Experiments in Information Retrieval

Kevin C. Ó Kane
Professor Emeritus
Computer Science Department
University of Northern Iowa
Cedar Falls, IA 50613
# Table of Contents

1 Introduction ................................................................................................................ 7

1.1 What is Information Retrieval? .............................................................................. 7

1.2 Information Retrieval System Examples ................................................................. 12

1.2.1 Text Retrieval ........................................................................................................ 12

1.2.2 Genomic Retrieval ................................................................................................. 13

1.2.3 Non-text Retrieval ................................................................................................. 15

2 Experimental Database and Software ...................................................................... 16

2.1 Overview .................................................................................................................. 16

2.2 Software Overview ................................................................................................. 18

2.3 Getting the Software .............................................................................................. 18

2.4 Mumps Interpreter Database Options ..................................................................... 18

2.5 Script files .................................................................................................................. 20

2.5.1 index.script .......................................................................................................... 20

2.5.1.1 Parallel Processing .......................................................................................... 20

2.5.2 iterate.script ....................................................................................................... 21

2.5.3 boolean-grep-0*.script ..................................................................................... 22

2.5.4 Mesh.script ......................................................................................................... 22

2.6 Experimental Corpus .............................................................................................. 22

2.7 Modified OHSUMED File ..................................................................................... 23

2.8 Document Preprocessing ....................................................................................... 25

2.9 Using Other Source Input Files ............................................................................. 30

3 Boolean Searching and Inverted Files ....................................................................... 31

3.1 Simple Boolean Searching Using grep .................................................................... 33

3.1.1 Regular Expression Searching with grep .......................................................... 33

3.1.2 Boolean Searching with Regular Expressions .................................................... 34

3.2 A Mumps Program to Search OSUMED .............................................................. 40

4 ISR Database Models in Mumps ................................................................................. 43

4.1 Hierarchical and Multidimensional Indexing ......................................................... 43

5 Basic Indexing, Searching and Information Retrieval .............................................. 47

5.1 Indexing Models ...................................................................................................... 47

5.2 Indexing Vocabularies .......................................................................................... 47

6 Document Indexing: the Big Picture ......................................................................... 53

7 Identifying Good Indexing Terms ............................................................................. 57

7.1 Stop Lists .................................................................................................................. 57

7.2 Building an Enhanced Stop List ............................................................................. 59

7.3 Basic Word Dictionary ............................................................................................ 62

7.4 Zipf's Law ................................................................................................................ 64

7.5 KWIC/KWOC Indices ............................................................................................ 66

7.6 Reducing Words to Stems ....................................................................................... 67

7.6.1 Words Stems ....................................................................................................... 69

8 Vector Space Model ................................................................................................... 71
8.1 Overview..................................................................................................................71
8.2 Basic Similarity Functions.......................................................................................73
8.3 Implementing the Vector Space Model in Mumps....................................................74
8.4 Vector Space Global Arrays....................................................................................75
8.5 Result Files................................................................................................................76
8.6 Run Time Settings and Defaults..............................................................................77

9 Document-Term Matrix.............................................................................................79
9.1 Building a Document-Term Matrix.........................................................................79
9.2 Normalized Word Frequencies..............................................................................82
9.3 Assigning Word Weights.......................................................................................82
9.4 Inverse Document Frequency Weight.....................................................................87
  9.4.1 OSU MEDLINE Data Base IDF Weights...............................................................87
  9.4.2 Calculating IDF Weights....................................................................................88
9.5 Discrimination Coefficients....................................................................................89

10 Weighted Matrices.....................................................................................................97
10.1 Document-Term Matrix.........................................................................................97
10.2 Title Global Array..................................................................................................98
10.3 SQL Support...........................................................................................................98

11 Thesaurus construction..............................................................................................99
11.1 Building a Term-Term Co-Occurrence Matrix.........................................................101
11.2 Term-Term-Term Connection Matrix....................................................................108
11.3 Construction of Term Phrases...............................................................................108
  11.3.1 Term Cohesion.....................................................................................................109
  11.3.2 Jaccard Term-Term Similarity...........................................................................110
  11.3.3 Term Proximity..................................................................................................111
  11.3.4 Adding Phrases to the Collection....................................................................116
11.4 Term-Term clustering.............................................................................................117
  11.4.1 Adding Clusters to the Collection....................................................................120

12 Document-Document Matrix..................................................................................121
12.1 Document Clustering.............................................................................................122

13 Retrieval..................................................................................................................127
13.1 Retrieval Using the Doc-Term Matrix....................................................................127
13.2 Retrieval Using the Term-Doc Matrix....................................................................127
13.3 Weighted Retrieval using the Term-Doc Matrix.....................................................129
13.4 Simple Cosine Weighted Retrieval Part 1...............................................................130
13.5 Simple Cosine Weighted Retrieval Part 2...............................................................132
13.6 Retrieval Enhancements........................................................................................134

14 Controlled Vocabulary Indexing Example.............................................................139
14.1 The Medical Subject Headings (MeSH).................................................................139
14.2 Building a MeSH Structured Global Array............................................................140
14.3 Displaying the MeSH Global Array Part I.............................................................143
14.4 Printing the MeSH Global Array Part II...............................................................146
14.5 Displaying Global Arrays in Key Order.................................................................149
14.6 Searching the MeSH Global Array.........................................................................150
14.7 Display OHSUMED Collection by Embedded MeSH Headings............................152
14.8 Display of OHSUMED Documents by MeSH Hierarchy............................................... 155
14.9 Searching By MeSH Terms.......................................................................................... 156

15 Evaluation of Retrieval Systems..................................................................................... 158
15.1 Precision and Recall........................................................................................................ 158

16 Alternative Text Encodings............................................................................................ 160
16.1 Truncation........................................................................................................................ 160
16.2 Soundex Encoding........................................................................................................... 160
16.3 N-gram encoding............................................................................................................ 161

17 Semantic Approaches: WordNet.................................................................................... 162

18 References....................................................................................................................... 174

Index of Figures
Figure 1 DR vs IR System.............................................................. 8
Figure 2 DBMS Data Table.......................................................... 11
Figure 3 Result Table................................................................. 11
Figure 4 Two dimensional display of data................................. 11
Figure 5 Multidimensional display........................................... 12
Figure 6 Example DNA Sequence............................................. 13
Figure 7 Example BLAST Result............................................... 15
Figure 8 Views of Old State House, Boston.............................. 15
Figure 9 Parallel processing example...................................... 21
Figure 10 Original OSUMED format........................................ 23
Figure 11 OSUMED modified format........................................ 24
Figure 12 OSUMED modified format........................................ 24
Figure 13 MEDLINE Codes...................................................... 25
Figure 14 Modified OSUMED database.................................... 29
Figure 15 Inverted file organization......................................... 32
Figure 16 Example Boolean search with grep/egrep............... 39
Figure 17 Boolean search results from Program 10............... 42
Figure 18 Global array tree...................................................... 45
Figure 19 Creating a global array............................................ 45
Figure 20 Overview of Indexing............................................... 47
Figure 21 ACM classification system...................................... 49
Figure 22 Overview of basic document indexing..................... 53
Figure 23 Best indexing terms................................................ 58
Figure 24 Frequency of top 75 OSUMED words.................. 60
Figure 25 Frequency of top 75 Wikipedia words.................... 61
Figure 26 Dictionary sorted by frequency................................ 63
Figure 27 Zipf constants - The Dead........................................ 65
Figure 28 Zipf constants - OHSUMED................................. 65
Figure 29 KWIC output.......................................................... 66
Figure 30 KWOC output......................................................... 67
Program 31 loadStems.mpss................................................. 70
Figure 32 Vector space model.................................................. 71
Figure 33 Vector space queries............................................... 72
Figure 34 Vector space clustering.......................................... 72
Figure 35 Calculating vector space similarities...................... 73
Figure 36 Similarity functions............................................... 74
Figure 37 Example similarity coefficient calculations.............. 74
Figure 38 Output Files............................................................. 77
Figure 39 Run-time settings.................................................... 78
Figure 40 Example word weights.......................................... 87
Figure 41 Good Discriminators............................................... 96
Figure 42 Poor Discriminators............................................... 96
Figure 43 SQL Select statements........................................... 99
Figure 44 SQL commands.................................................... 99
Figure 45 Term-Term Matrix.................................................. 100
Figure 46 term-term-matrix-sorted.txt................................ 106
Figure 47 Frequency of term co-occurrences.......................... 107
Figure 48 Term Term Cohesion Results................................. 110
Figure 49 Jaccard Term Term Relationships......................... 111
Figure 50 Proximity calculations........................................... 113
Figure 51 Ranked Proximity Correlations............................... 116
Figure 52 Term Clusters........................................................ 120
Figure 53 Example Document Document Clustering.............. 124
Figure 54 Enhanced retrieval output...................................... 137
Figure 55 Sample MeSH Hierarchy........................................ 139
Figure 56 Global Array Commands....................................... 140
Figure 57 MeSH Tree............................................................. 142
Figure 58 Creating the Mesh tree.......................................... 144
Figure 59 Printed the Mesh tree............................................. 146
Figure 60 Alternative MeSH printing output.......................... 148
Figure 61 MeSH global array codes...................................... 149
Figure 62 MeSH global printed.............................................. 150
Figure 63 MeSH keyword search results............................... 151
Figure 64 Titles organized by MeSH code............................ 154
Figure 65 Hierarchical MeSH concordance............................ 156
Figure 66 Precision/recall example........................................ 158
Figure 67 Precision/recall graph........................................... 159
Figure 68 Soundex codes....................................................... 161

Index of Programs
Program 1 map.mpss......................................................................... 27
Program 2 Document pre-processing........................................... 28
Program 3 reformat.mpss........................................................... 29
Program 4 Boolean OR search with grep.................................... 35
Program 5 Boolean AND and OR search with grep.................... 36
Program 6 Parallel Boolean search with grep............................ 37
Program 7 Display document numbers from grep search.......... 38
Program 8 Display titles from grep search.................................. 38
Program 9 Program to retrieve and display titles...................... 39
Program 10 Boolean search in Mumps........................................ 41
1 Introduction

1.1 What is Information Retrieval?

Information retrieval (IR), sometimes revered to as information storage and retrieval was defined by Gerard Salton, one of the pioneers in the area, as:

... a field concerned with the structure, analysis, organization, storage, searching, and retrieval of information. [Salton 1968]

Modern information retrieval systems work on many data formats. These include collections of text, images, videos, audio, chemical/pharmaceutical/biological, and many multi-media formats.

It is the goal of an IR system to analyze these objects for the purpose of (1) identifying and extracting tokens that are indicative of the object's content and meaning, and, (2) organizing and storing these in a structured knowledge base with the ultimate goal that it is searchable by users.

Users interact with IR systems in many ways including text queries that are matched against the knowledge base as well as a wide variety of graphical user interfaces that facilitate user navigation, visualization and interpretation of the content.

A key difference between an IR system and, say, a database management system (DBMS) is that an IR system matches queries with objects based on partial matches, and probabilistic analysis of the similarities between queries and objects, while a DBMS query is satisfied only by complete and exact matches.

Thus, for example, a query to an IR system operating on a collection of medical text seeking information on antibiotics will probably identify many documents. Some of these will deal exclusively with antibiotics while many others will, in all likelihood, deal with antibiotic related topics. A well designed retrieval engine will display a ranking of the results so as to indicate which documents its algorithms determined most likely best satisfied the user's query.

In a DBMS system, on the other hand, all results are equally valid since they all fully satisfied the search criteria. Results are presented in any order unless the user explicitly requests sorting based on one or more data attributes.

For example, if you query an DBMS system for red Ford sedans, the system would not be expected to retrieve red Chevrolets. An IR system, however, probably would because they are somewhat similar although the Ford results would be ranked first.

In this regard, C. J. van Rijsbergen [van Rijsbergen 1979] developed the table shown in Figure 1.

<table>
<thead>
<tr>
<th>Function</th>
<th>Data Retrieval System</th>
<th>IR System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matching</td>
<td>Exact match</td>
<td>Partial match, best match</td>
</tr>
<tr>
<td>Inference</td>
<td>Deduction</td>
<td>Induction</td>
</tr>
<tr>
<td>Model</td>
<td>Deterministic</td>
<td>Probabilistic</td>
</tr>
<tr>
<td>Classification</td>
<td>Monothetic</td>
<td>Polythetic</td>
</tr>
</tbody>
</table>
Matching. In a data retrieval (DR) system, we expect exact matching where each result is equally valid whereas in an IR system we expect partial matches and the results ranked according to which are the better or worse matches.

Inference. With regard to inference, in a data retrieval system, inference is deductive in nature. If $a \mathcal{R} b$ and $b \mathcal{R} c$ are true, then $a \mathcal{R} c$ is true for objects $a$, $b$, and $c$ and relation $\mathcal{R}$. On the other hand, in an IR system, there are degrees of uncertainty in the relationships and the matching. The validity of an inference is variable. In many cases, Bayesian probabilities are used to construct inferences.

