MAI4CAREU

Master programmes in Artificial Intelligence 4 Careers in Europe

University of Ruse

# Information Retrieval

**Yordan Kalmukov**

May 2023

Co-financed by the European Union
Connecting Europe Facility

This Master is run under the context of Action No 2020-EU-IA-0087, co-financed by the EU CEF Telecom under GA nr. INEA/CEF/ICT/A2020/2267423

Content-based document retrieval

# Vector Space Model

## I. Vector Space Model - introduction

The vector space model [1] is an implicit method where the meaning of a document is described (or extracted) from its content.

As its name suggests, according to the vector space model, all the documents as well as the query are represented in the form of vectors. However, unlike set-theoretic models, the vectors are not binary, but contain real numbers representing term weights.

For example:

$d_1$ = {0.15, 0.83, 2.12, 0.87, 0, 0, 0, 0.43, 0.25, 0, 0, … }

It is important to note that the size of the vector is not equal to the number of words in the document, but to the number of unique words in the entire document collection. If the number of unique words in the entire collection is *n*, then all the vectors representing the documents will be presented in the *n*-dimensional space. I.e. they will have *n* number of elements. In practice, this means that the vast majority (over 95%) of the elements of each vector will be zeros, because most words will not occur in the corresponding document at all, i.e. their weight with respect to it will be 0.

A relatively small collection of about 5,000 documents, each of 150-200 words, typically results in over 20,000 unique words. Working with multiple vectors consisting of tens, even hundreds, thousands of elements requires a lot of memory and computing resources. Therefore, in practice the model is implemented in a slightly different way with the use of an inverted index. How exactly this is done is described in details later.

The accuracy of the calculated similarity factor between two documents (or between a query and a given document) largely depends on how the term weights are calculated. Numerous term-weighting models for their calculation have been proposed in the scientific literature. All of them are based on two main features:

- Frequency of occurrence of a given word $t_i$ within the j-th document $t_i$. It is called ***term-frequency*** – $\boldsymbol{tf_{i,j}}$. This is a local feature of the word, within the current document only. The presumption is that the more frequently a word occurs within a document, the more important it is to its description and meaning.
- Number of documents containing the given word $t_i$. It is called ***document frequency*** - $\boldsymbol{df_i}$. It represents a global feature of the word with respect to the entire collection of documents. Unlike tf, however, it is an inversely proportional measure of word informativeness. I.e. the more documents this word appears in, the less specific information it carries. Why? Because the words that occur in all documents, regardless of their thematic orientation, are usually conjunctions, prepositions, definite articles (in English), etc. These are the so-called semantically insignificant words. Therefore, when

calculating term weights, not this feature is used, but its "inversion" - ***inverse document frequency*** - ***idf$_i$***. The presumption here is that the fewer documents the word is involved in, the more specific meaning it carries. *idf$_i$* is not 1 over df but is calculated by equation 1.

In general, the term weight is calculated by the product of tf and idf. But they are usually not used directly, with their "raw" values. Why? Imagine you are looking for a "large pizza". If "pizza" occurs 1 time in document $d_1$ and 10 times in document $d_2$, then clearly $d_2$ should have a greater degree of similarity to the query, but not quite 10 times greater. In addition, a word that appears in the document many times can significantly reduce the weight of the other words, which of course is not very desirable. Therefore, when calculating the weights, tf is usually logarithmized. Likewise with idf. There, logarithmization is mandatory (equation 1), because otherwise, if the document collection is very large, idf may have a disproportionately larger value than tf.

$$idf_i = \log(\frac{d}{df_i}) \quad (1)$$

where:

*idf$_i$* – inverse document frequency of $t_i$ in the entire document collection.

*d* – number of documents in the collection.

*df$_i$* – number of documents that contain $t_i$.

