Bibliography

[1] S. Gauch, J. Wang, and S. Rachakonda. "A Corpus Analysis Approach for Automatic Query Expansion and Its Extension to Multiple Databases". ACM Transactions on Information Systems, 1999, pp. 250-269.

[2] W. Frakes, and R. Baeza-Yates. "Information Retrieval: Data Structures and Algorithms". Prentice-Hall, 1992.

[3] G. Salton and M. McGill. *Introduction to Modern Information Retrieval.* New York: McCraw-Hill, 1983.

[4] C. Yu, and W. Meng. *Principles of Database Query Processing for Advanced Applications.* Morgan Kaufmann, San Francisco, 1998.

utilize the same list. These locations can be arranged in the order they appear in the document (for example in ascending line number). Given two such lists of two terms, it is rather straightforward to check if there are occurrences of the two terms in close proximities of each other, say the two locations differ by no more than a certain distance. For example, the list of one term can be (2, [4, 24]), indicating that the document has two occurrences of the term and they appear in line 4 and line 24. The list of another term can be (3, [5, 18, 31]). When these two lists are merged, it is easy to identify the pairs of locations which are identical or close to each other. For example, when "4" is compared against "5", the locations are judged to be close, as they differ by 1 only. Then, the next location after "4" in the same list, namely "24", is compared to "5". Since "24" and "5" are very different, the proximity condition is not satisfied by this pair of locations. Then "24" is compared against the next location after "5" in the same list, namely "18". This is repeated until one list runs out. In general, after each comparison of two locations, loc_1 and loc_2, the following situations are possible. $loc_1 = loc_2$, $loc_1 > loc_2$ and $loc_1 < loc_2$. In the first case, if we assume that it is sufficient to match the pair with the closest location, we can compare the next location after loc_1 with the next location after loc_2 in the two lists. In the second case, we compare loc_1 with the location after loc_2. In the last case, we compare the next location after loc_1 with loc_2. In each situation, at least one element can be eliminated from one of the two lists. Thus, if there are n_1 and n_2 elements in the two lists, then the time complexity to identify the pairs of locations which are in proximity is $O(n_1 + n_2)$. By applying this process to pairs of terms in the user query, the number of pairs of occurrences of terms which satisfy the proximity condition can be counted. This count can be used to supplement the computation of similarity between a query and a document. By using the list of locations associated with each term, it is possible to identify sets of occurrences of terms which satisfy certain proximity conditions. For example, the occurrences of three terms within a document need to be no more than 1 line apart.

algorithm. These columns contain the inverted file lists of the query terms. Since a typical query contains only a few terms, this means that typically only the non-zero entries in a few columns are accessed to evaluate a query. This is in sharp contrast with the direct computation method in which the query is compared against all documents. The direct method would require all non-zero entries in all columns need to be accessed. In the proposed efficient method, the needed columns (i.e., inverted file lists) can be located quickly using the hash table (or a B-tree).

2. Let D be a document that has terms in common with a query q. It is typical that most terms in D do not appear in q. It can be seen that the efficient algorithm does not need to process those terms of D which do not appear in the query q.

3. If a document D does not have any term in common with a query q, then document D will not be involved in the evaluation of q by the efficient algorithm. This is because in this case, the inverted file lists of the query terms will not contain any information regarding D. In contrast, the direct computation method needs to compare q with every document, including those documents that do not share any term with q.

1.7.1 Boolean Query Evaluation

The data structures used above for evaluating vector queries can also be used to evaluate Boolean queries efficiently. For a query that requires each qualified document to contain a number of terms (i.e., an and-query), we can first obtain the inverted file list of each query term and then perform an intersection among document numbers. The document numbers in each inverted file list are usually in ascending order so that the intersection can be performed efficiently. Specifically, a multi-way intersection similar to a multi-way merge in merge-sort can be performed. Only one scan of each list is sufficient for the intersection as all lists are sorted. For a query that requires a qualified document to contain at least one query term (i.e., an or-query), a union of the inverted file lists of all query terms can be performed. Again, only one scan of each list is sufficient.

1.7.2 Proximity Query Evaluation

In order to evaluate proximity queries, location information of terms need to be stored. This location information can be placed in an extended inverted list. As an example, consider term t. If document D contains term t with weight w, then there is a pair (D, w) in the inverted file list of t. In order to support the evaluation of proximity queries, this pair can be extended into $(D, (total_occ, [loc_1, loc_2, ..., loc_k]), w)$, where $total_occ$ is the total number of occurrences of the term in the document, and the locations of these occurrences in the document are given by $loc_1, loc_2, ..., loc_k$. Note that term weights may not be needed for processing proximity queries. But they are normally kept so that both proximity queries and vector queries (or hybrid queries) can

	t_1	t_2	t_3	t_4	t_5
D_1	2	1	1	0	0
D_2	0	2	1	1	0
D_3	1	0	1	1	0
D_4	2	1	2	2	0
D_5	0	2	0	1	2

Figure 1.4: A Example Document-Term Matrix

$0.2774, nf[5] = 0.3333, I(t_1) = [(D_1, 2), (D_3, 1), (D_4, 2)], I(t_2) = [(D_1, 1), (D_2, 2), (D_4, 1), (D_5, 2)], I(t_3) = [(D_1, 1), (D_2, 1), (D_3, 1), (D_4, 2)], I(t_4) = [(D_2, 1), (D_3, 1), (D_4, 2), (D_5, 1)], I(t_5) = [(D_5, 2)]$. Let q be a query with two terms t_1 and t_3 and their weights are both 1. The normalization factor for the query is $1/|q| = 0.7071$.

