Chapter 9

Information Retrieval and Search Engines

> "Making a wrong decision is understandable. Refusing to search continually for learning is not."—Phil Crosby

9.1 Introduction

Information retrieval is the process of satisfying user information needs that are expressed as textual queries. Search engines represent a Web-specific example of the information retrieval paradigm. The problem of Web search has many additional challenges, such as the collection of Web resources, the organization of these resources, and the use of hyperlinks to aid the search. Whereas traditional information retrieval only uses the content of documents to retrieve results of queries, the Web requires stronger mechanisms for quality control because of its open nature. Furthermore, Web documents contain significant meta-information and zoned text, such as title, author, or anchor text, which can be leveraged to improve retrieval accuracy. This chapter discusses the following aspects of information retrieval:

1. What types of data structures are most suitable for retrieval applications? The classical data structure for enabling search in text is the inverted index, and it is surprisingly versatile in handling various types of queries. The discussion of the inverted index will be paired with that of query processing.

2. The additional design issues associated with Web-centric search engines will be discussed. For example, we will discuss the collection of document resources from the Web, which is referred to as *crawling*.

3. How does one decide which Web documents are of high quality? Documents that are pointed to by many other pages are often considered more reputable, and it is desirable to assign such documents higher ranks in the search results.

© Springer International Publishing AG, part of Springer Nature 2018

C. C. Aggarwal, *Machine Learning for Text*,
https://doi.org/10.1007/978-3-319-73531-3_9

4. Given a search query, how does one score the matching between the keywords and the document? This is achieved with the use of *information retrieval models*. In recent years, such models have been enhanced with machine learning techniques in order to account for user feedback.

From the aforementioned discussion, it is evident that the Web-centric application of information retrieval (i.e., a search engine) has several additional layers of complexity. This chapter will discuss these additional layers.

The query processing can either provide a 0-1 response (i.e., *Boolean* retrieval), or it can provide a score that indicates the relevance of the document to the query. The Boolean model is the traditional approach used in classical information retrieval in which all results satisfying a logical keyword query are returned. The scoring model is more common for queries on very large document collections like the Web, because only a tiny fraction of the top-ranked results are relevant. Although thousands of Web pages might exactly match the keywords specified by the user, it is crucial to rank the results with various relevance- and quality-centric criteria in order to ease the burden on the user. After all, a user cannot be expected to browse more than ten or twenty of the top results. In such cases, quality-scoring techniques and learning techniques on user feedback are often used to enhance the search results. Although traditional forms of information retrieval are unsupervised, a supervised variant of information retrieval has gained increasing attention in recent years. Search can be viewed as a ranking-centric variant of classification. This is because a user query to a document collection is a binary classification problem over the entire corpus in which a label of "*relevant*" indicates that the document is relevant, and a label of "*non-relevant*" indicates otherwise. This is the essence of the learning-to-rank approach, which is also discussed in this chapter.

9.1.1 Chapter Organization

This chapter is organized as follows. Indexing and query processing are discussed in the next section, whereas scoring models are covered in Sect. 9.3. Methods for Web crawling are discussed in Sect. 9.4. The special issues associated with query processing in search engines are discussed in Sect. 9.5. The different ranking algorithms such as *PageRank* and *HITS* are discussed in Sect. 9.6. A summary is given in Sect. 9.7.

9.2 Indexing and Query Processing

Queries in information retrieval are typically posed as sets of keywords. The older *boolean retrieval systems* were closer to database querying systems in which users could enter sets of keywords connected with the "AND" and "OR" clauses:

text AND *mining*
(*text* AND *mining*) OR (*recommender* AND *systems*)

Each keyword in the aforementioned expression implicitly refers to the fact that the document contains the relevant keywords. For example, the first query above can be viewed as the *conjunct* of two conditions:

(*text* $\in$ Document) AND (*mining* $\in$ Document)

Most natural keyword queries in information retrieval systems are posed as conjuncts. Because of the ease in providing keywords as sets of relevant terms, it is often implicitly assumed that a query like "*text mining*" really refers to a conjunct of two conditions without explicitly using the "AND" operator. The use of the "OR" operator is increasingly rare in modern retrieval systems both because of the complexity of using it, and the fact that too many results are returned with queries containing the "OR" operator unless the individual conjuncts are very restrictive. In general, the most common approach is to simply pose the query as a set of keywords, which implicitly uses the "AND" operator. However, search engines also use the relative ordering of the keywords when interpreting such queries. For example, the query "*text mining*" might not yield the same result as "*mining text.*" For simplicity in discussion, we will first discuss the case in which queries are posed as sets of keywords that are implicitly interpreted as conjuncts of membership conditions. Later, we will show how to extend the approach to more complex settings.

In all keyword-centric queries, two important data structures are commonly used:

1. *Dictionary:* Given a query containing a set of terms, the first step is to discover whether that term occurs in the vocabulary of the corpus. If the term does occur in this vocabulary, a pointer is returned to a second data-structure indexing the documents containing this term. The second data structure is an *inverted list*, which is a component of the *inverted index*.

2. *Inverted index:* As the name implies, the inverted index can be viewed as an "inverted" representation of the document-term matrix, and it comprises a set of inverted lists. Each inverted list contain the identifiers of documents containing a term. The inverted index is connected to the dictionary data structure in the sense that the dictionary data structure contains pointers to the heads of the inverted lists of each term. These pointers are required during query processing.

For a given query, the dictionary is first used to locate the pointers to the relevant term-specific inverted lists, and subsequently these inverted lists are used for query processing. The intersection of the different inverted lists provides the list of document identifiers that are relevant to a particular query. In practice, too many or too few documents might satisfy all query keywords. Therefore, other types of scoring criteria such as partial matches and word positions are used to rank the results. The inverted index is versatile enough to address such ranking queries, as long as the scoring function satisfies certain convenient *additivity* properties with respect to the query terms. In the following, we will describe these query processing techniques together with their supporting data structures.

9.2.1 Dictionary Data Structures

The simplest dictionary data structure is a hash table. Each entry of the hash table contains the (1) string representation of the term, (2) a pointer to the first element of the inverted list of the term, and (3) the number of documents in which the term occurs. Consider a hash table containing N entries. The data structure is initialized to an array of NULL values. The hash function $h(\cdot)$ uses the string representation of the term t_j to map it to a random value $v = h(t_j)$ in $[0, N-1]$. In the event that the vth entry in the hash table is empty, the term t_j is inserted as the vth entry in the hash table. The main problem arises in cases where the vth entry is already occupied, which results in a *collision*. Collisions can be resolved in two ways, depending on the type of hash table that is used:

Chained hashing: In the case of chained hashing, one creates a linked list of multiple terms, which is pointed to by each hash table entry. When the vth entry is already occupied, it is first checked whether term t_j already exists within the linked list. If this is the case, then an insertion does not need to be performed. Otherwise, the linked list is augmented with the term t_j, and its length increases by 1. The entries of the linked list contain the string representation of the term, the number of documents in which it occurs, and a pointer to the first item on the inverted list of the term. The linked list is maintained in (lexicographically) sorted order[1] to enable faster searching of terms. When a term is to be checked against the linked list, one simply scans the linked list in sorted order until the term is found or a lexicographically larger term is reached.

Linear probing: In linear probing, a linked list is not maintained at each position in the hash table. Rather each position in the hash table contains the meta-information (e.g., string representation, inverted list pointer, and inverted list size) for a single term. For a given term t_j, the $h(t_j)$th position is checked to see if it is empty. If the position is empty, then the string for term t_j is inserted at that position along with its meta-information (document frequency and inverted list pointer). Otherwise, it is checked if the occupied position already contains term t_j. If the occupied position does not contain the term t_j, the same check is repeated with the $[h(t_j) + 1]$th position. Thus, one "probes" successive positions $h(t_j), h(t_j) + 1, \ldots h(t_j) + r$, until the term t_j is encountered or an empty position is reached. If the term t_j is encountered, then nothing needs to be done, since the hash table already contains the term t_j. Otherwise, the term t_j is inserted at the first empty position encountered during the linear probing process. This probing process is also useful during query time, when a term needs to be searched in the dictionary to obtain the pointer to its inverted list.

The hash table data structure does not provide any natural way to identify terms with closely related spellings. It is often useful to identify such terms as query suggestions to the user, when they make a mistake in entering a query. For example, if a user enters the (misspelled) query term "*recieve*," it is often desirable to suggest the alternate query term "*receive*." One can find such terms in the context of the hash table data structure by creating a separate dictionary of misspelled words (from historical queries) together with the possible spellings that might be correct. A more challenging case arises when users misspell words to their *homonyms*. For example, the term "*school principle*" is most likely intended to be "*school principal*." Such a spelling correction is referred to as a *contextual* spelling correction, and it can be detected only by using the surrounding phrase in the form of a k-gram dictionary of incorrect query phrases.

An alternative that allows the detection of closely related spellings is to implement the dictionary as a variant of the binary search tree in which terms are stored only at the leaf levels of the tree, and the internal nodes contain the meta-information in order to find the relevant leaf efficiently. In the binary search tree, the entire set of terms can be viewed as a lexicographically sorted list, which is partitioned at some intermediate letter between 'a' and 'z.' For example, all terms starting with letters between 'a' and 'h' belong to the left branch of the tree, whereas all terms starting with letters between 'i' and 'z' belong to the right branch of the tree. Similarly, the left branch may be divided into two parts, corresponding to the beginning portions between $[a, de]$, and $[df, h]$, respectively. This type of recursive division is shown in Fig. 9.1. The leaf nodes of the binary search tree contain the actual terms. The process of searching for a term is a relatively simple matter. One only

[1] A lexicographically sorted order refers to the order in which terms occur in a dictionary.

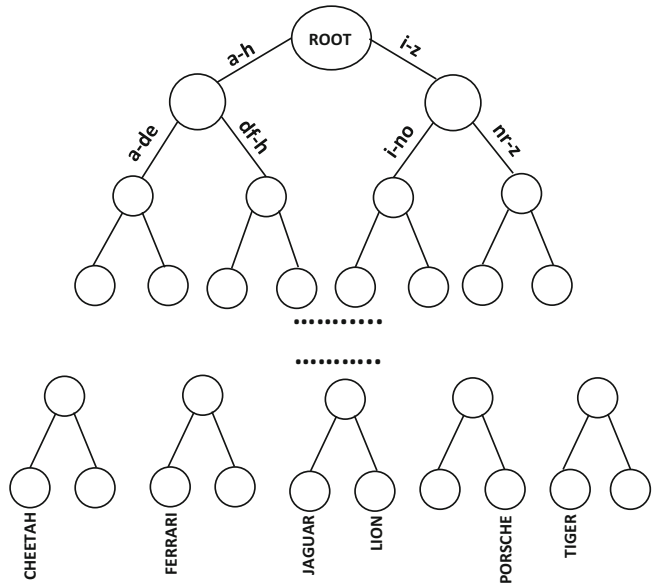


Figure 9.1: A binary tree structure for storing a searchable dictionary of terms. The leaf nodes point to the data structures indexed by the terms, which are the inverted lists.

needs to traverse the path corresponding to the front portion of the query term until the appropriate leaf node is reached (or it is determined that the binary tree does not contain the search term).

If the binary tree is relatively well balanced, the search process is efficient because the depth of the tree is $O(\log(d))$ over a dictionary of d terms. It is often difficult to fully balance a binary tree in the presence of dynamic updates. One way of creating a more balanced tree structure is to use a *B-Tree* instead of a binary search tree. Interested readers are referred to [427] for details of these data structures. Although the tree-like structures do offer better search capabilities, the hash table is often the data structure of choice for dictionaries. One advantage of the hash table is that it has $O(1)$ lookup and insertion time.

9.2.2 Inverted Index

The inverted list is designed to identify all the document *identifiers* related to a particular term. Each *inverted list* or *postings list* corresponds to a particular term in the lexicon, and it contains a list of the identifiers of all documents containing the term. Each element of this list is also referred to as a *posting*. The document identifiers of the inverted list are often (but not always) maintained in sorted order to enable efficient query processing and index update operations. The relevant term frequencies are often stored with document identifiers.

An example of an inverted representation of a document-term matrix is shown in Fig. 9.2. The hash-based dictionary data structure, which is tied to this index, is also included in this figure. Note that the dictionary data structure also contains the document-wise frequency of each term (i.e., number of occurrences across distinct documents), whereas each individual posting of the inverted list contains the document-specific term frequency. These additional statistics are required to compute match-based scores between queries and documents with

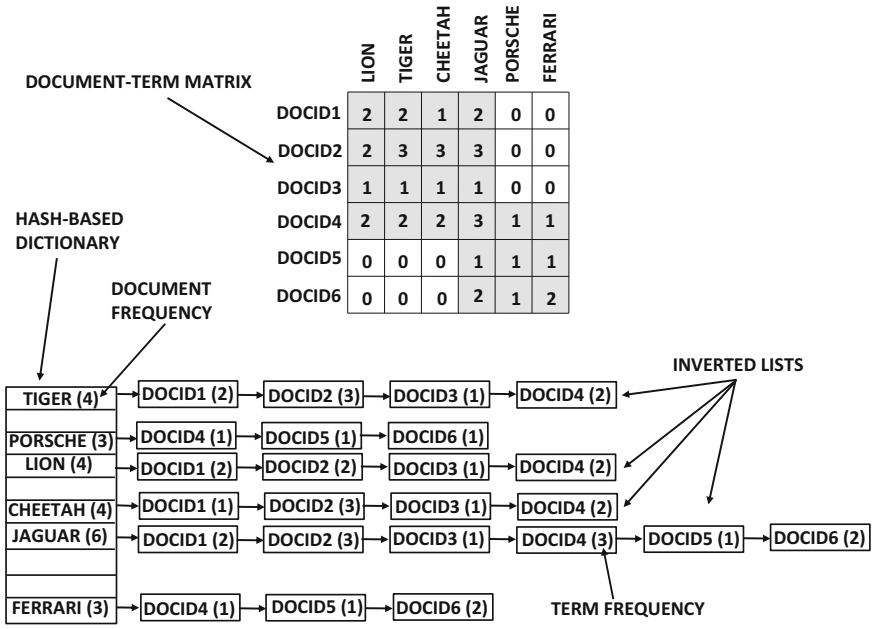


Figure 9.2: A hash-based dictionary and an inverted index together with its parent document-term matrix. The dictionary is used to retrieve the pointer to the first element of the inverted list during query processing.

inverse document frequency normalization. As we will see later, the postings list might also contain other meta-data about the position of that term in the document. Such meta-data can be useful for positional queries.

It is common to use linked lists to store the inverted index when it is maintained in main memory. Even when inverted lists are too long to be stored in main memory, smaller portions of them are often maintained in main memory for fast query processing. Linked lists can be used to insert a document identifier at any position in the inverted list efficiently by a single pointer deletion and two pointer additions. Therefore, such data structures are efficient from the perspective of incremental updates. The first element of each inverted list is pointed to by the entry of the relevant term in the dictionary data structure. This mapping is crucial for query processing.

One issue with the inverted list is that many of the lists of uncommon or unique terms are extremely short. Storing such lists as separate files is inefficient. In practical implementations, multiple inverted lists are consolidated into files on disk, and the dictionary data structure contains the pointer to the offset in the relevant file on disk. This pointer provides the first posting in the inverted list of the term being queried.

9.2.3 Linear Time Index Construction

Given a document corpus, how does one create the dictionary and the inverted lists? Modern computers usually have sufficient memory to maintain the dictionaries in main memory. However, the construction of inverted lists is a completely different matter. The space required by an inverted index is of the same magnitude required by a sparse representation of the document-term matrix within a constant of proportionality (see Exercise 1). A document corpus is usually too large to be maintained in main memory and so is its inverted index.

Converting one disk-resident representation (i.e., corpus) to another (i.e., inverted index) is often an inefficient task, if care is not taken to limit the reads and writes to disk. The most important algorithm design criterion is to minimize random accesses to disk and favor sequential reads as far as possible during index construction. The following will describe a linear-time method, which is referred to as *single-pass in-memory indexing*. The basic idea is to work with the available main memory and build both the dictionary and inverted index within the memory until it is exhausted. When memory is exhausted, the current dictionary and inverted index structure are both stored on disk with care being taken to store the inverted lists in sorted lexicographic order of the terms. At this point, a new dictionary and inverted index structure is started, and the entire process is repeated. Therefore, at the end of the process, one will have multiple dictionaries and inverted index structures. These are then merged in a single pass through the inverted lists. The following discussion explains both the phases of multiple index construction and merging.