The simplest tf-idf model for calculation of term weights is:

$$w_{i,j} = tf_{i,j} * idf_i = \left(tf_{i,j}\right) * \log\left(\frac{d}{df_i}\right) \quad (2)$$

where:

$w_{i,j}$ – the weight of the *i*-th term in the *j*-th document.

$tf_{i,j}$ – term frequency of $t_i$ in the *j*-th document.

The rest notations are already explained.

However, as mentioned, tf is usually also logarithmized rather than used directly. Thus, the most frequently used tf-idf scheme is actually equation (3). It is by no means perfect, but it is easy to implement and gives good results. There are other more complex term-weighting models, some of which are discussed in the next point.

$$w_{i,j} = \left(1 + \log(tf_{i,j})\right) * \log(\frac{d}{df_i}) \quad (3)$$

The weights of the same terms in different documents will be different, because the local frequency of occurrence of the corresponding words within the given document (tf) also takes part in them.

After the term weights are calculated and the document vectors are formed, the degree of similarity between them can be calculated by various similarity measures, including the algebraic versions of Dice (Equation 4) and Jaccard (Equation 5), but most commonly used is the *cosine similarity* – equation 6.

$$Sim_{Dice}\left(q, d_j\right) = \frac{2\sum_{i=1}^{n} w_{qi} w_{d_j i}}{\sum_{i=1}^{n}(w_{qi})^2 + \sum_{i=1}^{n}(w_{d_j i})^2} \quad (4)$$

where:

$Sim(q, d_j)$ – similarity between the query *q* and the dopcument *d<sub>j</sub>*.

$w_{qi}$ – weight of the *i*-th term in the query vector.

$w_{d_ji}$ – weight of the *i*-th term in the *j*-th document vector.

*n* – number of unique terms in the entire document collection.

$$Sim_{Jaccard}(q, d_j) = \frac{\sum_{i=1}^{n} w_{qi} w_{d_ji}}{\sum_{i=1}^{n} (w_{qi})^2 + \sum_{i=1}^{n} (w_{d_ji})^2 - \sum_{i=1}^{n} w_{qi} w_{d_ji}} \quad (5)$$

Cosine similarity calculates the cosine of the angle between the two vectors. If the angle is 0 degrees, ie. vectors coincide, the cosine (similarity) is 1. If the angle between the vectors is 90 degrees, i.e. they have nothing to do with each other, the cosine is 0. The idea is illustrated in figure 1, for the case of vectors represented in two-dimensional space. Of course, if the dimensions are more than 3, there is no way to draw such a graph, but it is not necessary.

$$Sim_{cosine}(q, d_j) = \frac{\sum_{i=1}^{n} w_{qi} w_{d_ji}}{\sqrt{\sum_{i=1}^{n} (w_{qi})^2} \sqrt{\sum_{i=1}^{n} (w_{d_ji})^2}} \quad (6)$$

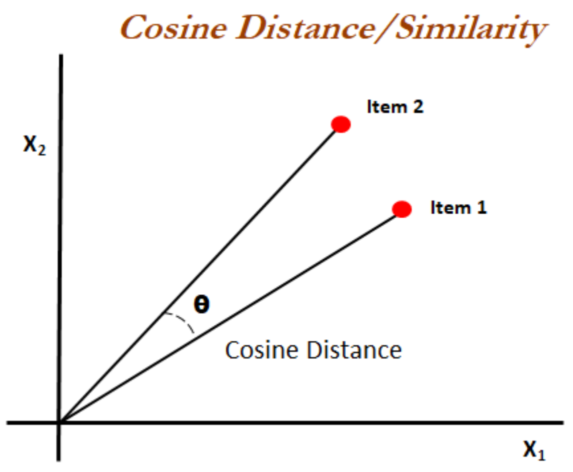


*Figure 1. Illustration of the idea for cosine similarity between two documents by calculating the cosine of the angle between their vectors.*

The numerator of equation 6 represents the dot (scalar) product of the two vectors — sum of the positional product of each element of one vector with the element standing at the same position in the other vector.

