

Chapter 5: Query Processing

Objectives

- Learn how compute a measure of relevance
- Learn the basic query processor of Simple IR

1 Introduction

Until now, we focused solely on the preprocessing of text for the moment when a user issues a query. All of the work on token identification and index building was all about building a special access structure to make it so we could quickly identify queries. Recall that without an inverted index, we are relegated to a sequential scan of the document collection – a prospect that is simply not feasible with a large collection.

From the very first Information Retrieval papers, researchers have tried to compute a measure of relevance between a query and a document. We refer to this task as the development of a *retrieval strategy*. In our earlier book [REF], we defined a *retrieval strategy* as a *means by which a measure of relevance is assigned between a query and a document*. While researchers were working on the problem of computing a measure of relevance in the mid 1960's, commercial systems ignored the problem and focused simply on Boolean retrieval. The idea behind Boolean retrieval is that users issue queries as Boolean expressions. A query such as *((apple AND pie) NOT (peanut AND butter) OR (dessert))* finds documents whose keywords satisfy this logical expression. With a Boolean query, there is no notion of a system attempting to interpret what the user is interested in. Instead, the system merely verifies the truth of the

expression as it relates to a given document. Boolean systems were and still are popular with librarians and other expert users who are able to form very sophisticated Boolean searches. They are primarily used on reasonably small document collections such as a card catalog systems, personal document collections, etc.

For many years, researchers argued with academics about the need for relevance ranking systems, but the reality was that text search did not become a focus for large numbers of users until web search engines were developed. From the beginning, web search engines have assigned a measure of relevance between a query and each document. All of the web search engines do support Boolean retrieval as well, but the need for relevance ranking became evident as the document collections grew and the number of users who are not computer savvy increased.

We primarily focus on relevance ranking. There are many books written in the 1980's that have lengthy dialogs on implementation issues of Boolean systems. Essentially the inverted index used is the same as with Boolean based systems, but the query processing is somewhat different. Since all web search engines implement relevance ranking, we focus on relevance ranking here and refer to reader to [xxxx] for discussions on Boolean retrieval.

2 Retrieval Strategies: An Overview

A retrieval strategy takes a query and a document as input and identifies a measure of relevance between the query and the document. In research papers, the function that does this is often referred to as a *similarity coefficient* (*sc*) or a *retrieval status value* (*rsv*).

Hence the $SC(Q, D_i)$ identifies the similarity between query Q and the i^{th} document. The

most commonly used retrieval strategies are the Boolean, vector space, and probabilistic models, inference networks, latent semantic indexing (LSI), and neural networks.

We will briefly give an overview of these models and then provide a detailed example of the vector space model. Finally, we describe how SimpleIR implements the vector space model.

2.1 **Boolean Model**

<Note to students – this will be filled in later>

2.2 **Vector Space Model**

Salton introduced the vector space model in 1975 [xx]. In the vector space model, each document is represented as a vector and each query as a vector in n -dimensional space. If we refer to the document vector as D and the query vector as Q , to measure the relevance of D and Q we simply take the Euclidean distance between D and Q as $SC(Q, D_i) = Q D_i$. Additionally, the cosine of the angle between Q and D is often used – this is written as

$$SC(Q, D_i) = Q D_i / \|D_i\|.$$

The hard part of the vector space model is identifying the components of the vectors. That is, one must first define which features of a document should be used to represent the document. Ideally, we would use something that represented the *meaning* of a document. If we could do this than two documents that say the same thing but happen to use different words “e.g., D_1 : *my favorite ice cream is chocolate* and D_2 : *it is always a joyous occasion when I eat my beloved chocolate ice cream*” would be

represented identically. The whole field of natural language understanding strives to *understand* text like this and derive a canonical *meaning* for each document. At present, it remains an open research problem. So, we are left with using terms as features and hoping that enough terms will match the query and document pair so that we will find some relevant documents.

2.2.1 Binary Vector Space Model

As stated, terms are used as features to represent documents. Since vectors must be of equal length, we assign a component in each vector for *each distinct term in the document collection*. The value of a given component may vary. The *binary vector space model* assigns a one if the term occurs in the document and a zero if it does not. This simplistic approach is easily derived and used, but it fails to consider the importance or document discrimination factor of a term or the fact that some terms occur multiple times within a document.

To simplify our discussion, assume we have deciphered the language of elephants and that they only use three words. Since we do not know how elephants write, let us assume that they use words A, B, and C. To represent, a document written in this language, we need only three components. A document D1 with terms {A, A, B} would be represented with the vector $\langle 1, 1, 0 \rangle$ where the first component identifies a binary value for the occurrence of A, the second, B and finally the third C. A query with terms A and B would have a vector of $\langle 1, 1, 0 \rangle$. Taking the dot product of Q with D, we obtain $(1)(1) + (1)(1) + (0)(0) = 2$. Assume we have a document D2 with only term A; it would have a vector of $\langle 1, 0, 0 \rangle$ and would have a similarity of $(1)(1) + (1)(0) + (0)(0) = 1$. A

document comprising of only C will have a vector of $\langle 0,0,1 \rangle$ and would have a similarity of 0 to the query.

2.2.2 Term Frequency (tf)

Notice that in Document D1 two occurrences of the term A exist, but with the binary vector space model, this document is represented no differently than if only one occurrence of A exists. A document with multiple occurrences of a query term is probably more relevant to a query than a document with only one occurrence of the given term. To take this into account, the *term frequency (tf)* is often used in each component of a document vector. Using the *term frequency*, we have a vector for D1 that is $\langle 2, 1, 0 \rangle$ and its similarity to the query would be $(2)(1) + (1)(1) + (0)(0) = 3$. This is higher than the similarity between the query and D2 (the document with only one occurrence of term A).