We now apply the algorithm to calculate the similarities of documents with respect to q. For query term t_1, we first obtain $I(t_1)$. Since $I(t_1)$ contains D_1, D_3 and D_4, the following intermediate similarities are obtained after t_1 is processed: $sim(q, D_1) = 1 * 2 = 2, sim(q, D_3) = 1 * 1 = 1, sim(q, D_4) = 1 * 2 = 2$. For query term t_3, we obtain $I(t_3)$ and compute the following intermediate similarities: $sim(q, D_1) = 2 + 1 * 1 = 3, sim(q, D_2) = 1 * 1 = 1, sim(q, D_3) = 1 + 1 * 1 = 2, sim(q, D_4) = 2 + 1 * 2 = 4$. After normalization factors are multiplied, the following final similarities can be obtained: $sim(q, D_1) = 0.87, sim(q, D_2) = 0.29, sim(q, D_3) = 0.82, sim(q, D_4) = 0.78$. ■

A special feature of the above algorithm is that it computes the similarities between a given query and a number of documents at the same time. Specifically, after the first query term is processed, a temporary similarity between every document containing this term and the query will be obtained. If a document contains more than one query term, then this temporary similarity of the document is not the final similarity of the document. In other words, it is only an intermediate result. Only after the last term of the query is processed, we can be sure that the final similarity is computed (subject to multiplying the normalization factors). During the evaluation of a query, the intermediate similarity of a document may be accessed several times for adding contributions by new query terms. There is a need to locate the intermediate result of each query quickly. This can be accomplished by another hash table. If document D_i contains query term t_j, then i is hashed into a bucket in the hash table. If D_i has an entry (i.e., intermediate result) already in the hash table, the intermediate similarity will be in the same bucket and can be updated; otherwise a new entry will be created to contain the intermediate similarity of D_i with q due to term t_j.

The following observations about the above algorithm explains why the algorithm is efficient.

1. For each given query, only the non-zero entries in the columns in the document-term matrix corresponding to the query terms are needed to evaluate the query using the efficient

for any given term the storage location of the inverted file list of the term. Consider the case when a hash table is used. In this case, the hash table consists of a number of buckets. For each term t in the collection, a hash function $h()$ is applied to the term and the hash value $h(t)$ determines the bucket that contains the disk address of the inverted file list $I(t)$.

3. If the similarity function employed by the text retrieval system uses the norm of a document to compute the similarity of the document (e.g., the *Cosine* function), then an array say $nf[n]$ can be used to store the normalization factors of documents, where n is the number of documents in the collection. Specifically, $nf[i]$ stores the normalization factor of the ith document D_i, namely $nf[i] = 1/|D_i|$.

The above data structures permit efficient calculation of similarities of documents with any query q. For each term t in q, the hash function $h()$ is applied to find the address of the inverted file list $I(t)$. For each document-weight pair (D, w) in $I(t)$, where w is the weight of t in D, the similarity between q and D is increased by $q_t * w$, where q_t is the weight of t in q. After all terms in q are processed as above, the normalization factor of each encountered document as well as the normalization factor of the query are multiplied to the accumulated similarity of the document to produce the final similarity. Finally, documents are sorted in descending similarities and the top k documents are displayed for some user specified k (If the user did not specify such a k when submitting his/her query, a system default can be used). This process is summarized into the following procedure:

Algorithm Efficient_Retrieval(q, k)
begin
  initialize all $sim(q, D_i) = 0$;
  for each term t in q
    { find I(t) using the hash table;
     for each (D_j, w_j) in $I(t)$
       $sim(q, D_j) += q_t * w_j$;
    }
  for each encountered document D_j
    $sim(q, D_j) = sim(q, D_j) * (1/|q|) * nf[j]$;
  sort documents;
  display the top k documents;
end;

Example 1.4 An example of document-term matrix is shown in Figure 1.4.

From this matrix, we can obtain $nf[1] = 0.4082, nf[2] = 0.4082, nf[3] = 0.5772, nf[4] =$

sentences S2 and S4 is (null, (0,0,0,0,0,1), (0,1,0,0,1,0), (1,0,0,1,0,0)). Null can be replaced by (0,0,0,0,0,0). A similarity function can then be applied to the two vectors to measure their degree of relatedness. (In [1], a mutual information measure is used instead.) ■

1.7 Efficient Query Evaluation

It is very inefficient to compute the similarity between a user query and every document directly as most documents do not have any terms in common with the query. To illustrate this point more clearly, let us consider the following *document-term matrix*:

	t_1	t_2	$\cdots$	t_j	$\cdots$	t_m
D_1	w_{11}	w_{12}	$\cdots$	w_{1j}	$\cdots$	w_{1m}
D_2	w_{21}	w_{22}	$\cdots$	w_{2j}	$\cdots$	w_{2m}
$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\cdots$	$\vdots$
D_i	w_{i1}	w_{i2}	$\cdots$	w_{ij}	$\cdots$	w_{im}
$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\cdots$	$\vdots$
D_n	w_{n1}	w_{n2}	$\cdots$	w_{nj}	$\cdots$	w_{nm}

Figure 1.3: A Document-Term Matrix

where n is the number of documents in the collection, m is the number of distinct terms of the collection, and w_{ij} is the weight of the jth term in the ith document. If the jth term does not appear in the ith document, then w_{ij} will be zero. Most entries in the matrix would be zero as most documents have relatively small number of distinct terms in comparison to the total number of distinct terms in a large collection. If a query is compared with each document directly to compute the similarity of the document, then all or most non-zero entries in the matrix would need to be accessed. This would be very inefficient. A much more efficient way to compute document similarities is to utilize several special data structures as described below:

1. The first data structure is called an *inverted file index*. For each term t_j, an inverted list of the format $[(D_{j_1}, w_{j_1 j}), ..., (D_{j_k}, w_{j_k j})]$ is generated and stored, where D_{j_i} is the identifier of a document containing t_j and $w_{j_i j}$ is the weight of t_j in D_{j_i}, $1 \leq i \leq k$, and k is the number of documents containing t_j. In other words, the inverted file list of t_j contains the non-zero entries in the jth column in the document-term matrix, plus the corresponding document identifiers. The inverted file index consists of the inverted file list for all terms and is usually stored on disk as the size of the index can be very large. Let $I(t)$ denote the inverted file list of term t.