The denominator represents the so-called cosine normalization - the value obtained from the numerator is normalized with the product of the lengths of the two vectors. The length of a vector is the square root of the sum of all elements squared. Why is normalization necessary? Because the longer a given document is, the more likely it is to have words from the query, resulting in greater similarity to it than to another shorter document. Since the scalar product is a sum of products, it can grow indefinitely (especially for words with high weights), making it difficult to compare and rank individual similarity factors between documents. Therefore, it is necessary to fit all calculated similarities in the same interval, for example from 0 to 1, so that they can then be objectively compared with each other. This is exactly what normalization does - it fits the resulting value into the range from 0 to 1, regardless of the length of the query and of the documents.

In general, it is possible to normalize terms weights in the vectors before calculating the similarity. This is done when it is necessary to apply (and) another type of normalization. In this case, the denominator of equation 6 can be dropped, but not necessarily – it depends on what normalization is applied to the terms.

Example: Let's there are two vectors:

$d_1$ = {0.15, 0.83, 2.12, 0.87, 0, 0, 0, 0.43, 0.25, 0, 0}

$d_2$ = {0.67, 0, 0, 0.19, 0, 0.98, 1.27, 0.43, 0.53, 0, 0}

Then

Numerator = 0.15*0.67 + 0.83*0 + 2.12*0 + 0.87*0.19+ 0*0 + 0*0.98 + 0*1.27 + 0.43*0.43 + 0.25*0.53 + 0*0 + 0*0 = 0.1005 + 0 + 0 + 0.1653 + 0 + 0+ 0 + 0.1849 + 0.1325 + 0 + 0 = 0.5832

Denominator = 6.2101 * 3.5241 = 21.8850

or

$Sim(d_1, d_2)$ = 0.5832/21.8850 = 0.026648

**II. Term-weighting models**

Since the similarity between two vectors is computed by their dot product, longer documents that have a greater number of non-zero elements in their vectors will logically yield a greater degree of similarity to the query. This is precisely why the normalization in equation 6 is available to reduce this unwanted effect and the influence of the length of the vectors. However, Amit Singhal noticed that as a result of cosine normalization, short documents yield higher similarity factors with the query, and therefore have a higher probability of retrieval [2]. In an attempt to solve this problem, he compares the probability that a document will be retrieved with the probability that it will be adequate, depending on its length. If the two probabilities are plotted in two-dimensional space, ideally the two curves should coincide. However, they do not coincide, but intersect at a point called pivot - figure 2.

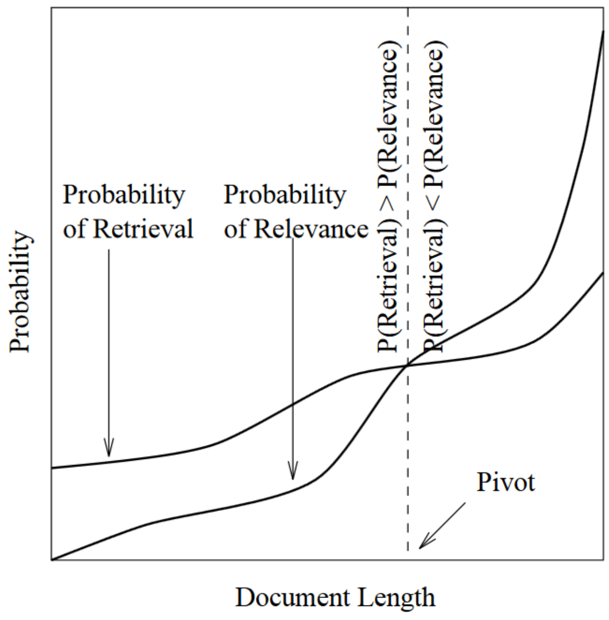


*Figure 2. Curves describing the probability of retrieval and the probability of relevance of the document depending on its length. An illustration of the idea of "pivoted" normalization. The graphic is borrowed from Singhal's publication [2] in which he proposes his model.*