The same term can certainly appear more than once in a query. Although this is somewhat unlikely with the short queries used on the web (most queries are about 3 words in length [xx]), but it could happen. The *query term frequency (qtf)* is stored in each component of the query vector using the same way that terms were stored in the document vector.

2.2.3 Inverse Document Frequency (idf)

Using only term frequency does not consider the *weight* of a term in a document. The idea is that some terms are simply more important than others. That is, some terms, due to their rarity of use, do a better job of identifying the relevant documents to queries than others. Users can be asked to designate which terms are more important to them and to

weight these query terms accordingly, but manually weighting every term within every document is an arduous and prohibitive task. No user is likely to assign a weight to each word in a document.

Computing a weight automatically is done based on the frequency of the term across the entire document collection, often referred to as the *document frequency (df)* for a term. If term A occurs 500 times but only appears in 50 documents, its document frequency is fifty. The *inverse document frequency (idf)* is $\log_{10} (N / df)$ where N is the number of documents in the collection. Karen Sparck-Jones first introduced the *idf* concept in the early 1970's. The idea is that a term that occurs rarely should have a high weight and frequently occurring terms should be weighted low. Taken to the extreme, stop-words, like the word "the", that appear in almost all documents should ideally receive a score of roughly 0.

Consider a document collection with 1,000,000 documents. Assume term A appears in only one document. Its weight is $\log_{10} (1,000,000 / 1) = 6$. A term, B that occurs in every document will have a weight of $\log_{10} (1,000,000 / 1,000,000) = 0$. The use of the *log* function scales the weight into some reasonably small range; for our one million document collection, it is from zero to six. This prevents one term from having a weight that completely dominates the computation of the similarity measure.

Overall, the idea of the *idf* measure is to increase the weight of infrequent terms. The notion is that a term such as *microscope* probably is more meaningful because it probably occurs less often than a term such as *black*. Using the *idf*, a query *black microscope* will have a higher similarity to a document that only contains *microscope* than to a document that contains only *black*.

2.2.4 tf-idf

Frequently, the weight of *tf-idf* is used to populate components of document vectors because it combines the impact of a term occurring multiple times with the impact of the importance of a term. Returning to our example with our three word language, assume term A has an *idf* of 3 and term B an *idf* of 4. Our document D1 would now be represented as $\langle 6, 4, 0 \rangle$ since A occurs twice. Computing the similarity with our query vector with terms A and B occurring only once $\langle 3, 4 \rangle$ we obtain $SC(Q, D1) = (3)(6) + (4)(4) = 34$. Frequently, in research papers people describe a model as using *tf-idf*. Often, in such cases, they are referring to the vector space model with term components populated by the combination of *tf* and *idf*.

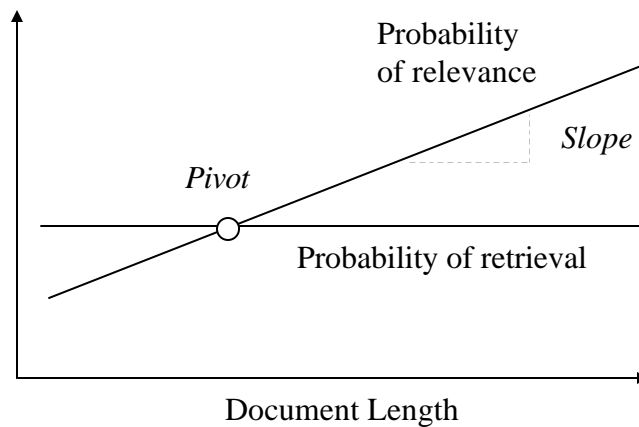
2.2.5 Document Length

The alert reader will have noticed that documents that simply have a whole bunch of words will tend to score higher than smaller documents. The odds of a long document having more words that match the query are certainly higher than a document with only a few words. To normalize for document length, the cosine measure is often used which divides each similarity measure by the *size* of the document vector. The notion is that this measure levels the playing field and prevents longer documents from being deemed more relevant to queries simply because of their size.

In a PhD thesis [Singhal], Singhal showed that dividing by document length *over normalized* for the size of a document. He found that, for many test collections, larger documents happen to be more relevant to users queries than smaller documents. By normalizing using only the document length, we obtain a measure that gives significant preference to documents that are small. The figure below shows how normalization

results in the probability of retrieval being less than the probability of relevance for small documents and the probability of retrieval being larger than the probability of relevance for large documents. The crossing point at location *pivot* (on the horizontal axis) gives the coordinates of the line needed for adjusting the normalization factor. Similarly the slope of this line can be computed as well.

Document Length Normalization



Since, Singhal showed that relevant documents tend to be larger, he suggested an adjustment to the overcompensation provided by dividing by the document length. The adjustment is referred to as *pivoted document length normalization* and is based on modifying the normalization by pivoting based on the point at which document size becomes a detriment to the likelihood of relevance. This pivot point may vary for different collections, but one that is currently used is xxxx. The new equation is:

$$\text{sum}(q_i \times d_i) / ((1-s) * \text{avgNormalization})$$

where the avgNormalization is computed as the average of $\|D_i\|$ for a reasonable sample of documents. s refers to the slope of the adjustment line that modifies the average document normalization. In the original papers [REF], the *slope* was found to be 0.7; for other document collections, it is necessary to identify a slope optimized for the specific document collection.