Model. Likewise, the overall model of the systems differ. DR systems are deterministic and IR systems are probabilistic. In a DR system, there is only one set of results for a given query. In an IR system, there could be many sets of results depending on system tuning, automatic query enhancement, system learning, and so on.

Classification. At to classification of the data, in a monothetic classification system, members of a class of similar objects are defined by defined attributes both necessary and sufficient to belong to a class. On the other hand, in a polythetic classification system only some of the attributes will apply to the members of a class. For example, an class of articles on antibiotics will not only contain articles exclusively dealing with antibiotics. The results may also contain other related articles dealing with related topics such as antibiotic therapies, administration, cost, manufacture, toxicity and so on. Some may not even use the term *antibiotic*. In a DR system, each result retrieved will contain all the attributes specified and in the relationship specified by the query.

Similarly, in a DR, a query for paint brushes would not contain instances of paint rollers, sprayers or other methods of paint application even though the ultimate function might be related. In an IR system, depending upon design, they probably would.

Query Language. Query languages are another significant difference. DR systems used highly structured query languages such as SQL whereas IR systems use natural language query expressions. A properly formed expression in a DR data query language completely describes the data sought whereas a query to an IR system is ordinarily incomplete and fuzzy. IR queries are often somewhat ambiguous with regard the final results desired especially when users are just beginning their exploration of a topic.

Query context is also often a factor. For example, a query on antibiotics by a grade school user should be answered differently than one from a medical student and the...
IR system should know the context in which it operates either through user input or by means of machine learning through user feedback.

**Items Wanted.** DR systems return all matched data items without ranking or sorting unless explicitly specified. IR systems, on the other hand, return only those items deemed by the system's algorithms to be relevant to the query and these are normally ranked by the system according to calculated probable relevance.

**Errors.** Finally, how does the system deal with errors. DR systems are sensitive to errors in data encoding, query syntax, and query semantics and ordinarily fail or reject queries that fall short of rigid syntax and semantics requirements. IR systems, on the other hand, are less sensitive to errors. For example, the spellings *color* and *colour* are deemed to be equivalent in most IR systems but they are separate objects in a data retrieval system. Likewise, grammar is rarely taken into account.

DR systems come in many forms. Some of these are:

1. Database management systems (DBMS). These consist of systems that manipulate data, usually in tables, according to queries expressed in a structured language. The dominant DBMS systems at present are relational database management systems\(^1\) (RDBMS) but there have been others such as CODASYL network based systems\(^2\) (some of which are still supported\(^3\)) and newer NoSQL systems such as Apache Hadoop\(^4\), MongoDB\(^5\), and Amazon Redshift\(^6\).

RDBMS systems use an algebraic or calculus based query language, SQL, which is standardized across many vendor implementations. Non-relational systems use *ad hoc* query languages or program APIs\(^7\) specific to the implementation. In all these systems, the data elements stored and retrieved are instances of highly structured and constrained data domains. Queries are matched exactly to the data.

Example RDBMS implementations include *PostgreSQL*, *MySQL*, *Oracle*, and *Microsoft SQL Server*. The data are stored in tables and the results returned as data tables. An example data table is shown in Figure 2. A user query on such a table might be to list all the red cars in the inventory, their make and the number on hand which might be stated in SQL as:

\[
SELECT \text{MAKE, QUANTITY FROM INVENTORY WHERE COLOR = 'RED';}
\]

The result is a small table with two columns and two rows as shown in Figure 3.

---

5 [http://www.mongodb.org/](http://www.mongodb.org/)
7 Application Program Interface
2. Another type of DR systems are fact based question answering systems. These systems, often extensions of artificial intelligence, retrieve specific facts about specific domains of knowledge based on queries usually expressed in natural language. For the most part, as these deal in natural language, the information domains are subject to semantic interpretation which, places them somewhat nearer IR systems in many cases. A current example would be Ask.com. A typical query might be of the form: *Which city in the USA gets the most snow?* Results tend to be short answers and specific, much other DR systems. The query cites would not be expected to suggest Canadian or European cities with large snow fall but an IR system might.

3. Management information and on-line analytical processing (OLAP) systems are other forms of DR. These are systems, in concert with DBMSs, that integrate, analyze, synthesize and visualize information for the purpose of creating multidimensional views, designing corporate strategies, projecting trends and optimizing deployment of resources. Examples would be systems that seek correlate customer buying habits so as to optimize sales. These systems deal with facts from well constrained domains and manipulate them according to well formal procedures. A typical query might be of the form: *Identify those items from the grocery department that are likely to be purchased by customers redeeming coupons for hand soap.* Results can be tables, graphs or other visual representations of the information.

For example, consider the the presentation of data on car sales. In a DBMS system, the data will be stored in a table such as shown in Figure 2. Queries to a DBMS will produce only tabular results that are themselves data tables (see in Figure 3). That is, the results are tables with column headings and the rows that are instances of rows from one or more tables that match the search criteria and they are presented in no predefined order unless the user instructs otherwise.\(^8\)

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|}
\hline
\textbf{MAKE} & \textbf{TYPE} & \textbf{COLOR} & \textbf{QUANTITY} \\
\hline
Buick & 2 DR & red & 3 \\
\hline
Buick & convertible & blue & 5 \\
\hline
Buick & SUV & white & 4 \\
\hline
Toyota & 4 DR & silver & 6 \\
\hline
Toyota & truck & black & 10 \\
\hline
Toyota & truck & blue & 5 \\
\hline
Toyota & SUV & red & 3 \\
\hline
Ford & truck & black & 6 \\
\hline
Ford & truck & gray & 4 \\
\hline
Ford & 4 DR & green & 4 \\
\hline
Ford & SUV & yellow & 3 \\
\hline
\end{tabular}
\caption{Inventory}
\end{table}

\(^8\) SQL permits the user to specify a sort on the results based on the contents of one or more columns.
<table>
<thead>
<tr>
<th>Make</th>
<th>Color</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford</td>
<td>2 DR</td>
<td>silver</td>
</tr>
<tr>
<td>Ford</td>
<td>2 DR</td>
<td>black</td>
</tr>
<tr>
<td>Honda</td>
<td>SUV</td>
<td>red</td>
</tr>
<tr>
<td>Honda</td>
<td>van</td>
<td>gray</td>
</tr>
<tr>
<td>Honda</td>
<td>van</td>
<td>blue</td>
</tr>
<tr>
<td>Honda</td>
<td>van</td>
<td>white</td>
</tr>
</tbody>
</table>

Figure 2 DBMS Data Table

<table>
<thead>
<tr>
<th>Make</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buick</td>
<td>3</td>
</tr>
<tr>
<td>Toyota</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 3 Result Table

However, other data representations are possible. A two dimensional summary of three of the four columns from Figure 2 is shown in Figure 4. Here, the information has aggregated in a different visualization that highlights colors and models. Likewise, an even more revealing rotatable three dimensional view is also possible as seen in Figure 5. These representations are alternative visualizations of the data and built from the results of one or more database queries.

<table>
<thead>
<tr>
<th>Model</th>
<th>Color</th>
<th>red</th>
<th>white</th>
<th>blue</th>
<th>black</th>
<th>gray</th>
<th>yellow</th>
<th>silver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buick</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Toyota</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ford</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Honda</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Totals</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>15</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4 Two dimensional display of data
1.2 Information Retrieval System Examples

1.2.1 Text Retrieval

For most people, a query to an IR system means a query to Yahoo, Bing, Google, DuckDuckGo, Baidu, AOL, Ask, Excite and other similar Internet based services.

In these web search engines, the user phrases his or her query as a sequence of words or, in some cases, as a well form natural language sentence. Most queries, however, are just a sequence of one or more words.

A basic query might be of the form: *aviation*. Typically, the results returned will include many references including articles early pioneers in the field, technical reports on aircraft design, flight schedules on airlines, information on airports and so on. For example, the term *aviation* when typed into Google resulted in about 111,000,000 hits, all of which had something to do with aviation.

Search engines present results in a ranked or ordered arrangement depending on algorithms that determine relevancy of each result to the query. These algorithms are normally proprietary and take into account many factors including user search history.
When confronted with poor search results, the typical user usually rephrases the query to include more specific terminology so as to narrow the search and obtain more relevant results. Most web search engines provide limited support at query reformulation as their primary interest is to display advertisements. The more user interaction, the more ads the user is exposed to!

Many IR systems incorporate knowledge of the user into account. This can take the form of user feedback, user query history or other profiles. This information helps the IR system to more closely attenuate the results to the individual. For example, in a system that incorporates user profiles, articles retrieved in response to a query from a grade school student on the subject of nuclear power are significantly different than those returned for a graduate student.

In summary, a well designed IR system should involve many levels of user feedback in which the system interacts with the user. A well designed system will learn from users and adapt accordingly. A well designed IR system should also provide assistance in query formulation and database navigation.

1.2.2 Genomic Retrieval

Another text based example is in the area of Bioinformatics. Here, researchers with DNA or protein sequences need to search massive databases for similar and, sometimes, only distantly related, DNA or protein sequences.

For example, consider the the DNA sequence\(^9\) shown in Figure 6.

>gi|2695846|emb|Y13255.1|ABY13255 Acipenser baeri mRNA for immunoglobulin heavy chain, TGGTTACAAACACTTTTCCTTCTTTCAATAACCACAATActCGAGTACAATGAGGATTTTTAAACAGCTCTCTGTATAATAATGA CAGCCTCTATCAAGTGTCGGTCTGTAGTGAGTTGCTGAGTCTCGGAACAGCAGATTATATAAAAGCTTGAGAGTTCCCATAAAA CTGTCCTGTAAAGCCTCTGGATTACATCTACAGCCAAGCCTACATGAGCTGTTCCGAAGCATCCTGGGAAAGGGTCTGGA ATGCGGTGCTTTATATATTACTTACAGTGAGTGAGTAGACATATACTAGCCAGTCTCTCGACCGGAGATCGCATCCATCCAGAG ACGATTCCTCAACCAGATGCTGTATTATTTTTAAATAATGAAACGCCCTGAGACTGAGAAGACACTGCGGTATATTACTTGCTGCGGCC GGGCTGGGTTGATCTTTAGTACCTGGGGAAGGGCAACATGATGTACCCTAACTCTGCTACCCCATACCCACCCGACAGT GTTCTCCCTATAGGATCAATTGTGTGTGGGATGCGATCTTGCCCTGTGTCGTCTGCTTAGGCAACCCGGATCTGTGCCC TACCCCGCGACCTTCGTTGAACTGATCAATCTTGAAAAGCTTTT

Figure 6 Example DNA Sequence

The sequence in Figure 6 is that for a known and archived DNA sequence. It is shown in what is referred to as FASTA format\(^10\). In FASTA format, the first line gives the name and library accession numbers of the sequence. The subsequent lines are the DNA nucleotide codes.

All DNA is expressed as a sequence of letters from a four character alphabet. The letters are A, C, G, and T which represent the chemicals Adenine, Cytosine, Guanine, and Thymine, respectively\(^11\).

\(^9\) Note: there are also databases of protein sequences not discussed here. See the web page for the National Center for Biotechnology Information (NCBI) for additional details: http://www.ncbi.nlm.nih.gov/

\(^10\) http://en.wikipedia.org/wiki/FASTA_format
For many years several large national and international agencies have been collecting DNA sequences that have been identified by researchers. There are millions of such sequences and the number grows daily.

Researchers, upon identifying a new sequence, need to find out if the sequence is already known and what it may be related to whether known or unknown. But DNA sequences present a problem. Even if they are for the same genetic function in the same species, they may vary slightly from one individual to another. For example, some people have red hair; others have blond. This is determined by DNA sequences that are similar but differ slightly. Similarly, a DNA sequence in one species is likely to be related to a similar sequence in another species. Due to evolution, these will vary and, where the evolutionary distance is great, the variance may be large.

The question is, how do you build a search engine that can quickly look for similar but not identical sequences (called homologues)? In some cases, the relationship may be very faint but nonetheless real.

There are several IR systems to do these searches but a program known as BLAST\(^\text{12}\) (Basic Local Alignment Sequencing Tool) is one of the most popular. It can be used to find similar sequences in online databases of known DNA and protein sequences.

If you submit the sequence from Figure 6 to NCBI BLAST (National Center for Biotechnology Information), they will conduct a search of their \(nr\) database. The result will be a ranked list of hits of sequences in the database ordered according to their similarity to the query sequence. BLAST will also give a statistical evaluation of whether the match is significant and not due to random chance alone. Sequences found whose similarity scores exceed a threshold are displayed.