The graphic shows that before the pivot (i.e., for the shorter documents), the probability of the document being retrieved is greater than that of being relevant. And after the pivot – vice versa. I.e. after the cosine normalization, the short documents get a bonus. So, his idea is to find such a correction of the normalization that the curve of the probability of retrieval turns around the pivot and coincides more precisely with the probability of relevance. For this purpose, Singhal suggests applying the correction from (equation 7) [2] to the already calculated term weights. As a result, similarity with short documents decreases, and with long documents increases. In his paper, Singhal doesn't specify a specific model for calculating term weights before correction, but he mentions that the traditional logarithmic normalization of tf does a good job .

$$w_{i,j} = \frac{tf\_idf\ weight}{1 - s + s\dfrac{dl(d_j)}{avdl}} \tag{7}$$

where:

*s* – slope of rotation. $s \in [0,1]$. By default s = 0.2.

*dl(d*$_j$*)* – length of the *j*-th document.

*avdl* – average document length (for the entire collection). Exactly this average length determines the projection of the pivot point onto the abscissa. It is defined as the average number of characters in the documents or the average number of unique words in the documents.

The slope value of 0.2 was proposed by Singhal himself and subsequently experimentally confirmed as the most appropriate by other scientists and experts in their own experiments and projects. In my own experimental studies, it was found that for the specific subject domain (calculation of similarities between papers and reviewers), the value of the slope is not significant. Yes, different values of s lead to different similarity factors, but the relationship between them remains the same in almost all cases. Just to remind that the relationships between the similarities is what determines the order of the results.

In a later publication [3], Singhal proposed to normalize tf by double-logarithmization, which is particularly effective for cases where the same words occur many times in the same documents. Thus, the model for calculating word weights that he proposes takes the form (8).

$$w_{i,j} = \frac{1 + \log\left(1 + \log(tf_{i,j})\right)}{1 - s + s\frac{dl(d_j)}{avdl}} \log(\frac{d+1}{df_i}) \tag{8}$$

Equation 8 represents a term-weighting model that consists of a composite tf function - first a twofold logarithmic normalization of tf is done, then the proposed pivoted normalization. The latter is also called average document length normalization, since the axis is chosen to intersect the abscissa at the location that corresponds to the arithmetic average length of all documents in the collection. The length can be determined by number of characters or number of unique words.

Stephen E. Robertson offers an algebraic variant of his probabilistic model BM 25 [4,5] (equation 9) which also uses a composite tf function, applying Singhal normalization first, then k-concavity normalization.

$$w_{i,j} = \frac{(k_1 + 1)(tf_{i,j})}{k_1\left(1 - s + s\frac{dl\left(d_j\right)}{avdl}\right) + tf_{i,j}} \log(1 + \frac{d}{df_i}) \tag{9}$$

where:

$k_1$ – constant set to 1.2 by default.

My experimental studies show that this seems to be the best algebraic model for computing term weights used by the vector space model.

Rousseau and Vazirgiannis [6] propose another model for computing term weights using a composite tf function (10), which also relies on Singhal's normalization on average document length. Singhal's normalization is performed first, then lower bound δ normalization and finally double logarithmic normalization.

$$w_{i,j} = 1 + \log\left(1 + \log\left(\frac{tf_{i,j}}{1 - s + s\frac{dl\left(d_j\right)}{avdl}} + \delta\right)\right)\log(\frac{d+1}{df_i}) \tag{10}$$

where:

δ – lower bound. In fact, this is a constant. It should be 0.5, if δ is applied right after pivoted normalization (as in this case).