An important assumption is that the document identifiers are processed in sorted order, which is easy to implement when the document identifiers are created during index construction. The practical effect of this design choice is that the elements of the inverted lists are arranged in sorted order as identifiers are appended to the end of each list. Furthermore, the document identifiers in the list of an earlier block are all strictly smaller than those in a later block, which enables easy merging of these lists. The approach starts by initializing a hash-based dictionary $\mathcal{H}$ and an inverted index $\mathcal{I}$, to empty structures and then updating them as follows:

while remaining memory is sufficient to process next document **do begin**
  Parse next document with identifier DocID;
  Extract set S of distinct terms in DocID with term frequencies;
  Use $\mathcal{H}$ to identify existing and new terms in S;
  For each new term in S, create a new entry in $\mathcal{H}$ pointing to a newly created
    singleton inverted list in $\mathcal{I}$ containing DocID and term frequency;
  For each existing term in S, add DocID to end of corresponding inverted list
    in $\mathcal{I}$ together with the term frequency;
end while
Sort the entries of $\mathcal{H}$ in lexicographic order of term;
Use the sorted entries of $\mathcal{H}$ to create a single disk file containing
    the inverted lists of $\mathcal{I}$ in lexicographic order of term;
Store sorted dictionary $\mathcal{H}$ on disk containing file offset pointers to inverted lists;

The sorted dictionaries can be stored on disk as lists of sorted term-string/document-frequency/offset triplets rather than as hash tables. After processing the entire corpus, the (multiple) disk files containing partial inverted indexes need to be merged. Let these disk files containing the inverted lists be denoted by $\mathcal{I}_1, \mathcal{I}_2, \ldots \mathcal{I}_k$. The merging is a simple matter because (1) each inverted list in $\mathcal{I}_j$ is sorted by document identifier, (2) the different inverted lists within each $\mathcal{I}_j$ are arranged in lexicographically sorted order of term, and (3) all document identifiers in earlier block writes are smaller than the document identifiers in later block writes. The conditions (1) and (3) are consequences of the fact that document identifiers are selected (or created) in sorted order for parsing. An example of two partial indexes containing three documents each is shown in Fig. 9.3. Note that the first index only contains sorted lists with document identifiers between DocId1 and DocId3, whereas the second index contains sorted inverted lists with document identifiers between DocId4 and DocId6. Therefore, the merged and sorted list of any term (e.g., *Jaguar*) can be obtained by concatenating one list after the other.

In order to merge the inverted lists, one can simultaneously open all the files containing $\mathcal{I}_1 \ldots \mathcal{I}_k$ and $\mathcal{H}_1 \ldots \mathcal{H}_k$. We do not need to read these files in memory but scan them sequentially in order to process each term in sorted order. The merging can be achieved

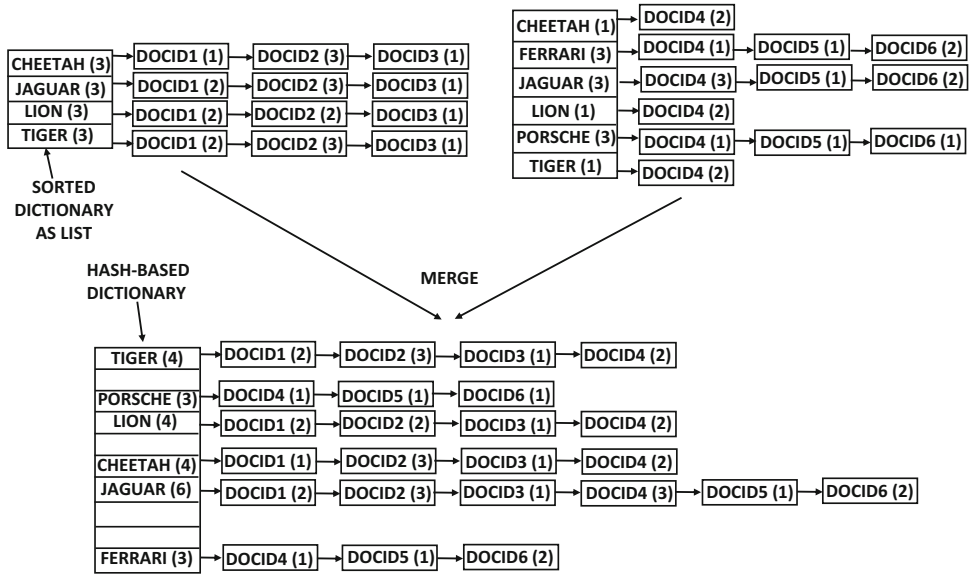


Figure 9.3: Merging a pair of partial indexes

with a simultaneous linear scan of the different files, because the inverted lists of various terms are stored in lexicographically sorted order. For any particular term, its inverted list is identified in each block (if it exists), and these lists are simply concatenated. The lists of later blocks are concatenated after those of earlier blocks to ensure that document identifiers remain in sorted order. At the same time, the dictionary of the merged index is created from scratch. For each term-specific merging, a new entry is added to the dictionary containing the pointer to the merged postings list and the number of documents containing that term (which is the length of the merged list). It is also possible to create the index without processing the document identifiers in sorted order, although doing so will increase the running time slightly (see Exercise 2).

9.2.4 Query Processing

Query processing is of two types. One of them is *Boolean retrieval*, in which documents are returned only when they *exactly* match a particular query. There is no focus on ranking the results, even when a large number of them are returned. Furthermore, in the Boolean retrieval model, one can construct Boolean expressions for queries containing "AND," "OR," and "NOT," whereas ranked retrieval generally uses free text queries. As a practical matter, however, ranked retrieval is almost always necessary in order to distinguish between the varying levels of matches between target queries and documents.

9.2.4.1 Boolean Retrieval

In Boolean retrieval, results are returned depending on whether or not they match a particular query. The query can be in forms such as the following:

(*text* AND *mining*) OR (*data* AND *mining*)
(*text* OR *data*) AND *mining*

The two queries above are actually equivalent, which also reflect the different ways in which they can be resolved. Consider the case where one wishes to use the first form of the query. The first step is to use the dictionary data structure in order to locate the terms "*text*," "*data*," and "*mining*," and their corresponding inverted lists. First, the lists of "*text*" and "*mining*" are intersected into a sorted list L_1, and then the lists of "*data*" and "*mining*" are intersected into another sorted list L_2. Subsequently, the lists L_1 and L_2 are merged with a union operation in order to implement the "OR" operator. An important point about Boolean retrieval is that the returned results are unordered, and there is a single correct result set for a given search query.

The following will describe the process of intersecting two sorted lists in linear time. Assume that each inverted list is defined as a linked list in which the document identifiers are in sorted order. Then, the algorithm uses two pointers, which are initialized to the beginning of the two lists. These pointers are used to scan through the two linked lists in order to identify common document identifiers. The query-result list, which is denoted by Q, is initialized to the empty list. If the document identifiers at the current pointer values in the two inverted lists are the same, then this document identifier is appended to the end of Q and both pointers are incremented by 1. Otherwise, it is determined which pointer corresponds to the smaller document identifier. Consider the case where the pointer of the list corresponding to the keyword "*mining*" has the smaller document identifier. The pointer to the inverted list for "*mining*" is incremented until the corresponding document identifier is either the same or larger than that of "*text*." This process of advancing the pointers to the two lists is continued (with the list Q growing continuously) until the end of at least one of the lists is reached. When the intersection of more than two lists is performed in succession, it is advisable to start with the most restrictive pair of words first to perform the intersection, so that the size of the intermediate result is as small as possible. In other words, the inverted lists are used in decreasing order of inverse document frequency in the intersection process. This is done in order to ensure that smaller documents are processed first. The process of merging two lists with the "OR" operator uses a similar approach as that of intersection (see Exercise 3).

9.2.4.2 Ranked Retrieval

Boolean retrieval is rarely used in information retrieval and search engines, because it provides no understanding of the ranking of the retrieved results. Even though the Boolean retrieval model does allow the ability to combine different logical operators to create potentially complex queries, the reality is that it is often cumbersome for the end user to effectively use this type of functionality. Most practical applications use *free text queries*, in which users specify sets of keywords. Although a free text query can be interpreted in terms of maximizing the match over the query keywords, there are often many other factors that influence the matching. In ranked retrieval, the results need to be *scored* and *ranked* in response to the query, and the system often performs this type of ranking using a variety of different factors (e.g., relative positions of terms in document or document quality) that are not always specified explicitly in the query. In this sense, ranked retrieval allows the use of a variety of different *models* for retrieving search results, and there is no single model that is considered fundamentally "correct" in a way that can be crisply defined. This is a different concept from Boolean retrieval, in which the correct set of results is exactly defined, and the returned results are unordered.

For large-scale applications like Web search, the Boolean relevance of the (possibly thousands of) documents to a set of search terms is not quite as important as ensuring

that the tiny set of results at the very top of the search are relevant to the user. This is a far more difficult problem than Boolean search, and many aspects of it have a distinct machine learning flavor. Although one can restrict the search results based on relevance criteria (e.g., all query terms must be present), the ability to correctly score the large number of valid search results remains exceedingly important in these settings. Most natural scoring functions satisfy the following properties:

1. The presence of a term in the document that matches a query term increases the score, and the score increases with the frequency of the term.

2. Matching terms that are rarely present in the document collection (i.e., terms with high inverse document frequency) increase the score to a greater degree. This is because rare terms are less likely to be matched by chance.

3. The score of candidate documents with longer length is penalized because terms might be matched to the query purely by chance.

The cosine similarity function with tf-idf normalization satisfies all of the above properties, although it does not account for many factors used in modern search engines such as the ordering of the terms or their proximity. Furthermore, when multiple factors are used for computing similarity, it is helpful to be able to weight the relative importance of these factors. This problem has the flavor of supervised learning, which leads to the notion of machine learning in information retrieval. This section will provide a broad overview of the index structures, query processing, and scoring functions, whereas Sect. 9.3 will focus more deeply on the basic principles with which various scoring functions are designed.

There are two fundamental paradigms for query processing in ranked retrieval, which correspond to *term-at-a-time* and *document-at-a-time* query processing with the inverted index. Many nicely behaved scoring functions like the cosine can be computed using either paradigm because they can be expressed as additive functions over query terms. However, the document-at-a-time processing is more convenient for complex functions that use various factors involving multiple terms, such as the relative positions of the terms. In both cases, the document identifiers are accessed using the inverted lists and their scores are continually updated using *accumulator variables* (each of which is associated with a document identifier). In the following, we will describe each of these paradigms.

9.2.4.3 Term-at-a-Time Query Processing with Accumulators

Accumulators are intermediate aggregation variables that can help in evaluating surprisingly general scoring functions between queries and documents, as long as the scoring function is computed in an additive way over the target query terms. For small subsets of query terms, even more general functions incorporating positional information between terms can be computed with accumulators. Consider a query $\overline{Q} = (q_1 \ldots q_d)$ with a small number of query terms in which most values of q_i are 0. Consider a document $\overline{X} = (x_1 \ldots x_d)$ defined over the same lexicon of size d. Now consider a simple scoring function $F(\overline{X}, \overline{Q})$ of the following form:

$$F(\overline{X}, \overline{Q}) = \sum_{j:q_j>0} g(x_j, q_j) \tag{9.1}$$

Note that the summation is only over the small number of terms satisfying $q_j > 0$, and $g(\cdot, \cdot)$ is another function that increases with both x_j and q_j. For example, using $g(x_j, q_j) = x_j q_j$ yields the dot product, which is the unnormalized variant of the cosine function.

The inverted lists of all the terms with $q_j > 0$ are accessed one after another to perform the scoring. Every time a new document identifier is encountered on an inverted list, a new accumulator needs to be created to track the score of that document. For each document identifier encountered on the inverted list of a query term with $q_j > 0$, the value of $g(x_j, q_j)$ is added to the accumulator of that document identifier. In cases where the corpus is large, too many document identifiers might be encountered and one might run out of space to create new accumulators. There are several solutions for addressing this issue. First, the inverted lists should always be accessed in decreasing order of inverse document frequency, so that the most number of terms are used when one runs out of memory. Furthermore, since the terms with higher inverse document frequency are assumed to be more discriminative, this ordering is also helpful in ensuring that the accumulators are more likely to be assigned to relevant documents. A hash table is used to keep track of the accumulators for various documents. When one runs out of memory in the hash table, the results are returned with respect to only[2] those identifiers that have been encountered so far. New accumulators are no longer added because such documents are not assumed to be strong matches. However, the counts of existing accumulators continue to be updated.

Finally, the documents with the largest accumulators are returned. The naïve approach would be to scan the accumulators to identify the top-k values. A more efficient approach is to scan the accumulator values and maintain the top-k in a min-heap (i.e., a heap containing the minimum value at the root). The heap is initialized by inserting the first k scanned accumulators. Subsequent accumulators are compared with the value at the root of the heap, and dropped if they are less than the value at the root. Otherwise, they are inserted into the heap, and the minimum value at the root is deleted. This approach requires time that is $O(n_a \cdot \log(k))$ time, where n_a is the number of accumulators.

It is noteworthy that term-at-a-time query processing does not require the elements on the inverted list to be sorted by document identifier. In fact, for term-at-a-time query processing, it makes sense to sort the lists by decreasing order of term-frequency in the various document identifiers and use only those documents whose term-frequency is above a particular threshold. Furthermore, one can also handle more general functions than Eq. 9.1, which are of the following form:

$$F(\overline{X}, \overline{Q}) = \frac{\sum_{j:q_j>0} g(x_j, q_j)}{G(\overline{X})} + \alpha \cdot Q(\overline{X}) \tag{9.2}$$

Here, $G(\overline{X})$ is some normalization function (like the length of the document), α is a parameter, and the function $Q(\overline{X})$ is some global measure of the quality of the document (such as the *PageRank* of Sect. 9.6.1). It is not difficult to see that the cosine is a special case[3] of this measure. It is also assumed that such global measures for document normalization or quality are pre-stored up front in a hash table indexed by document identifier. This type of scoring function can be addressed by using an additional processing step at the end in which the values of $G(\overline{X})$ and $Q(\overline{X})$ are accessed from the hash table to adjust the scores.

[2]If all query terms must be included in the result, then the intersection of the inverted lists can be performed up front and accumulators are assigned only to document identifiers that lie in this intersection. There are many such *index elimination* tricks that one can use to speed up the process.

[3]One can set $Q(\overline{X}) = 0$ and select $G(\overline{X})$ to be the length of document $\overline{X}$. Normalization with the query length is not necessary because it is constant across all documents and does not change the relative ranking.

9.2.4.4 Document-at-a-Time Query Processing with Accumulators

Unlike the term-at-a-time query processing paradigm, the document-at-a-time approach requires each inverted list to be sorted by document identifier. The document-at-a-time approach can handle more general query functions than the term-at-a-time approach because it accesses all the inverted lists for the query terms simultaneously in order to identify all the query-specific meta-information associated with a document identifier. For a given query vector $\overline{Q} = (q_1 \ldots q_d)$, let $\overline{Z}_{X,Q}$ represent all the meta-information in the document $\overline{X}$ about the *matching terms* in the document with respect to the query. This meta-information could correspond to the position of the matching terms in the document $\overline{X}$, the portion of the document in which matching terms lie, and so on. As we will discuss later, such meta-information can often be stored along with the inverted lists. Then, consider the following scoring function, which is a generalization of Eq. 9.2:

$$F(\overline{X}, \overline{Q}) = \frac{H(\overline{Z}_{X,Q})}{G(\overline{X})} + \alpha \cdot Q(\overline{X}) \tag{9.3}$$

Here, $G(\overline{X})$ and $Q(\overline{X})$ are global document measures as in Eq. 9.2. The function $H(\overline{Z}_{X,Q})$ is more general than the additive form of Eq. 9.2 because it could include the effect of the interaction of multiple query terms. This function could, in principle, be quite complex and include factors such as the positional distance between the query terms in the document. However, to enable such a query, the inverted index needs to contain the meta-information about the positions of query terms (cf. Sect. 9.2.4.7).

In such a case, one simultaneously traverses the inverted list for *each* term t_j satisfying $q_j > 0$ (i.e., terms included in the query). As in the case of list intersection, one traverses each of the *sorted* lists in parallel until one reaches the same document identifier. At this point, the value of $H(\overline{Z}_{X,Q})$ is computed (using the meta-information associated with document identifiers) and added to the accumulator variable for that document identifier. The other post-processing steps in document-at-a-time querying are identical to those of term-at-a-time query processing. If the space for accumulator variables is limited, the document-at-a-time processing maintains the best scores so far, which turns out to be a more sensible approach for obtaining the best results. In such cases, it might also make sense to incorporate the impact of global document measures like $G(\overline{X})$ and $Q(\overline{X})$ at the time the document is processed rather than leaving it to the post-processing phase.

Although it is possible to enable scoring functions like Eq. 9.3 with term-at-a-time querying, it increases the space overhead in impractical ways. One would need to store all the meta-information in the traversed lists along with the accumulator variables and then evaluate Eq. 9.3 in the final step.

9.2.4.5 Term-at-a-Time or Document-at-a-Time?

The two schemes have different advantages and disadvantages. The document-at-a-time approach allows the maintenance of the best k results found so far dynamically. Furthermore, the types of queries that can be resolved with document-at-a-time processing are more complex, because one can use the relative positions of terms and other statistics that use the properties of multiple query terms. On the other hand, the document-at-a-time processing requires multiple disk seeks and buffers because multiple inverted lists are explored simultaneously. In term-at-a-time processing, one can read in large chunks of a single inverted list at one time in order to perform the processing efficiently.