When the sequence from Figure 6 was submitted to BLAST, a number a matches were returned. One of these is shown in Figure 7. In Figure 7 BLAST displays the sections from the query sequence that match a portion of the sequence from the database. The numbers at the beginning and ends of the lines are the starting and ending points of the subsequence (relative to one, the start of all sequences). Where there are vertical lines between the query and the subject, there is an exact match. Where there are blanks, there was a mismatch.

It should be clear that, even though the subject differs from the query in many places, the two have a high degree of similarity. BLAST calculates a score and an expectation value which is \"...the number of hits one can "expect" to see by chance when searching a database of a particular size. It decreases exponentially as the Score (S) of the match increases\(^\text{13}\) ...\" In this case, the likelihood is 4e-33 (4 x 10\(^{-33}\)) that these two sequences are related by chance alone.

However, note the first lines from the found sequence and the query sequence. They are from different organisms but nonetheless related.

\(>\text{gb|U17058.1|LOU17058 Lepisosteus osseus Ig heavy chain V region mRNA, partial cds}\)

\(^{11}\) Note: protein sequences are ordinarily written with an alphabet of 20 letters corresponding to the amino acids found in proteins.

\(^{12}\) http://blast.ncbi.nlm.nih.gov/Blast.cgi

<table>
<thead>
<tr>
<th>Score = 151 bits (76), Expect = 4e-33</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identities = 133/152 (87%), Gaps = 0/152 (0%)</td>
</tr>
<tr>
<td>Strand=Plus/Plus</td>
</tr>
<tr>
<td>Query 242 TGGGTGGCTTATATTTACTCATGGTAGTAGCTCTGTCCAGGGGA 301</td>
</tr>
<tr>
<td>Sbjct 4 TGGGTGGCTTATATTTACACCGATGGGAGCAATACATACTATTCCCAGTCTGTCCAGGGGA 63</td>
</tr>
<tr>
<td>Query 302 AGATTCGCCATCTCCAGAGAAGCTGTATTACGACTGTACATTTAAATGAACACGCTG 361</td>
</tr>
<tr>
<td>Sbjct 64 AGATTCACCATCTCCAGAGAAGCTGTATTACGACTGTACATTTAAATGAACACGCTG 123</td>
</tr>
<tr>
<td>Query 362 AAGACTGAAGACACTGCGTTGTATTACTGTGC 393</td>
</tr>
<tr>
<td>Sbjct 124 AAGACTGAAGACACTGCGTTGTATTACTGTGC 155</td>
</tr>
</tbody>
</table>

### Figure 7 Example BLAST Result

1.2.3 Non-text Retrieval

Information retrieval, however, isn't restricted to text retrieval. So, if you have a cut from a musical piece such as from the Beethoven 9th Symphony and you want to find other music similar to it, such as from the Beethoven *Choral Fantasy*, you need a retrieval engine that can detect the obvious similarities, but not match with a chorus from von Weber's *der Freischutz*! Such systems exist but, clearly, their indexing algorithms differ significantly from those in use with text based systems.

![Figure 8 Views of Old State House, Boston](image)

Also, consider the search for similar images. Again, this involves searching for similarities, not identity. For example, a human observer would clearly see the pictures in Figure 8 as dealing with the same subject, despite the differences. An obvious question would be, can you write a computer program to see the obvious similarity?
2 Experimental Database and Software

2.1 Overview

In this book we will concentrate on text retrieval systems and examine several algorithms that have been used in the past. Accompanying this book is an open source collection of computer code used to implement these algorithms and to allow the reader to experiment with different settings and orderings as well as implement additional procedures. The programs implement one of the more popular IR models known as the Vector Space model originally due to G. Salton [Salton 1968, 1971, 1983, 1992].

The workbench presently consists of about fifty modular programs\(^{14}\) written in Mumps\(^{15}\) and/or Bash\(^{16}\) script. These programs implement the basic Vector Space Model for document classification and retrieval as originally developed by G. Salton [Salton, 1968, 1983, 1988, 1992] and others. Also included is a collection of approximately 294,000 medical abstracts for testing and experiments.

The purpose of this package is to facilitate teaching, exploration and experimentation with the vector space model and the development of new algorithms and techniques. The modular design of the code together with the Mumps multidimensional database model enable the user to experiment, augment, and measure various indexing strategies.

Currently, the package contains programs that perform:

1. word frequency analysis,
2. stop list generation,
3. word stemming,
4. term weighting,
5. synonym detection,
6. phrase identification,
7. term clustering,
8. document clustering,
9. document hyper-clustering, and
10. several retrieval methods.

The programs build:

1. document-term matrix,
2. term-document matrix,
3. term-term matrix
4. document-document matrix,

---

\(^{14}\) For use with Linux or Cygwin.

\(^{15}\) Mumps (Massachusetts General Hospital Utility Multi-Programming System) is a programming language originally developed in the late 1960s and often used in medical settings.

\(^{16}\) Bash is a shell normally available in Linux and Cygwin installations that is based on the Bourne shell originally written for Unix (bourne again shell).
5. dictionary vectors giving:
   (a) word frequency,
   (b) document frequency,
   (c) Zipf's Law coefficients,
   (d) inverse document frequency weights [Salton 1968] and
   (e) discrimination coefficients [Willet 1985].

There are programs to calculate:

1. term phrases,
2. term cohesion,
3. proximity weighted term similarities,
4. term clusters.
5. document clusters and
6. clusters of document clusters.

The package includes routines to retrieve documents based on:

1. simple sequential searches,
2. inverted file searches and
3. weighted inverted file searches using document similarity metrics such as
   Cosine [Salton 1983].

There also indexing routines to organize the documents by:

1. controlled vocabularies such as MeSH\textsuperscript{17},
2. KWIC/KWOC\textsuperscript{18} indices,
3. n-grams [Manning 1999] and

The experimental corpus provided (details given below) is the OSU Medline
collection used at the National Institute of Standards (NIST) Text Retrieval
Conference 9 (TREC-9) [NIST 2000]. Other user provided collections may also be
used if their source text is formatted according to the input model.

Most of the code in these modules is written in Mumps, a language developed in
medicine in the late 1960s [Barnett 1970, Bowie 1976, O'Kane 2008] which supports
a string handling and a multidimensional database model which is ideally suited for
vector space model implementations. The Mumps modules are invoked by \textit{bash}
scripts which control flow of data and multitasking.

The Mumps interpreter software used in these experiments are available for free
download (GPL License) at:

http://www.cs.uni.edu/~okane

\textsuperscript{17} National Library of Medicine Medical Subject Headings
\textsuperscript{18} Key Word In Context, Key Word Out of Context
2.2 Software Overview

The software, as noted, is written as a combination of Mumps and bash script. Bash script is the programming language used by the bash (Bourne-again shell) shell developed by Brian Fox for the GNU Project. It is generally available in all Linux distributions as well as Cygwin for Windows\(^{19}\) (there are also versions that run directly in Microsoft Windows as well but they are not used here).

The bash scripts in this project are used to:

1. control the order of program execution,
2. calculate threshold parameters,
3. control multitasking, and
4. provide inter-process communication.

This project assumes that the user is running in a Linux or Windows-based Cygwin\(^{20}\) installation. Linux is the preferred platform.

The Mumps language was originally developed for medical applications in the late 1960s. It is a high level, typeless, string oriented language that supports a hierarchical and multidimensional database. The database model is, as will be shown, well suited for applications involving document classification through string indexed multidimensional arrays. The version of Mumps used here in addition to the standard functions, also supports many string and matrix manipulation functions designed specifically for document indexing applications.

The GPL Mumps interpreter used in this project is a free, open-source system licensed under the GNU General Public License V2.

2.3 Getting the Software

To download Mumps, the ISR Workbench software, and related documentation, please see:

http://www.cs.uni.edu/~okane

You will also need to configure your host system with several packages including the gcc/g++ compiler, the Perl Regular Expression library and autoconf. These are free downloads in both Linux and Cygwin.

2.4 Mumps Interpreter Database Options

The database used by the GPL version of Mumps can be stored in three ways\(^{21}\). These include:

1. in a MySQL or PostgreSQL relational database,
2. in a native B-tree server and
3. in a single user native B-tree.

---

\(^{19}\) http://www.cygwin.com/

\(^{20}\) A better choice for Windows users, however, is to use the free Oracle Virtual Box for Windows and install a virtual Linux image.

\(^{21}\) Not all these options are available in the Cygwin version at present.
Of these, the first is fully ACID\textsuperscript{22} compliant while the last is the fastest but has the most limitations with regard to concurrent users (only one with read-write access) and integrity of the data\textsuperscript{23}.

As the database used in these programs is not \textit{mission critical}, the single user native B-tree is the best choice from a performance point of view. Consequently, when configuring the interpreter, please use the script file to build the system:

\texttt{BuildMumpsWithGlobalsInNative.script}

This script will build two Mumps interpreters: \texttt{/usr/bin/mumps} and \texttt{/usr/bin/mumpsRO}. The first of these can be used to both read \textit{and} write the Mumps database while the second is for \textit{read-only} database access.

In \textit{read-write} mode, only one instance of the interpreter may access the database files at a time. When using the \textit{read-only} version, however, multiple instances may concurrently access the database. The main indexing script (\texttt{index.script}) in the Workbench takes advantage of this in order to improve performance.

The \textit{read-write} and \textit{read-only} distinctions only apply to single user, native B-tree builds of the Mumps interpreter and not the server version nor the versions which use relational database systems for data storage. These other versions of Mumps all permit multiple concurrent \textit{read-write} instances\textsuperscript{24} of the interpreter but their database access speeds are considerably slower.

The \texttt{bash} script \texttt{index.script} is the primary script used to run the indexing procedures. In several places, multiple Mumps programs are initiated concurrently. For the most part, these use the \textit{read-only} version of the interpreter and write their results to files for merger into the larger database at a later stage. This significantly improves overall performance.

In a few cases, where read-write access is needed, \texttt{index.script} makes copies of the Mumps database files and initiates multiple Mumps programs to work with these in \textit{read-write} mode. The results are later merged or written to files and the database copies discarded at the end of the script. This is permitted because GPL Mumps has, in single user mode, an added command\textsuperscript{25} named \texttt{database} which is used to tell the Mumps interpreter where to find its database. Multiple \textit{read-write} instances of Mumps may run concurrently if they each have separate database files. This allows concurrent \textit{read-write} execution of processes whose results are not required by other processes. The extra database copies are deleted at the end of the script.

The user should be cautious about re-arranging the order of program execution as some routines depend upon the results of others to function correctly. Please note the use of the \texttt{bash wait} commands in \texttt{index.script}. These cause the script to wait until one or more processes have completed before additional processes are initiated.

\textsuperscript{22} Atomicity, Consistency, Isolation, Durability
\textsuperscript{23} See Mumps documentation for details.
\textsuperscript{24} In these versions, there is no separate read-only build of the interpreter.
\textsuperscript{25} Not part of the original Mumps language standard
2.5 Script files

The following is an overview of the main *bash* script files found in the Workbench.

2.5.1 index.script

The primary *bash* script to process and index a document collection is *index.script*. This script permits two optional command line parameters giving (1) the number of documents from the collection to be processed and, (2) the database to be processed. It also permits the code -? which causes basic usage to be printed but with no document processing.

If no parameters are given, a default of 1000 documents and *osu* will be used.

The following are examples of how the script may be invoked either standalone or by means of *nohup* with:

```
# process the first 10000 docs from osu.medline
index.script 10000

# 10000 docs from osu.medline in the background, output to nohup.out
nohup index.script 10000 &

# process the first 10000 docs from abc.medline
index.script 10000 abc

# display usage
index.script -?
```

For basic test runs, a small number such a 1000 may be useful but, due to the small sample size, the final indexing results may be poor. For full test runs, at least 10000 documents should be processed, perhaps more.

As the number of documents to be processed rises, internal thresholds and cutoffs will need to be altered to reflect the larger database size. One size does not fit all!

2.5.1.1 Parallel Processing

The *bash* script in *index.script* is designed to run several tasks in parallel. On a multi-core machine, this greatly reduces the time required to process a large collection of documents.

As noted above in section 2.4, there are several ways in which the Mumps interpreter can store the global arrays. In these examples, the native global array option is used as it is the fastest. However, with this option, only one task may have access to the global array database at a time. This, normally, would preclude parallel execution of Mumps tasks.

However, as noted in section 2.4, there is a *read-only* version of the interpreter. That is, a version which can only read but not write to the global array database. As a

---

26 The convention of using the *script* file extension for *bash* scripts is followed throughout.
consequence, multiple read-only instance can access the database at a time. Note that in some of the Mumps script files the interpreter invocation (first line) is for:

```
/usr/bin/mumpsRO
```

rather than the read-write version:

```
/usr/bin/mumps
```

Also note that, in a few instances, copies of the underlying database files are temporarily created in order to permit more than one instance of a read-write interpreter to execute concurrently.

Overall, as can be seen in Figure 9, the effect is to maximize CPU utilization. In this figure, all six cores are actively engaged in indexing.