2. The second data structure is a *hash table* or a B-tree data structure used to quickly locate

word together with these four words make up the window of five words.) Consider the following two sentences.

S1: The beautiful color of the dress is observed.

S2: The beautiful colour of the shirt was observed.

Supposed that the context words are "beautiful", "shirt', "dress" and "observed" in this example (i.e., the stopped words "the", "are" and "of" are not used as context words). In sentence S1, the context of "color" can be represented by a vector $V = (v_1, v_2, v_3, v_4)$, where v_1 and v_2 are the two words to the left of "color" and v_3 and v_4 are the two words to the right. Since "the" is not considered a context word, v_1 is null. v_2 is "beautiful"; v_3 is "dress" and v_4 is "observed", assuming that the non-context words are not included in the context window. Similarly, the context of "colour" in sentence S2 is $W = (w_1, w_2, w_3, w_4)$, where w_1 is null, w_2 is "beautiful", w_3 is "shirt" and w_4 is "observed". By comparing the corresponding positions of the two vectors V and W, $v_2 = w_2$ and $v_4 = w_4$. Thus, we may argue that "color" and "colour" are related, since their contexts are similar. Determining the contexts of the two words based on two sentences is not statistically significant. Thus, we need to find contexts of the two words using all sentences in which these two words occur. This involves all documents in the database. Thus, in general, each context position (say v_i) contains a set of context words which occur in that position relative to the target word. As an example, v_1 is the set of words which are two words to the left of "color". When v_i is compared against w_i, the set of context words in common in the same context position is obtained. In order to facilitate computation, the jth context word may be represented by a vector $(0, 0, ..., 0, 1, 0, ..., 0)$ in which the "1" occurs in the jth position. If a set of context words appear in a context position, say position i of a target word, then the context position will have a vector of the form $(o_1, o_2, ..., o_m)$, where o_j is the number of times the jth context word occurs in the ith context position. If two target words have the jth context word occurring in the ith context position multiple times, then the corresponding o_j's in the two vectors will have positive values. A simple similarity function such as the *Cosine* function can be utilized to compute the degree of relatedness of the two target words, based on their context vectors.

Example 1.3 In addition to the above two sentences, assume that there are 2 other sentences:

S3: The color of the sky is blue.

S4: The colour of the dress is blue.

Let the set of context words be "blue", "dress", "sky", "observed", "shirt" and "beautiful". Let them be ordered from 1 to 6 respectively. Thus, "blue" is represented by the vector $(1, 0, 0, 0, 0, 0)$. The context vector of color in sentence S3 = (null, null, "sky", "blue"). Replacing the words by their vector forms, the vector of color in S3 = (null, null, (0,0,1,0,0,0), (1,0,0,0,0,0)) or simply (null, null, 0,0,1,0,0,0,1,0,0,0,0,0). The corresponding vector for color in sentence S1 = (null, (0,0,0,0,0,1), (0,1,0,0,0,0), (0,0,0,1,0,0)). Combining the effects of the two sentences, the context vector of color is then (null, (0,0,0,0,0,1), (0,1,1,0,0,0), (1,0,0,1,0,0)). Similarly, the context vector of colour in

S2 are better represented by the phrase "programming language". The phrase, if applicable to those documents, is a more precise description than the combination of the individual terms. Documents which are not represented by the phrase will still contain the individual terms, depending on which terms they originally have. Clearly, the document frequency of the phrase is no higher (and usually much lower) than those of the individual terms. In addition, the number of documents which now have either the term "program" or the term "language" is decreased, as some of the these documents are now indexed by the phrase "programming language" instead. In general, a phrase t can be used to index some of the documents in the intersection of the documents containing the individual terms, if the individual terms form a phrase representing t. Those documents having one or more component terms but not the phrase will continue to be indexed by the individual terms. It is also possible that some documents having the phrase may also have isolated instances of some of the individual terms as well.

1.6.3 Finding Related Terms

There are several ways to identify terms which are related to certain terms given in the query. By adding related terms to the query, it may be possible to improve retrieval effectiveness. One common technique to identify related terms is to make use of co-occurrence information. Two terms are related if they co-occur in many documents. For example, the term "snow" and the term "cold" may co-occur in many documents and they can be considered as related terms. To identify such terms automatically, we can measure the deviation in which they actually co-occur from the number of documents they are expected to co-occur as if they occur independently. For example, if the term "snow" occurs in n_1 out of n documents, the term "cold" occurs in n_2 out of n documents, then the number of documents containing both terms is estimated to be $EXP = n * n_1/n * n_2/n$, if the terms are distributed independently. If the number of documents having both terms, n-cooccur, is significantly higher than EXP, say one or two standard deviation away, then the two terms are likely to be related.

Related terms need not co-occur in documents. For example, the term "color" and its British counterpart "colour" may not have a single document containing both terms. However, the two terms are likely to occur in similar contexts. In order to determine such related terms automatically, it is essential to define the contexts. First, a context word of a word w is a word which occurs near w. It is usually a word which occurs frequently in documents. For example, all words which are the most frequent k words, for some integer k, can be used as context words. (In [1], all the k most frequent words are context words, although it may be argued that the stop-words or non-content words should be eliminated.) Less frequent words which we want to determine relationships among them are called target words. The context of a target word consists of a "window" of a set of context words and their positions. For example, if the window size is 5, then the context of a target word consists of two words to the left of the target word and two words to its right. (The target

Experiments with several document collections indicate the the relationship between document frequency of a term and the quality of the term (see Figure 1.2). Poor terms are usually those with high document frequencies, i.e., terms which occur in many documents. Good terms tend to occur in few documents. Yet the best terms are found to be those with medium document frequencies, i.e., they do not occur in too many nor too few documents. The reason that the rare terms are good but not the best terms is as follows. Rare terms do not appear in queries frequently and as a result, their effects on overall retrieval performance may be insignificant.