9.2.4.6 What Types of Scores Are Common?

In many search engines, global meta-features of the document such as its provenance or its citation structure are included in the final similarity score. In fact, modern search engines often learn the importance of various meta-features (cf. Sects. 9.2.4.9 and 9.2.4.10) by leveraging user click-through behavior. For example, Eqs. 9.2 and 9.3 contain the parameter α, which regulates the importance of page quality in ranking. Such a parameter can be learned using machine learning models from previous user click-through behavior. It needs to be pointed out that most of the popular scoring functions in information retrieval and search engines (including advanced machine learning models) can be captured using Eqs. 9.2 and 9.3 by instantiating the various terms in these equations appropriately. Several such models will be explored in this section and in Sect. 9.3.

9.2.4.7 Positional Queries

It is often desirable for query processing to account for the positions of the query terms. There are several ways in which the positioning can be taken into account. The first is to include common phrases as "terms" and created inverted lists for them. However, this approach greatly expands the term set. Furthermore, for a given query, there are multiple ways in which one can process the query using either the phrases or the individual terms.

In order to resolve queries with the positional index, the same inverted list is maintained, except that all the positions of a term in the document are maintained as meta-information along with a document identifier in the inverted list. Specifically, in the inverted list for any particular term, the following meta-information is retained along with document identifiers:

DocId, $freq$, (Pos_1, Pos_2, ..., Pos_{freq})

Here, $freq$ denotes the number of times the term occurs in the document with identifier DocId. For example, if the term "*text*" occurs at position 7 and 16 in DocId, and the term "*mining*" occurs at positions 3, 8, and 23 of DocId, then all these positions are stored with the document identifiers in the inverted list. Therefore, in the first case, the meta-information DocId,2, (7, 16) is maintained as one of the entries in the inverted list of "*text*," whereas in the second case, the meta-information DocId,3, (3, 8, 23) is maintained in one of the entries of the inverted list of "*mining*." In this particular case, it is evident that the term "*text*" occurs at position 7 in the document with identifier DocId, whereas the term "*mining*" occurs at position 8 in the same document. Therefore, it is evident that the phase "*text mining*" is present in DocId. For Boolean queries, one will need to check these positions at the time of intersecting the inverted lists of "*text* and "*mining*."

In ranking queries (like search engines), the relative positions of terms in a document can affect the scoring function used to quantify the degree to which a document matches a specific set of keywords in a particular order. Therefore, the queries "*text mining*" and "*mining text*" will not return the same ranking of the results, when using a search engine like Google. It turns out that this type of query processing can be performed with accumulator variables, because the effect of relative positioning can be captured by Eq. 9.3. In particular, the function $H(\overline{Z}_{X,Q})$ of Eq. 9.3 should be *defined* by the search engine architect in order to capture the impact of term positioning. The natural approach in these cases is to use document-at-a-time query processing (see page 270).

It is common to combine phrase-based indexes with positional indexes. The basic idea is to keep track of the commonly queried phrases, and maintain inverted lists for these frequent

phases in addition to the lists of the individual terms. For a given query, the frequent phrases in it are combined with positional indexes in order to use the positioning information. How is such a combination achieved? An important point here is that search engines typically use *free text queries*, which are often mapped to an internal representation by the system with the use of a *query parser*. In many cases, the query parser can issue multiple queries, in which phrase indexes are used in combination with positional indexes to yield an efficient query result. For example, the following approach might be used:

1. The inverted lists of frequent phrases might be used in order to provide a first response to the query. Note that it is not necessary that the query phrase is frequent and is available in the index. In such a case, one might try different 2-word subsets of the query phrase to check if it is available in the inverted index.

2. In the event that sufficient query results cannot be generated using the aforementioned approach, one might try to use the positional index in order to generate a query result that scores the documents based on relative proximity of query terms.

The specific heuristic used to resolve a query depends on the goals of the search system at hand. Modern search engines use a number of query optimizations that include all types of meta-data about the document to score and rank results. Examples include *zoned scoring* and *machine learned scoring*, which are discussed in Sects. 9.2.4.8 and 9.2.4.9, respectively. Furthermore, qualitative judgements about the document are inferred based on the co-citation structure, and incorporated in the final ranking. These issues are discussed in a later section (cf. Sects. 9.5 and 9.6).

9.2.4.8 Zoned Scoring

In zoned scoring, different parts of a document, such as the author, title, keywords, and other meta-data are given varying amounts of weight. These different parts are referred to as *zones*. Although zones seem similar to fields at first sight, they are different in the sense that they might contain arbitrary and free-form text. For example, in search engines, the title of a document is quite important as compared to the body of the document. In some cases, the zoning can be implemented by simply adding a more important zone to the vector space representation with a higher weight. For example, the title can be given a higher weight than the body of the document. However, in most cases, zoning is implemented by storing the information about the zoning within the inverted list. Specifically, consider the inverted list of each term that contains the frequency as well as the positioning information. Along with each positioning information, we also maintain the zone in which the term occurs. In other words, consider a position-based inverted index in which one of the entries in the inverted list of "*text*" is of the form DocId, 2, (7, 16). Therefore there are two occurrences of the term "*text*" in DocId with positions 7 and 16, respectively. This type of entry, however, assumes that the positioning is defined with respect to a document with a single zone. More generally, the entry can be of the form DocId, 2, (2-Title, 9-Body). In this case, the term "*text*" occurs as the second token of the title and the ninth position in the body. This type of meta-information can be used easily for *weighted zone scoring*, in which the matched document identifiers during the intersection of inverted lists are scored based on the specific zones in which they reside. An important point here is in deciding how much weight to give each zone. While it is clear that some zones such as the title are more important, the process of finding specific weights for zones has the flavor of a machine learning algorithm.

9.2.4.9 Machine Learning in Information Retrieval

How can one find the appropriate weights for each zone in an information retrieval setting? Consider a situation in which the documents of a corpus have r weights $w_1 \ldots w_r$ over the r zones. Furthermore, the frequencies of a particular document-term combination over these zones are $x_1 \ldots x_r$. For simplicity, one can also assume that each $x_i \in \{0, 1\}$ depending on whether the term is present in the zone. However, it is also possible to have non-binary values of x_i. For example, if a term occurs more than once in a ith zone, the value of x_i might be larger than one. In practical settings, many values of x_i might be 0. Then, the contribution of that document-term combination to the scoring is given by $\sum_{j=1}^{r} w_j \cdot x_j$, where w_j is an unknown weight. It is relatively easy to compute this type of additive score with accumulators at query processing time, if one knew the values $w_1 \ldots w_r$ of the weights. Therefore, the weights are learned up front in offline fashion.

In order to learn the appropriate values of w_i, one can use the *relevance feedback values from the user* over a set of training queries. The training data contains a set of engineered features that are extracted from each document in response to a query (such as the zones in which query words lie) together with a user relevance judgement of whether the document is relevant to the corresponding query. The relevance judgement might be a binary quantity (i.e., relevant/not relevant), a numerical quantification of the relevance judgement, or a ranking-based judgement between pairs of documents. The Web-centric approach of collecting relevance feedback leverages user click-through behavior, which is discussed later.

Learning the importance of zones is not the only application of such weight-learning techniques. There is significant meta-information associated with both terms and documents, whose importance can be learned for query processing. Examples include the following:

1. *Document-specific features:* The meta-information associated with a document on the Web, such as its geographical location, creation date, Web linkage-based co-citation measures (cf. Sect. 9.6.1), number of words in pointing anchor text, or Web domain can be used in the scoring process. Document-specific meta-data is often independent of the query at hand, and has been shown to be effective for improving retrieval performance with machine learning techniques [406].

2. *Impact features:* The impact of several terms in a scoring function is often regulated with parameters. For example, the scoring functions of Eqs. 9.2 and 9.3 contain the parameter α, which regulates the importance of document quality. Sometimes multiple scoring functions like the cosine, binary-independence model, and the BM25 model (see Sect. 9.3) can be combined with weights. The importance of these weights can be learned with user feedback.

3. *Query-document pair-specific meta-data:* The zones of the document in which query terms occur and query-term ordering/positioning within the candidate document can be used in the scoring process.

The scoring function in modern search engines is quite complex and is often tuned using machine learning. In general, one might have any arbitrary set of parameters $w_i \ldots w_m$. In other words, these weight parameters include the zoning weights, and they might represent the importance of different *features* $x_1 \ldots x_m$ extracted from a document-query pair. Note that the same set of features is extracted from any document-query pair, which allows the learning to be generalized from one query to another. In the zoning example, the features correspond to the different zones of the documents in which the query words lie, and their corresponding frequencies. Therefore, if the user feedback data consistently shows the user

preferentially clicking on search results with query words in the title (over the body), then this fact will be learned by the algorithm irrespective of the query at hand. This is achieved with a relevance function $R(w_1x_1, \ldots w_mx_m)$, which is defined in terms of the weighted features. For example, a possible relevance function could be as follows:

$$R(w_1x_1, \ldots, w_mx_m) = \sum_{j=1}^{m} w_jx_j \tag{9.4}$$

The values of $w_1 \ldots w_m$ are learned from user feedback. The choice of the relevance function is a part of search engine design, and virtually all functions that are defined in terms of the meta-data about matching terms between the query and the target can be modeled with scoring functions like Eq. 9.3. Such scores can be efficiently computed at query processing time with accumulators.

It is noteworthy that relevance judgements can also be inferred using *implicit feedback* based on user actions rather than their explicit judgements. For example, search engines provide a large amount of implicit feedback based on user clicks on returned query results. Such feedback should, however, be used carefully because top-ranked items are more likely to be clicked by a user, and therefore one must adjust for the rank of the returned items during the learning process. Consider a situation in which a search engine ranks document $\overline{X_j}$ above document $\overline{X_i}$, but the user clicks on the document $\overline{X_i}$ but not $\overline{X_j}$. Because of the preferential clicking pattern of the user, there is evidence that document $\overline{X_i}$ might be more relevant than document $\overline{X_j}$ to the user. In such a case, the training data is defined in terms of *ranked pairs* like $(\overline{X_i}, \overline{X_j})$, which indicate relative preference. This type of data is highly noisy but the saving grace is that copious amounts of it can be collected easily. Machine learning methods are particularly good at learning from large amounts of noisy data.

For the purpose of this section and the next, each $\overline{X_i}$ refers to the query-specific features extracted from the document (e.g., impact features), rather than text vectors. This choice requires an understanding of the importance of various characteristics of the document-query pair, such as the impact features, including the use of zones, physical proximities, ordering of matched words, document authorship, domain, creation date, page citation structure, and so on. *The extraction of the features depending on the match between the query and the document is an important modeling and feature engineering process, which depends on the search application at hand.* Each of these features either need to be pre-stored (e.g., document-specific *PageRank*), or they need to be computed on-the-fly using accumulators. At query time, they need to be combined using the linear condition in Eq. 9.4.

9.2.4.10 Ranking Support Vector Machines

The previous section discusses the importance of learning methods by extracting m query-specific features and learning their associated parameters $w_1 \ldots w_m$ in order to quantify their relevance to a new query. How can one use such pairwise judgements by the end user in order to learn key parameters such as $w_1 \ldots w_m$ that are used in ranking the results? This is typically achieved by *learning-to-rank* algorithms. A classical example of such an algorithm is the *ranking support vector machine*, which is also referred to as the *ranking SVM*. The ranking SVM uses previous queries to create training data for the extracted features. The features for a document contain attributes corresponding to the various zones in which the query terms occur, meta-information about the document such as its geographical location, and so on. The training data contains pairs of documents (in this query-centric representation), denoted by $(\overline{X_i}, \overline{X_j})$, which signifies the fact that $\overline{X_i}$ should occur earlier

than $\overline{X_j}$. We would like to learn $\overline{W} = (w_1 \ldots w_m)$, so that $\overline{W} \cdot \overline{X_i} > \overline{W} \cdot \overline{X_j}$ for the training documents $\overline{X_i}$ and $\overline{X_j}$, which *contain the query-specific "match" features* (see Sect. 9.2.4.9 for examples). Once such weights have been learned from a training corpus (created by past queries and user feedback), they can be used in real time to rank the different documents.

We will now formalize the optimization model for the ranking SVM. The training data $\mathcal{D}_R$ contains the following set of ranked pairs:

$$\mathcal{D}_R = \{(\overline{X_i}, \overline{X_j}) : \overline{X_i} \text{ should be ranked above } \overline{X_j}\}$$

For each such pair in the ranking support vector machine, the goal is learn $\overline{W}$, so that $\overline{W} \cdot \overline{X_i} > \overline{W} \cdot \overline{X_j}$. However, we impose an additional *margin requirement* to penalize pairs where the difference between $\overline{W} \cdot \overline{X_i}$ and $\overline{W} \cdot \overline{X_j}$ is not sufficiently large. Therefore, we would like to impose the following stronger requirement:

$$\overline{W} \cdot (\overline{X_i} - \overline{X_j}) > 1$$

Any violations of this condition are penalized by $1 - \overline{W} \cdot (\overline{X_i} - \overline{X_j})$ in the objective function. Therefore, one can formulate the problem as follows:

$$\text{Minimize } J = \sum_{(\overline{X_i}, \overline{X_j}) \in \mathcal{D}_R} \max\{0, [1 - (\overline{W} \cdot [\overline{X_i} - \overline{X_j}])]\} + \frac{\lambda}{2} ||\overline{W}||^2$$

Here, $\lambda > 0$ is the regularization parameter. Note that one can replace each pair $(\overline{X_i}, \overline{X_j})$ with the new set of features $\overline{X_i} - \overline{X_j}$. Therefore, one can now assume that the training data simply contains n instances of the m-dimensional *difference features* denoted by $\overline{U_1} \ldots \overline{U_n}$, where n is the number of ranked pairs in the training data. In other words, each $\overline{U_p}$ is of the form $U_p = \overline{X_i} - \overline{X_j}$ for a ranked pair $(\overline{X_i}, \overline{X_j})$ in the training data. Then, the ranking SVM formulates the following optimization problem:

$$\text{Minimize } J = \sum_{i=1}^{n} \max\{0, [1 - \overline{W} \cdot \overline{U_i}]\} + \frac{\lambda}{2} ||\overline{W}||^2$$

One can also write this optimization formulation in terms of the *slack penalty* $C = 1/\lambda$ in order to make it look cosmetically more similar to a traditional SVM:

$$\text{Minimize } J = \frac{1}{2} ||\overline{W}||^2 + C \sum_{i=1}^{n} \max\{0, [1 - \overline{W} \cdot \overline{U_i}]\}$$

Note that the only difference from a traditional support-vector machine is that the class variable y_i is missing in this optimization formulation. However, this change is extremely easy to incorporate in all the optimization techniques discussed in Sect. 6.3 of Chap. 6. In each case, the class variable y_i is replaced by 1 in the corresponding gradient-descent steps of various methods discussed in Sect. 6.3. The linear case is particularly easy to extend using the aforementioned approach, although one can also extend the techniques to kernel SVMs with some minor modifications. The main point to keep in mind is that kernel SVMs work with dot products between training instances. In this case, the training instances are of the form $\overline{U_p} = \Phi(\overline{X_i}) - \Phi(\overline{X_j})$, where $\Phi(\cdot)$ is the nonlinear transformation that is (implicitly) used by a particular kernel similarity function. Let the kernel similarity function define a similarity matrix $S = [s_{ij}]$ over the training instances so that we have the following:

$$s_{ij} = \Phi(\overline{X_i}) \cdot \Phi(\overline{X_j}) \tag{9.5}$$

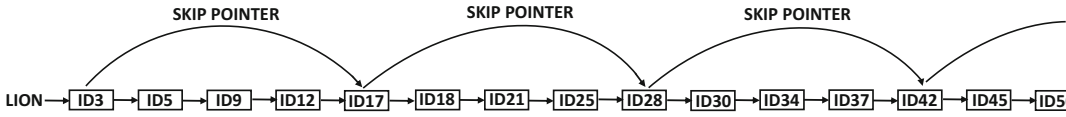


Figure 9.4: Skip pointers with skip values of 4

In kernel methods, only the similarity values s_{ij} are available (as a practical matter) rather than the explicit transformation.

Consider two training instances $\overline{U_p}$ and $\overline{U_q}$ in the following form:

$$\overline{U_p} = \Phi(\overline{X_i}) - \Phi(\overline{X_j}) \quad [\overline{X_i} \text{ ranked higher than } \overline{X_j}]$$
$$\overline{U_q} = \Phi(\overline{X_k}) - \Phi(\overline{X_l}) \quad [\overline{X_k} \text{ ranked higher than } \overline{X_l}]$$

Then, the dot product between $\overline{U_p}$ and $\overline{U_q}$ can be computed as follows:

$$\begin{aligned}\overline{U_p} \cdot \overline{U_q} &= (\Phi(\overline{X_i}) - \Phi(\overline{X_j})) \cdot (\Phi(\overline{X_k}) - \Phi(\overline{X_l})) \\ &= \{\Phi(\overline{X_i}) \cdot \Phi(\overline{X_k}) + \Phi(\overline{X_j}) \cdot \Phi(\overline{X_l})\} - \{\Phi(\overline{X_i}) \cdot \Phi(\overline{X_l}) + \Phi(\overline{X_j}) \cdot \Phi(\overline{X_k})\} \\ &= \underbrace{\{s_{ik} + s_{jl}\}}_{\text{Similarly ranked}} - \underbrace{\{s_{il} + s_{jk}\}}_{\text{Differently ranked}}\end{aligned}$$

One can use these pairwise similarity values to adapt the kernel methods discussed in Sect. 6.3 to the case of the ranking SVM. As a practical matter, however, linear models are preferable because they can be efficiently used in conjunction with an inverted index with the use of accumulators. The basic idea here is that the inverted index can be used in conjunction with all the meta-data available in it to efficiently compute $\overline{W} \cdot \overline{Z}$ for a test (candidate) document $\overline{Z}$, once the weights in $\overline{W}$ have been learned during the (offline) scoring phase.