Because of parallel processing code, the user should be cautious when modifying index.script.

![Figure 9 Parallel processing example](image)

### 2.5.2 iterate.script

The file iterate.script is a bash script to run multiple instances of index.script, with multiple parameters, in parallel. It does this by building sub-directories and concurrently executing instances of index.script in these.
2.5.3 boolean-grep-0*.script

These are some *bash* script examples of how to perform Boolean searches on the database using common Linux utility software (grep, egrep, sort, uniq, wc, etc.).

2.5.4 Mesh.script

A *bash* script to index the document collection using the controlled vocabulary Medical Subjects Headings.

2.6 Experimental Corpus

The experimental corpus of text provided in the distribution and used in many of subsequent examples and experiments is the OSU MEDLINE Data Base which was obtained from the TREC-9 conference. TREC (Text REtrieval Conferences) are annual events sponsored by the National Institute for Standards and Technology (NIST). These data sets are (as of April 2014) at:

http://trec.nist.gov/data.html

http://trec.nist.gov/data/t9_filtering.html

The original OHSUMED data sets can be found here:

http://ir.ohsu.edu/ohsumed/

The TREC-9 Filtering Track data base consisted of a collection of medically related titles and abstracts which requires the following disclosures:

"... The OHSUMED test collection is a set of 348,566 references from MEDLINE, the on-line medical information database, consisting of titles and/or abstracts from 270 medical journals over a five-year period (1987-1991). The available fields are title, abstract, MeSH indexing terms, author, source, and publication type. The National Library of Medicine has agreed to make the MEDLINE references in the test database available for experimentation, restricted to the following conditions:

1. The data will not be used in any non-experimental clinical, library, or other setting.

2. Any human users of the data will explicitly be told that the data is incomplete and out-of-date.

The OHSUMED document collection was obtained by William Hersh (hersh@OHSU.EDU) and colleagues for the experiments described in the papers below:

2.7 Modified OHSUMED File

For purposes of this text, the OHSUMED file was modified and edited into a revised format similar to that currently used by MEDLINE. For examples of this format, see:


The original format used many very long lines which were inconvenient to manipulate as well as a number of fields that were not of interest for this study. A sample of the original file format is given in Figure 10 and the revised data base format is shown in Figure 12.

.I 54711
.U 88000001
.S Alcohol Alcohol 8801; 22(2):103-12
.M Acetaldehyde/*ME; Buffers; Catalysis; HEPES/PD; Nuclear Magnetic Resonance; Phosphates/*PD; Protein Binding; Ribonuclease, Pancreatic
/AI/*ME; Support, U.S. Gov't, Non-P.H.S.; Support, U.S. Gov't, P.H.S.;
.T The binding of acetaldehyde to the active site of ribonuclease: alterations in catalytic activity and effects of phosphate.
.P JOURNAL ARTICLE.
.W Ribonuclease A was reacted with [1-13C,1,2-14C]acetaldehyde and sodium cyanoborohydride in the presence or absence of 0.2 M phosphate. After several hours of incubation at 4 degrees C (pH 7.4) stable acetaldehyde-RNase adducts were formed, and the extent of their formation was similar regardless of the presence of phosphate. Although the total amount of covalent binding was comparable in the absence or presence of phosphate, this active site ligand prevented the inhibition of enzymatic activity seen in its absence. This protective action of phosphate diminished with progressive ethylation of RNase, indicating that the reversible association of phosphate with the active site lysyl residue was overcome by the irreversible process of reductive ethylation. Modified RNase was analysed using 13C proton decoupled NMR spectroscopy. Peaks arising from the covalent binding of enriched acetaldehyde to free amino groups in the absence of phosphate were as follows: NH2-terminal alpha amino group, 47.3 ppm; bulk ethylation at epsilon amino groups of nonessential lysyl residues, 43.0 ppm; and the epsilon amino group of lysine-41 at the active site, 47.4 ppm. In the spectrum of RNase ethylated in the presence of phosphate, the peak at 47.4 ppm was absent. When RNase was selectively premethylated in the presence of phosphate, to block all but the active site lysyl residues and then ethylated in its absence, the signal at 43.0 ppm was greatly diminished, and that arising from the active site lysyl residue at 47.4 ppm was enhanced. These results indicate that phosphate specifically protected the active site lysine from reaction with acetaldehyde, and that modification of this lysine by acetaldehyde adduct formation resulted in inhibition of catalytic activity.
.A Mauch TJ; Tuma DJ; Sorrell MF.

Figure 10 Original OSUMED format

The meaning of the codes used in Figure 10 are shown in Figure 11
The binding of acetaldehyde to the active site of ribonuclease: alterations in catalytic activity and effects of phosphate.

Ribonuclease A was reacted with [1-13C,1,2-14C]acetaldehyde and sodium cyanoborohydride in the presence or absence of 0.2 M phosphate. After several hours of incubation at 4 degrees C (pH 7.4) stable acetaldehyde-RNase adducts were formed, and the extent of their formation was similar regardless of the presence of phosphate. Although the total amount of covalent binding was comparable in the absence or presence of phosphate, this active site ligand prevented the inhibition of enzymatic activity seen in its absence. This protective action of phosphate diminished with progressive ethylation of RNase, indicating that the reversible association of phosphate with the active site lysyl residue was overcome by the irreversible process of reductive ethylation. Modified RNase was analysed using 13C proton decoupled NMR spectroscopy. Peaks arising from the covalent binding of enriched acetaldehyde to free amino groups in the absence of phosphate were as follows: NH2-terminal alpha amino group, 47.3 ppm; bulk ethylation at epsilon amino groups of nonessential lysyl residues, 43.0 ppm; and the epsilon amino group of lysine-41 at the active site, 47.4 ppm.

In the spectrum of RNase ethylated in the presence of phosphate, the peak at 47.4 ppm was absent. When RNase was selectively premethylated in the presence of phosphate, to block all but the active site lysyl residues and then ethylated in its absence, the signal at 43.0 ppm was greatly diminished, and that arising from the active site lysyl residue at 47.4 ppm was enhanced. These results indicate that phosphate specifically protected the active site lysine from reaction with acetaldehyde, and that modification of this lysine by acetaldehyde adduct formation resulted in inhibition of catalytic activity.
The meaning of the codes used in Figure 12 are shown in Figure 13.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MH</td>
<td>MeSH heading term</td>
</tr>
<tr>
<td>TI</td>
<td>title</td>
</tr>
<tr>
<td>AB</td>
<td>abstract</td>
</tr>
</tbody>
</table>

All data fields begin in column 7 and all descriptors begin in column 1. Each entry begins with the text STAT- MEDLINE.

This revised file is named osu.medline in the distribution.

2.8 Document Preprocessing

The bash script file index.script has code to pre-process documents which are in the format shown in Figure 5. It works on an input file named $DB.medline where $DB is a command line parameter to index.script with osu the default value for $DB if none is supplied.

The bash script file index.script checks if $DB.converted exists. If not, it is created (this can be time consuming for large input files). This is shown in Figure 2.

If $DB.converted is to be build it is first preprocessed by a group of sed commands that remove punctuation, numbers, and short words to create tmp.tmp. which is passed to reformat.mps which creates $DB.converted.

The final format of $DB.converted is:

1. Each document is on one (very long) line;
2. each line begins with:
   1. the offset, in bytes, of the start of the abstract entry in osu.medline,
   2. followed by one blank,
   3. followed by the document number,
   4. followed by one blank,
3. the remainder of the line consists of continuous text taken from:
   1. the title, and
   2. the abstract from the original document.
4. All text is converted to lower case.

Initially map.mps, shown in Program 1, locates the offset where each document begins in the original $DB.medline file. These offsets are stored in the global array ^map() indexed by document number. The contents of this array are used in reformat.mps to write the final output.

Input to the series of piped sed commands in Program 2 is initially from $DB.medline. Each successive sed command sends its output to the next and, on a

---

28 "...sed (stream editor) is a Unix utility that parses and transforms text, using a simple, compact programming language. sed was developed from 1973 to 1974 by Lee E. McMahon of Bell Labs, and is available today for most operating systems. sed was based on the scripting features of the interactive editor ed ("editor", 1971) and the earlier qed ("quick editor", 1965–66). sed was one of the earliest tools to support regular expressions, and remains in use for text processing, most notably with the substitution command....” Wikipedia
multi-processor machine, these will run concurrently. The backslash at the end of each line indicates that the next line is a continuation. That is, all the sed commands are, from the Linux point of view, one long command line. The output from the final sed command is written to tmp.tmp which will become input to the next step.
The sed commands consist of a regular expression to search for followed by the replacement for what is found (a blank in each case). The commands do the following:

1. Find and remove the characters\(^{29}\): []+"=,;():/
2. Find and remove STAT-MEDLINE
3. Find and remove numbers like 123.456
4. Find and remove numbers like 1234.
5. Find and remove all remaining numbers
6. Find and remove all decimal points / periods
7. Find and remove doubled hyphens
8. Find and remove hyphens preceded by a blank
9. Find and remove hyphens followed by a blank
10. Find and remove hyphens preceded and followed by blanks
11. Find and remove asterisks

\(^{29}\) Note that ] and [ appear at the beginning and in that order. This is because [...] has a meaning in sed.
# Check to see if input files exist and create if not

if [ -f "$DB.converted" ] && [ -f "titles.list" ]
  then
    echo "Not rebuilding $DB.converted - files exist"
else
  echo "Rebuilding $DB.converted"
  echo "This will take several minutes..."

  map.mps < $DB.medline
  
  sed -e 's/\[\][+"%=,;:()/\]/g' $DB.medline ||
  sed -e 's/STAT-MEDLINE/STAT-MEDLINE/' ||
  sed -e 's/[0-9][0-9]*.\[0-9][0-9]*/ /g' ||
  sed -e 's/[0-9][0-9]*/ /g' ||
  sed -e 's/\[0-9]/ /g' ||
  sed -e 's/\//g' ||
  sed -e 's/[\-\[\\{2,5\}\]/ /g' ||
  sed -e 's/-/ /g' ||
  sed -e '/\[a-zA-Z]\{1,3\}/ /g' ||
  reformat.mps < tmp.tmp ||
  sed -e "s/'//g" \
    -e 's/ [a-zA-Z]\\{1,3\} / /g' \
    -e 's/^[a-zA-Z]\\{1,3\} / /g' \
    -e 's/ [a-zA-Z]\\{1,3\}$/ /g' \
  sed -e 's/ \{2,6\}$/ /g' \
  > $DB.converted
fi

Program 2 Document pre-processing

The file from the sed commands is passed to reformat.mps which is shown in Program 3. This program reads the medline format and converts it to a more easily managed format. It also writes to titles.list the document number followed by a blank followed by the title of each document truncated to 93 characters.

Output from reformat.mps is processed by several sed expressions that remove words of three of fewer characters and short fragments of words with hyphens.
When processing the documents, we begin with the full text where the only modification that all letters have been converted to lower case. In later steps, we:

1. the original text begins at offset 148710⁴ in the osu.medline file, and
2. the document number is 130.
1. remove commonly occurring terms (stop list words), and
2. replace words with their semantic stem,
3. remove numbers and short words.

By semantic stem, we mean a term which is at the root of the meaning of several variant forms of the word. For example, computer, computed, computing, compute, computes, computational and so forth. In many cases, this may simply mean changing plural forms to singular while in others is can result in significant prefix and suffix removal.

2.9 Using Other Source Input Files

Files other than the provided versions of osu.medline/osu.converted may be used as input to the processing routines. These files must have the same file name prefix and the suffixes .medline and .converted. The file with the .medline prefix is optional. If it does not exist, the byte offset in the file with the .converted suffix should be zero or it may be any value you chose so long as it contains NO embedded blanks.

You must also create a file named titles.list which consists of, for each document in your collection, a line containing:

1. the document number beginning in column one (one means one, not two or three ...),
2. one blank (one means one, not a couple or a <tab>),
3. the text of a title (less that 2048 characters in length.).

The file with the suffix .converted should be derived from the collection you wish to index. It consists of, for each document in your collection, a line containing:

1. the byte offset, beginning in column one, of the original document in the file with the .medline suffix (or zero or a value of your choosing containing no embedded blanks). If there is no file with the .medline extension, this value needs to be zero or a token of your own design so long as it contains no embedded blanks.
2. one blank.
3. the document number of the document which must correspond to the document number used in titles.list.
4. one blank.
5. the text and title of the document which all characters reduced to lower case. This line may be arbitrarily long. You may add additional terms to the line if you want the terms to be used in the indexing process.
3 Boolean Searching and Inverted Files

A number of popular commercial information retrieval systems over the years have been based on queries rooted in Boolean logic. In these systems, query terms are typically connected by operators such as AND, OR, XOR (exclusive or) and NOT. In its simplest form, a Boolean query seeks to find those documents which contain (or not contain) terms matching the query.