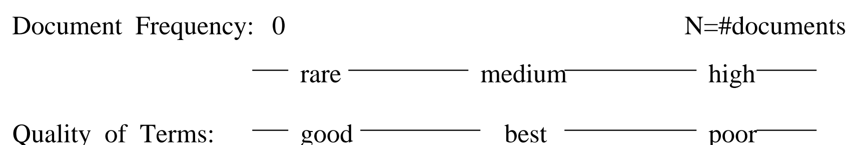

Figure 1.2: Relationship between Term Quality and Document Frequency

Based on the relationship between the quality of terms and their document frequencies, it is desirable to transform terms with both low document frequencies and high document frequencies into terms with medium document frequencies.

One way to transform low document frequency terms into medium document frequency terms is to recognize synonyms and replace them by a single term. As an example, suppose "journal", "periodical" and "magazine" are three terms with low document frequencies. Suppose df(journal) = 15, df(periodical) = 5, df(magazine) = 20, where df(t) denotes the document frequency of term t. Since these terms are synonyms, we can replace them by a single term, say "journal", such that this term will appear in a document whenever "journal", "periodical" or "magazine" appears in the document. After the replacement, "periodical" and "magazine" disappear from the term list of the document collection and if the above three terms appear in different documents, then df(journal) $= 15 + 5 + 20 = 40$. In general, a term t can be used to represent a set of synonymous terms $\{t_i\}$ such that whenever one or more component terms t_i occurs in a document D, t is considered to have occurred in D. Clearly, the document frequency of t is at least as high as that of highest document frequency of its component terms. That is, $df(t) \geq \max_i\{df(t_i)\}$. Note that this transformation may also change the term frequency of a term in a document. For example, if a document contains "journal" 2 times, "magazine" 3 times and "periodical" 0 times, then after the above replacement, the term frequency of "journal" would become 5 and that of "magazine" become 0. Also note that the same replacement should also applied to user queries. For example, if a user query contains "magazine", then the term should be replaced by "journal" before the query is used to calculate similarities.

We can also transform high document frequency terms into medium document frequency terms. As an example, suppose there are two terms "program" and "language", which appear in two sets of documents S1 and S2, respectively. Furthermore, suppose "program" and "language" are high document frequency terms. It is possible that quite a few documents in the intersection of S1 and

where C_1 and C_2 are constants; q' is the modified user query. A common value for C_1 is $1/N_1$, where N_1 is the cardinality of RR; a corresponding value for C_2 is $1/N_2$ where N_2 is the cardinality of RI. Usually, N_2 is significantly larger than N_1. By having this normalization, the vector for the modified query q' will not have most of the terms having negative entries. The additions and the subtractions in the above formula are in vector form. Addition of the documents in RR is to move the query towards the retrieved and relevant documents (i.e., find more documents like them), while the subtraction is to shift the query away from the retrieved and irrelevant documents. By combining the additions and the subtractions, the terms which are responsible for retrieving the relevant documents in RR but not the irrelevant documents in RI will be emphasized. At the same time, those terms responsible for retrieving the irrelevant documents in RI but not the relevant documents in RR will be deemphasized. If the documents which are relevant but not retrieved by q are close to the documents in RR, the modified query q' is likely to retrieve them. If all relevant documents are clustered together, then q' will be able to retrieve relevant documents which are not retrievable by q. If the user is satisfied with the retrieved result by q', then the feedback process is terminated; otherwise, another modified query q'' is generated based on the feedback to the returned documents by q' to retrieve more documents. This process may be repeated more times if the user wants.

There are many proposed relevance feedback algorithms and it is beyond the scope of this book to cover them.

1.6.2 Better Document Representation

One way to improve the representation of documents is to use better terms to index them. The following definition is useful for determining the quality of a term in indexing documents.

Definition 1.1 *Let $sim(D_1, D_2)$ be the similarity between two documents D_1 and D_2 according to some similarity function (say the Cosine function). The compactness of a set of documents in a collection is defined to be $\sum_{i \neq j} sim(D_i, D_j)$, where the summation is over all different pairs of documents.*

Intuitively, if all documents are close together (i.e., have high similarity), then the compactness is high and it is difficult to differentiate the documents of interest from the other documents with respect to a given query. Thus, it is desirable to have a low compactness. Based on this intuition, we can define the quality of a term as follows.

Definition 1.2 *If the removal of a term from all document representations increases (decreases) the compactness of the collection, then the term is a good (bad) term. The more the compactness increases (decreases), the better (worse) the term is.*

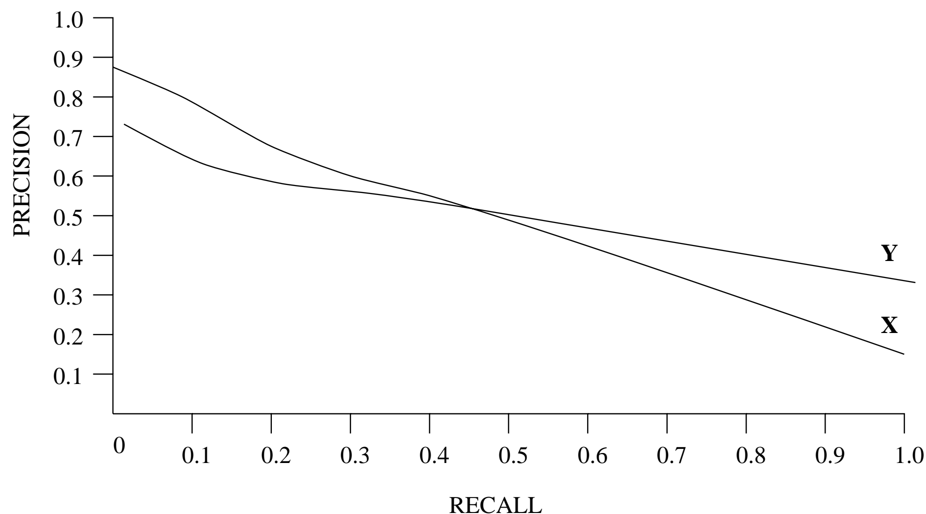

Figure 1.1: Comparing the Effectiveness of Retrieval Systems

3. Document representation and query representation are not perfect. In other words, the representations usually do not preserve the meanings of the original document and query. For example, the employed stemming algorithm may work incorrectly for some terms. As another example, when documents and queries are represented as vectors, information regarding the order in which terms appear would likely be lost.