2.3 *Probabilistic Model*

2.3.1 Original Probabilistic Models

The probabilistic model takes a more mathematical approach to the problem. The vector space model does use simple vector equations, but the reality is that there is no inherent reason to treat a document as a vector. True enough, that once we make this leap we can do some interesting things with the vectors, but to people who developed the probabilistic model it is simply too much of a leap. The probabilistic model is founded on the notion that the problem of assigning a measure of a relevance between a document and a query can be cast as finding the *probability* that a document will be relevant to a query [Robertsonxx, SparckJonesxx]. That is, words in the query and the document can be used to estimate the probability. For example, if we somehow knew that the word “green” occurred in *every* relevant document regardless of a query and that if the word “green” occurred in none of the non-relevant documents, we could estimate that the probability $p(\text{Rel} \mid \text{“green”}) = 1.0$. This equation reads as with “absolute certainty, namely a probability of one, a document containing the term *green* is relevant”. Having determined the probability of a single term, we can extend this for a multi-term query.

Assume a 50% probability that a document is relevant if it contains “red” and, independently, there is a 50% probability that a document is relevant if it contains “black”. Given *both* “red” and “black”, the $p(\text{Rel} \mid \text{“red”}, \text{“black”})$ can be computed (by applying Bayes theorem) as the $p(\text{Rel} \mid \text{“red”}) * p(\text{Rel} \mid \text{“black”}) = (.5)(.5) = .25$. The real question is how can we compute the probability that a document will be relevant given a particular term. Hence, a probabilistic similarity measure can be defined for a query Q with n terms: (t_1, t_2, \dots, t_n)

$$SC(Q, D_i) = p(\text{Rel} \mid t_1, t_2, \dots, t_n) = \text{Product} (p(R \mid t_1) * p(R \mid t_2) * p(R \mid t_1) \dots * p(R \mid t_n))$$

The probabilities: $p(R \mid t_i)$ are referred to as *parameters* that must be estimated. The best way to estimate these is to take millions of queries and millions of documents and identify for each distinct term in the collection the probability that the term appears in a relevant document. Another way is to do this in a query-specific manner. We would take one query and then estimate the parameters for the terms in this query based on which documents were relevant and which were non-relevant. This approach may at first seem ridiculous since an initial reaction might be to assume that all queries are *ad-hoc* in nature and impossible to predict. The truth is that many queries can be estimated in advance. (A large percentage of queries in search engine query logs are duplicates. Some search engines go so far as to include rules that say if query = x then retrieve the following documents!) For these queries, it may be possible to estimate the parameters. Another approach is to use the *idf* as an estimate for the probability that given a term it will be relevant. The idea is that terms with a high *idf* are relatively unusual and, as such, may

appear in more relevant documents than a typical term. (Our other choice is to simply assign a guess of 0.5 for each parameter). This leaves us with the similarity measure:

$$SC(Q, D_i) = p(\text{Rel} | t_1, t_2, \dots, t_n) = \text{idf}(t_1) * \text{idf}(t_2) * \text{idf}(t_i) \dots * \text{idf}(t_n)$$

Initial versions of the probabilistic model [REF] included estimates based on the probability that a document was relevant as well as the probability that a document was not relevant. Some independence assumptions and ordering principles that were used are:

Independence Assumptions:

I1: The distribution of terms in relevant documents is independent and their distribution in all documents is independent.

I2: The distribution of terms in relevant documents is independent and their distribution in non-relevant documents is independent.

Ordering Principles

O1: Probable relevance is based only on the presence of search terms in the documents.

O2: Probable relevance is based on both the presence of search terms in documents and their absence from documents.

Given the following values:

r	Number of relevant documents that contain the term
R	Total number of relevant documents
n	Number of documents that contain the term
N	Total number of documents.

we can apply these different assumptions and ordering principles to obtain the following estimates for $p(R | t_i)$.

I1 and O1: $(r/R) / (n/N)$
I2 and O1: $(r/R) / ((n-r)/(N-R))$
I1 and O2: $(r/(R-r)) / (n / (N-n))$
I2 and O2: $(r/(R-r))/((n-r)/((N-n)-(R-r)))$

Typically, an extra 0.5 was added into the model to account for errors in estimation: This results in:

$$((r+.5)/(R-r+.5)) / ((n-r+.5) / ((N-n)-(R-r))+.5)$$

2.3.2 Incorporating tf and Document Length

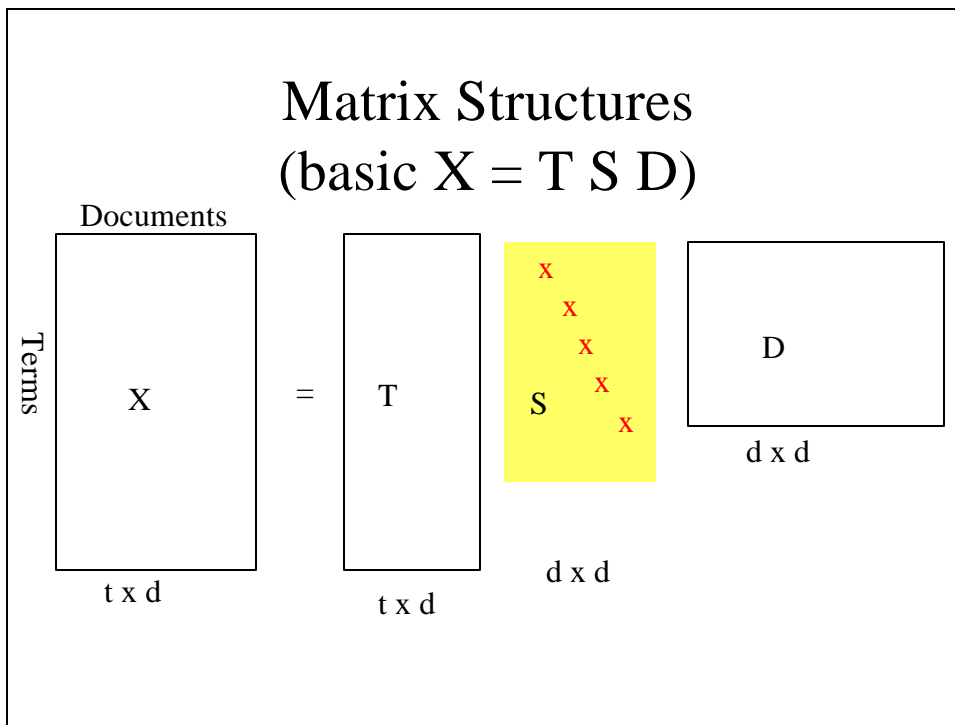
Using *idf* as a basis for parameter estimation is a convenient way to include an automatically identified weight, but as such does not provide a way to incorporate the term frequency or the document length into the probabilistic model. Initial benchmarks without these measures resulted in that the retrieval engines using the probabilistic model consistently performed worse than those using the vector space model. In recent years, both tf and the document length are incorporated into the parameter estimation, and the search results for engines using the probabilistic model perform comparably to those using the vector space model. The new probabilistic estimate is: <<fill this in later>>

2.4 Latent Semantic Indexing

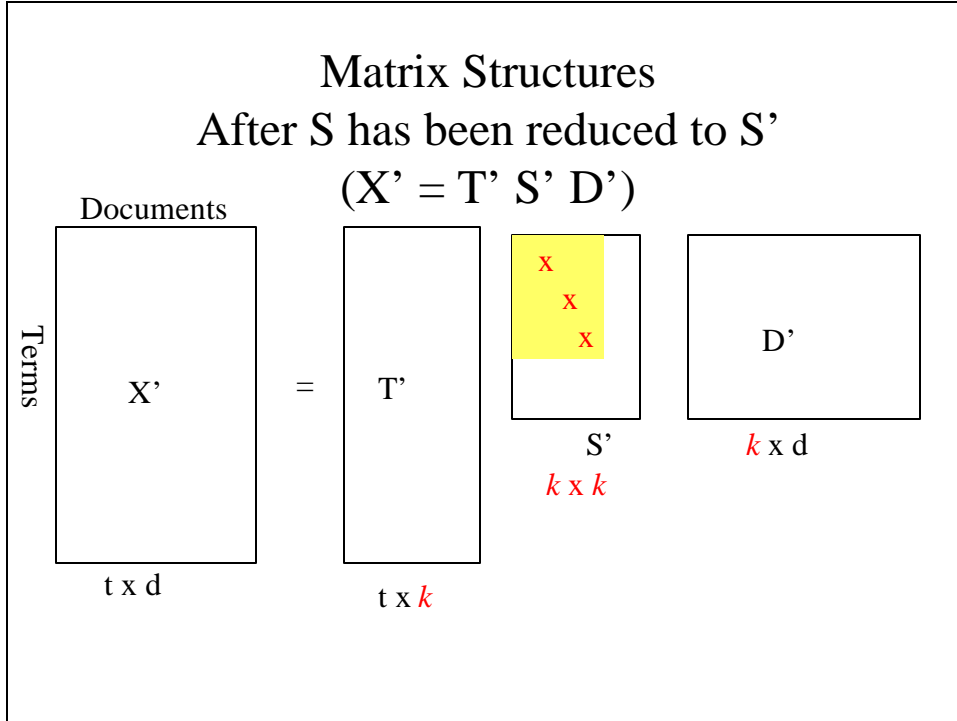
Latent Semantic Indexing [REF] is an approach to relevance ranking based on matrix algebra. First, the inverted index is represented as a term-document matrix. This matrix has a row for each term and a column for each document. A given cell of the matrix contains the number of occurrences of the term in a document (e.g; the term frequency – tf). Clearly, there are many terms that do not occur in many documents. In these cases, the term frequency is zero. Hence, the matrix is a very sparse. In fact, an inverted index

may be viewed as simply an efficient data structure to represent the non-zero elements of the sparse term-document matrix.

Once we have the term document matrix, the idea is to identify the *singular values* for this matrix. Computing these values is done via a technique called *singular value decomposition (SVD)*. SVD's are often used in matrix algebra to identify the key elements of a large matrix. With a term document matrix X , the singular value decomposition will identify three matrices T , S , and D' such that $X = T S D'$. The matrix S contains the $|D|$ singular values where $|D|$ is the number of documents. The idea is that the top k singular values can be used to find the key content of X . Obtaining a correct value of k is done via experimentation. The figure below shows the result of a singular value with a term document matrix X decomposed into a T , S , and D' matrix.



After choosing the top k singular values, we obtain the following:



The T' matrix can be thought of as mapping terms to higher level *concepts*, and the D' matrix can be thought of as mapping document concepts. Essentially, the singular value decomposition merges terms into concepts if they have similar posting lists. The idea is that terms that co-occur in documents are likely to be related. One of the first descriptions of the latent semantic indexing included the following description:

Singular value decomposition (SVD) is used to allow the arrangement of the term-document space to reflect the major associative patterns in the data, and ignore the smaller, less important differences. (Deerwester, JASIS, 1990).