9.2.5 Efficiency Optimizations

There are several other optimizations associated with query processing. Some of these optimizations are particularly important in the context of Web retrieval in which the inverted lists are long and lead to many disk space accesses.

9.2.5.1 Skip Pointers

Skip pointers are like shortcuts in the inverted lists at various positions in order to be able to skip over irrelevant portions of the lists in the intersection process. Skip pointers are useful for intersecting lists of unequal size. In such cases, the skip pointers in the longer list can be useful in performing efficient intersection, because the longer inverted list will have large segments that are irrelevant to the intersection. Consider a term t_j with an inverted list of length n_j. We assume that the inverted list is sorted with respect to the document identifiers. Skip pointers are placed only at positions in the inverted list of the form $s \cdot k + 1$ for fixed skip value s and $k = 0, 1, 2, \ldots, \lfloor n_j/s \rfloor - 1$. An example of skip pointers with $s = 4$ is shown in Fig. 9.4.

Now consider the simple problem of intersecting a long list with an extremely short list of length 1 containing a single document identifier. In order to determine whether or not the long list contains this document identifier, we simply traverse its skip pointers, until we

identify the segment in which the identifier lies. Subsequently, only this segment is scanned in order to determine whether or not the document identifier lies inside it. In this case, if we use $s = \sqrt{n_j}$, then it can be shown that at most $2\sqrt{n_j}$ traversals will be required. Now, if we need to intersect a short list of length n_t with a longer list of length n_j, then we can repeat this process one by one with elements of the shorter list in sorted order. For best efficiency, care must be taken to use the starting point in the longer list in each case where the search for the previous element of the shorter list was concluded. In the worst case, this approach might incur a small overhead, whereas one will generally do extremely well in cases where the lists have asymmetric lengths. In general, the use of the square-root of the length of the inverted list is a good heuristic for setting the skip values. The main drawback of skip pointers is that they are best suited to static lists that do not change frequently. For a dynamically changing list, it is impossible to maintain the structured pattern of the skip pointers without incurring large update overheads.

9.2.5.2 Champion Lists and Tiered Indexes

One problem with the solutions in all of the above cases is that the query processing can be quite slow for larger collections in which the inverted lists are very long. In such cases, one typically does not even need all the responses to the query, as long as the top-ranked results can be identified reasonably accurately.

Since large term frequencies often have a favorable impact on the scores, they can be used to identify the portions of the inverted lists that are most likely to yield good matches. A natural approach is to use *champion lists*, in which only the subset of document identifiers in which only the top-p documents with highest frequency with respect to a term are maintained in a *truncated* inverted list. Any additional meta-information such as term frequency and term positions can also be maintained along with the document identifiers. In order to resolve a query, the first step is to determine if a "sufficient" number (say, q) of documents is returned by using only the champion lists. If a sufficient number of documents is returned, then one does not need to use the entire inverted index. Otherwise, the query has to be resolved using the full inverted index. The values of p and q are therefore parameters in this process, which need to be chosen in an application-specific way. Champion lists are particularly useful in document-at-a-time querying in which inverted lists are sorted by document identifier. In the event that the inverted lists are sorted in decreasing order of term frequency (for term-at-a-time querying), the effect of champion lists can be realized by using only the initial portions of the inverted lists. Therefore, champion lists need not be explicitly maintained in such cases.

A generalization of the notion of champion lists is the use of *tiered indexes*. In tired indexes, the idea is that the inverted list only contains the subset of document identifiers with frequency more than a particular threshold. Therefore, the highest threshold corresponds to tier 1, which has the shortest inverted lists. The next higher threshold corresponds to tier 2, and so on. If a query can be resolved using only tier 1 lists, then the results are accepted. Otherwise, the query is processed using the next tier. This approach is continued, until a sufficient number of results can be returned.

9.2.5.3 Caching Tricks

In query processing systems, large numbers of users might be simultaneously making queries, as a result of which many inverted lists will be accessed repeatedly. In such cases, it makes sense to store the inverted lists of frequently queried terms in fast caches for quick retrieval.

The query processing system first checks the cache to retrieve the inverted list for the term. If the inverted list is not available in the cache, then the pointer to the disk (available in the dictionary) is used.

Caches are expensive and therefore only a small fraction of the inverted lists can be stored. Therefore, one needs an *admission control* mechanism to decide which inverted lists to store in the cache. The admission control mechanism must be sufficiently adaptive that the inverted lists stored in the cache are statistically likely to have been accessed frequently in the recent past. The time-tested method for achieving this goal is to use a least recently used (LRU) cache. The cache maintains the last time that each inverted list in it was accessed. When an inverted list for a term is requested, the cache is checked to see if its is available. If the inverted list is found in the cache, its time stamp is updated to the current time. On the other hand, if the inverted list is not found in the cache, it needs to be accessed from disk. Furthermore, it is now inserted in the cache at the expense of one or more existing lists in the cache. This is achieved by removing a sufficient number of least recently used inverted lists from the cache to make room for the newly inserted list. Refer to the bibliographic notes for pointers to multilevel caching methods that are used in information retrieval applications.

9.2.5.4 Compression Tricks

Both the dictionary and the inverted index are often stored in compressed form. Although compression obviously saves on storage, a more important motivation for compression is that it improves efficiency. This is because smaller files improve the caching behavior of the system. Furthermore, it takes less time to read a file from disk and load it to main memory.

Dictionaries are often stored in main memory because they require much less space than the inverted index. However, for some systems even the memory requirements of a dictionary become a burden. Therefore, one needs to reduce its memory footprint as much as possible to ensure that it fits in main memory and possibly free up storage for other parts of the index. How does one allocate memory for the terms in the dictionary? One approach is to allocate *fixed-width* for the string representation of the term in a hash-based dictionary. For example, if 25 characters are allocated for each term in the hash table, but a term like "*golf*" requires only four characters. Therefore, 21 characters are being wasted. Furthermore, the fixed-length approach would cause problems in storage of long terms or phrases with more than 25 characters. One approach is to allocate space only for pointers within the hash table to the string representation of each term. Such pointers are referred to as *term pointers*. Dictionaries are compressed by concatenating all the terms in the lexicon into a single string, in which the terms occur in lexicographically sorted order. The delimiters between two terms can be obtained by using term pointers. Therefore, instead of the hash table, one now maintains a lexicographically sorted array of entries containing these term pointers. A term pointer points to the position on the string at which the term starts. Because of the sorting of both the string dictionary and the array in the same way, the next pointer in the array also provides the end delimiter of the current term in the string. Aside from the term pointers, the array also contains a numerical entry with the number of documents containing the term and the pointer to the first element of the inverted list of the term. Therefore each entry in the sorted table contains 4 bytes each for two pointers (to terms and postings), and 4 bytes to store the document frequency. An example of a compressed dictionary for the example of Fig. 9.2 is shown in Fig. 9.5. When a query is entered by the user, one needs to efficiently locate the pointer to the relevant inverted lists. To achieve this goal, binary search can be used on this dictionary with the help of term pointers. Once the

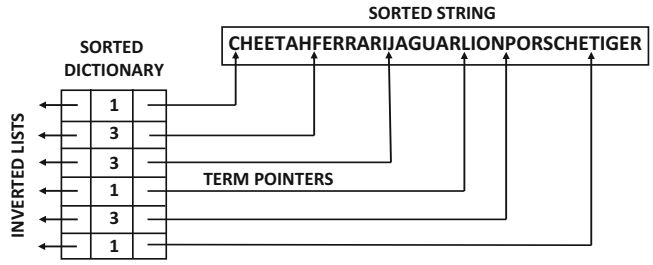


Figure 9.5: Compressing a dictionary by avoiding fixed width allocation of terms

relevant entry of the array has been isolated, the pointer to the inverted list can be returned. For a dictionary containing a million terms of 8 characters per term, the size required by the string is 8 MB and the size required by the array is 12 MB. Although these requirements might seem tiny (and unimportant to compress), they do enable the use of very fast caches or severely constrained hardware settings. One can also use hash tables with dynamically allocated memory for terms, if space is not at a premium.

Inverted lists can also be compressed. The most common approach is to use *variable byte codes*, in which each number is encoded using as many bytes as needed. Only 7 bits within the byte are used for encoding, and the last bit is a continuation indicator telling us whether or not the next byte is part of the same number. Therefore any number less than $2^7 = 128$ requires a single byte, and any number less than 128^2 requires at most two bytes. Most term frequencies in a document are small values less than 128, and they can be stored in a single byte. However, document identifiers can be arbitrarily long integers. In the case of inverted lists, which are sorted by document identifier, one can use the idea of *delta encoding*.

When the document identifiers are in sorted order, one can store the *differences* between consecutive document identifiers using variable byte codes (or any other compression scheme that favors small numbers). For example, consider the following sequence of document identifiers:

23671, 23693, 23701, 23722, 23755, 23812

One does have to store the first document identifier, which is rather large. However, subsequent document identifiers can be stored as successive offsets, which are the differences between consecutive values in the aforementioned sequence:

22, 8, 21, 33, 57

These values are also referred to as *d-gaps*. Each of these values is small enough to be stored in a single byte in this particular example. Another important point to keep in mind is that these differences between successive document identifiers will be small for more frequent terms (with larger inverted lists). This means that larger inverted lists will be compressed to a greater degree, which is desirable for storage efficiency. This is a recurring idea in many compression methods where frequently occurring items are represented using codes of smaller length, whereas rarer items are allowed codes of longer length. We refer the reader to the bibliographic notes for pointers to various compression schemes.

9.3 Scoring with Information Retrieval Models

The previous section provides a broad idea of the scoring process in information retrieval with the use of different types of indexes. A broad picture is provided about the various types of factors that are used for scoring and ranking documents (e.g., aggregate matches or proximity of keywords). However, it does not discuss the specific types of models that are used in information retrieval applications for scoring and ranking documents. Such models can often be used in combination with relevance judgements by combining them with weights. It is noteworthy that most of the models in this section can be captured using scoring functions of the form discussed in Eqs. 9.2 and 9.3. This fact enables the use of efficient term-at-a-time or document-at-a-time query processing methods for these models (see page 268).

9.3.1 Vector Space Models with tf-idf

The simplest approach is to use the tf-idf representation discussed in Sect. 2.4 of Chap. 2. We briefly recap some of the concepts in using the tf-idf representation.

Consider a document collection containing n documents in d dimensions. Let $\overline{X} = (x_1 \ldots x_d)$ be the d-dimensional representation of a document after the term extraction phase. The square-root or the logarithm function may be applied to the frequencies to reduce the effect of terms that occur too often in a document. In other words, one might replace each x_i with either $\sqrt{x_i}$, $\log(1 + x_i)$, or $1 + \log(x_i)$.

It is also common to normalize term frequencies based on their presence in the entire collection. The first step in normalization is compute the inverse document frequency of each term. The inverse document frequency id_i of the ith term is a decreasing function of the number of documents n_i in which it occurs:

$$id_i = \log(n/n_i) \tag{9.6}$$

Note that the value of id_i is always nonnegative. In the limiting cases in which a term occurs in every document of the collection, the value of id_i is 0. The term frequency is normalized by multiplying it with the inverse document frequency:

$$x_i \Leftarrow x_i \cdot id_i \tag{9.7}$$

Once the normalized representation of each document in the corpus is computed, it is used to respond to similarity-based queries. The most common similarity function is the cosine function, which is introduced in Sect. 2.5 of Chap. 2.

Consider a target document $\overline{X} = (x_1 \ldots x_d)$ and the query vector $\overline{Q} = (q_1 \ldots q_d)$. The query vector might either be binary or it might be based on term frequencies. The cosine function is defined as follows:

$$\begin{aligned} \text{cosine}(\overline{X}, \overline{Q}) &= \frac{\sum_{i=1}^{d} x_i q_i}{\sqrt{\sum_{i=1}^{d} x_i^2} \sqrt{\sum_{i=1}^{d} q_i^2}} \\ &\propto \frac{\sum_{i=1}^{d} x_i q_i}{\sqrt{\sum_{i=1}^{d} x_i^2}} = \frac{\sum_{i=1}^{d} x_i q_i}{||\overline{X}||} \end{aligned}$$

It is sufficient to compute the cosine to a constant of proportionality because we only need to rank the different instances for a particular query. It is easy to see that this scoring function is of the form captured by Eqs. 9.2 and 9.3, which can be computed with either term-at-a-time or document-at-a-time query processing (see pages 268 and 270).

9.3.2 The Binary Independence Model

The binary independence model uses binary relevance judgements about training documents in order to score previously unseen documents with the use of a naïve Bayes classifier. In particular, the Bernoulli classifier of Sect. 5.3.1 in Chap. 5 is used. Let $R \in \{0, 1\}$ indicate whether or not a the document is relevant to be particular query. As discussed in Sect. 5.3.1, the Bernoulli model implicitly assumes that each document $\overline{X}$ is represented in a vector space representation with Boolean attributes containing information about whether or not each term t_j is present in $\overline{X}$.

Assume that we have some training data available, which tells us whether or not a document is relevant to a particular query. Note that relevance judgement data that is *query-specific*[4] is often hard to come by, although one can allow the user to provide feedback to query results to collect data about document relevance or non-relevance. Note that the collected data is useful only for that specific query, which is different from the machine learning approach of previous sections in which the training data for importance of specific types of meta-features are learned over *multiple* queries. In order to use these models without human intervention or training data, we will eventually make a number of simplifying assumptions. In that sense, these models also provide the intuitions necessary for query processing without these (query-specific) relevance judgements.

Let $p_j^{(0)}$ be the fraction of non-relevant documents in the training data (i.e., user relevance judgements) that do not contain term t_j, and $p_j^{(1)}$ be the fraction of documents containing term t_j. Similarly, let α_0 be the fraction of non-relevant documents and α_1 be the fraction of relevant documents in the training data. The scoring function for a given query is represented in terms of the Bayes classification probabilities. We wish to find the ratio of the probability $P(R = 1|\overline{X})$ to that of $P(R = 0|\overline{X})$. Based on the results from the Bernoulli model in Sect. 5.3.1, we can state the following:

$$P(R = 1|\overline{X}) = \frac{P(R = 1) \cdot P(\overline{X}|R = 1)}{P(\overline{X})} = \frac{\alpha_1 \prod_{t_j \in \overline{X}} p_j^{(1)} \prod_{t_j \notin \overline{X}} (1 - p_j^{(1)})}{P(\overline{X})}$$

$$P(R = 0|\overline{X}) = \frac{P(R = 0) \cdot P(\overline{X}|R = 0)}{P(\overline{X})} = \frac{\alpha_0 \prod_{t_j \in \overline{X}} p_j^{(0)} \prod_{t_j \notin \overline{X}} (1 - p_j^{(0)})}{P(\overline{X})}$$

Then, the ratio of the two quantities may be computed after ignoring the prior probabilities (because they are not document-specific and do not affect the ranking):

$$\frac{P(R = 1|\overline{X})}{P(R = 0|\overline{X})} \propto \frac{\prod_{t_j \in \overline{X}} p_j^{(1)} \prod_{t_j \notin \overline{X}} (1 - p_j^{(1)})}{\prod_{t_j \in \overline{X}} p_j^{(0)} \prod_{t_j \notin \overline{X}} (1 - p_j^{(0)})} \tag{9.8}$$

The constant of proportionality is used here because the document-independent ratio α_1/α_0 is ignored in the above expression. We can rearrange the above expression to within a constant of proportionality and make it dependent only on the terms occurring in $\overline{X}$. This is achieved by multiplying both sides of Eq. 9.8 with a document-independent term, and then dropping it only from the left-hand side (which retains the proportionality relationship

[4] In all the previous discussions on machine learned information retrieval, the training data is not specific to a particular query. However, each set of values of the extracted features is query-specific and multiple queries are represented in the same training data. The importance of the query-specific values of the meta-features (e.g., zones, authorship, location) of the document is learned with feedback data.

because of document-independence):

$$\frac{P(R=1|\overline{X})}{P(R=0|\overline{X})} \cdot \underbrace{\frac{\prod_{t_j}(1-p_j^{(0)})}{\prod_{t_j}(1-p_j^{(1)})}}_{\text{Document Independent}} \propto \frac{\prod_{t_j \in \overline{X}} p_j^{(1)}(1-p_j^{(0)})}{\prod_{t_j \in \overline{X}} p_j^{(0)}(1-p_j^{(1)})} \quad \text{[Multiplying both sides]}$$

$$\frac{P(R=1|\overline{X})}{P(R=0|\overline{X})} \propto \prod_{t_j \in \overline{X}} \frac{p_j^{(1)}(1-p_j^{(0)})}{p_j^{(0)}(1-p_j^{(1)})} \quad \text{[Dropping document-independent term on LHS]}$$