Initially, the lower bound δ is proposed by Lv and Zhai [7]. With it, they try to solve the following situation, which would occur with very long documents: If two documents exist, and in one of them more words from the query are found, and in the other less, but more often, then the similarity of the two to the query would be more or less commensurate. So, it is not clear which will be ranked over the other. Using this lower bound δ, Lv and Zhai want to prioritize the document that contains a higher number of unique words from the query and get a higher degree of similarity with it than the other document that contains less number of words from the query, but which occur more often in it.

According to the experimental studies of Rousseau and Vazirgiannis, their model (10) gives slightly better results than BM 25, but my experiments do not fully confirm this statement. Yes, in some cases their model outperforms BM25, especially in cases without stemming the words, but with stemming in the vast majority of cases BM25 is better.

## III. Influence of IDF on the search accuracy

You have probably already noticed that in the above models the inverse document frequency (idf) of a given word is calculated in different ways.

$$idf_i = \log(\frac{d}{df_i}) \tag{11}$$

$$idf_i = \log(\frac{d+1}{df_i}) \tag{12}$$

$$idf_i = \log(\frac{d}{df_i} + 1) \tag{13}$$

$$idf_i = \log(\frac{d-df_i+0.5}{df_i+0.5}) \tag{14}$$

where

d – number of documents in the collection.

$df_i$ – number of document, containing the term $t_i$.

Option (11) is the so-called "classical" or traditional way of calculating the IDF. However, there is a problem with it - if a word occurs in all documents, then idf will have a logarithm of 1, which is actually 0. I.e. in this case, the resulting value for idf will cause the word to be completely ignored - as if it is not present in the query and/or documents at all. This can be a problem for some highly specialized document collections related to one or a few very close subject areas. And if a search query contains a small number of words, many of which are terms that occur in all documents, then the query may return nothing. Even if this scenario doesn't come true, ignoring specialized terms is not a good idea and will in no way contribute to increasing search accuracy. Therefore, option (12) makes a very slight correction by adding +1 to the numerator, ensuring that idf will always have a non-zero value and the term that occurs in all documents will be given a very low weight, but still not completely ignored. Variant (13) performs a more serious (called “smoothing”) correction, greatly reducing the "depersonalization" of the word, and even if it occurs in all documents, it will no longer receive too low a weight. This actually reduces the impact of idf on words in the query and/or documents.

Variant (14) was originally proposed by Robertson and Sparck Jones for the BM25 model. However, it also has a problem, and Robertson notes it in his publication [4] - if the given word occurs in more than half of the documents, then the idf will become negative, i.e. the word will not just be ignored, but its presence will make the document more likely to be avoided. To overcome this "strange" effect, Robertson suggests simply dropping $df_i$ from the numerator.

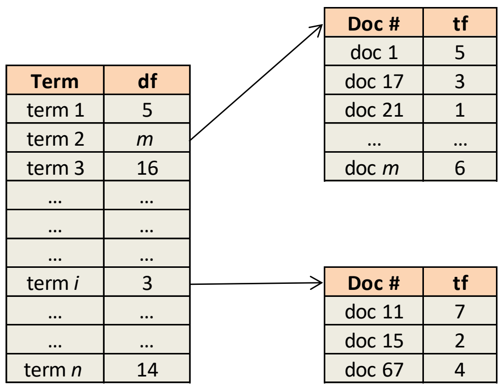
Of the presented options for calculating the idf, the most used are (12) and (13), because they solve the problem of the classical option without creating new ones.

The accuracy and adequacy of the results is influenced not only by the way idf is calculated, but also by which documents it is applied to. It is known from the scientific literature (see [8] and [9]) that the best results are obtained when idf is applied only to the words in the query, but not to the words in the documents. There is logic in that. My experimental studies also support this claim. When inverse frequency of occurrence (idf) is applied to words in a query, it significantly reduces the weight of those that occur in all or most of the documents. And often these are semantically insignificant words and parts of speech such as conjunctions, prepositions, adverbs, pronouns, etc., which were not removed during the pre-processing of the text. In this sense, applying idf to the term weights in the query is reasonable. Applying it to words in documents, however, can greatly reduce the weight of common terms that truly describe the subject area of the documents. This is especially true in the case of the highly specialized document collections discussed earlier. In fact, it is possible for them to ignore the idf, even for the words in the query. Or if idf is applicable, it should be used in its “smoothing correction” option.