To compare two terms, the dot product of the corresponding rows of X' is computed. To compare two documents, we use the columns of X' . Finally, to compute a similarity measure, we must adjust the query vector q to resemble another document. To do this, we take the inverse of the singular value matrix. This is $O(k)$ since it is a diagonal

matrix. Then, we compute the query matrix Q as: $q^{T(T')(S')^{-1}}$. Once computed, the final similarity measure is obtained by taking the dot product of Q with any particular column in T' .

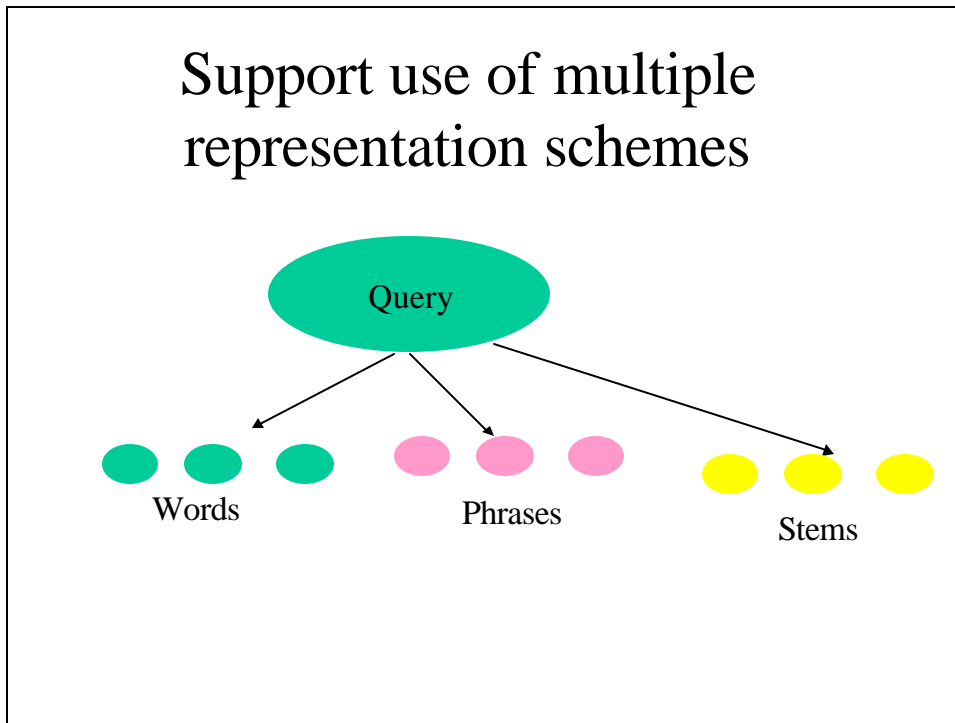
Overall, LSI is commonly referenced by database researchers who describe LSI as a “popular” technique used in information retrieval. To our knowledge, no existing web search engine or intranet search engine uses LSI because of the computational overhead required to compute the SVD. However, research continues on faster SVD algorithms; so this may well ultimately become a popular algorithm.

2.5 Inference Networks

Inference networks are used for information retrieval since the early 1990's. The general idea of an inference network is to reduce the search space when computing the belief that the probability of an event will occur. For a given set of events, A, B, C, and D, we should compute the probability of all events occurring, $p(ABCD)$, as the probability of A occurring without B,C,D which is $p(A B'C'D')$, the probability of A and B occurring without C and D ($A B C' D'$), etc. For any n events, we need 2^n assignments. The inference network reduces the need to enumerate all of these assignments because it combines events wherever possible. So, if we know event A and B and have occurred, we can build a node AB that accepts as inputs A and B. The belief in AB is then used throughout the remainder of the network.

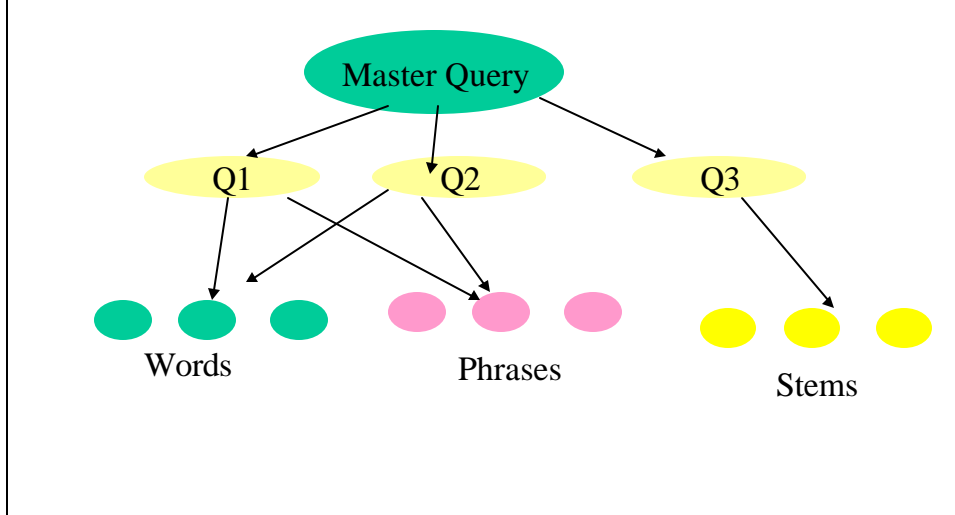
To show how an inference network can be used to obtain a measure of relevance, we step through two sub-networks. Two network types exist, namely query and document networks. Each is described separately. Query Network

The query network takes the original query and exhibits numerous representations of the same query. A query can be represented with words, phrases, or stems. The figure below shows how the query network uses these.