In this section, we briefly discuss some techniques that have proposed to address the above issues.

1.6.1 Relevance Feedback

Relevance feedback is a mechanism for a user to interact with a text retrieval system to modify the user's original query to a new and better query. A good relevant feedback process has the potential to improve the quality of a user query so that the information needs of the user can be described more precisely by the new query. This in turn can improve the retrieval quality for the query by the retrieval system.

Relevance feedback operates as follows. After a set, R, of documents is retrieved by the text retrieval system for a user query q and is shown to the user, the user is asked to identify among R which documents are relevant. From the response of the user, R is partitioned into two sets, RR, containing the retrieved relevant documents, and RI, containing the retrieved irrelevant documents. The initial query q is then modified, taking into consideration the additional feedback information from the user. One such modification is as follows:

$$q' = q + C_1 * \sum_{D_i \in RR} D_i - C_2 * \sum_{D_j \in RI} D_j \tag{1.5}$$

document, but the user wants to find multiple relevant documents. Thus, the effectiveness of a system should be measured based on the precisions at different recall levels. As each recall and precision pair can be plotted as a point in the plane as determined with recall being the x-axis and precision being the y-axis, a curve can be plotted based on a number of recall and precision pairs.

To evaluate the effectiveness of a text retrieval system, a set of test queries is used. For each query, the set of relevant documents is identified in advance. For each such query, a recall-precision curve can be obtained. The average recall-precision curve over all test queries is used as the measure of the effectiveness of the system.

If a text retrieval system is perfect, then its precision should stay at 1 for all recalls. This essentially means that for every query the system must rank each relevant document ahead of all irrelevant documents. In practice, it is nearly impossible to achieve perfect effectiveness. Usually, as recall increases, precision decreases. In other words, when the number of relevant documents to be retrieved increases, the number of documents to retrieve for each additional relevant document also increases. The first few relevant documents may require modest effort to retrieve, but later relevant documents require more and more efforts. In comparing the retrieval effectiveness of two systems, two recall-precisions curves for the two systems are plotted. The system whose curve is above the other is a better system, because at any given recall, the precision is higher for this system. It is possible that the two curves would intersect at certain locations. The intersections partition the range of recall from 0 to 1 into different intervals such that within each interval, one curve is above the other. Thus, for a given interval I, if curve A is above curve B, then the system which produces curve A is better than the other system within the interval I.

In Figure 1.1, the two curves represent the performance of two text retrieval systems. It can be seen that System X is better than System Y when the recall is in interval [0.0, 0.4], but the reverse is true when the recall is in the interval [0.5, 1.0].

1.6 Improving Retrieval Effectiveness

There are many reasons that the perfect effectiveness is very hard to achieve. Some of the reasons are as follows.

1. User's inability to describe information needs precisely. In practice, most users either cannot describe their information needs precisely or don't bother to spend too much time to precisely describe their information needs.

2. Many terms have multiple meanings and different terms may have similar meanings. Term matching based on exact spellings is not capable of dealing with inaccuracies caused by these problems.

together, then only occurrences of the two terms in proximity of each other in a document satisfy the constraint imposed by the query.

1.5 Measuring Retrieval Effectiveness

For a query submitted by a user, a document is said to be *relevant* if it is determined by the user to be useful; otherwise, the document is said to be *irrelevant*. For a give query, a text retrieval system computes the similarity of each document with the query and displays documents in descending similarity values. As users usually examine documents in the order in which the documents are displayed, this order can have a big impact on the ultimate performance of the system. For example, if many relevant documents appear near the top of the displayed list, then they are likely to be identified by the user. On the other hand, if very few relevant documents are displayed near the top, then many relevant may be missed. Thus, intuitively, a good text retrieval system should compute similarities in such a way such that when documents are displayed in descending similarity values, relevant documents are displayed ahead of irrelevant documents.

From the discussions in previous sections, we can see that the similarity between a document and a query depends on several factors such as the indexing scheme used, the term weighting formula used and the similarity function employed. It can also depend on whether term proximity information is used and how it is used. Different choices of the above factors lead to different text retrieval systems. The question here is how to compare the performance or effectiveness of different text retrieval systems.

A common measure for retrieval effectiveness is *recall* and *precision*. For a given query submitted by a user, suppose that the set of relevant documents with respect to the query in the document collection can be determined. The two quantities recall and precision can be defined as follows:

$$recall = \frac{the\ number\ of\ retrieved\ relevant\ documents}{the\ number\ of\ relevant\ documents} \tag{1.3}$$

$$precision = \frac{the\ number\ of\ retrieved\ relevant\ documents}{the\ number\ of\ retrieved\ documents} \tag{1.4}$$

Example 1.2 Suppose for a given query, there are 12 relevant documents in the collection. When 10 documents are retrieved, 8 are identified as relevant. In this case, recall = 8/12 = 2/3 and precision = 8/10 = 0.8. Suppose when 20 documents are retrieved, 12 are identified as relevant. Then in this case, recall = 12/12 = 1 and precision = 12/20 = 0.6. Usually, when recall increases, precision decreases. ■