## IV. Using inverted index for more efficient implementation

In the vector space model, each document, including the query, is represented as a vector with as many elements as the number of unique words in the entire document collection, i.e. usually tens of thousands. Even more. In this case, about 99% of the elements in a given vector are likely to be zeros, simply because the corresponding words are not present in the document at all, and their tf=0. But even zero takes up space and needs to be processed. Processing is actually the bigger problem. If the number of unique words in the collection is 50,000 (a traditional situation), and the collection contains 5,000 documents, then to calculate the similarity between a query and a document, the dot product between their vectors must be computed and then normalized. But vectors are 50,000 elements in size. I.e. one loop with 50 thousand iterations must be implemented to find the dot product and the lengths of the vectors. That's just to calculate the similarity of the query with a single document! Then all this must be repeated another 4999 times to compute the similarity with the other documents. Obviously, this calculation will happen very slowly. But more importantly, it's also largely meaningless, because 99% of the elements in the vectors will be 0, and so will their "local" product. Therefore, in order to optimize the practical implementation of the VSM, we should use the so-called ***inverted index***.

Instead of storing thousands of vectors with a dimension of tens of thousands of elements, filled almost entirely with zeros, *a dictionary* is constructed. It's like an associative array whose indexes are the unique words themselves. The value of each element is another array that indicates in which document ($d_j$) the given word ($term_i$) occurs and how many times ($tf_{i,j}$). The idea is presented in Figure 3.



| Term | df |
|---|---|
| term 1 | 5 |
| term 2 | *m* |
| term 3 | 16 |
| … | … |
| … | … |
| … | … |
| term *i* | 3 |
| … | … |
| … | … |
| term *n* | 14 |

| Doc # | tf |
|---|---|
| doc 1 | 5 |
| doc 17 | 3 |
| doc 21 | 1 |
| … | … |
| doc *m* | 6 |

| Doc # | tf |
|---|---|
| doc 11 | 7 |
| doc 15 | 2 |
| doc 67 | 4 |

*Figure 3. Inverted index.*

Building an inverted index is easy - when a new document is added to the document collection, each word in it is processed individually. Each is checked if it is already present in the index. If yes, its global frequency of occurrence is updated (df+1) and a new element is added to the array containing information in which documents it occurs and how often. If the word is not present in the index, then it is added, its global df property is set to 1, and a (sub)array is created whose first element contains the ID of the document in which it is found, as well as its frequency of occurrence (tf) in it.

The figure shows multiple arrays (the number of words in the dictionary + 1). The left one is the main one, in which words are stored, and the others store the identifiers of the documents in which these words occur and their local tf characteristics. In practice, however, this complete dictionary can also be implemented in a single dynamic multidimensional data structure of type associative array, object, or structure.

It is known that associative arrays are usually represented as hash tables. Their elements are not stored sequentially in memory, which immediately eliminates the requirement that they have the same size, and hence also eliminates the requirement that they be of the same type. This allows different elements in the array to be of different types and have different sizes. Not only the values can be of different types, but also the indices themselves (in associative arrays, the indices are called keys).