The logarithm of the above quantity is used in order to compute an additive form of the relevance score. The additive form of the ranking score is always desirable because it can be computed using inverted indexes and accumulators, as discussed on page 268. In the binary independence model, the *retrieval status value* $RSV_{bi}(\overline{X}, \overline{Q})$ of document $\overline{X}$ with respect to query vector $\overline{Q}$ can be expressed in terms of a summation only over the terms present in the document $\overline{X}$ as follows:

$$RSV_{bi}(\overline{X}, \overline{Q}) = \sum_{t_j \in \overline{X}} \log \left(\frac{p_j^{(1)}(1-p_j^{(0)})}{p_j^{(0)}(1-p_j^{(1)})} \right) \tag{9.9}$$

For any term t_j that is not included in the binary query vector $\overline{Q}$, it is assumed that the term is distributed in a similar way across relevant and non-relevant documents. This is equivalent to assuming that $p_j^{(0)} = p_j^{(1)}$ for terms not in $\overline{Q}$, which results in the dropping of non-query terms from the right-hand side of Eq. 9.9. This results in a retrieval status value that is expressed only as a summation over the *matching* terms in the query and the document:

$$RSV_{bi}(\overline{X}, \overline{Q}) = \sum_{t_j \in \overline{X}, t_j \in \overline{Q}} \log \left(\frac{p_j^{(1)}(1-p_j^{(0)})}{p_j^{(0)}(1-p_j^{(1)})} \right) \tag{9.10}$$

An important point here is that the quantities such as $p_j^{(0)}$ and $p_j^{(1)}$ are not usually available on a query-specific basis, unless the human is actively involved in providing relevance feedback to the results of queries. Some systems do allow the user to actively enter relevance feedback values. In these cases, a list of results is presented to the user based on some matching model for retrieval. The user then indicates which results are relevant. As a result, one can now label documents as *relevant* or *non-relevant*, which is a straightforward classification setting. In such cases, the problem of parameter estimation of $p_j^{(0)}$ and $p_j^{(1)}$ becomes *identical* to the way in which parameters are estimated in the Bernoulli model in Sect. 5.3.1. As in Sect. 5.3.1, Laplacian smoothing is used in order to provide more robust estimates of the probabilities. Note that such an approach has to perform the parameter estimation in real time after receiving the feedback, so that the next round of ranking can be presented to the user on the basis of computed values of $RSV_{bi}(\overline{X}, \overline{Q})$. If R is the number of relevant documents out of N documents, and r_j is the number of relevant documents containing term t_j out of n_j such documents, then the values of $p_j^{(0)}$ and $p_j^{(1)}$ are set as follows:

$$p_j^{(1)} = \frac{r_j + 0.5}{R+1}, \qquad p_j^{(0)} = \frac{n_j - r_j + 0.5}{N - R + 1} \tag{9.11}$$

The constant values of 0.5 and 1 are respectively added to the numerator and denominator for smoothing.

However, not all systems are able to use the feedback to specific queries in real time. In such cases, a number of simplifying assumptions are used in order to estimate quantities such as $p_j^{(0)}$ and $p_j^{(1)}$. For any term t_j that is included in the binary query vector $\overline{Q}$, the value of $p_j^{(1)}$ is assumed to be a large constant[5] such as 0.5 (relative to fractional occurrences of random terms in documents) because the terms in the queries are generally extremely relevant. However, such terms can also be present in non-relevant documents at a statistical frequency that is similar to that of the remaining collection. The value of $p_j^{(0)}$ is computed based on the statistical frequency of the term t_j across the whole collection. The frequency of the term across the whole collection is n_j/n, where n_j is the number of documents in the whole collection in which the term t_j occurs and n be the total number of documents. Then, the value of $p_j^{(0)}$ is set to n_j/n.

Therefore, if relevance feedback is not used, the retrieval status value with the binary independence model, specific to query $\overline{Q}$ and candidate document $\overline{X}$ is given by substituting $p_j^{(1)} = 0.5$ and $p_j^{(0)} = n_j/n$ in Eq. 9.10:

$$RSV_{bi}(\overline{X}, \overline{Q}) = \sum_{t_j \in \overline{X}, t_j \in \overline{Q}} \log\left(\frac{n - n_j}{n_j}\right) \approx \sum_{t_j \in \overline{X}, t_j \in \overline{Q}} \log\left(\frac{n}{n_j}\right) \tag{9.12}$$

An alternative version of the expression above is also used, when Laplacian smoothing is desired:

$$RSV_{bi}(\overline{X}, \overline{Q}) = \sum_{t_j \in \overline{X}, t_j \in \overline{Q}} \log\left(\frac{n - n_j + 0.5}{n_j + 0.5}\right) \tag{9.13}$$

It is easy to see that the expression on the right-hand side of Eq. 9.12 is equal to the sum of the inverse document frequency (idf) weights over matching terms. Therefore, one can even view this probabilistic model as a theoretical confirmation of the soundness of using inverse document frequencies in other similarity functions (e.g., cosine similarity) for computing matching scores. There are, however, several key differences of this model from the cosine similarity. First, the term frequencies of documents are not used, and secondly, document length normalization is missing. The fact that the term frequency is missing is a consequence of the fact that the documents are treated as binary vectors. Unfortunately, the missing term frequencies and document-length normalization do hurt the retrieval performance. Nevertheless, the binary independence model provides an initial template for constructing a more refined probabilistic model using the term frequencies that accounts for the same factors as the cosine similarity, but is better grounded in terms of probabilistic interpretation. This model is referred to as the *BM25* model, which is discussed in the next section.

9.3.3 The BM25 Model with Term Frequencies

The BM25 model, which is also referred to as the *Okapi* model, augments the binary independence model with term frequencies and document length normalization in order to improve the retrieved results. Let $(x_1 \ldots x_d)$ represent the raw term frequencies in document $\overline{X}$ without any form of frequency damping[6] or inverse document frequency (idf) normaliza-

[5] This was one of the earliest ideas proposed by Croft and Harper [119]. However, other alternatives are possible. Sometimes, a few relevant documents may be available, which can be used to estimate $p_j^{(1)}$. The other idea is to allow $p_j^{(1)}$ to rise with the number of documents n_j containing term t_j. For example, one can use $p_j^{(1)} = \frac{1}{3} + \frac{2 \cdot n_j}{3 \cdot n}$ [184].

[6] As discussed earlier, the square root or logarithm is frequently applied to term frequencies to reduce the impact of repeated words.

tion. Similarly, let $\overline{Q} = (q_1 \ldots q_d)$ represent the term frequencies in the query Q. Then, the retrieval status value $RSV_{bm25}(\overline{X}, \overline{Q})$ is closely related to that of the binary independence model:

$$RSV_{bm25} = \sum_{t_j \in \overline{Q}} \underbrace{\left(\log \frac{p_j^{(1)}(1 - p_j^{(0)})}{p_j^{(0)}(1 - p_j^{(1)})} \right)}_{\approx idf_j} \cdot \underbrace{\frac{(k_1 + 1)x_j}{k_1(1 - b) + b \cdot L(\overline{X}) + x_j}}_{\text{doc. frequency/length impact}} \cdot \underbrace{\frac{(k_2 + 1)q_j}{k_2 + q_j}}_{\text{query impact}} \tag{9.14}$$

The values k_1, k_2, and b are parameters, which respectively regulate impact of document term frequency, query term frequency, and document length normalization, respectively. The first term is identical to that in the binary independence model and can be simplified in a similar way to Eq. 9.13. The second term incorporates the impact of term frequencies and the document lengths. Small values of k_1 lead to the frequencies of the term being ignored, and large values of k_1 lead to linear weighting with the term frequency x_j. Intermediate values[7] of $k_1 \in (1, 1.5)$ have the same effect as that of applying the square-root or logarithm to the term frequency in order to reduce excessive impact of repeated term occurrences. The expression $L(\overline{X})$ is the normalized length of document $\overline{X}$, which is the ratio of its length to the average length of a document in the collection. Note that $L(\overline{X})$ will be larger than 1 for long documents. The parameter b is helpful for document length normalization. Setting the value of b to 0 results in no document length normalization, whereas setting the value of $b = 1$ leads to maximum normalization. A typical value of $b = 0.75$ is used. The parameter k_2 serves the same purpose as k_1, except that it does so for the query document frequencies. The choice of k_2 is, however, not quite as critical because the query documents are typically short with few repeated occurrences of terms. In such cases, almost any choice of $k_2 \in (1, 10)$ will yield similar results, and in some cases the entire term for query frequency normalization is dropped. Query length normalization is unnecessary because it is a proportionality factor that does not affect ranking of documents. Unlike the binary independence model, the summation is over all the terms in the query $\overline{Q}$ rather than only the matching terms between $\overline{Q}$ and $\overline{X}$. However, since the value of x_j is multiplicatively included in the expression, the absence of the query term in a document will automatically set its retrieval status value to 0. This is important because it means that only documents with matching terms contribute to the ranking score, and it is possible to perform query processing with an inverted index. One can express the first term in a data-driven manner, which is similar to Eq. 9.13:

$$RSV_{bm25} = \sum_{t_j \in \overline{Q}} \underbrace{\left(\log \frac{N - n_j + 0.5}{n_j + 0.5} \right)}_{\approx idf_j} \cdot \underbrace{\frac{(k_1 + 1)x_j}{k_1(1 - b) + b \cdot L(\overline{X}) + x_j}}_{\text{doc. frequency/length impact}} \cdot \underbrace{\frac{(k_2 + 1)q_j}{k_2 + q_j}}_{\text{query impact}} \tag{9.15}$$

The aforementioned expression is for cases where relevance feedback is not available. If relevance feedback is available, then the values of $p_j^{(0)}$ and $p_j^{(1)}$ in the first term are set using Eq. 9.11. Since the retrieval status value is computed in an additive way over query terms, and only matching documents are relevant, one can use the document-at-a-time (page 270) query processing technique in order to evaluate the score.

[7] Such values of k_1 are recommended in TREC experiments.

9.3.4 Statistical Language Models in Information Retrieval

A statistical language model assigns a probability to a sequence of words in a given language in a data-driven manner. In other words, given a corpus of documents, the language model estimates the probability that it was generated using this model. The use of language models in information retrieval is based on the intuition that users often formulate queries based on terms that are likely to appear in the returned documents. In some cases, even the ordering of the terms in the query might be chosen on the basis of the expected sequence of terms in the document. Therefore, if the user creates a *language model* for each document, it effectively provides a language model for the query. In other words, the assumption is that the document and query were generated from the same model. Documents can then be scored by computing the posterior probability of generating the document from the same model as the query. This is a fundamentally different notion from the concept of relevance that is used in the binary independence and BM25 models for ranking documents.

A language model for a document provides a generative process of constructing the document. The most *primitive* language model is the *unigram* language model in which no sequence information is used and only the frequencies of terms are used. The basic assumption is that each *token* in a document is generated by rolling a die independently from the previous tokens in the document, where each face of the die shows a particular term. Note that the unigram language model creates a multinomial distribution of terms in a document, as discussed in Chaps. 4 and 5. Therefore, the unigram language model can be fully captured by using the probabilities of the different terms in the documents and no information about the sequence of the terms in the collection.

More complex language models such as bigram and n-gram language models use sequence information. A bigram language model uses only the previous term to predict the term at a particular position, and a trigram model uses the previous two words. In general, an n-gram model uses the previous $(n-1)$ terms to predict a term at a particular position. In this case, the parameters of the model correspond to the conditional probabilities of tokens, given a fixed set of previous $(n-1)$ tokens. An n-gram model falls in the broad category of Markovian models, which refers to a *short-memory assumption*. In this particular case, only a history of $(n-1)$ terms in the sequence is used to predict the current term, and therefore the amount of memory used for modeling is limited by the parameter n. Large values of n result in theoretically more accurate models (i.e., lower bias), but sufficient data is often not available to estimate the exponentially increasing number of parameters of the model (i.e., higher variance). As a practical matter, only small values of n can be used in a realistic way because of the rapid increase in the amount of data needed to estimate the parameters at large values of n. A broader discussion of language models is provided in Sect. 10.2 of Chap. 10, although this section will restrict the discussion to unigram language models. In general, unigram language models are used frequently because of their simplicity and the ease in estimation of the parameters with a limited amount of data.

9.3.4.1 Query Likelihood Models

How are language models used for information retrieval? Given a document $\overline{X}$, one can estimate the parameters of the language model, and then compute the posterior probabilities of $\overline{X}$, given the additional knowledge about the query $\overline{Q}$. Therefore, the overall approach may be described as follows:

1. Estimate the parameter vector $\overline{\Theta}_X$ of the language model $\mathcal{M}$ being used with the use of each candidate document $\overline{X}$ in the corpus. For example, if a unigram language

model is used, the parameter vector $\overline{\Theta}_X$ will contain the probabilities of the different faces of the die that generated $\overline{X}$. The value of $\overline{\Theta}_X$ can therefore be estimated as the fractional presence of various terms in $\overline{X}$. Note that each parameter vector $\overline{\Theta}_X$ is specific only to a particular document $\overline{X}$.

2. For a given query $\overline{Q}$, estimate the posterior probability $P(\overline{X}|\overline{Q})$. The documents are ranked on the basis of this posterior probability.

In order to compute the posterior probability, the Bayes rule is used:

$$P(\overline{X}|\overline{Q}) = \frac{P(\overline{X}) \cdot P(\overline{Q}|\overline{X})}{P(\overline{Q})} \propto P(\overline{X}) \cdot P(\overline{Q}|\overline{X})$$

The constant of proportionality above is identified as the document-independent term, which does not affect relative ranking. A further assumption is that the prior probability $P(\overline{X})$ is uniform over all documents. Therefore, we have the following:

$$P(\overline{X}|\overline{Q}) \propto P(\overline{Q}|\overline{X})$$

Finally, the value of $P(\overline{Q}|\overline{X})$ is computed by using the underlying language model, whose parameters were estimated using $\overline{X}$. This is the same as estimating $P(\overline{Q}|\overline{\Theta}_X)$.

In the context of a unigram model, this estimation takes on a particularly simple form. Let $\overline{\Theta}_X = (\theta_1 \ldots \theta_d)$ be the probabilities of the different terms in the collection, which were estimated using $\overline{X} = (x_1 \ldots x_d)$. The parameter θ_j can be estimated as follows:

$$\theta_j = \frac{x_j}{\sum_{j=1}^{d} x_j} \tag{9.16}$$

Then, the estimation of $P(\overline{Q}|\overline{\Theta}_X)$ can be accomplished using the multinomial distribution:

$$P(\overline{Q}|\overline{\Theta}_X) = \prod_{j=1}^{d} \theta_j^{q_j} \tag{9.17}$$

Note that the logarithm of Eq. 9.17 is additive in nature, and it can be computed efficiently with the use of an inverted index and accumulator variables (cf. page 268). Furthermore, the use of the logarithm avoids the multiplication of very small probabilities.

One way of understanding the query likelihood probability in the context of the unigram language model is that it estimates the probability that the query is generated as a sample of terms from the document. Of course, this interpretation would not be true for more complex language models like a bigram model.

One issue with this estimation is that it would give nonzero scores for a document only if it contained all the terms in that document. This is because the parameter vector $\overline{\Theta}_X$ is computed using a single document $\overline{X}$, which inevitably leads to a lot of zero values in the parameter vector. One can use the Laplacian smoothing methods that are commonly used for multinomial distributions, as discussed in Chaps. 4 and 5. Another option is to use *Jelinek-Mercer smoothing*, in which the value of θ_j is estimated using the statistics of both the document $\overline{X}$ and the whole collection. Let θ_j^X and θ_j^{All} be these two estimated values. The parameter θ_j^X is estimated as before, whereas the estimation of θ_j^{All} is simply he fraction of the tokens in the whole collection that are t_j. Then, the estimated value of θ_j is a convex combination of these two values with the use of the parameter $\lambda \in (0, 1)$:

$$\theta_j = \lambda \theta_j^X + (1 - \lambda)\theta_j^{All} \tag{9.18}$$

Using $\lambda = 1$ reverts to the aforementioned model without smoothing, whereas using $\lambda = 0$ causes so much smoothing that all documents tie with the same ranking score.

9.4 Web Crawling and Resource Discovery

Web crawlers are also referred to as *spiders* or *robots*. The primary motivation for Web crawling is that the resources on the Web are dispensed widely across globally distributed sites. While the Web browser provides a graphical user interface to access these pages in an interactive way, the full power of the available resources cannot be leveraged with the use of only a browser. In many applications, such as search and knowledge discovery, it is necessary to download all the relevant pages *at a central location* (or a modest number of distributed locations), to allow search engines and machine learning algorithms to use these resources efficiently. In this sense, search engines are somewhat different from information retrieval applications; even the compilation of the corpus for querying is a difficult task because of the open and vast nature of the Web.