Additionally, a single query may be represented in many different ways. If we take a query and rewrite it using different terms we might start with a “master query” and then represent each rendition of the query as a version of the master query. This is illustrated with the query network shown in the figure below.

Results from different queries and query types to be combined

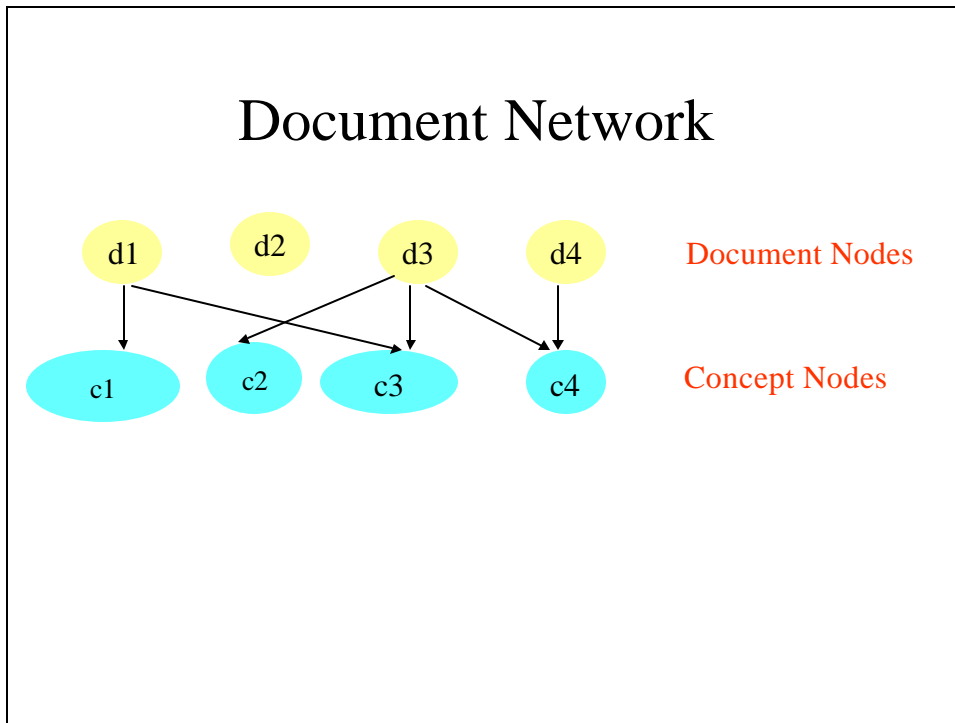


2.5.1 Document Network

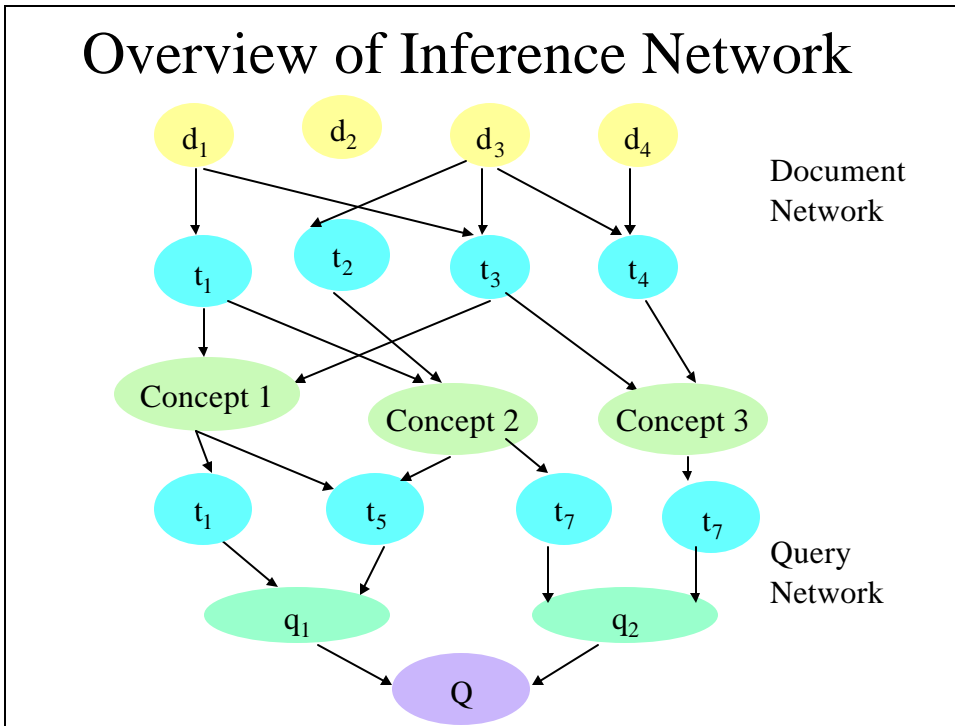
The document network consists of *document nodes* (one for each document in the collection), *text representation nodes*, and *concept representation nodes*. A text representation node summarizes information about how a document is represented. Most of this book has focused on processing ASCII text versions of documents, but if the image of a document exists, or a video of the document, these may all be represented with different text representation nodes. A single document may have many different representations.