A recall of 100% may not indicate excellent retrieval because the system may simply return all documents to the user who does not have the time to read these documents. A precision of 100% does not necessarily indicate good retrieval results, as the system may return a single relevant

Numerous other similarity functions exist. Usually, they return values between 0 and 1, indicating no match and perfect match respectively. Very often, proximity information about terms can be utilized to yield more relevant documents. For example, consider two documents D_1 and D_2 which have exactly the same length and exactly the same set of terms in common with a query and the normalized weights of the common terms are identical in both documents. Suppose in document D_1, the set of common terms appear in close proximity, while in document D_2, they appear in different locations which are far apart. It is possible to argue that document D_1 is semantically more similar to the query than document D_2 and a good similarity function, say $sim()$, would assign a higher similarity to D_1 than D_2. As an example, suppose the query is "programming language". In document D_1, the phrase "programming language" appears in a sentence (note that after stemming is applied, "programming" becomes "program"), while in document D_2, the phrase "TV program" appears in one sentence and "the German language" appears in another sentence. In this example, it is likely that the document D_1 is useful to the user, but the document D_2 is not. One way to implement a similarity function which takes into consideration the proximity of words is as follows. First, compute the similarity between the query and the documents using a standard similarity function such as $Cosine$. This will have the similarity of D_1 with the query identical to that of D_2. Second, each document is pre-partitioned into chunks. For example, for a book, each chapter is a partition of a book. A finer partition would have sections within a chapter as partitions. Then, the same similarity function is applied to each partition. In the case of D_1, there is a partition of the document which have all the common terms between the document and the query. As a result, the similarity of this partition with the query will be high. In the case of D_2, since the common terms are spread out over different partitions, the similarity of the query with any one partition in D_2 will be small. Thus, the partition in D_1 with all the common terms will be retrieved ahead of any partition in D_2. In other words, the retrieval process is a two step process. First, the similarities of whole documents with the query are obtained. Those documents which are reasonably similar to the query will undergo the second step. In the second step, the similarities of the partitions of each retrieved document with the query are computed. Those partitions which yield high similarities (possibly together with the whole documents containing them) are presented to the user.

The above scheme is by no means the only way to implement the computation of similarities which take into consideration the proximities of terms. Assume that each occurrence of a term is associated with a location. It could be a logical one such as a chapter number, a section number within a chapter, a paragraph number within a section and a sentence number within a paragraph. It could be a physical one such as a line number within the document. Based on the location information, it is possible to compute the degree of proximity between the occurrences of two terms. For example, if the two occurrences are in the same line or differ by at most one line, then they are assumed to be in proximity of each other. Thus, if a query requires two terms to be close

two vectors. One simple similarity function is the following dot product function:

$$dot(q,d) = \sum_{k=1}^{n} q_i * d_i \tag{1.1}$$

where $q = (q_1, ..., q_n)$ is a query and $d = (d_1, ..., d_n)$ is a document. When the two vectors are binary vectors, i.e., containing 0's and 1's only, where 0 and 1 represent the absence and presence of a term in the document/query, the similarity given by the dot product function yields the number of terms in common between the two vectors. If the ith term is present in both vectors, the contribution to the dot product due to this term is 1; otherwise, the contribution is 0. The dot product function sums the contributions due to the n terms. In practice, the terms are weighted, permitting more important terms to contribute more to the similarity. Thus, the dot product function is a weighted sum of terms in common between the two vectors. A main drawback of the dot product function is that it tends to favor long documents over short documents. The reason is that the chance of having more terms in common between a document and a given query is higher for a longer document than a shorter document as a longer document has more terms. Another drawback of the dot product function is that similarities computed using this function has no clear upper bound.

One popular way to overcome the problems associated with the dot product function is to divide the dot product by the product of the lengths of the two vectors, namely the document vector and the query vector. The new similarity function is known as the *Cosine* function [3].

$$Cosine(q,d) = \frac{dot(q,d)}{|q| * |d|} \tag{1.2}$$

where $|q|$ and $|d|$ are respectively the lengths (or norms) of the query vector and the document vector. For a given vector $X = (x_1, ..., x_n)$, its length is defined by $|X| = \sqrt{x_1^2 + ... + x_n^2}$. The Cosine function overcomes the problems of the dot product function as follows. First, a longer document will have a large norm and since the norm is the denominator of the fraction it will compensate the likely larger dot product achieved by the document. As a result, longer documents will no longer be favored over shorter ones. Second, if the weight of each term in a document or a query is non-negative, then the *Cosine* function returns a value between 0 and 1. It gets the value 0 if there is no term in common between the query and the document; its value is 1 if the query and the document are identical or one vector is a positive multiplicative constant of the other. In fact, the Cosine function measures the angular distance between the query vector and the document vector. Specifically, if θ is the degree of the angle between the query vector and the document vector, then $Cosine(q,d) = cos\theta$. In practice, we are mostly interested in the relative similarity values of documents with respective to a given query so that the documents can be ranked in descending similarity values. Since all similarities computed based on the Cosine function has the same factor $1/|q|$ for a given query, $|q|$ could be dropped (i.e., replaced by 1). However, such a replacement may yield similarities not upper bounded by 1.

$tf > 0$, where max_tf is the largest term frequency for all terms in the document; otherwise, the weight is 0. In the formula, c_1 and c_2 are constants. The choice $c_1 = c_2 = 0.5$ is widely used. The second factor affecting the weight of a term is the *document frequency* (df), which is the number of documents having the term. Usually, the higher the document frequency, the less important the term is in differentiating different documents. In the extreme case where all documents have a term, then this term is not very useful for differentiating one document from another. In the other extreme case where there is only one document having the term, the document having the term can be easily distinguished from other documents. Thus, the weight of a term based on its document frequency is usually monotonically decreasing and is called the *inverse document frequency weight* ($idfw$). One such formula is $\log \frac{N}{df+c}$, where N is the total number of documents and c is a constant (say 1). The weight of a term in a document can be the product of its term frequency weight and its inverse document frequency weight, i.e., $tfw * idfw$.