Queries are constructed as logical expressions involving the query terms. Each query term may be thought of a a set of documents. When two words are and'ed, the sets are intersected; when two words are or'ed, the sets are combined (duplicate identifiers are removed). When a NOT is used, the not'ed set is subtracted from the first set. Parentheses are used as needed to express the order of evaluation. For example:

1. COMPUTERS AND MEDICINE
2. COMPUTERS AND (ONCOLOGY OR GASTROENTEROLOGY OR CARDIOLOGY)
3. COMPUTERS NOT ANALOG

The first query would retrieve those documents containing both the terms COMPUTERS and MEDICINE. In the second, those documents which contained the term COMPUTERS and one (or more) of: ONCOLOGY, GASTROENTEROLOGY, or CARDIOLOGY. In the third example, those documents containing COMPUTERS but not ANALOG.

In Boolean based systems, the documents themselves are indexed by words or terms derived either from:

1. the documents themselves or
2. assigned to the documents from a controlled vocabulary or dictionary.

A Boolean search can be conducted in two ways:

1. Each document in the collection can be inspected and evaluated in terms of the Boolean search expression, or,
2. The Boolean search expression can be evaluated by means of an inverted index file.

An inverted file is one in which, for each word in the indexing vocabulary, there is a list of pointers to the documents which contain those words.
In an inverted file system, sets of document numbers (or other referential tokens) are intersected or joined. The resulting document numbers are used to retrieve the actual text.

For example, the query: *COMPUTERS AND MEDICINE* would be processed by intersecting the set of document numbers of those documents containing the term *COMPUTERS* with the set of document numbers containing the term *MEDICINE*. The resulting document numbers are those of documents containing both terms.

Normally, the results are presented without ranking but some systems rank the retrieved documents according to the relative frequency of query words in the document versus other documents and other techniques [Blair 1996].

Additional operators can be used such as:

1. *ADJ* requiring words to be adjacent or (*ADJ 5*) requiring the words be within 5 words of one another, or
2. *WITH* requiring the words to be in the same sentence, or
3. *SAME* requiring the words to be in the same paragraph.
4. *SYN* indicating possible term synonyms.

These examples are taken from IBM STAIRS [Blair 1996] (known as SearchManager/370 in later versions) and Lockheed's original DIALOG systems.

Wildcard truncation characters is also possible. For example *COMPUT?* would match the terms:
Most systems of this kind retain the results of searches during a session and permit prior results to be used in new queries:

1: COMPUTERS AND MEDICINE
2: 1 AND ONCOLOGY

In some systems, a user might be asked to rank the importance of the terms. Documents are then scored based on the sum of user assigned weights for the search terms and only those exceeding a threshold are displayed. For example:

<table>
<thead>
<tr>
<th>Term</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONCOLOGY</td>
<td>4</td>
</tr>
<tr>
<td>CARDIOLOGY</td>
<td>5</td>
</tr>
<tr>
<td>VIROLOGY</td>
<td>3</td>
</tr>
<tr>
<td>GASTROENTEROLOGY</td>
<td>2</td>
</tr>
<tr>
<td>THRESHOLD</td>
<td>6</td>
</tr>
</tbody>
</table>

If the threshold for document display is 6, a document with only VIROLOGY and GASTROENTEROLOGY would not be displayed (weight of 5) but another document with CARDIOLOGY and GASTROENTEROLOGY (weight of 7) would be displayed. These weights might also be used to rank the documents.

### 3.1 Simple Boolean Searching Using `grep`

The purpose of this section is to explore some simple ways to search a text database using built-in Linux utility programs such as `grep`, `egrep`, `wc`, `sort` and `uniq`.

#### 3.1.1 Regular Expression Searching with `grep`

The most basic form of searching involves simply scanning the documents (the medical abstracts in this case) for words. In Linux and other systems this can be done with built-in programs such as `grep` or `egrep`. `Grep` was originally written by Ken Thompson for Unix in the early 1970s. Since then it has been extended and expanded and is now available on Linux, Unix and Windows.

At its simplest, `grep` scans one or more files for lines containing an instance of a word. For example:

```
grep -i alcohol osu.converted
```

will scan the file `osu.converted` for instances of the word `alcohol`. The lines containing the word `alcohol` will be written to standard out (stdout). `Grep` is case sensitive by default. By adding the switch `-i`, it becomes case-insensitive.
The search argument to *grep* may be a regular expression [Aho 1990]. Regular expressions are used in many utilities and programming languages (including Mumps) to specify pattern matching criteria.

For example, if you want those abstracts from *osu.converted* that contained any word beginning with the term *alcohol*, you could type:

```
grep -i "alcohol[a-z]*" osu.converted
```

where the figure \[a-z\] means any letter in the range of \(a\) through \(z\) and the asterisk means repeated zero or more times.

Similarly, specific combinations of letters may be used such as:

```
grep -i "alcohol[ic|ics]" osu.converted
```

which means that the word matched must begin with *alcohol* and be followed by either *ic* or *ics* (*alcoholic* or *alcoholics*).

To see the count of number of documents containing, for example, the words *alcoholic* or *alcoholics* you type:

```
grep -i "alcohol[ic|ics]" osu.converted | wc -l
```

which passes (*pipes*) the output from *grep* to *wc* which, in turn, displays the number of lines it received. Since each document in this file is on one (long) line, the number written by *wc* is the number of documents containing one of the words. Note: all the *stdout* output of *grep* is passed to *wc* so you will only see the final line count. The -l parameter to *wc* tells it to output only the line count.

The Perl language also has an extended pattern match features and is very well suited for searches such as these.

### 3.1.2 Boolean Searching with Regular Expressions

Note: in the experimental files supplied and built, by default, are *osu.medline* and *osu.converted* and these are the names used in this text. However, in the *bash* script files, the *osu* prefix is a *bash* variable substitution.

The *grep* searches shown above can be extended to handle Boolean expression based searches.

For example, if the query is to find those abstracts containing:

```
  alcohol* AND gambl*
```

you could construct the following command line\(^{31}\):

```
grep "alcohol[a-z]*" osu.converted | \  
```

\(^{31}\) The backslash at the line end indicates that the line is continued on the next line.
grep "gambl[a-z]*e" osu.converted | wc -l

which first finds all articles containing words that begin with alcohol and then, from this set, selects those articles containing words that begin with gambl. This effectively ands the result which is piped to wc for the final count.

Alternatively, the expression:

alcohol OR gambling

could be rendered as shown in Program 4.32

```
#!/bin/bash
# Copyright 2014 Kevin C. O'Kane
# boolean-grep-01.script
if [ -f "DBPREFIX" ]
    then
    db=`cat DBPREFIX`
    echo "Using $db.converted"
else
    echo "File DBPREFIX not found"
    exit
fi

pid=$$

grep "alcohol[a-z]*" $db.converted > /tmp/$pid.tmp

grep "gambl[a-z]*" $db.converted >> /tmp/$pid.tmp

sort /tmp/$pid.tmp | uniq | wc -l ; rm /tmp/$pid.tmp
```
Program 4 Boolean OR search with grep

The $$ figure in Program 4 is a builtin bash variable that contains the process ID of the current shell. Since no two process IDs are concurrently same33, this value will be unique. The process ID is stored in the bash shell variable pid.

The first grep command stores its results in a temporary file placed in /tmp whose name consists of the process ID followed by .tmp.

The second grep concatenates34 its results onto the end of this file.

The last line runs several programs with the output of each becoming the input of the next.

---

32 Note that the prefix for the file with the .converted suffix is obtained from the file DBPREFIX. DBPREFIX is created by index.script which should be executed prior to doing these examples. It stores the file name prefix (osu by default) in DBPREFIX and the number of documents processed in the file MAXDOCS. The file DocCount is also created by index.script and contains the number of documents actually in the indexed collection. This number may be significantly less that MAXDOCS as some documents may be eliminated during indexing.

33 Process IDs are ultimately re-used but the same process ID is never assigned to two processes executing at the same time.

34 the operator >> means concatenate output at the end of the target file.
The `sort` command sorts the results from `$pid.tmp` based on the first whitespace delimited token in each line. In our case, this is the unique (and identifying) file offset of the abstract in the original OSUMED file.

The output of the `sort` is then piped to `uniq` which collapses duplicate lines to one line. For example, a document which contains both word stems would be in the output file twice.

The output of `uniq` is piped to `wc` which counts the number of abstracts that contain either or both of the search terms. The `rm` command deletes the temporary file.

A more complex query such as:

```
(alcohol OR gambling) AND smoking
```

could be written as shown in Program 5. In Program 5 the result of the OR operation from lines 6 and 7 in `/tmp/$pid.2.tmp`, becomes input to the AND operation in the third `grep` command.

```bash
#!/bin/bash
# Copyright 2014 Kevin C. O'Kane
# boolean-grep-02.script

if [ -f "DBPREFIX" ]
    then
db=`cat DBPREFIX`
echo "Using $db.converted"
else
    echo "File DBPREFIX not found"
    exit
fi

pid=$$
grep "alcohol[a-zA-Z]*" $db.converted > /tmp/$pid.1.tmp
grep "gambl[a-zA-Z]*" $db.converted >> /tmp/$pid.1.tmp
sort /tmp/$pid.1.tmp | uniq > /tmp/$pid.2.tmp
grep "smoking" /tmp/$pid.2.tmp | wc -l; rm /tmp/$pid.*.tmp
```

Program 5 Boolean AND and OR search with grep

An alternative `grep` expression for line 6 Program 5 might be:

```
grep "alcohol[a-zA-Z]*\|gambl[a-zA-Z]*" osu.converted > /tmp/$pid.1.tmp
```

where the OR operation is performed by `grep` (the `\|` means OR). This would eliminate the second `grep` on line 7. However, in tests, this modification actually resulted in slower execution. Not all intuitive solutions work! Always test.

---

35 See `boolean-grep-02b.script`
36 On a 6 core AMD processor under Linux Mint LMDE in April, 2014. Your mileage may vary.
Program 5 could be modified to improve performance by piping the output of line 8 directly as input to the \texttt{grep} in line 9. This would reduce the number of intermediate files.

Another variation of Program 5 would involve parallel execution of the \texttt{grep} functions on lines 6 and 7. On a multi-core processor, this should improve execution substantially. The modified code is given in Program 6. In this program, the \texttt{grep} functions on lines 6 and 7 are executed in parallel (because of the \\& at the end of the lines). The two output files become input to the \texttt{sort} function on line 9. The \textit{bash wait} command on line 8 causes the script to pause until all background processes in the current shell have completed.

```
#!/bin/bash
# Copyright 2014 Kevin C. O'Kane
# boolean-grep-02a.script

if [ -f "DBPREFIX" ]
  then
data=`cat DBPREFIX`
  echo "Using $db.converted"
else
  echo "File DBPREFIX not found"
  exit
fi

pid=$$

grep "alcohol[a-z]*" $db.converted > /tmp/$pid.1.tmp &
grep "gambl[a-z]*" $db.converted > /tmp/$pid.2.tmp &
wait

sort /tmp/$pid.1.tmp /tmp/$pid.2.tmp | uniq > /tmp/$pid.3.tmp

grep "smoking" /tmp/$pid.3.tmp | wc -l; rm /tmp/$pid.*.tmp
```

Program 6 Parallel Boolean search with grep

Rather than just print the count of the number of documents fulfilling the search criteria, we may want to know the specific document numbers and file offsets. This can be done using a command sequence such as shown in Program 7

Final output (from \texttt{egrep}) will appear on stdout. \textit{Egrep} is an extended version of \texttt{grep} that permits additional operators, the + operator in this case. In the above, the \texttt{egrep} receives the output of the final \texttt{grep} and matches the first two numeric sequences on each line. The ^ operator means \textit{at the beginning} of line and the + means 1 or more instances of the preceding token (a number in the range 0 through 9 in this case). A blank must separate the two numbers. The output, determined by the -o switch, will only consist of the portion of the input line matched (that is, the first two numbers)\footnote{Without arguments, \texttt{wait} waits for all background processes to complete. If you specify one or more process IDs as arguments to \texttt{wait}, \texttt{wait} waits only for those you specify. The process ID of the most recent command is in the \textit{bash} builtin variable $! which can be assigned to a shell variable for later reference.}

\footnote{As the ^ operator means beginning of line, the $ operator means end of line.}

37
38
#!/bin/bash
# Copyright 2014 Kevin C. O'Kane
# boolean-grep-03.script

if [ -f "DBPREFIX" ]
then
    db=`cat DBPREFIX`
    echo "Using $db.converted"
else
    echo "File DBPREFIX not found"
    exit
fi

pid=$$

grep "alcohol[a-z]*" $db.converted > /tmp/$pid.1.tmp
grep "gambl[a-z]*" $db.converted >> /tmp/$pid.1.tmp
sort < /tmp/$pid.1.tmp | uniq > /tmp/$pid.2.tmp
grep "smoking" /tmp/$pid.2.tmp | egrep -o "^[0-9]+ [0-9]+ " | get-docs.mps
rm /tmp/$pid.*.tmp

Program 7 Display document numbers from grep search

To display the titles from the above, use Programs 8 and 9.