The inverted index could be stored in the following associative array:

```
index[<term>][df] = df;
index[<term>][documents][<docId_1>] = tf_1;
...
index[<term>][documents][<docId_m>] = tf_m;
```

In this case, the array is three-dimensional. The index in the first dimension is the word/term itself (in its respective language, stemmed or not). In the second dimension, there are two possible indices – df and documents. The df element contains the number of documents in which that word occurs. The documents element is another one-dimensional array whose indices are the identifiers of the documents in which the word is contained, and the values - the tf characteristics of the word, relative to the particular document. For example, if word *i* in the figure is "rocket" and it, as indicated in the figure, occurs in 3 documents (11, 15 and 67), this can be written in the three-dimensional associative array like this:

```
    index['rocket']['df'] = 3;
    index['rocket']['documents']['doc_11'] = 7;
    index['rocket']['documents']['doc_15'] = 2;
    index['rocket']['documents']['doc_67'] = 4;
```

Where "rocket" is the dictionary word itself, and "doc_11", "doc_15", and "doc_67" are the actual identifiers of the documents in which the word "rocket" occurs.

The time complexity to create the index is apparently linear with respect to the total number of words (including duplicates) in the document collection. This may seem like a lot at first glance, since in a large collection of long documents, the number of words can reach hundreds of thousands, even millions. But firstly, the dependence is linear, and secondly, every single word of every single document has to be processed individually anyway, regardless of whether it will be searched through an inverted index or the extremely long document vectors will be formed. I.e. the individual processing of every single word from every single document is unavoidable in principle.

Calculating the degree of similarity between the query and the documents in the collection using an inverted index is also easy to implement, but it is not done by applying the similarity formulas quite directly, because when using the inverted index, there are de facto no vectors formed. Instead, a similarity between each word from the query and each document containing it is computed first, then these "partial" similarities are iteratively superimposed to obtain the full similarity between the query and the documents. The process is presented in figure 4.

The main data structures used in the pseudo code are:

**`similarities[docId]`** – an array whose indices are the identifiers of the documents, and in the values of the elements – the cumulative scalar products between the query and the corresponding documents in which the words from the query are found. Since the words in the query are processed iteratively, until the last word is reached, each scalar product is "partial", calculated by summing the products of the weights of the already processed words of the query with the weights of the same words in the corresponding documents. After processing all the words and normalizing the full scalar products, the elements in this array contain the final similarity coefficients between the query and each of the documents identified by its docId identifier. After the algorithm ends, the similarities array can be sorted in descending order before displaying the results. In this way, the documents most similar to the query will be displayed at the top.

**`documentLen[docId]`** – an array whose indices are the IDs of the documents, and in the value of each element the length of the corresponding document is calculated and stored.

**`index`** - a multidimensional associative array that stores the inverted index. The structure of this array is detailed above.

**`calculateQueryWeight()` –** function that calculates the weight of a given word/term from the query, using its local tf and global df characteristics. Any of the models for calculating the term weights described earlier can be implemented in this function.