Web crawlers have numerous applications. The most important and well-known application is search, in which the downloaded Web pages are indexed to provide responses to user keyword queries. All the well-known search engines, such as Google and Bing, employ crawlers to periodically refresh the downloaded Web resources at their servers. Such crawlers are also referred to as *universal crawlers* because they are intended to crawl all pages on the Web irrespective of their subject matter or location. Web crawlers are also used for business intelligence, in which the Websites related to a particular subject are crawled or the sites of a competitor are monitored and incrementally crawled as they change. Such crawlers are also referred to as *preferential crawlers* because they discriminate between the relevance of different pages for the application at hand.

9.4.1 A Basic Crawler Algorithm

While the design of a crawler is quite complex, with a distributed architecture and many processes or threads, the following describes a simple sequential and universal crawler that captures the essence of how crawlers are constructed.

A crawler uses the same mechanism used by browsers to fetch Web pages based on the *Hypertext Transfer Protocol* (HTTP). The main difference is that the fetching is now done by an automated program using automated selection decisions, rather than by the manual specification of a *Uniform Resource Locator (URL)* by a user with a Web browser. In all cases, a particular URL is fetched by the system. Both browsers and crawlers typically[8] use GET requests to fetch Web pages, which is a functionality provided by the HTTP protocol. The difference is that the GET request is invoked in a browser when a user clicks a link or enters a URL, whereas the GET request is invoked in an automated way by the crawler. In both cases, a *domain name system (DNS) server* is used to translate the URL into an internet protocol (IP) address. The program then connects to the server using that IP address and sends a GET request. In most cases, servers listen to requests at multiple *ports*, and port 80 is typically used for Web requests.

The basic crawler algorithm, described in a very general way, uses a seed set of Universal Resource Locators (URLs) S, and a selection algorithm $\mathcal{A}$ as the input. The algorithm $\mathcal{A}$ decides which document to crawl next from a current *frontier list* of URLs. The frontier list represents URLs extracted from the Web pages. These are the candidates for pages that can eventually be fetched by the crawler. The selection algorithm $\mathcal{A}$ is important because it regulates the basic strategy used by the crawler to discover the resources. For example, if new

[8]Browsers also use POST requests, when additional information is needed by the Web server. For example, an item is usually bought on the POST request. However, such requests are not used by crawlers because they might inadvertently causes actions (such as buying), which were not desired by the crawler.

```
Algorithm BasicCrawler(Seed URLs: S, Selection Algorithm: A)
begin
  FrontierList = S;
  repeat
    Use algorithm A to select URL X ∈ FrontierSet;
    FrontierList = FrontierList − {X};
    Fetch URL X and add to repository;
    Add all relevant URLs in fetched document X to
      end of FrontierList;
  until termination criterion;
end
```

Figure 9.6: The basic crawler algorithm

URLs are appended to the end of the frontier list, and the algorithm $\mathcal{A}$ selects documents from the beginning of the list, then this corresponds to a breadth-first algorithm.

The basic crawler algorithm proceeds as follows. First, the seed set of URLs is added to the frontier list. In each iteration, the selection algorithm $\mathcal{A}$ picks one of the URLs from the frontier list. This URL is deleted from the frontier list and then fetched using the GET request of the HTTP protocol. The fetched page is stored in a local repository, and the URLs inside it are extracted. These URLs are then added to the frontier list, provided that they have not already been visited. Therefore, a separate data structure, in the form of a hash table, needs to be maintained to store all visited URLs. In practical implementations of crawlers, not all unvisited URLs are added to the frontier list due to Web spam, spider traps, topical preference, or simply a practical limit on the size of the frontier list. After the relevant URLs have been added to the frontier list, the next iteration repeats the process with the next URL on the list. The process terminates when the frontier list is empty. If the frontier list is empty, it does not necessarily imply that the entire Web has been crawled. This is because the Web is not strongly connected, and many pages are unreachable from most randomly chosen seed sets. Because most practical crawlers such as search engines are *incremental* crawlers that refresh pages over previous crawls, it is usually easy to identify unvisited seeds from previous crawls and add them to the frontier list, if needed. With large seed sets, such as a previously crawled repository of the Web, it is possible to robustly crawl most pages. The basic crawler algorithm is described in Fig. 9.6.

Thus, the crawler is a graph-search algorithm that discovers the outgoing links from nodes by parsing Web pages and extracting the URLs. The choice of the selection algorithm $\mathcal{A}$ will typically result in a bias in the crawling algorithm, especially in cases where it is impossible to crawl all the relevant pages due to resource limitations. For example, a breadth-first crawler is more likely to crawl a page with many links pointing to it. Interestingly, such biases are sometimes desirable in crawlers because it is impossible for any crawler to index the entire Web. Because the indegree of a Web page is often closely related to its *PageRank*, a measure of a Web page's quality, this bias is not necessarily undesirable. Crawlers use a variety of other selection strategies defined by the algorithm $\mathcal{A}$.

Because most universal crawlers are incremental crawlers that are intended to refresh previous crawls, it is desirable to crawl frequently changing pages. The explicit detection of whether a Web page has been changed can be done at a relatively low cost using the HEAD request of the HTTP protocol. The HEAD request receives only the header information from a Web page at a lower cost than crawling the Web page. The header information also contains the last date at which the Web document was modified. This date is compared with that obtained from the previous fetch of the Web page (using a GET request). If the date has changed, then the Web page needs to be crawled again.

The use of the HEAD request reduces the cost of crawling a Web page, although it

still imposes some burden on the Web server. Therefore, the crawler needs to implement some internal mechanisms in order to estimate the frequency at which a Web page changes (without actually issuing any requests). This type of internal estimation helps the crawler in minimizing the number of fruitless requests to Web servers. Specific types of Web pages such as news sites, blogs, and portals might change frequently, whereas other types of pages may change slowly. The change frequency can be estimated from repeated previous crawls of the same page or by using learning algorithms that factor in specific characteristics of the Web page. Some resources such as news portals are updated frequently. Therefore, frequently updated pages may be selected by the algorithm $\mathcal{A}$. Other than the change frequency, another factor is the *popularity and usefulness* of Web pages to the general public. Clearly, it is desirable to crawl popular and useful pages more frequently. Therefore, the selection algorithm $\mathcal{A}$ may specifically choose Web pages with high *PageRank* from frontier list. The computation of *PageRank* is discussed in Sect. 9.6.1. The use of *PageRank* as a criterion for selecting Web pages to be crawled is closely related to that of preferential crawlers.

9.4.2 Preferential Crawlers

In the preferential crawler, only pages satisfying a user-defined criterion need to be crawled. This criterion may be specified in the form of keyword presence in the page, a topical criterion defined by a machine learning algorithm, a geographical criterion about page location, or a combination of the different criteria. In general, an arbitrary predicate may be specified by the user, which forms the basis of the crawling. In these cases, the major change is to the approach used for updating the frontier list during crawling, and also the order of selecting the URLs from the frontier list.

1. The Web page needs to meet the user-specified criterion in order for its extracted URLs to be added to the frontier list.

2. In some cases, the anchor text may be examined to determine the relevance of the Web page to the user-specified query.

3. In context-focused crawlers, the crawler is trained to learn the likelihood that relevant pages are within a short distance of the page, even if the Web page is itself not directly relevant to the user-specified criterion. For example, a Web page on "*data mining*" is more likely to point to a Web page on "*information retrieval*," even though the data mining page may not be relevant to the query on "*information retrieval*." URLs from such pages may be added to the frontier list. Therefore, heuristics need to be designed to learn such context-specific relevance.

Changes may also be made to the algorithm $\mathcal{A}$. For example, URLs with more relevant anchor text, or with relevant tokens in the Web address, may be selected first by algorithm $\mathcal{A}$. A URL such as http://www.golf.com, with the word "*golf*" in the Web address may be more relevant to the topic of "*golf*," than a URL without the word in it. The bibliographic notes contain pointers to a number of heuristics that are commonly used for preferential resource discovery.

A number of simple techniques, such as the use of *PageRank* to preferential crawl Web pages, greatly enhance the popularity of the Web pages crawled. This type of approach ensures that only "hot" Web pages are crawled, which are of interest to many users. Therefore, these types of preferential crawlers are desirable when one has limited resources to crawl pages, although the goal is often similar to that of a universal crawler algorithm.

Other types of preferential crawlers include *focused crawling* or *topical crawling*. In focused crawling, the crawler is biased using a particular type of classifier. The first step is to build a classification model based on an open resource of Web pages such as the *Open Directory*[9] *Project (ODP)*. When a crawled Web page belongs to a desired category, then the URLs inside it are added to frontier list. The URLs can be scored in various ways for ordering on frontier list, such as the level of classifier relevance of their parent Web pages as well as their recency.

In topical crawling, labeled examples are not available to the crawler. Only a set of seed pages and a description of the topic of interest is available. In many cases, a description of the topic of interest is provided by the user with a short query. A common approach in this case is to use the *best-first* algorithm in which the (crawled or seed) Web pages containing the most number of user-specified keywords are selected, and the URLs inside them are preferentially added to the frontier list. As in the case of focused crawling, the topical relevance (e.g., matching between query and Web page) of the parent Web pages of the URLs on frontier list as well as their recency can be used to order them.

9.4.3 Multiple Threads

Crawlers typically use multiple threads to improve efficiency. You might have noticed that it can sometimes take a few seconds for your Web browser to fulfill your URL request. This situation is also encountered in a Web crawler, which idles while the server at the other end satisfies the GET request. It makes sense for Web crawlers to use this idle time in order to fetch more Web pages. A natural way to speed up the crawling is by leveraging concurrency. The idea is to use multiple threads of the crawler that update a shared data structure for visited URLs and the page repository. In such cases, it is important to implement concurrency control mechanisms for locking or unlocking the relevant data structures during updates. The concurrent design can significantly speed up a crawler with more efficient use of resources. In practical implementations of large search engines, the crawler is distributed geographically with each "sub-crawler" collecting pages in its geographical proximity.

One problem with this approach is that if hundreds of requests are made to the Web server at a single site, it would result in an unreasonable load on the server, which also has to serve URL requests from other clients. Therefore, Web crawlers often use politeness policies, in which a page is not crawled from a Web server, if one has been crawled recently. This is achieved by creating per-server queues, and disallowing a fetch from a particular queue, if a page has been crawled from it within a particular time window.

9.4.4 Combatting Spider Traps

The main reason that the crawling algorithm always visits distinct Web pages is that it maintains a list of previously visited URLs for comparison purposes. However, some shopping sites create dynamic URLs in which the last page visited is appended at the end of the user sequence to enable the server to log the user action sequences within the URL for future analysis. For example, when a user clicks on the link for *page2* from http://www.examplesite.com/page1, the new dynamically created URL will be http://www.examplesite.com/page1/page2. Pages that are visited further will continue to be appended to the end of the URL, even if these pages were visited before. A natural way to combat this is to limit the maximum size of the URL. Furthermore, a maximum limit may also be placed on the number of URLs crawled from a particular site.

[9] http://www.dmoz.org.

9.4.5 Shingling for Near Duplicate Detection

One of the major problems with the Web pages collected by a crawler is that many duplicates of the same page may be crawled. This is because the same Web page may be mirrored at multiple sites. Therefore, it is crucial to have the ability to detect near duplicates. An approach known as *shingling* is commonly used for this purpose.

A k-shingle from a document is simply a string of k consecutively occurring words in the document. A shingle can also be viewed as a k-gram. For example, consider the document comprising the following sentence:

Mary had a little lamb, its fleece was white as snow.

The set of 2-shingles extracted from this sentence is "*Mary had*", "*had a*", "*a little*", "*little lamb*", "*lamb its*", "*its fleece*", "*fleece was*", "*was white*", "*white as*", and "*as snow*". Note that the number of k-shingles extracted from a document is no longer than the length of the document, and 1-shingles are simply the set of words in the document. Let S_1 and S_2 be the k-shingles extracted from two documents D_1 and D_2. Then, the shingle-based similarity between D_1 and D_2 is simply the Jaccard coefficient between S_1 and S_2.

$$J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \tag{9.19}$$

The advantage of using k-shingles instead of the individual words (1-shingles) for Jaccard coefficient computation is that shingles are less likely than words to repeat in different documents. There are r^k distinct shingles for a lexicon of size r. For $k \geq 5$, the chances of many shingles recurring in two documents becomes very small. Therefore, if two documents have many k-shingles in common, they are very likely to be near duplicates. To save space, the individual shingles are hashed into 4-byte (32-bit) numbers that are used for comparison purposes. Such a representation also enables better efficiency.

9.5 Query Processing in Search Engines

The broad framework for query processing in search engines is inherited from traditional information retrieval, as discussed in Sects. 9.2 and 9.3. All the data structures introduced in earlier sections, such as dictionaries (cf. Sect. 9.2.1) and inverted indexes (cf. Sect. 9.2.2) are used in search engines, albeit with some modifications to account for the large size of the Web and the immense burden placed on Web servers by the large numbers of queries.

After the documents have been crawled, they are leveraged for query processing. The following are the primary stages in search index construction:

1. *Preprocessing:* This is the stage in which the search engine preprocesses the crawled documents to extract the tokens. Web pages require specialized preprocessing methods, which are discussed in Chap. 2. A substantial amount of meta-information about the Web page is also collected, which is often useful in enabling query processing. This meta-information might include information like the date of the document, its geographical location, or even its *PageRank* based on the methodology in Sect. 9.6.1. Note that measures such as *PageRank* need to be computed up front because they are expensive to compute, and cannot be reasonably computed during query processing time.

2. *Index construction:* Most of the data structures discussed in Sect. 9.2 carry over to the search engine setting. In particular, the inverted indexes are required to respond to queries, and dictionary data structure is required to map terms to offsets in inverted files and conveniently access the relevant inverted lists for each term. However, there are many issues of scalability when using such data structures. For example, the size of the Google index is of the order of a hundred trillion documents as of 2018, and it continues to grow over time. In most cases, it becomes cost effective and economical to build distributed indexes that are highly fault tolerant. Fault tolerance is achieved by replicating the index over multiple machines. Therefore, distributed *MapReduce* methods [128] are often used in the process of index construction.

3. *Query processing:* This preprocessed collection is utilized for online query processing. The relevant documents are accessed and then ranked using both their relevance to the query and their quality. The basic technique of query processing with an inverted index is discussed in Sect. 9.2.4, and a number of additional information retrieval models for scoring are discussed in Sect. 9.3. User feedback on search results is often used to construct machine learning methods on extracted meta-features such as zones, positional data, document meta-information, and linkage features such as the *PageRank*. As long as all these characteristics can be combined in an additive way to create a score, accumulators can be used in conjunction with inverted indexes for efficient query processing (cf. page 268). However, query processing in search engines is more challenging than in a traditional information retrieval system because of the high volume of the search and the distributed nature of the indexes. In such cases, the use of techniques like *tiered* inverted lists and skip pointers becomes essential. These types of optimizations can lead to large speedups because huge portions of the inverted lists are not accessed at all.

Some of these issues are discussed in the following subsections.

9.5.1 Distributed Index Construction

In most large search engines, the indexes are distributed over multiple nodes. One can partition the inverted indexes by storing different inverted lists at different nodes (term-wise partitioning) or by partitioning the different documents over multiple nodes. Although partitioning by terms might seem to be natural in the inverted list setting, it creates some challenges for query processing. Consider a situation, where the user enters the keywords "*text mining*" and the inverted lists for "*text*" and "*mining*" are located in different nodes. This means that one now has to send one of the two inverted lists from one node to the other to perform the intersection operation. This can create inefficiencies during query processing.

On the other hand, document-wise partitioning distributes the documents across different nodes and maintains all the inverted lists for that subset of documents in a single node. This means that all the intersection operations and intermediate computations can occur within the individual nodes. However, the global statistics such as the frequencies of terms in documents (for computing idf) need to be computed in a global way across all documents using distributed processes and then stored at individual nodes along with the terms. A hash function is used on each URL in order to distribute the different Web pages to nodes in a uniform way. For query processing, a given query is first broadcast to all the nodes, and computations/mergings are performed within the nodes. The top results from each node are returned and then combined in order to yield the top results for the query.

9.5.2 Dynamic Index Updates

Another important issue with search engines is that new documents are continually created and deleted, which creates problems when multiple inverted lists are stored in a single file. This is because inserting postings in a single file containing multiple inverted lists will affect the offset locations of all the postings in the dictionary. Although some techniques like *logarithmic merging* are available in the literature, the drawback is that such methods increase the complexity of query processing, and it is crucially required to very efficient at query processing time. Therefore, many large search engines periodically reconstruct the indexes from scratch as new crawled pages are discovered by spiders, and old pages are removed. The trade-offs associated with different ways of dynamic index maintenance are discussed in [281].