#!/bin/bash
# Copyright 2014 Kevin C. O'Kane
# boolean-grep-04.script

if [ -f "DBPREFIX" ]
then
    db=`cat DBPREFIX`
    echo "Using $db.converted"
else
    echo "File DBPREFIX not found"
    exit
fi

pid=$$

grep "alcohol[a-z]*" $db.converted > /tmp/$pid.1.tmp
grep "gambl[a-z]*" $db.converted >> /tmp/$pid.1.tmp
sort < /tmp/$pid.1.tmp | uniq > /tmp/$pid.2.tmp
grep "smoking" /tmp/$pid.2.tmp | egrep -o "^[0-9]+ [0-9]+ " | get-docs.mps
rm /tmp/$pid.*.tmp

Program 8 Display titles from grep search

where get-docs.mps is given in Program 9.

39 Note: The indexing procedure in index.script creates the global array ^title(d) that contains the titles for the given document number (d) which we assume exists for this program.
As an example, the query corresponding to *(alcohol OR gambling) AND smoking* yields the results in Figure 16. based on the first 5,000 abstracts (long lines truncated).

When multiple functions are run on the same line and if the output of one becomes the input of the next, on a machine with multiple processors, the system will schedule normally the separate functions concurrently onto different processors thus improving speed by parallel execution. *Bash* also has facilities to permit programs initiated by multiple command lines to be executed concurrently. However, even with parallel execution, the operations described above are sequential in nature and inherently slow on very large collections.

Finally, it should be noted that functions written in most programming languages have the ability to create and issue shell commands. In GPL Mumps this can be done several ways including the *shell* command and the $zsystem() built-in function. The *shell command* can be used to read or write (but not both at the same time) from/to the shell. In the C/C++ language, the system() and popen() functions provide this functionality.

Overall speed of these operations can be increased on a multi-core machine if the underlying database is *sharded* (that is, split into multiple pieces) and the search conducted in parallel on each shard or piece with the final results combined (similar to Map-Reduce).

As an exercise, write a program that accepts a Boolean expression involving vocabulary terms and the operators AND, OR, and NOT, parses the expression, and then executes the appropriate *bash* shell commands to search the database.
3.2 A Mumps Program to Search OSUMED

The code in Program 10 is a simple full-text, sequential, boolean search of the OSUMED collection written in Mumps. Program 10 produces output as shown in Figure 17. It operates on the modified OHSUMED database osu.converted an example of which was shown in Figure 14 on page 29.

The program first reads a query and then loads it into a Mumps internal line buffer. The query consists of one or more search terms separated by the operators AND (&), OR (!), NOT ('), and matching sets of parentheses such as:

\[(\text{term}_1 \& \text{term}_2) \mid (\text{term}_3 \& \text{term}_4) \& \neg \text{term}_5\]

\[(\text{term}_1 \text{ AND} \text{term}_2) \text{ OR} (\text{term}_3 \text{ AND} \text{term}_4) \text{ AND NOT} \text{term}_5\]

where \text{term}_1,...,\text{term}_5 are words or terms from the vocabulary.

The loop extracts tokens ($zwparse$) from the query buffer and builds a Mumps expression in the string variable exp. This expression is a Mumps translation of the user's Boolean query using Mumps logical operators and the Mumps $\text{find()}$ function. So, for example, a user expression such as:

\[(\text{apples} \text{ AND} \text{oranges}) \text{ OR} \text{pears}\]

becomes:

\[(\text{\$find(line,"apples")}\&\text{\$find(line,"oranges")})\neg\text{\$find(line,"pears")}\]

The Mumps expression, when executed on a line from the text (remember: in osu.converted, each document is one line), the individual $\text{find()}$ functions search the contents of line for an instance of the search term (for example, apples). If the term is found, the result for the function is $true$ (non-zero), $false$ (zero) otherwise. The results of these searches are combined with the logical operators to find a final result ($true$ or $false$). This result will determine if a document meets the search criteria. If the final result is $true$, the line (document) is said to fulfill the search criteria.

Once a user query is converted to a Mumps expression, the syntax is tested. The expression is then applied to consecutive input lines from osu.converted in a loop. Each line of osu.converted, read into the Mumps variable line, contains the full text of the original abstract converted to lower case, stemmed, and devoid of punctuation.

The loop executes the expression in exp in the if statement. If the whole expression results in $true$, the program extracts from line the offset of the document in the original file and the document number. The title\(^{40}\) is displayed along with the document number.

---

\(^{40}\) We assume the $^\text{\$title(docnbr)}$ has been built and contains the text of the titles for all document numbers $docnbr$. 

40
#!/usr/bin/mumpsRO
# Copyright 2014 Kevin C. O'Kane
# boolean.mps February 14, 2014
# assumes that ^titles(docnbr) exists

read "Enter query terms ",query
set query=$zlower(query)
set i=$zwstore(query)

set exp=""
for w=$zwparse do
  if w="" break
  if $find("()",w) set exp=exp_w continue
  if w="|"!(w="OR") set exp=exp_"!" continue
  if w="-"!(w="NOT") set exp=exp_"!" continue
  if w="&"!(w="AND") set exp=exp_"&" continue
  set exp=exp_"$f(line,","_w_"""")"
write !,"Mumps expression to be evaluated on the data set: ",exp,!!
set $noerr=1 // turns off error messages
set line=" " set i=@exp // test trial of the expression
if $noerr<0 write "Expression error number ",-$noerror,!, got to again
open 2:"MAXDOCS,old"
if '$test write "MAXDOCS not found.",! halt
use 2 read M close 2
use 5

open 2:"DBPREFIX,old"
if '$test write "DBPREFIX not found.",! halt
use 2 read P close 2
use 5

set file=P_".converted,old"
open 1:file
if '$test write "file error",! halt
set i=0
for j=1:1:M do
  use 1
  . read line
  . if '$test break
  . if @exp do
    . set off=$piece(line," ",1)
    . set docnbr=$piece(line," ",2)
    . use 5
    . write docnbr,?10,$e(^title(docnbr),1,80),!
  use 5
write !,"M, documents searched",!!
halt

Program 10 Boolean search in Mumps
The program in Program 10 is slow, however, because it searches each document sequentially. A better way would be to build an inverted index of all significant words (with file offset pointers into the original OHSUMED file) and process the queries against the inverted index. This is discussed below.

Enter query: drink & alcohol

Mumps expression to be evaluated on the data set: $f(line,"drink") & f(line,"alcohol")

<table>
<thead>
<tr>
<th>Document</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Drinkwatchers--description of subjects and evaluation of laboratory markers of</td>
</tr>
<tr>
<td>7</td>
<td>Bias in a survey of drinking habits.</td>
</tr>
<tr>
<td>1490</td>
<td>Self-report validity issues.</td>
</tr>
<tr>
<td>1491</td>
<td>A comparison of black and white women entering alcoholism treatment.</td>
</tr>
<tr>
<td>1492</td>
<td>Predictors of attrition from an outpatient alcoholism treatment program for</td>
</tr>
<tr>
<td>1493</td>
<td>Effect of a change in drinking pattern on the cognitive function of female social</td>
</tr>
<tr>
<td>1494</td>
<td>Alcoholic beverage preference as a public statement: self-concept and social image</td>
</tr>
<tr>
<td>1496</td>
<td>Influence of tryptophan availability on selection of alcohol and water by men.</td>
</tr>
<tr>
<td>1497</td>
<td>Alcohol-related problems of children of heavy-drinking parents.</td>
</tr>
<tr>
<td>1499</td>
<td>Extroversion, anxiety and the perceived effects of alcohol.</td>
</tr>
<tr>
<td>2024</td>
<td>Psychiatric disorder in medical in-patients.</td>
</tr>
</tbody>
</table>

3648 documents searched

-----

Enter query: (drink | alcohol) & problem

Enter query terms

Mumps expression to be evaluated on the data set: ($f(line,"drink")! $f(line,"alcohol")) & $f(line,"problem")

<table>
<thead>
<tr>
<th>Document</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Bias in a survey of drinking habits.</td>
</tr>
<tr>
<td>1056</td>
<td>Reduction of adverse drug reactions by computerized drug interaction scree</td>
</tr>
<tr>
<td>1069</td>
<td>Suicide attempts in antisocial alcoholics.</td>
</tr>
<tr>
<td>1487</td>
<td>Childhood problem behavior and neuropsychological functioning in persons a</td>
</tr>
<tr>
<td>1496</td>
<td>Influence of tryptophan availability on selection of alcohol and water by</td>
</tr>
<tr>
<td>1497</td>
<td>Alcohol-related problems of children of heavy-drinking parents.</td>
</tr>
<tr>
<td>1959</td>
<td>Native American postneonatal mortality.</td>
</tr>
<tr>
<td>2024</td>
<td>Psychiatric disorder in medical in-patients.</td>
</tr>
<tr>
<td>4430</td>
<td>Family history of problem drinking among young male social drinkers: behav</td>
</tr>
<tr>
<td>4435</td>
<td>Dose-related effects of alcohol among male alcoholics, problem drinkers an</td>
</tr>
<tr>
<td>4439</td>
<td>Reliability and validity of the MAST, Mortimer-Filkins Questionnaire and C</td>
</tr>
<tr>
<td>4440</td>
<td>Social drinking and cognitive functioning in college students: a replicati</td>
</tr>
<tr>
<td>5453</td>
<td>Fatal occupational injuries of women, Texas 1975-84.</td>
</tr>
<tr>
<td>7532</td>
<td>Substance use and mental health problems among sons of alcoholics and cont</td>
</tr>
</tbody>
</table>

Figure 17 Boolean search results from Program 10
In the following sections we will conduct experiments illustrating several approaches to indexing and retrieving information from very large text data sets. These will require very large files and substantial amounts of computer time.

Many of the basic programming models in IS&R assume the existence of large, sparse, string indexed, multidimensional matrices. However, for the most part, data structures such as these are not well supported, if at all, in most programming languages.

Rather than implement the models from scratch in C/C++, Java or PHP, this text uses the Mumps language. Mumps is a very simple interpretive database scripting language that easily supports the disk based data structures needed for our purposes and it can be learned in a matter of hours.

Mumps (also referred to as $M$) is a general purpose programming language that supports a native hierarchical and multidimensional data base facility. It is supported by a large user community (mainly biomedical), and a diversified installed application software base. The language originated in the mid-60's at the Massachusetts General Hospital and it became widely used in both clinical and commercial settings.

As originally conceived, Mumps differed from other mini-computer based languages of the late 1960's by providing:

1. an easily manipulated hierarchical and multidimensional database that was well suited to representing the tree-like structure of medical records;
2. flexible string handling support; and
3. multiple concurrent tasks in limited memory on very small machines.

Syntactically, Mumps is based on an earlier language named JOSS and has an appearance that is similar to early versions of BASIC that were also based on JOSS.

### 4.1 Hierarchical and Multidimensional Indexing

In the following sections are examples of Mumps programs used to store and manipulate basic IS&R hierarchical indexing and multidimensional structures.

While table based Relational Database Management Systems (RDBMS) such as IBM DB2, MySQL, Microsoft SQL Server, PostgreSQL, Oracle RDBMS, dominate the commercial realm, not all data models are well suited to a tabular approach. Dynamically organized and hierarchically structured data with varying tree path lengths are not well suited to the relational model.

The hierarchical/multidimensional approach used in Mumps is also found in IBM IMS which was also originally developed in the 1960s and still widely used to this day. IMS is reputed to be IBM's highest revenue software product.
In Mumps and similar hierarchically organized systems, the organization of the data can be viewed either as a tree with varying length paths from the root to an ultimate leaf node, or a multidimensional sparse matrix.

In Mumps, persistent data, that is, data that can be accessed after the program which created it terminates, is stored in global arrays. Global arrays are disk resident and are characterized by the following:

1. They are not declared or pre-dimensioned.
2. The indices of an array are specified as a comma separated list of numbers or strings.
3. Arrays are sparse. That is, if you create an element of an array, let us say element 10, it does not mean that Mumps has created any other elements. In other words, it does not imply that there exist elements 1 through 9. You must explicitly create these if you want them.
4. Array indices may be positive or negative numbers or character strings or a combination of both.
5. Arrays may have multiple dimensions limited by the maximum line length (nominally 512 characters but most implementations permit longer lengths).
6. Arrays may be viewed as either matrices or trees.
7. When viewed as trees, each successive index is part of the path description from the root to a node.
8. Data may be stored at any node along the path of a tree.
9. Not all nodes will have data. In fact, some trees may have no data stored.
10. Global array names are prefixed with the up-arrow character (^).