**calculateDocumentWeight() -** function that calculates the weight of a word/term from the j-th document, using its local tf and global df features. This function is purposefully defined separately from the previous one so that the term weights in the query and in the documents can be calculated by different tf-idf models.

```
===================================================================================

// Adding query to the index
addDocument(query, index);

// Calculating similarity between the query and each document
// The result is stored in similarities[docId] array.

similarities = array(); // initializing as an empty array

documentLen = array();

queryLen = 0;

for every word term_i from the query {

   if (term_i is present in the stopWords array) {

      // Since the word is semantically-insignificant, we skip it
      continue;

   }

   // tf_iq - tf (number of occurrences) of the i-th word in the query
   tf_iq = index[term_i]['documents']['query'];

   // df_i – number of documents that contain the i-th word
   df_i = index[term_i]['df'];

   // Calculating term weight (w_iq) of the i-th word in the query
   w_iq = calculateQueryWeight(tf_iq, df_i);

   // Calculating query length (without the square root)
   // i.e. the first multiplication of the denominator in eq. 6
   queryLen = queryLen + (w_iq * w_iq);

   for every document docId_j, that contains the word term_i
            in the index[term_i]['documents'] array {

      if (empty(similarities[docId_j])) {
         // the cumulative partial similarity between the processed, so far,
         // words from the query with the document docId_j
         // is stored in similarities[docId_j]

         similarities[docId_j] = 0;

         documentLen[docId_j] = 0;

      }

      tf_ij = index[term_i]['documents'][docId_j];

      // Calculating term weight (w_ij)
      // of the i-th word in the j-th document
      w_ij = calculateDocumentWeight(tf_ij, df_i);

      // Calculating partial cumulative
      // scalar (dot) product of the processed so far
      // words from the query
      // with the same words in docId_j,
      // i.e. the numerator of the eq. 6

      similarities[docId_j] =
                     similarities[docId_j] + (w_iq * w_ij);
```

```
        // Calculating vector's length of the j-th document
        // i.e. the second multiplication in the denominator of eq. 6
         documentLen[docId_j] =
                          documentLen[docId_j] + (w_ij * w_ij);

    } // endfor every document docId_j

} // endfor every word term_i in the query


// Calculating final length of the query vector
queryLen = sqrt(queryLen);

// So far we have calculated the numerator of eq. 6
// as well as the two sums for the denominator.
// So the enture eq. 6 should be reconstructed
for every element with index docId in the similarities array {

   if (docId == 'query') continue;

   // Calculating final length of the docId document vector
   documentLen[docId] = sqrt(documentLen[docId]);

   // Calculating cosine normalization
   // i.e. the denominator of eq. 6
   cosNorm = queryLen * documentLen[docId];

   // normalizing the scalar (dot) product
   // with the query and document vectors' length (i.e. cos norm)
   similarities[docId] = similarities[docId] / cosNorm;

}
=================================================================================
```

*Figure 4. An algorithm for content-based textual search by using an inverted index*

Where does the improvement/speedup come from? Because the dot product is calculated not between vectors with the size of the number of *unique words in the collection* (i.e. tens of thousands), but between "vectors" with the size of the number of *unique words in the query*. Thus, in the worst case (if all documents contain the query words), *<number of documents> * <number of unique words in the query>* iterations will be performed. In comparison, if similarity is calculated in the original way using vectors with length the number of words in the collection, then *<number of documents> * <number of unique words in the collection>* iterations will need to be performed. The number of unique words in the collection is typically in the tens of thousands, while the unique words in the query most often range from just a few to a few hundred at most.

**References:**

1. Salton, G., A. Wong , C. S. Yang, A vector space model for automatic indexing, Communications of the ACM, v.18 n.11, p.613–620, Nov. 1975
2. Singhal, A., C. Buckley, and M. Mitra. Pivoted document length normalization. In Proceedings of SIGIR'96, pages 21–29, 1996.
3. Singhal, A., J. Choi, D. Hindle, D. Lewis, and F. Pereira. AT&T at TREC-7. In Proceedings of TREC-7, pages 239–252, 1999.
4. Robertson, Stephen. "Understanding inverse document frequency: on theoretical arguments for IDF." Journal of documentation, vol. 60 no. 5, pp 503–520, 2004.
5. Robertson, S. E., S. Walker, K. Spärck Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In Proceedings of TREC-3, pages 109–126, 1994.
6. Rousseau, F., and M. Vazirgiannis. "Composition of TF normalizations: new insights on scoring functions for ad hoc IR." In Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval, pp. 917-920. 2013.
7. Yuanhua Lv and ChengXiang Zhai. 2011. Lower-bounding term frequency normalization. In Proceedings of the 20th ACM international conference on Information and knowledge management (CIKM '11). Association for Computing Machinery, New York, NY, USA, 7–16. DOI:https://doi.org/10.1145/2063576.2063584
8. Manning, C., P. Raghavan, H. Schütze. An Introduction to Information Retrieval, Cambridge University Press, England, 2009.
9. Grossman, D., Frieder, O. Information Retrieval: Algorithms and Heuristics 2nd Ed. Springer, The Netherlands, 2004, ISBN: 1-4020-3004-5.