9.5.3 Query Processing

Web pages have different types of text in them, which can be used for various types of ad hoc query optimizations. An example is that of treating the anchor and title text differently from the body of a document during query processing. For a given set of terms in a query, all the relevant inverted lists are accessed, and the intersection of these inverted lists is determined. One can use accumulators to score and rank the different documents. Typically, to speed up the process, two indexes are constructed. A smaller index is constructed on only the titles of the Web page, or anchor text of pages *pointing to the page*. If enough documents are found in the smaller index, then the larger index is not referenced. Otherwise, the larger index is accessed. The logic for using the smaller index is that the title of a Web page and the anchor text of Web pages pointing to it, are usually highly representative of the content in the page.

If only the textual content of a Web page is used, the number of pages returned for common queries may be of the order of millions or more. Obviously, such a large number of query results will not be easy for a human user to assimilate. A typical browser interface will present only the first few (say 10) results to the human user in a single view of the search results, with the option of browsing other less relevant results. Therefore, the methodology for ranking needs to be very robust to ensure that the top results are highly relevant, As discussed in Sect. 9.3, information retrieval techniques always extract meta-features beyond the tf-idf scores in order to enable high-quality retrieval. While the exact scoring methodology used by commercial engines is proprietary, a number of factors are known to influence the content-based score:

1. A word is given different weights, depending upon whether it occurs in the title, body, URL token, or the anchor text of a pointing Web page. The occurrence of the term in the title or the anchor text of a Web page pointing to that page is generally given higher weight. This is similar to the notion of zoning discussed in Sect. 9.2.4.8. The weights of various zones can be learned using the machine learning techniques discussed in Sect. 9.2.4.9.

2. The prominence of a term in font size and color may be leveraged for scoring. For example, larger font sizes will be given a larger score.

3. When multiple keywords are specified, their relative positions in the documents are used as well. For example, if two keywords occur close together in a Web page, then this increases the score.

4. The most important meta-feature that is used by search engines is a *reputation-based* score of the Web page, which is also referred to as the *PageRank*.

The weighting of many of the aforementioned features during query processing requires the use of machine learning techniques in information retrieval. When a user chooses a Web page from among the responses to a search result in preference to earlier ranked results, this is clear evidence of the relevance of that page to the user. As discussed in Sect. 9.2.4.9 such data can be used by search engines to learn the importance of the various features. Furthermore, as long as the scoring computation is additive over the various meta-features, one can use the inverted index in combination with accumulator variables in order to enable efficient query processing (cf. page 268).

9.5.4 The Importance of Reputation

One of the most important meta-features used in Web search is the *reputation*, or the *quality*, of the page. This meta-feature corresponds to the *PageRank*. It is important to use such mechanisms because of the uncoordinated and open nature of Web development. After all, the Web allows anyone to publish almost anything, and therefore there is little control on the quality of the results. A user may publish incorrect material either because of poor knowledge on the subject, economic incentives, or with a deliberately malicious intent of publishing misleading information.

Another problem arises from the impact of *Web spam*, in which Website owners intentionally serve misleading content to rank their results higher. Commercial Website owners have significant economic incentives to ensure that their sites are ranked higher. For example, an owner of a business on golf equipment, would want to ensure that a search on the word "*golf*" ranks his or her site as high as possible. There are several strategies used by Website owners to rank their results higher.

1. *Content-spamming:* In this case, the Web host owner fills up repeated keywords in the hosted Web page, even though these keywords are not actually visible to the user. This is achieved by controlling the color of the text and the background of the page. Thus, the idea is to maximize the content relevance of the Web page to the search engine, without a corresponding increase in the *visible* level of relevance.

2. *Cloaking:* This is a more sophisticated approach, in which the Website serves different content to crawlers than it does to users. Thus, the Web site first determines whether the incoming request is from a crawler or from a user. If the incoming request is from a user, then the actual content (e.g., advertising content) is served. If the request is from a crawler, then the content that is most relevant to specific keywords is served. As a result, the search engine will use different content to respond to user search requests from what a Web user will actually see.

It is obvious that such spamming will significantly reduce the quality of the search results. Search engines also have significant incentives to improve the quality of their results to support their paid advertising model, in which the *explicitly marked* sponsored links appearing on the side bar of the search results are truly paid advertisements. Search engines do not want advertisements (disguised by spamming) to be served as *bona fide* results to the query, especially when such results reduce the quality of the user experience. This has led to an adversarial relationship between search engines and spammers, in which the former use reputation-based algorithms to reduce the impact of spam. At the other end of Website owners, a *Search Engine Optimization (SEO)* industry attempts to optimize search results by using their knowledge of the algorithms used by search engines, either through the general principles used by engines or through reverse engineering of search results.

For a given search, it is almost always the case that a small subset of the results is more informative or provides more accurate information. How can such pages be determined? Fortunately, the Web provides several natural voting mechanisms to determine the reputation of pages. The citation structure of Web pages is the most common mechanism used to determine the quality of Web pages. When a page is of high quality, many other Web pages point to it. A citation can be logically viewed as a vote for the Web page. While the number of in-linking pages can be used as a rough indicator of the quality, it does not provide a complete view because it does not account for the quality of the pages pointing to it. To provide a more holistic citation-based vote, an algorithm referred to as *PageRank* is used.

It should be pointed out, that citation-based reputation scores are not completely immune to other types of spamming that involve coordinated creation of a large number of links to a Web page. Furthermore, the use of anchor text of *pointing* Web pages in the content portion of the rank score can sometimes lead to amusingly irrelevant search results. For example, a few years back, a search on the keyword "*miserable failure*" in the Google search engine, returned as its top result, the official biography of a previous president of the United States. This is because many Web pages were constructed in a coordinated way to use the anchor text "*miserable failure*" to point to this biography. This practice of influencing search results by coordinated linkage construction to a particular site is referred to as *Googlewashing*. Such practices are less often economically motivated, but are more often used for comical or satirical purposes.

Therefore, the ranking algorithms used by search engines are not perfect but have, nevertheless, improved significantly over the years. The algorithms used to compute the reputation-based ranking score will be discussed in the next section.

9.6 Link-Based Ranking Algorithms

The *PageRank* algorithm uses the linkage structure of the Web for reputation-based ranking. The *PageRank* method is independent of the user-query, because it only precomputes the reputation portion of the score. Eventually, the reputation portion of the score is combined with other content-scoring methods like BM25, and the weights of the different components are regulated by learning-to-rank methods (e.g., ranking SVM of Sect. 9.2.4.10). The *HITS* algorithm is query-specific. It uses a number of intuitions about how authoritative sources on various topics are linked to one another in a hyperlinked environment.

9.6.1 PageRank

The *PageRank* algorithm models the importance of Web pages with the use of the citation (or linkage) structure in the Web. The basic idea is that highly reputable documents are more likely to be cited (or in-linked) by other reputable Web pages.

A random surfer model on the Web graph is used to achieve this goal. Consider a random surfer who visits random pages on the Web by selecting random links on a page. The long-term relative frequency of visits to any particular page is clearly influenced by the number of in-linking pages to it. Furthermore, the long-term frequency of visits to any page will be higher if it is linked to by other frequently visited (or *reputable*) pages. In other words, the *PageRank* algorithm models the reputation of a Web page in terms of its long-term frequency of visits by a random surfer. This long-term frequency is also referred to as the *steady-state probability*. This model is also referred to as the *random walk model*.

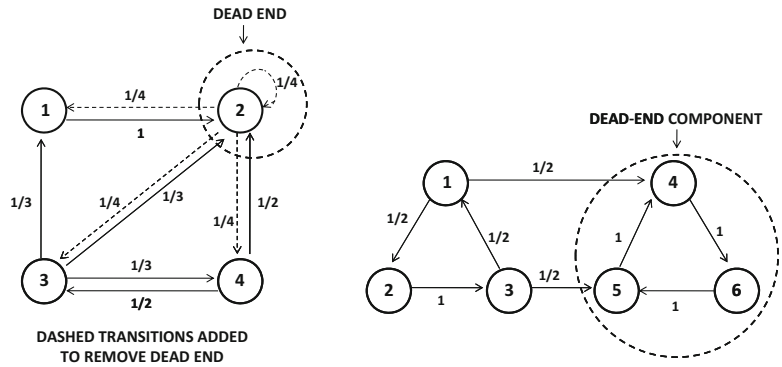
The basic random surfer model does not work well for all possible graph topologies. A critical issue is that some Web pages may have no outgoing links, which may result in the random surfer getting trapped at specific nodes. In fact, a probabilistic transition is not even meaningfully defined at such a node. Such nodes are referred to as *dead ends*. An example of a dead-end node is illustrated in Fig. 9.7a. Clearly, dead ends are undesirable because the transition process for *PageRank* computation cannot be defined at that node. To address this issue, two modifications are incorporated in the random surfer model. The first modification is to add links from the dead-end node (Web page) to all nodes (Web pages), including a self-loop to itself. Each such edge has a transition probability of $1/n$. This does not fully solve the problem, because the dead ends can also be defined on *groups of nodes*. In these cases, there are no outgoing links from *a group of nodes* to the remaining nodes in the graph. This is referred to as a *dead-end component*, or *absorbing component*. An example of a dead-end component is illustrated in Fig. 9.7b.

Dead-end components are common in the Web graph because the Web is not strongly connected. In such cases, the transitions at individual nodes can be meaningfully defined, but the steady-state transitions will stay trapped in these dead-end components. All the steady-state probabilities will be concentrated in dead-end components because there can be no transition out of a dead-end component after a transition occurs into it. Therefore, as long as even a minuscule probability of transition into a dead-end component[10] exists, *all* the steady-state probability becomes concentrated in such components. This situation is not desirable from the perspective of *PageRank* computation in a large Web graph, where dead-end components are not necessarily an indicator of popularity. Furthermore, in such cases, the final probability distribution of nodes in various dead-end components is not unique and it is dependent on the starting state. This is easy to verify by observing that random walks starting in different dead-end components will have their respective steady-state distributions concentrated within the corresponding components.

While the addition of edges solves the problem for dead-end nodes, an additional step is required to address the more complex issue of dead-end components. Therefore, aside from the addition of these edges, a *teleportation*, or *restart step* is used within the random surfer model. This step is defined as follows. At each transition, the random surfer may either jump to an arbitrary page with probability α, or it may follow one of the links on the page with probability $(1-\alpha)$. A typical value of α used is 0.1. Because of the use of teleportation, the steady state probability becomes unique and independent of the starting state. The value of α may also be viewed as a *smoothing* or *damping probability*. Large values of α typically result in the steady-state probability of different pages to become more even. For example, if the value of α is chosen to be 1, then all pages will have the same steady-state probability of visits.

How are the steady-state probabilities determined? Let $G = (N, A)$ be the directed Web graph, in which nodes correspond to pages, and edges correspond to hyperlinks. The total number of nodes is denoted by n. It is assumed that A also includes the added edges from dead-end nodes to all other nodes. The set of nodes incident on i is denoted by $In(i)$, and the set of end points of the outgoing links of node i is denoted by $Out(i)$. The steady-state probability at a node i is denoted by $\pi(i)$. In general, the transitions of a Web surfer can be visualized as a *Markov chain*, in which an $n \times n$ transition matrix P is defined for a Web graph with n nodes. The *PageRank* of a node i is equal to the steady-state probability $\pi(i)$

[10] A formal mathematical treatment characterizes this in terms of the *ergodicity* of the underlying Markov chains. In ergodic Markov chains, a necessary requirement is that it is possible to reach any state from any other state using a sequence of one or more transitions. This condition is referred to as *strong connectivity*. An informal description is provided here to facilitate understanding.



(a) Dead-end node (b) Dead-end component

Figure 9.7: Transition probabilities for *PageRank* computation with different types of dead ends

for node i, in the Markov chain model. The probability[11] p_{ij} of transitioning from node i to node j, is defined as $1/|Out(i)|$. Examples of transition probabilities are illustrated in Fig. 9.7. These transition probabilities do not, however, account for teleportation which will be addressed[12] separately below.

Let us examine the transitions into a given node i. The steady-state probability $\pi(i)$ of node i is the sum of the probability of a teleportation into it and the probability that one of the in-linking nodes directly transitions into it. The probability of a teleportation into the node is exactly α/n because a teleportation occurs in a step with probability α, and all nodes are equally likely to be the beneficiary of the teleportation. The probability of a transition into node i is given by $(1-\alpha)\cdot\sum_{j\in In(i)}\pi(j)\cdot p_{ji}$, as the sum of the probabilities of transitions from different in-linking nodes. Therefore, at steady-state, the probability of a transition into node i is defined by the sum of the probabilities of the teleportation and transition events are as follows:

$$\pi(i) = \alpha/n + (1-\alpha)\cdot\sum_{j\in In(i)}\pi(j)\cdot p_{ji} \tag{9.20}$$

For example, the equation for node 2 in Fig. 9.7a can be written as follows:

$$\pi(2) = \alpha/4 + (1-\alpha)\cdot(\pi(1) + \pi(2)/4 + \pi(3)/3 + \pi(4)/2)$$

There will be one such equation for each node, and therefore it is convenient to write the entire system of equations in matrix form. Let $\overline{\pi} = (\pi(1)\ldots\pi(n))^T$ be the n-dimensional column vector representing the steady-state probabilities of all the nodes, and let $\overline{e}$ be an n-dimensional column vector of all 1 values. The system of equations can be rewritten in

[11] In some applications such as bibliographic networks, the edge (i, j) may have a weight denoted by w_{ij}. The transition probability p_{ij} is defined in such cases by $\frac{w_{ij}}{\sum_{j\in Out(i)} w_{ij}}$.

[12] An alternative way to achieve this goal is to modify G by multiplying existing edge transition probabilities by the factor $(1-\alpha)$ and then adding α/n to the transition probability between each pair of nodes in G. As a result G will become a directed clique with bidirectional edges between each pair of nodes. Such strongly connected Markov chains have unique steady-state probabilities. The resulting graph can then be treated as a Markov chain without having to separately account for the teleportation component. This model is equivalent to that discussed in the chapter.

matrix form as follows:

$$\overline{\pi} = \alpha \overline{e}/n + (1-\alpha) P^T \overline{\pi} \tag{9.21}$$

The first term on the right-hand side corresponds to a teleportation, and the second term corresponds to a direct transition from an incoming node. In addition, because the vector $\overline{\pi}$ represents a probability, the sum of its components $\sum_{i=1}^{n} \pi(i)$ must be equal to 1.

$$\sum_{i=1}^{n} \pi(i) = 1 \tag{9.22}$$

Note that this is a linear system of equations that can be easily solved using an iterative method. The algorithm starts off by initializing $\overline{\pi}^{(0)} = \overline{e}/n$, and it derives $\overline{\pi}^{(t+1)}$ from $\overline{\pi}^{(t)}$ by repeating the following iterative step:

$$\overline{\pi}^{(t+1)} \Leftarrow \alpha \overline{e}/n + (1-\alpha) P^T \overline{\pi}^{(t)} \tag{9.23}$$

After each iteration, the entries of $\overline{\pi}^{(t+1)}$ are normalized by scaling them to sum to 1. These steps are repeated until the difference between $\overline{\pi}^{(t+1)}$ and $\overline{\pi}^{(t)}$ is a vector with magnitude less than a user-defined threshold. This approach is also referred to as the *power-iteration method*. It is important to understand that *PageRank* computation is expensive, and it cannot be computed on the fly for a user query during Web search. Rather, the *PageRank* values for *all* the known Web pages are pre-computed and stored away. The stored *PageRank* value for a page is accessed only when the page is included in the search results for a particular query for use in the final ranking. Typically, this stored value is used as one of the features in a learning-to-rank procedure (cf. Sect. 9.2.4.10).

The *PageRank* values can be shown to be the n components of the largest left eigenvector[13] of the stochastic transition matrix P, for which the eigenvalue is 1. However, the stochastic transition matrix P needs to be adjusted to incorporate the restart within the transition probabilities. This approach is described in Sect. 11.3.3 of Chap. 11.

9.6.1.1 Topic-Sensitive PageRank

Topic-sensitive PageRank is designed for cases in which it is desired to provide greater importance to some topics than others in the ranking process. While personalization is less common in large-scale commercial search engines, it is more common is smaller scale site-specific search applications. Typically, users may be more interested in certain combinations of topics than others. The knowledge of such interests may be available to a personalized search engine because of user registration. For example, a particular user may be more interested in the topic of automobiles. Therefore, it is desirable to rank pages related to automobiles higher when responding to queries by this user. This can also be viewed as the *personalization* of ranking values. How can this be achieved?

The first step is to fix a list of base topics, and determine a high-quality sample of pages from each of these topics. This can be achieved with the use of a resource such as the *Open Directory Project (ODP)*,[14] which can provide a base list of topics and sample Web pages for each topic. The *PageRank* equations are now modified, so that the teleportation is only

[13] The left eigenvector $\overline{X}$ of P is a row vector satisfying $\overline{X}P = \lambda \overline{X}$. The right eigenvector $\overline{Y}$ is a column vector satisfying $P\overline{Y} = \lambda \overline{Y}$. For asymmetric matrices, the left and right eigenvectors are not the same. However, the eigenvalues are always the same. The unqualified term "eigenvector" refers to the right eigenvector by default.