For example, consider an array reference of the form ^root("p2","m2","d2"). This could be interpreted to represent a cell in a three dimensional matrix named ^root indexed by the values ("p2","m2","d2") or, alternatively, it could be interpreted as a path from the origin (^root) to a final (although not necessarily terminal) node d2.

In either the array or tree interpretation, values may be stored not only at an end node, but also at intermediate nodes. That is, in the example above, data values may be stored at nodes ^root, ^root("p2"), ^root("p2","m2") as well as ^root("p2","m2","d2") or, at none at all.

Because Mumps arrays can have many dimensions (limited by implementation defined maximum line length), when viewed as trees, they can be of many levels of depth and these levels may differ in depth from one sub tree to another.

In Mumps, arrays can be accessed directly by means of a set of valid index values or by navigation of a global array tree primarily by means of the built-in functions $data() and $order(). The first of these, $data(), reports if a node exists, if it has data and if it has descendants. The second, $order(), is used to navigate from one sibling node to the next (or prior) at a given level of a tree.

In the example shown in Figure 18, each successive index added to the description leads to a new node in the tree. Some branches go deeper than others. Some nodes may have data stored at them, some have no data. The $data() function, described in detail below, can be used to determine if a node has data and if it has descendants.
In the example in Figure 18, only numeric indices were used to conserve space. In fact, however, the indices of global arrays are more often character strings.

In a global array tree, the order in which siblings appear in the tree is determined by the collating sequence, usually ASCII. That is, the index with the lowest overall collating sequence value is first branch at a given level of the tree, and the index with the highest value is last branch. The \$\text{order()}\$ function, described below, can be used to navigate from one sibling to the next at any given level of the tree. The tree from Figure 18 can be created with the code shown Figure 19.

```
1  set ^root(1,37)=1
2  set ^root(1,92,77)=2
3  set ^root(1,92,177)=3
4  set ^root(5)=4
5  set ^root(8,1)=5
6  set ^root(8,100)=6
7  set ^root(15)=7
8  set ^root(32,5)=8
9  set ^root(32,5,3)=9
10 set ^root(32,5,8)=10
11 set ^root(32,123)=11
```

In this construction example, note that several nodes exist but have no data stored. For example, the nodes ^root(1), ^root(8) and ^root(32) exist because they have descendants but they have no data stored at them. On the other hand, the node ^root(32,5) exists, has data and has descendants. Node ^root(32,123) has data but no descendants.

---

41 Mumps assignment statements require the keyword \text{set}.
5 Basic Indexing, Searching and Information Retrieval

5.1 Indexing Models

Information retrieval involves matching a user query with one or more documents in a database. Unlike a relational database system, the match is, in most cases, approximate. That is, some retrieved items will be more closely related to the query while others will be more distantly related. Results are usually presented from most relevant to least relevant with a cutoff beyond which documents are not shown. For example, the query *information retrieval* to Google resulted in nearly 14 million hits. However, only a sample of these, the most relevant, are actually displayed.

In its simplest form, an information retrieval system consists of a collection of documents and one or more procedures to calculate the similarity between queries and the documents.

[Figure 20 Overview of Indexing]

Determining the similarity between queries and documents is usually not done directly. Instead, the queries and documents are mapped into an internal representation or indexing model upon which the similarity functions can be calculated directly as seen in Figure 20. In this approach, queries and documents are mapped to an internal representation. The results of a query are determined by calculating the similarity between the query and documents in the context of the indexing model. Basically, queries become small documents. Similarity functions then calculate the similarities between documents.

In most cases, a user will interact with the system and refine his or her query in an attempt to hone in on a final set of answers. The system should provide aids in this process and learn from and adapt to the user's needs.

5.2 Indexing Vocabularies

Historically, indexing was conducted manually by experts in a subject who read each document and classified it according to content. Increasingly, manual indexing has
been overtaken by automated indexing, of the kind performed by search engines such as Bing, Google and other online indexing and information storage and retrieval systems.

In any indexing scheme, there is a distinction between a *controlled* and *uncontrolled* vocabulary scheme. A *controlled vocabulary* indexing scheme is one in which previously agreed upon standardized terms, categories and hierarchies are employed. On the other hand, an *uncontrolled vocabulary* based system is one that derives these from the text directly.

In a controlled vocabulary based system, subjects are described using the same preferred term each time and place they are indexed, thus ensuring uniformity across user populations and making it easier to find all information about a specific topic during a search. Many controlled vocabularies exist in many specific fields. These take the form of dictionaries, hierarchies, and thesauri which structure the content of the underlying discipline into commonly accepted categories. For the most part, these are constructed and maintained by government agencies (such as the National Library of Medicine in the U.S. or professional societies such as the ACM).

For example, the Association for Computing Machinery Computing Classification System (1998):

http://www.acm.org/about/class/1998/

is used to classify documents published in computing literature. This system is hierarchical. The author or reviewer of a document indexes the document under one or more categories to which the document most specifically applies and at the level in the tree that best corresponds to the generality of the document. For example, consider the extract of the ACM hierarchy shown in Figure 21. This is a controlled vocabulary.

Numerous other examples abound, especially in technical disciplines where nomenclature is precise. For example:

1. MeSH (Medical Subject Headings) for medicine and related fields
   

2. International Classification of Diseases - Clinical Modification version 9 (ICD9-CM) and related codes for diagnostic and forensic medicine:
   
   http://icd9cm.chrisendres.com/

3. National Library of Medicine Classification Schedule for medically related works
   

4. Mental disorders:
   
   http://www.psyweb.com/Mdisord/DSM_IV/dsm_iv.html
# D.4 OPERATING SYSTEMS (C)

* D.4.0 General

* D.4.1 Process Management
  o Concurrency
  o Deadlocks
  o Multiprocessing/multiprogramming/multitasking
  o Mutual exclusion
  o Scheduling
  o Synchronization
  o Threads NEW!

* D.4.2 Storage Management
  o Allocation/deallocation strategies
  o Distributed memories
  o Garbage collection NEW!
  o Main memory
  o Secondary storage
  o Segmentation [**]
  o Storage hierarchies
  o Swapping [**]
  o Virtual memory

* D.4.3 File Systems Management (E.5)
  o Access methods
  o Directory structures
  o Distributed file systems
  o File organization
  o Maintenance [**]

* D.4.4 Communications Management (C.2)
  o Buffering
  o Input/output
  o Message sending
  o Network communication
  o Terminal management [**]

* D.4.5 Reliability
  o Backup procedures
  o Checkpoint/restart
  o Fault-tolerance
  o Verification

* D.4.6 Security and Protection (K.6.5)
  o Access controls
  o Authentication
  o Cryptographic controls
  o Information flow controls
  o Invasive software (e.g., viruses, worms, Trojan horses)
  o Security kernels [**]
  o Verification [**]

Figure 21 ACM classification system

---

5. International Union for Pure and Applied Chemistry Names (IUPAC Names) for chemistry Diagnostic and Statistical Manual IV nomenclature:
In a manually indexed collection that uses a controlled vocabulary, experts trained in the field and the vocabulary read and assign vocabulary or hierarchy codes to the documents. Historically, because of the complexity of the terminology and the expense of conducting online searches, these systems were accessed mainly by trained personnel who intermediated a user's needs and translated them into the precise vocabulary of the discipline. Prior to the advent of the Internet, online database searching was expensive and time consuming. In recent years, however, with the advent of ubiquitous Internet access and vastly cheaper computer facilities, the end user is more likely to conduct a search directly.

Uncontrolled or derived vocabulary systems have been around for many years. These derive their terms directly from the text. Among the earliest forms were biblical concordances such as the King James Bible Hebrew and Greek Concordance Index:

http://www.sacrednamebible.com/kjvstrongs/CONINDEX.htm

This is an alphabetically organized index which references each occurrence of each term in the text. A more secular example can be found in the index for John Bartlett's Familiar Quotations, 10th ed. 1919:

http://www.bartleby.com/100/s0.html

Manual construction of concordances is tedious but well suited as a computer application. A computer based uncontrolled, also known as a derived vocabulary, can be constructed through statistical analysis of word usage in the collection as a whole. On the other hand, controlled vocabularies may also be used in computer based systems with the aid of training sets of documents.
In Chapter 14 on 139 we will examine indexing the OSUMEDLINE collection with the controlled MeSH vocabulary.
Figure 22 Overview of basic document indexing

Figure 22 attempts to graphically depict the overall flow of the document indexing process, the relationships between several of the methodologies to be discussed, and possible feedback loops to further improve and refine the indexing results.

For example, an initial pass through the system may result in term clusters, that is, collections of terms which frequently co-occur together. These might be regarded as semantically similar, perhaps even synonyms. These term clusters might be re-
introduced into the indexing scheme such that terms from the document might be replaced by a surrogate cluster token representing the entire class of terms to which the original term belonged. For example, the terms *antenna*, *receiver*, *yaggi* and *aerial* are sometimes used interchangeably and the similarity of usage might be detected by the term clustering technique. So, it might be better to reevaluate the documents and replace each instance of a cluster term with a token pointing to the cluster. Many other possibilities exist as well.

The main points in Figure 22 are:

1. The original corpus of documents is pre-processed into a format more easily manipulated by computer programs but with links to the original documents. Titles are usually stored in a separate file as these are normally the part of a document displayed during retrieval.

2. Documents are processed to remove common language words that convey no meaning from an indexing point of view.

3. Words in documents are reduced to semantic stems. In many cases, this involves prefix and suffix removal but in others it may involve a more complex semantic analysis. Words that are excessively short, long, or begin with numerics or special characters may also be removed at this time.

4. A dictionary is built giving a list of words remaining and their frequency of occurrence.

5. Based on word usage frequencies, documents are processed to remove very high and very low frequency terms as these are unlikely to be good indexing terms.

6. The dictionary is rebuilt to reflect the current vocabulary. Additionally, a document frequency dictionary is built giving the number of documents each word occurs in.

7. A *document-term matrix* is built where the row numbers are document numbers (arbitrarily assigned) and the columns are words. The value in a cell of the matrix gives the number of times a word appears in a given document. As each document contains only a small subset of the entire vocabulary, this is a very sparse (most cells have a value of zero) but potentially quite large matrix. For example, the OSU medical collection used in the examples has nearly 300,000 documents and a vocabulary, excluding stop words, of nearly 50,000 words. This implies a matrix of $15,000,000,000$ elements! In some systems, the document-term matrix also records the position in each document of each term occurrence.

8. A *term-document matrix* is calculated (the transpose of the document-term matrix) to provide inverted\(^{42}\) file access to the documents.

---

\(^{42}\) That is, access to the documents by word rather than by document number.
9. Various term weights are calculated including the \textit{inverse document frequency} weight and the \textit{discrimination coefficients}.

10. The document-term and term-document matrices are recalculated to reflect the term weights and the frequencies of occurrence of words in individual documents.

11. A \textit{term-term} matrix and a \textit{term-term proximity matrix} are calculated identifying terms whose usage and proximity are similar for possible synonym identification and query supplementation.

12. A \textit{document-document} matrix is built identifying documents whose word usage is similar.

13. A matrix of document clusters is built showing documents that are collectively related to one another. The centroid or average document vector from each cluster can be used as a surrogate for all documents in the cluster to improve retrieval speed.

14. Clusters of document clusters identified to provide a hierarchical view of the collection.

15. A matrix of term clusters is developed showing words whose usage is similar for possible use in augmenting queries and development of concept categories.

16. Documents are re-evaluated and original terms are replaced by concept tokens and the process re-iterates.
7 Identifying Good Indexing Terms

Information retrieval pioneer Hans Luhn [Luhn 1958] believed that the \textit{resolving power} of terms in a collection of text would be greatest in the middle-frequency range (see Figure 23). In this context, \textit{resolving power} is the ability of a term to differentiate between documents relevant and irrelevant to the query. Neither high frequency terms which are spread through many if not all documents nor low frequency terms whose usage is isolated to only a few documents, constitute good indexing terms.

Thus, a large part of the task ahead involves selection of terms, term surrogates, phrases, and synonyms that best reveal information content. But first we begin with elimination of words that are unlikely to be useful terms.

7.1 Stop Lists

All indexing techniques need to determine which words or combination of words are the better indexing terms and which terms are poor indications of content. However, in all languages, some words can be eliminated from further consideration immediately based on their frequency of occurrence.

Such a list of words is called a \textit{stop list}. A \textit{stop list} is a list of words which are not used for indexing (sometimes referred to as a \textit{null dictionary}).

For the most part, a stop list is composed of:

1. very high frequency terms conveying no real meaning for purposes document classification (words such as: \textit{the}, \textit{and}, \textit{was}, etc.) or,

2. very low frequency words that are one-of-a-kind and unlikely to be important in real world applications.