1.3 Query Representation

A query is simply a question written in text. Based on the vector query model, each query can be transformed into an n-dimensional vector as well. Specifically, the non-content words are eliminated by comparing the words in the query against the stop list. Then, words in the query are mapped into terms and finally, terms are weighted based on term frequency and/or document frequency information. While term frequency weights are used in both queries and documents, the inverse document frequency weight is usually used in either the queries or the documents but not both. Thus, the weight of a term in a document can be its term frequency weight and the weight of the term in a query can be the product of its term frequency weight and its inverse document frequency weight.

Although the terms submitted in a user query are indicative of the content of the documents the user wants to retrieve, documents of interest may not be retrieved based on these terms only. One reason is that a word in different contexts may have different meanings. Another reason is that synonyms or related terms but not the exact terms used in the query are present in the desired documents. If these related terms are added to the query, then the query is represented in a better context, resulting in the retrieval of more relevant documents. This is known as *query expansion*.

1.4 Document/Query Matching

After documents and query vectors are formed, document vectors which are close to the query vector are retrieved. A *similarity function* can be used to measure the degree of closeness between

more terms with similar meanings can be matched). A stemming program may sometimes yield incorrect results. Specifically, variations of the same words may be mapped to different terms and semantically different words can be mapped to the same term.

Several other simple techniques for mapping words to terms are also widely used. These include converting upper-case characters to lower-case characters and the removing of special symbols such as hyphen and punch signs. In a later section, additional techniques for determining terms will be discussed.

After the above steps, each document can be represented as a set of terms.

Example 1.1 Consider a document d = "The number of Web pages on the World Wide Web was estimated to be over 800 millions in 1999." The stop-words in this document are "the", "of", "on", "was", "to", "be", "over" and "in" and they should be removed. After the application of the stemming algorithm, the words "pages", "estimated" and "millions" become "page", "estimat" and "million", respectively. After converting upper-case characters to lower-case ones, this document can be represented by the following index terms (800, 1999, estimat, million, number, web, wide, world). ■

1.2.2 Term Weighting

As discussed in the above, each document can be represented as a vector of its index terms. Intuitively, we can see that not all index terms of a document have the same importance in representing the content of the document. While it might be possible to manually assign an appropriate importance factor (weight) to each term in a document, this method is not practical as it is too time-consuming. In this subsection, we introduce a widely used automated technique for assigning weights to terms.

For ease of discussion, each document is logically represented as a vector of n terms, where n is the total number of terms in the collection of all documents in a text retrieval system. The terms can be ordered in certain way (say in alphabetical order). Suppose the document d is represented by the vector $(d_1, ..., d_i, ..., d_n)$. Then, d_i is 0 if the ith term is absent from the document and it takes on a positive value if the ith term is present in the document. Most of the entries in the vector will be 0 because most terms are absent from any given document. The value of d_i is called the *weight* of the ith term in document d. Ideally, the weight of a term in a document should indicate the importance of the term in representing the content of the document.

When a term is present in a document, the weight assigned to the term in the document is usually based on two factors. The *term frequency* (tf) of a term in a document is the number of times the term occurs in the document. Intuitively, the higher the term frequency of the term, the more important the term is in representing the content of the document. As a consequence, the *term frequency weight* (tfw) of the term in the document is usually a monotonically increasing function of its term frequency. A possible weighting formula is $c_1 + c_2 * \frac{tf}{max_tf}$, if the term frequency

1.2 Document Representation

In order to facilitates efficient and accurate retrieval, documents in a text retrieval system need to be first preprocessed and then represented in suitable format. A text retrieval system typically characterizes each text document by a set of keywords. When a user query contains some keywords, documents having some or all of these keywords are retrieved. We now outline the process to characterize the documents. This process consists of primarily two steps. The first step is to determine what terms to use to represent the content of each document and this is known as *document indexing*. The second step is to determine the relative importance (or weight) of each term in a document representation and this is known as *term weighting*. Finally, the matching of documents against the query is performed to obtain the similarities of the documents with the query.

1.2.1 Document Indexing

A text document is usually rather lengthy and it often contains a high percentage of non-content words such as "a", "of" and "is". Keeping these words in the representation of a document will not only increase the overhead of storing the document but will also have an adverse effect on the retrieval accuracy as matching on non-content words can distract the attention from the matching of important content words. For example, a document matching many non-content words but few content words with the query may be ranked higher than another document having few non-content words but more content words in common with the query. To solve this problem, non-content words are typically not used to represent documents. In practice, non-content words can be identified in advance and manually placed in a collection of words called the *stop list*. Words appearing in the stop list are then discarded from document representations. The remaining words in the document are content words and can be used to represent the document. A possible way to identify words in the stop list automatically is to find for each word the number of documents having the word and then set a threshold. Any word in which the number of documents containing it exceeds the threshold is assumed to belong to the stop list.

Many words have different variations but with similar meanings. An example is the words "beauty", "beautiful" and "beautify". Due to their differences in spelling, word variations prevent words with similar meanings to be matched. A commonly used technique to deal with this problem is to apply a stemming algorithm to map word variations with the word stem into a single word (i.e., the stem). There are stemming programs such as Porter's stemming program [2] which perform such a task by removing some suffix of a word and possibly replace it by some other characters. For example, the above three words may be mapped to the term "beaut". A good stemming program can map most words to terms correctly and as a result can improve the efficiency of query evaluation (as fewer distinct terms need to be matched) and the retrieval accuracy (as

details). This is in sharp contrast with the Boolean query model where a document either satisfies to fails a query. This feature of the vector query model allows documents to be ranked according to how well they match with a given query. Furthermore, better matched documents can be displayed first to the user to facilitate the finding of useful documents. The ranking also makes it possible to control the number of documents in the result for each query. For example, if a user only wants 10 documents in the result, then the 10 highest ranked documents can be displayed.