[14] http://www.dmoz.org.

performed on this sample set of Web documents, rather than on the entire space of Web documents. Let $\overline{e_p}$ be an n-dimensional personalization (column) vector with one entry for each page. An entry in $\overline{e_p}$ takes on the value of 1, if that page is included in the sample set, and 0 otherwise. Let the number of nonzero entries in $\overline{e_p}$ be denoted by n_p. Then, the *PageRank* Equation 9.21 can be modified as follows:

$$\overline{\pi} = \alpha \overline{e_p}/n_p + (1-\alpha)P^T\overline{\pi} \tag{9.24}$$

The same power-iteration method can be used to solve the personalized *PageRank* problem. The selective teleportations bias the random walk, so that pages in the structural locality of the sampled pages will be ranked higher. As long as the sample of pages is a good representative of different (structural) localities of the Web graph, in which pages of specific topics exist, such an approach will work well. Therefore, for each of the different topics, a separate *PageRank* vector can be precomputed and stored for use during query time.

In some cases, the user is interested in specific *combinations of* topics such as sports and automobiles. Clearly, the number of possible combinations of interests can be very large, and it is not reasonably possible or necessary to prestore every personalized *PageRank* vector. In such cases, only the *PageRank* vectors for the base topics are computed. The final result for a user is defined as a weighted linear combination of the topic-specific *PageRank* vectors, where the weights are defined by the user-specified interest in the different topics.

9.6.1.2 SimRank

The notion of *SimRank* was defined to compute the structural similarity between nodes. *SimRank* determines *symmetric* similarities between nodes. In other words, the similarity between nodes i and j, is the same as that between j and i. Before discussing *SimRank*, we define a related but slightly different asymmetric ranking problem:

Given a target node i_q and a subset of nodes $S \subseteq N$ from graph $G = (N, A)$, rank the nodes in S in their order of similarity to i_q.

Such a query is very useful in recommender systems in which users and items are arranged in the form of a bipartite graph of preferences, in which nodes corresponds to users and items, and edges correspond to preferences. The node i_q may correspond to an item node, and the set S may correspond to user nodes. Alternatively, the node i_q may correspond to a user node, and the set S may correspond to item nodes. Refer to [3] for a discussion on recommender systems. Recommender systems are closely related to search, in that they also perform ranking of target objects, but while taking user preferences into account.

This problem can be viewed as a limiting case of topic-sensitive *PageRank*, in which the teleportation is performed to the *single node* i_q. Therefore, the personalized *PageRank* Equation 9.24 can be directly adapted by using the teleportation vector $\overline{e_p} = \overline{e_q}$, that is, a vector of all 0s, except for a single 1, corresponding to the node i_q. Furthermore, the value of n_p in this case is set to 1.

$$\overline{\pi} = \alpha \overline{e_q} + (1-\alpha)P^T\overline{\pi} \tag{9.25}$$

The solution to the aforementioned equation will provide high ranking values to nodes in the structural locality of i_q. This definition of similarity is *asymmetric* because the similarity value assigned to node j starting from query node i is different from the similarity value assigned to node i starting from query node j. Such an *asymmetric* similarity measure is suitable for *query-centered* applications such as search engines and recommender

systems, but not necessarily for arbitrary network-based data mining applications. In some applications, symmetric pairwise similarity between nodes is required. While it is possible to average the two topic-sensitive *PageRank* values in opposite directions to create a symmetric measure, the *SimRank* method provides an elegant and intuitive solution.

The *SimRank* approach is as follows. Let $In(i)$ represent the in-linking nodes of i. The *SimRank* equation is naturally defined in a recursive way, as follows:

$$SimRank(i, j) = \frac{C}{|In(i)| \cdot |In(j)|} \sum_{p \in In(i)} \sum_{q \in In(j)} SimRank(p, q) \tag{9.26}$$

Here C is a constant in $(0, 1)$ that can viewed as a kind of decay rate of the recursion. As the boundary condition, the value of $SimRank(i, j)$ is set to 1 when $i = j$. When either i or j do not have in-linking nodes, the value of $SimRank(i, j)$ is set to 0. To compute *SimRank*, an iterative approach is used. The value of $SimRank(i, j)$ is initialized to 1 if $i = j$, and 0 otherwise. The algorithm subsequently updates the *SimRank* values between all node pairs iteratively using Eq. 9.26 until convergence is reached.

The notion of *SimRank* has an interesting intuitive interpretation in terms of random walks. Consider two random surfers walking *in lockstep* backwards from node i and node j till they meet. Then the number of steps taken by each of them is a random variable $L(i, j)$. Then, $SimRank(i, j)$ can be shown to be equal to the expected value of $C^{L(i,j)}$. The decay constant C is used to map random walks of length l to a similarity value of C^l. Note that because $C < 1$, smaller distances will lead to higher similarity and vice versa.

Random walk-based methods are generally more robust than the shortest path distance to measure similarity between nodes. This is because random walks measures implicitly account for the *number* of paths between nodes, whereas shortest paths do not.

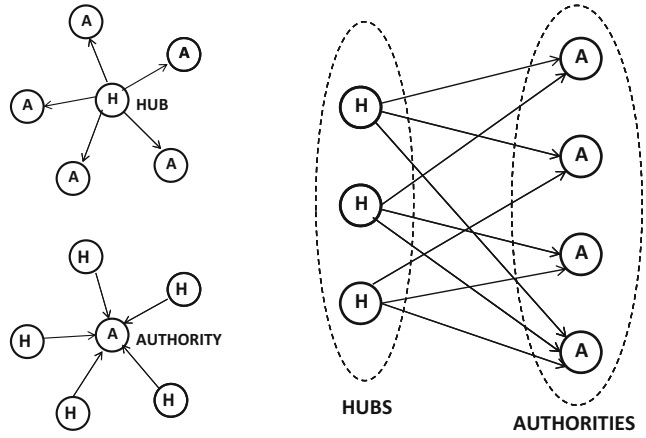
9.6.2 HITS

The *Hypertext Induced Topic Search (HITS)* algorithm is a *query-dependent* algorithm for ranking pages. The intuition behind the approach lies in an understanding of the typical structure of the Web that is organized into hubs and authorities.

An *authority* is a page with many in-links. Typically, it contains authoritative content on a particular subject, and, therefore, many Web users may trust that page as a resource of knowledge on that subject. This will result in many pages linking to the authority page. A *hub* is a page with many out-links to authorities. These represent a compilation of the links on a particular topic. Thus, a hub page provides guidance to Web users about where they can find the resources on a particular topic. Examples of the typical node-centric topology of hubs and authorities in the Web graph are illustrated in Fig. 9.8a.

The main insight used by the *HITS* algorithm is that good hubs point to many good authorities. Conversely, good authority pages are pointed to by many hubs. An example of the typical organization of hubs and authorities is illustrated in Fig. 9.8b. This mutually reinforcing relationship is leveraged by the *HITS* algorithm. For any query issued by the user, the *HITS* algorithm starts with the list of relevant pages and expands them with a *hub ranking* and an *authority ranking*.

The *HITS* algorithm starts by collecting the top-r most relevant results to the search query at hand. A typical value of r is 200. This defines the *root set* R. Typically, a query to a commercial search engine or content-based evaluation is used to determine the root set. For each node in R, the algorithm determines all nodes immediately connected (either in-linking or out-linking) to R. This provides a larger *base set* S. Because the base set S can



(a) Hub and authority examples (b) Network organization between hubs and authorities

Figure 9.8: Illustrating hubs and authorities

be rather large, the maximum number of in-linking nodes to any node in R that are added to S is restricted to k. A typical value of k used is around 50. Note that this still results in a rather large base set because *each* of the possibly 200 root nodes might bring 50 in-linking nodes, along with out-linking nodes.

Let $G = (S, A)$ be the subgraph of the Web graph defined on the (expanded) base set S, where A is the set of edges between nodes in the root set S. The entire analysis of the *HITS* algorithm is restricted to this subgraph. Each page (node) $i \in S$ is assigned both a *hub score* $h(i)$ and *authority score* $a(i)$. It is assumed that the hub and authority scores are normalized, so that the sum of the squares of the hub scores and the sum of the squares of the authority scores are each equal to 1. Higher values of the score indicate better quality. The hub scores and authority scores are related to one another in the following way:

$$h(i) = \sum_{j:(i,j)\in A} a(j) \quad \forall i \in S \tag{9.27}$$

$$a(i) = \sum_{j:(j,i)\in A} h(j) \quad \forall i \in S \tag{9.28}$$

The basic idea is to reward hubs for pointing to good authorities and reward authorities for being pointed to by good hubs. It is easy to see that the aforementioned system of equations reinforces this mutually enhancing relationship. This is a linear system of equations that can be solved using an iterative method. The algorithm starts by initializing $h^0(i) = a^0(i) = 1/\sqrt{|S|}$. Let $h^t(i)$ and $a^t(i)$ denote the hub and authority scores of the ith node, respectively, at the end of the tth iteration. For each $t \geq 0$, the algorithm executes the following iterative steps in the $(t+1)$th iteration:

for each $i \in S$ set $a^{t+1}(i) \Leftarrow \sum_{j:(j,i)\in A} h^t(j)$;
for each $i \in S$ set $h^{t+1}(i) \Leftarrow \sum_{j:(i,j)\in A} a^{t+1}(j)$;
Normalize L_2-norm of each of hub and authority vectors to 1;

For hub-vector $\overline{h} = [h(1) \ldots h(n)]^T$ and authority-vector $\overline{a} = [a(1) \ldots a(n)]^T$, the updates

can be expressed as $\overline{a} = A^T \overline{h}$ and $\overline{h} = A\overline{a}$, respectively, when the edge set A is treated as an $|S| \times |S|$ adjacency matrix. The iteration is repeated to convergence. It can be shown that the hub vector $\overline{h}$ and the authority vector $\overline{a}$ converge in directions proportional to the dominant eigenvectors of AA^T and $A^T A$, respectively. This is because the relevant pair of updates can be shown to be equivalent to power-iteration updates of AA^T and $A^T A$, respectively.

9.7 Summary

This chapter discusses the data structures and query processing methods involved in information retrieval, and their generalizations to search engines. A proper design of the inverted index is crucial in obtaining efficient responses to queries. Many types of additive functions over terms can be computed with an inverted index and accumulator variables. Many vector space and probabilistic models of retrieval are used by search engines. Some of these models use relevance feedback, whereas others are able to score documents with respect to queries without using relevance feedback. An important aspect of search engine construction is the discovery of relevant resources with the use of crawling techniques. In search engines, the linkages can be used to create a measure Web page quality with *PageRank* measures. These quality measures are combined with match-based measures to provide responses to queries.

9.8 Bibliographic Notes

A discussion of several data structures, such as hash tables, binary trees and B-Trees, may be found in [427]. All these data structures are useful for dictionary construction. The use of k-gram dictionaries for spelling corrections and error-tolerant retrieval may be found in [542, 543].

A detailed discussion of the inverted file in the context of search engines may be found in [506, 545], including various optimizations like skip pointers. Skip pointers were introduced in [354]. The inverted file is the dominant data structure for indexing documents, although some alternatives like the signature file [163] have also been proposed in the literature. A comparison of the inverted files with the signature file is provided in [544], and it is shown that the signature file is inefficient as compared to the inverted file. Several methods for constructing inverted indexes are discussed in [216, 506]. The construction of distributed indexes is discussed in [31, 72, 194, 334, 405]. Most recent techniques are based on the *MapReduce* framework [128]. Dynamic index construction methods like logarithmic merging are discussed in [71]. The trade-offs of different ways of index maintenance are discussed in [281].

The technique of using accumulators with early stopping is discussed in [354]. Other efficient methods for query processing with early termination and pruning are discussed in [22, 24]. Methods for using inverted indexes in phrase queries are discussed in [502]. Machine learning approaches for search engine optimization with the use of the ranking SVM were pioneered in [244]. However, the view of information retrieval as a classification problem was recognized much earlier and also appears in van Rijsbergen's classical book [480], which was written in 1979. The earliest methods for learning to rank with pairwise training data points were proposed in [105]. The work in [472] optimized the parameters of the BM25 function on the basis of the NDCG measure. Refer to Chap. 7 for a discussion of the NDCG measure. The ranking SVM idea has been explored in [74] in the context of information retrieval. A structured SVM idea that optimizes average precision is discussed in [522]. The

work in [70] discusses ways of ranking using gradient-descent techniques with the *RankNet* algorithm. It is stated in [307] that the *RankNet* algorithm was the initial approach used in several commercial search engines. The listwise approach for learning to rank is discussed in [75]. The extraction of document-specific features for improving ranking with machine learning approach is discussed in [406]. An excellent overview of different learning-to-rank algorithms with a focus on search engines is provided in [307].

The use of champion lists, pruning, and tiered indexes for large-scale search is discussed in [77, 309, 321, 366]. Dictionary compression is discussed in [506], and variable byte codes were proposed in [435]. Word aligned codes have also been proposed that improve over variable byte codes [23, 25]. The delta coding scheme was proposed in [153]. The use of caching for improving retrieval performance is studied in [31, 278, 429]. Many of these techniques show how one can use caches of multiple levels to improve performance. The combination of inverted list compression and caching for improved performance is explored in [532]. In general, compression improves caching performance as well. A good overview of caching and compression may also be found in Zobel and Moffat's survey on indexing [545].

The vector space model for information retrieval was introduced in [426], and term weighting methods were studied in [423]. Over the years, numerous term weighting and document length normalization methods have been proposed. Pivoted length document normalization is a notable approach that is often used [450]. The use of idf normalization was first conceived in [453]. The binary independence model was proposed in [411, 480], and the final form of the retrieval status value is a confirmation of the importance of idf in retrieval applications. A number of theoretical arguments for idf normalization may also be found in [410]. A number of experiments about the probabilistic model of information retrieval are provided in [456]. This paper also adapts the binary independence model into the BM25 model. The BM25 model has had a significant impact on the matching function used in the search engines. A variant of the BM25 model that uses the different fields in the document is referred to as BM25F [412]. The use of language models in information retrieval were pioneered by Ponte and Croft with the Bernoulli approach [385]. A language model by Hiemstra [214] also appeared at approximately the same time, which used a multinomial approach. The use of Hidden Markov Models for language modeling were proposed by Miller et al. [346]. The role of smoothing in language modeling approaches is studied by Zhai and Lafferty [528]. An overview of language models for information retrieval may be found in [527].

Detailed discussions on crawling and search engines may be found in several books [31, 71, 79, 120, 303, 321, 506]. Focused crawling was proposed in [83]. The work in [93] discusses the importance of proper URL ordering in being able to efficiently crawl useful pages. The *PageRank* algorithm is described in [64, 370]. The *HITS* algorithm was described in [262]. A detailed description of different variations of the *PageRank* and *HITS* algorithms may be found in [79, 303, 321, 280]. The topic-sensitive *PageRank* algorithm is described in [205].

9.8.1 Software Resources

Numerous open source search engines are available such as Apache Lucene and Solr [587, 588], Datapark search engine [586], and Sphinx [589]. Some of the search engines also provide crawling capabilities. The Lemur project [582] provide an open source framework for language modeling approaches in information retrieval. Numerous open source crawlers are available such as Heritrix [585] (Java), Apache Nutch [583] (Java), Datapark search engine [586] (C++), and Python Scrapy [584]. The package *scikit-learn* [550] has an implementation of the computation of the principal eigenvector, which is useful for *PageRank*

evaluation and HITS. The Snap repository at Stanford University also includes a *PageRank* implementation [590]. The **gensim** software package [401] has an implementation of some ranking functions like BM25.

9.9 Exercises

1. Show that the space required by the inverted index is exactly proportional to that required by a sparse representation of the document-term matrix.

2. The index construction of Sect. 9.2.3 assumes that document identifiers are processed in sorted order. Discuss the modifications required when the document identifiers are not processed in sorted order. How much does this modification increase the time complexity?

3. Discuss an efficient algorithm to implement the OR operator in Boolean retrieval with two inverted lists that are available in sorted form.

4. Show that a dictionary, which is implemented as a hash table with linear probing, requires constant time for insertions and lookups. Derive the expected number of lookups in terms of the fraction of the table that is full.

5. Write a computer program to implement a hash-based dictionary and an inverted index from a document-term matrix.

6. Suppose that the inverted index also contains positional information. Show that the size of the inverted index is proportional to the number of tokens in the corpus.

7. Consider the string $ababcdef$. List all 2-shingles and 3-shingles, using each alphabet as a token.

8. Show that the *PageRank* computation with teleportation is an eigenvector computation on an appropriately constructed probability transition matrix.

9. Show that the hub and authority scores in *HITS* can be computed by dominant eigenvector computations on AA^T and A^TA respectively. Here, A is the adjacency matrix of the graph $G = (S, A)$, as defined in the chapter.

10. Propose an alternative to the ranking SVM based on logistic regression. Discuss how you would formulate the optimization problem and how the stochastic gradient-descent steps are related to traditional logistic regression.

11. The ranking SVM is a special case of the classical SVM in which each class variable is $+1$ in the training data (but not necessarily at the time of prediction) and the bias variable is 0. Show that any classical SVM in which the bias variable is 0 but the class variables are drawn from $\{-1, +1\}$ can be transformed to the case in which each class variable is $+1$. Why do we need the bias variable to be 0 for this transformation?