A concept representation node describes the various *concepts* identified in the query. Attempts to automatically identify these concepts have occurred for over thirty years in the information retrieval field, but the notion is that if we somehow can derive concepts from a document, we would place them into concept nodes of the information network.

Crude approximations of concept nodes can be frequently occurring terms or phrases in the document. Removing the representation nodes (since we are focused on just processing ASCII text), the document network consists of document nodes and concept nodes as given below:

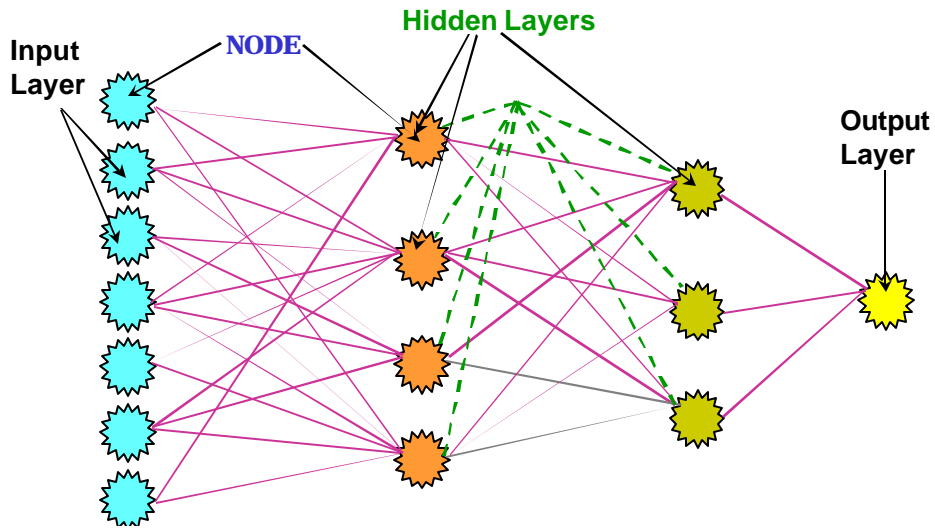


Finally, the document network and the query network can be combined to make it so, given a query, we can obtain an estimate for the “belief” that a document is relevant to the query. A simple network, which links concepts from the query to the document is given in the figure below. A link matrix is used for each node to identify, for a given set of inputs, the belief in the output. The link matrix makes it possible to weight different inputs higher than others. A simple link matrix simply sums the weights of the inputs. More sophisticated link matrices are described in [Grief, 99].

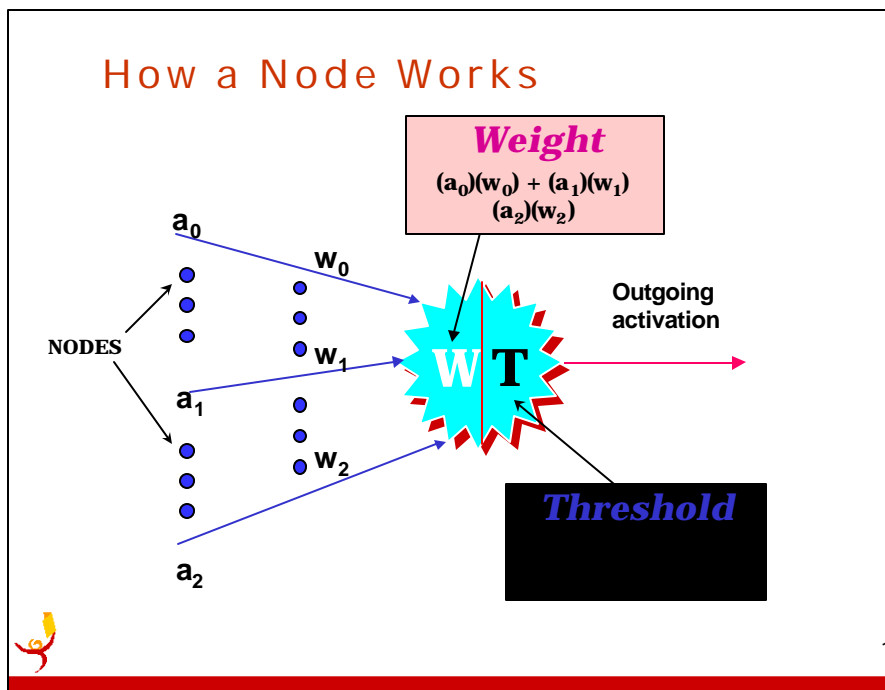


2.6 Neural Networks

Neural networks are also used to compute similarity measures. Typically, a neural network works by accepting input values, applying a function to the input values, and comparing the result to a known answer. The function that operates on the input values also contains internal weights. If the output of the function is incorrect, an algorithm is applied to modify the weights. The idea is that ultimately the network “learns” how to produce accurate output; the learning of the neural network is implemented by changes to the internal weights. The figure below shows the basic structure of a simple neural network.



The computation of a value inside of a given node is shown below:



With terms and documents, the query terms are used as inputs, and weights are assigned based on the occurrence of a term in a given document, typically, tf-idf. Algorithms such as back propagation exist to update the nodes of the neural network based on some indication of the accuracy of a given similarity measure to a particular query.

3 Implementing the Vector Space Model in SimpleIR

To implement the vector space model, many algorithms exist. A simple one is to create a *score* array for each distinct document in the collection and update elements in the *array* as posting lists are traversed. The basic algorithm is:

```
For each query term t
    Obtain the posting list for that term
    For each term in the posting list, get the posting list node [docID, tf]
         $score[docID] = score[docID] + tf \times idf(t)$ 
    End
End
```

The code that implements the basic driver for the similarity measure is given below. It can be seen that the exact similarity measure is selected as a SimpleIR configuration parameter. In this way, users can implement vector space, probabilistic, pivoted normalization, inner product or other measures as well. This code is found in the QueryProcessor object.