Vector queries are most widely used and supported on Web-based search engines. Throughout this book, vector queries will be the assumed query model in our discussion unless the otherwise is explicitly stated.

1.1.3 Proximity Query Model

A common weakness of the Boolean query model and the vector query model is that they do not take the closeness of query terms in documents into consideration. Intuitively, when query terms appear close to each other in a document, the document is more likely to be useful. For example, consider a query with two terms "space shuttle". The two terms have a special meaning when used together as a phrase. If the closeness of the two terms is ignored, then the statement "There is still space on that shuttle bus." will match the query as well as the statement "The space shuttle Challenger is taking off.". But clearly the latter statement should be a better match.

Proximity queries specify how the closeness of query terms should be taken into consideration when retrieving documents. Sometimes the order of the occurrences of query terms will also be utilized. As an example, the proximity query "proximity(t1, t2, 3)" can be used to retrieve all documents that contain both terms t1 and t2 with the two terms not being separated by more than 3 words. A phrase query can be considered as a special case of a proximity query. For example, the phrase query "space shuttle" can be used to find all documents that contain both terms "space" and "shuttle" with the term "space" appearing immediately before the term "shuttle". Most text retrieval systems support certain types of proximity queries such as phrase query.

Just as in the case of Boolean queries, documents satisfying a proximity query are usually not ranked.

In principle, it is possible for a text retrieval system to support combined query models so that the advantages of different query models can be utilized. For example, a query (t_1, w_1), (t_2, w_2) and $(t_1, t_2$, proximity 3) may mean documents retrieved by the query need to have both terms with no more than 3 words apart. Furthermore, these documents are ranked using the weights assigned to the terms in the query as well as weights assigned to the terms in the documents.

example, to retrieval documents that contain either the two terms t1 and t2 or term t3, as well as term t4 but not no term t5, the Boolean query can be expressed as "((t1 and t2) or t3) and t4 and not(t5)". It can be shown that any Boolean query can be transformed into one of two normal forms: the *conjunctive normal form* and the *disjunctive normal form*. The former connects or-clauses by "and" operators while the latter connects and-clauses by "or" operators. An or-clause contains one or more terms (or the negation) and uses the "or" operator to connect them and an and-clause contains one or more terms (or the negation) and uses the "and" operator to connect them. As an example, the above Boolean expression can be transformed into the conjunctive normal form "(t1 or t3) and (t2 or t3) and t4 and not(t5)" with 4 or-clauses. The same above Boolean expression can be transformed into the disjunctive normal form "(t1 and t2 and t4 and not(t5)) or (t3 and t4 and not(t5))" with 2 and-clauses.

Boolean queries are widely used in text retrieval systems. The Boolean query model has several advantages. First, it has good expressive power in terms of representing users' information needs. Second, it makes the retrieval deterministic. In other words, for any Boolean query, a document either satisfies it or not satisfies it. However, the Boolean query model also has several drawbacks. First, non-trivial Boolean queries are difficult to form and understand. Users need to be trained to write complex Boolean queries correctly. Second, documents satisfying a Boolean query cannot be ranked as they are all considered to satisfy the condition equally well. This is somewhat counter-intuitive as it is likely that some documents will match the user's information needs better than others. Third, a user can either get too many or too few results depending on how many documents can satisfy his/her Boolean query. Should it be possible to rank results, the user may be able to control the result size by requesting the system to return only the top k ranked documents for some desirable k.

1.1.2 Vector Query Model

A vector query contains one or more terms and there are no special operators. It can be entered either as an English sentence from which terms (content words) are extracted or as a set of terms directly. Each query can be represented as a vector of terms with weights where the weight of a term indicates the importance of the term in the query. For example, consider the query "text analysis and text retrieval". This query has three distinct content words, namely "text", "analysis" and "retrieval". The other word "and" is a non-content word (also known as *stopword*). Suppose among the three content words, "analysis" and "retrieval" are equally important and "text" is twice as important as the other two words as it appears twice in the query while the other two words appear only once. As a result, this query can be represented as ((analysis, 1), (retrieval, 1), (text, 2)) or simply as (1, 1, 2) if we know the order of these words.

A document can satisfy a vector query to different extents depending on several factors such as the number of query terms it contains and the importance of these terms (see Section 1.2.2 for

Chapter 1

Text Retrieval

A large amount of digital data are in text format. They can be text files in personal computers or web pages on the World Wide Web. They usually have less structure than formatted data in a traditional database such as a relational database. Sometimes they are referred to as unstructured data. *Text retrieval* (also known as *information retrieval* and *document retrieval*) is an area in computer science that addresses issues or techniques for the accurate and efficient retrieval of text documents. In this chapter, we provide a short introduction to this area. The content in this chapter is crucial for understanding later chapters.

1.1 Query Models

When a user wants to find some documents from a text retrieval system, he/she needs to submit a query to the system. The query should indicate the information need of the user. There are primarily three query models for users to express their information needs and they are the *Boolean query model*, the *vector query model* and the *proximity query model*. It is possible to form a query that has components in different query models.

1.1.1 Boolean Query Model

Queries in this model use Boolean operators *and*, *or* and *not* to connect query terms. For example, the Boolean query for retrieving all documents that contain both "text" and "retrieval" can be expressed as "text and retrieval" and the query for retrieving all documents that contain either "text" or "retrieval" or both words can be expressed as "text or retrieval". The query for retrieving all documents that contain "text" but not "retrieval" can be expressed as "text and not(retrieval)".

In general, a Boolean query is a Boolean expression of terms and Boolean operators. Any single term is a Boolean expression and if B, B1 and B2 are Boolean expressions, so are "B1 and B2", "B1 or B2" and "not(B)". A single Boolean query can contain zero or more Boolean operators. For