If as similarity we use the cosine similarity, then we can rewrite as:

$$\bar{q}_{opt} = \frac{1}{|C_r|} \sum_{\bar{d} \in C_r} \bar{d} - \frac{1}{|C_{nr}|} \sum_{\bar{d} \in C_{nr}} \bar{d}$$

i.e. we are performing the vector difference centroid of relevant documents − centroid of non-relevant documents.

We only have partial knowledge on the relevance of all documents, we have:

- a set D_r of *known* relevant documents;
- a set of D_{nr} of *known* non-relevant documents;
- the original query $\bar{q}_0$ performed by the user.

So we can perform a linear combination of: the centroid of D_r, the centroid of D_{nr} and the original query $\bar{q}_0$„ in such a way that, starting from $\bar{q}_0$ the new query moves some distance toward the centroid of the relevant documents and some distance away from the centroid of the non-relevant documents. This new query can be used for standard retrieval in the vector space model.

We need to select reasonable weights α, β, γ, we consider that:

- positive feedback is more valuable than negative feedback (consider positive feedback more than negative feedback), so $\gamma < \beta$;
- reasonable values might be $\alpha = 1, \beta = 0.75, \gamma = 0.15$. We can also have only positive feedbacks with $\gamma = 0$.

The query is hence updated as:

$$\bar{q}_m = \alpha \bar{q}_0 \frac{1}{|D_r|} \sum_{\bar{d} \in C_r} \bar{d} - \gamma \frac{1}{|D_{nr}|} \sum_{\bar{d} \in C_{nr}} \bar{d}$$

this is the Rocchio algorithm.

Pseudo-relevant feedback

Way of performing relevance feedback without the user (it automates the manual part of the RF, so that the user gets improved retrieval performance without an extended interaction): we perform the query $\bar{q}$ (normal retrieval to find an initial set of most relevant documents) as usual and we consider the first k ranked retrieved documents in the ranking as relevant (we are confident that our system works well at leats for the first k documents, we assume this) and finally do RF as before under this assumption. We need to balance how many times to perform this pseudo-relevant feedback, because it can happen that the modified query moves in a unwanted direction.