First a case statement is used to identify the type of similarity measure. The similarity measure fills the score array. At this point, we loop through the score entries and, for each non-zero entry, a Result object is created. Results objects are added to an arraylist named results. Finally, the result objects are sorted using the built in Collections.sort method and ordered in reverse order with the *reverse* method.

```

private ArrayList runRetrievalStrategy(Query query, InvertedIndex idx) {
    ArrayList results = new ArrayList();
    int    i;
    float  currentScore;
    int    docID;
    Result currentResult;
    Document currentDocument;

    /* now lets find out what kind of similarity measure to run */
    switch (scType) {
        case 0:
            runDotProduct(query, idx);
            break;
        default:
            System.out.println("Invalid SC Type");
            break;
    }
    printScore();

    /* now lets store the non-zero elements of score in the treemap */
    /* this sets things up nicely for the GUI */
    for ( i = 0; i < MAX_DOC; i++) {
        if (score[i] != 0) {
            currentScore = score[i];
            docID = i;
            currentDocument = idx.getDocument(docID);
            currentResult = new Result(currentScore, currentDocument);
            results.add(currentResult);
        }
    }

    /* now lets sort the results */

    Collections.sort(results);
    Collections.reverse(results);

    return results;
}

```

1

The simple result object contains the document identifier and the similarity measure for

```
public class Result implements Comparable {

    private float score;
    private Document currentDocument;
    private static final int MAX_LENGTH = 60;

    public Result(float s, Document d) {
        score = s;
        currentDocument = d;
    }

    public void put (float s, Document d) {
        score = s;
        currentDocument = d;
    }

    /* returns right justified char string representation of score with d digits */
    public String getScore (int d, DecimalFormat form) {
        String temp;
        String result;
        Float f;

        f = new Float(score);
        temp = form.format(f);
        return ResultsWindow.spaces(d - temp.length()) + temp;
    }

    public Document getDocument () {
        return currentDocument;
    }

    public int compareTo (Object r) {
        Result currentResult = (Result) r;
        if (currentResult.score > score) return -1;

        if (currentResult.score < score) return 1;
        return 0;
    }
}
```

the result. The figure above gives the details of the result object. The *comparable* interface is used so that the result objects may be sorted. The *compareTo* method is

defined to explain to the *sort* method how to compare two elements that are instances of a result object.

The actual computation of a similarity measure is implemented in the method *runDotProduct*. For other similarity measures, it is reasonable to expect that other methods would be developed such as *runProbabilistic*, etc. The *runDotProduct* method is given below; it implements the algorithm given at the start of this section.

First, we obtain the number of documents in the inverted index as this drives the computation of the inverse document frequency (*idf*). Next, we initialize the *score* array. An outer loop iterates through each query term and, for each query term, it find its posting list. For each posting list, an inner loop is used to traverse each postinglistnode (*pln*) in the posting list. For each node, the corresponding *score* array is updated. The *updateScore* method is used in an attempt to make the code generic enough to try different term weighting schemes simply by changing the *updateScore* method and avoiding changes to the *runDotProduct* method.

Having populated the score matrix, our work is done, and the *execute* method of the *QueryProcessor* handles sorting of the matrix.

We note that the *score* array might not be a scaleable data structure. First, we are forced to do a linear scan to identify the non-zero entries. A *heap* is a more appropriate data structure since it would keep the lowest score at the top. However, to keep SimpleIR simple, we do not use a heap.

```

private void runDotProduct(Query query, InvertedIndex idx) {
    String      currentToken;
    LinkedList  currentPostingList;
    PostingListNode pln;
    short      qtf;
    int        docID;
    short      tf;
    int        df;
    long       numberOfDocuments;
    double     idf;
    double     idfSquared;
    HashMap    queryTerms = new HashMap();

    numberOfDocuments = idx.getNumberOfDocuments();

    /* initialize the score array */
    for (int i = 0; i < MAX_DOC; i++) {
        score[i] = (float) 0.0;
    }

    /* loop through all query terms, get their posting lists */
    /* loop through the posting lists and update the score array */
    queryTerms = query.getTerms();
    Set querySet = queryTerms.keySet();
    Iterator i = querySet.iterator();
    while (i.hasNext()) {
        currentToken = (String) i.next();
        qtf = ((TermFrequency) queryTerms.get(currentToken)).getTF();

        currentPostingList = idx.getPostingList(currentToken);

        /* if a term has no posting list, skip rest of the loop -- no score to update */
        if (currentPostingList == null) {
            continue;
        }

        df = currentPostingList.size();
        idf = Math.log((double) (numberOfDocuments / df));
        idfSquared = idf * idf;

        Iterator j = currentPostingList.iterator();
        while (j.hasNext()) {
            pln = (PostingListNode) j.next();
            docID = (int) pln.getDocumentID();
            tf = pln.getTF();
            updateScore(docID,tf,qtf,idfSquared); /* increase score for document */
        }
    }
}

```

```
/* here's where we update the score for a document */  
  
/* different weighting schemes could easily be incorporated here */  
private void updateScore(int docID, short tf, short qtf, double idfSquared) {  
  
    score[docID] = score[docID] + ((float) tf * (float) qtf * (float) idfSquared) ;  
  
    return;  
}
```