The file \textit{basic.stop.words} contains, one word per line, a list of about 700 such words.
Once a stop list has been constructed, there are two ways to use it in a program. One way is to read each stop word into a Mumps global array and then test each input text word to see if it is in the stop list global array.

In the GPL version of the Mumps interpreter, however, there are two added functions, $zStopInit()$ and $zStopLookup()$. These use memory resident C++ lookup containers as illustrated in Program 11. At the beginning of the program, $zStopInit()$ creates the container and reads the stop list words (one word per line) into it. The parameter to the function is the name of the file to read that contains the stop list words.

In the body of the program, $zStopLookup()$ returns true (1) if the argument string is in the container and false (0) otherwise. It is very fast.

Another special function, $zzInput()$ is used. It reads all the words on a line and places them in a Mumps local array. The name of the array is given as a parameter. The function returns the number of elements in the array. The program processes elements

---

**Figure 23 Best indexing terms**

-derived from Salton 1983-
of the array \(w(j)\) except for elements zero and one which contain, respectively, the byte offset and document number.

The program in Figure 11 reads each input line where each line is a document (a very long line in most cases) and writes each word to \textit{stdout} if it is \textit{not} in the stop list. After all the words in a line have been processed, a \textit{new line} character is written. The output resembles the input except that stop list words have been removed.

```
#!/usr/bin/mumps
# stop.mps
# Copyright 2014 Kevin C. O'Kane
# input has an extra blank at end so array count is 1 high
set %=$zStopInit("stop.words")
for do
  . set i=$zzInput("w")
  . if i=0 break
  . set k=i-2
  . write w(0)," ",w(1)
  . for j=2:1:k do
    . set w(j)=$zlower(w(j))
    . if '$zStopLookup(w(j)) write " ",w(j)
  . write !
```

Program 11 Stop list removal

### 7.2 Building an Enhanced Stop List

While some words are common to all stop lists (such as \textit{are}, \textit{is}, \textit{the}, \textit{etc}.), other words may be discipline specific.

For example, while the word \textit{computer} may be a significant content word in a collection of articles about biology, it is a common term conveying little content in a collection dealing with computer science. Consequently, it is necessary to individually examine the vocabulary of each collection to identify discipline specific words to include in the stop list in addition to the basic set of words common to all disciplines.

In the OSUMED collection, the total number of documents is 293,857 and the total vocabulary consists of about 120,000 words after (1) stemming\(^{43}\), (2) rejection of words less than three or longer than 25 characters in length, and (3) words beginning with numbers.

A small number of words have very high frequencies of occurrence compared to the remainder of the file. While at the low end of the frequency spectrum, there were about 72,000 words that occur 5 or fewer times (60% of the total number of words). Figure 24 gives a graph of overall word usage in the OHSUMED collection. The vertical axis gives the number of times the word occurs and the horizontal the word rank\(^{44}\). The most frequently occurring word occurred about 180,000 times. As rank increases, frequency of occurrence drops dramatically.

\(^{43}\) Removal of common suffixes.

\(^{44}\) The most frequent word is rank 1, the next, rank 2 and so forth.
If we eliminate words with total frequency of occurrence of 5 or less and greater than 40,000 (the top ranking 101 words), this eliminates about half the words and results in a candidate vocabulary of about 64,000 words.

On the other hand, in a Wikipedia data base the vocabulary is very large. In the 179 MB sample used, there were 402,347 distinct words after stemming, rejection of word whose length was less than three or greater than 25, and rejection of words beginning with digits.

Figure 25 gives the frequency and rank of the 75 most frequently occurring Wikipedia words. As can be seen, a very small number of words have very high frequencies of occurrence.

The script file `index.script` initially removes common stop list words and then reduces words to stems. Next, a second stop list is prepared. The second top list is based on the frequency of occurrence of stemmed words, not original words. Words than exceed or
fall below user designated thresholds are removed from the collection. The result, `stemmed.docs.tmp`, is the basis for most of the subsequent classification procedures.

With regard to terms of very low frequency and terms of very high frequency, rather than elimination, it may be possible to rescue these terms based upon further analysis.

For example, low frequency terms that are determined to be semantically related to other low frequency terms may, collectively, if merged into a semantic token and thus be considered a middle frequency term. For example, highly specific medical terms relating to essentially the same condition when taken together may have a frequency of occurrence which places them in the middle frequency range.

Similarly, high frequency terms which by themselves convey little information may, when considered as phrases, be quite specific terms of middle frequency. For example: `programming` and `language` which separately are not very important in a database of computer abstracts. However, taken together as a phrase, they are somewhat more specific.

![Figure 25 Frequency of top 75 Wikipedia words](image-url)
7.3 Basic Word Dictionary

A simple Mumps program to build a basic dictionary of terms using one of the versions of osu.converted is shown in Program 12. The first two words on each line are ignored. These are the byte offset and document number.

```
#!/usr/bin/mumps
# dictionary.mps February 14, 2014
# Copyright 2014 Kevin C. O’Kane

kill ^dict

for do
  . set i=$zzInput("w")
  . if i=0 break
  . set k=i-1
  for j=2:1:k do
    .. if $data(^dict(w(j))) set ^dict(w(j))=^dict(w(j))+1
    .. else set ^dict(w(j))=1
  end
  for word=$order(^dict(word)) write ^dict(word)," ",word,!
end
halt
```

Program 12 Dictionary construction from original text

At the end, program 12 writes the dictionary in alphabetical word order to stdout, one word per line. The frequency of occurrence is first followed by one blank followed by the word. A more useful format would be in reverse frequency order. That is, arrange the words according to their frequency of usage from most to least. The program to do this is shown in Program 13.

Program 13 also includes some speed enhancing improvements. While access to global arrays is fat, each fetch and store operation is actually a complex database operation involving a large btree. So, rather than increment elements of the global array $dict() as was done in Program 12, Program 13 writes each instance of each word to a temporary file named dictionary.unsorted. This file is then processed by Linux system functions sort and uniq with the line:

```
shell sort < dictionary.unsorted | uniq -c | sort -nr > dictionary.sorted
```

This line sorts the final dictionary.unsorted file and passes the output to uniq. The output consists of an alphabetically sorted list of words, one word per line. If a word, say computer occurred twenty times in the collection, there would be twenty consecutive lines with the word computer on them. The function uniq reads the output of sort and writes, for each unique word, the number of times the word occurred followed by blanks, followed the word, one word per line. This is then re-sorted numerically according to the count, in reverse order. That is, the first line in the output file, will have the highest count and so on. The parameter -c tells uniq to count and the parameter -nr tells sort to perform a numeric (rather than alphabetic) sort with the results presented in reverse order.

45 These include osu.converted (1) in its original form, (2) after stop list removal, or (3) after stemming. All have the same file format but differ in content.

46 The letter n tells sort to sort numerically; the letter r telles it to present the results in reverse order. Many Linux functions parameters can be specified in this manner.
After the file is sorted, it is read in and \texttt{\textasciitilde dict()} is initialized. The function \texttt{\$zblanks()} removes extra blanks that \texttt{uniq} inserts. On balance, this procedure is substantially faster that incrementing individual elements of \texttt{\textasciitilde dict()}. The \texttt{sort} and \texttt{uniq} functions are very fast especially on a machine with multiple computing cores as the steps in the line above will be scheduled on separate CPU cores thus resulting in parallel execution.

| 58760 of  | 4371 as  | 1998 study  | 528 pressure |
| 50093 the  | 4125 an  | 1943 which  | 1509 one     |
| 39368 and  | 4084 be  | 1937 group  | 1497 patient |
| 35125 in   | 3861 had | 1894 normal | 1488 significant |
| 20814 to   | 3689 not | 1874 no     | 1484 all     |
| 17939 with | 3582 after | 1846 increased | 1482 levels |
| 17396 a    | 3366 letter | 1814 blood  | 1460 p      |
| 14042 was  | 3187 In   | 1813 1      | 1425 has     |
| 12065 were | 3119 less | 1780 two    | 1419 compared |
| 11083 patients | 3067 are | 1770 cells | 1418 control |
| 10257 The  | 2822 during | 1731 more  | 1407 These   |
| 10078 for  | 2757 A    | 1726 We     | 1405 clinical |
| 8147 by    | 2647 this | 1721 significantly | 1398 years |
| 7502 that  | 2452 disease | 1685 results | 1398 cases |
| 6636 is    | 2341 but  | 1655 rate   | 1380 associated |
| 6147 or    | 2196 treatment | 1639 2      | 1364 10     |
| 5740 than  | 2182 between | 1625 both  | 1355 found   |
| 5701 from  | 2098 may  | 1618 This    | 1350 per     |
| 5198 on    | 2045 have | 1571 who     | 1349 activity |
| 4762 at    | 2037 these | 1548 been   | 1345 3       |

\begin{figure}[h]
\centering
\begin{tabular}{llll}
58760 of & 4371 as & 1998 study & 528 pressure \\
50093 the & 4125 an & 1943 which & 1509 one \\
39368 and & 4084 be & 1937 group & 1497 patient \\
35125 in & 3861 had & 1894 normal & 1488 significant \\
20814 to & 3689 not & 1874 no & 1484 all \\
17939 with & 3582 after & 1846 increased & 1482 levels \\
17396 a & 3366 letter & 1814 blood & 1460 p \\
14042 was & 3187 In & 1813 1 & 1425 has \\
12065 were & 3119 less & 1780 two & 1419 compared \\
11083 patients & 3067 are & 1770 cells & 1418 control \\
10257 The & 2822 during & 1731 more & 1407 These \\
10078 for & 2757 A & 1726 We & 1405 clinical \\
8147 by & 2647 this & 1721 significantly & 1398 years \\
7502 that & 2452 disease & 1685 results & 1398 cases \\
6636 is & 2341 but & 1655 rate & 1380 associated \\
6147 or & 2196 treatment & 1639 2 & 1364 10 \\
5740 than & 2182 between & 1625 both & 1355 found \\
5701 from & 2098 may & 1618 This & 1350 per \\
5198 on & 2045 have & 1571 who & 1349 activity \\
4762 at & 2037 these & 1548 been & 1345 3 \\
\end{tabular}
\caption{Dictionary sorted by frequency}
\end{figure}

The final result is \texttt{dictionary.sorted} consisting of lines containing (1) a count and, (2) a word. Some sample output is shown in Figure 26. As can be seen, the word \textit{of} was the most frequently occurring term with the word \textit{the} in second place. Having a list of the most and least frequently occurring terms is required for determining maximum and minimum word frequency cutoffs.

\begin{verbatim}
1 #!/usr/bin/mumps
2 # dictionary1.mps
3 # stdin input from osu.converted
4
5 open 1:"dictionary.unsorted,new"
6 for  do
7   . use 5
8   . set i=$zzInput("w")
9   . if i=0 break
10  . set k=i-1
11  . use 1
12  . for j=2:1:k write w(j),!
13 close 1
14
16 shell sort < dictionary.unsorted | uniq -c | sort -nr > dictionary.sorted
17
18 kill ^dict
19 open 1:"dictionary.sorted,old"
20 use 1
21 for  do
\end{verbatim}
7.4 Zipf's Law

Zipf's Law states that the frequency ordered rank of a term in a document collection times its frequency of occurrence is approximately equal to a constant:

Frequency * rank \sim constant

where Frequency is the total number of times some term \( W \) occurs. Rank is the position number of the term when the terms have been sorted by Frequency. That is, the most frequently occurring term is rank 1, the second most frequently occurring term is rank 2 and so forth. Program 14 calculates Zipf's constants. The input to Program 14 was a dictionary created from the original collection. Words were unstemmed and stop words not removed.

Some Zipf results for James Joyce's short story The Dead are given in Figure 27 and the OHSUMED collection in Figure 28. A divisor is used to bring the constants, which have no units and correspond to no actual measurements, into a more readable range. The Zipf constant for these files differs due to their lengths.

<table>
<thead>
<tr>
<th>rank</th>
<th>word</th>
<th>rank</th>
<th>word</th>
<th>rank</th>
<th>word</th>
<th>rank</th>
<th>word</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>the</td>
<td>21</td>
<td>miss</td>
<td>22</td>
<td>or</td>
<td>22</td>
<td>my</td>
</tr>
<tr>
<td>11</td>
<td>and</td>
<td>22</td>
<td>is</td>
<td>22</td>
<td>freddy</td>
<td>23</td>
<td>been</td>
</tr>
<tr>
<td>12</td>
<td>of</td>
<td>22</td>
<td>all</td>
<td>23</td>
<td>by</td>
<td>22</td>
<td>mrs</td>
</tr>
<tr>
<td>15</td>
<td>to</td>
<td>21</td>
<td>be</td>
<td>22</td>
<td>have</td>
<td>22</td>
<td>came</td>
</tr>
<tr>
<td>17</td>
<td>a</td>
<td>20</td>
<td>mary</td>
<td>22</td>
<td>me</td>
<td>21</td>
<td>still</td>
</tr>
<tr>
<td>17</td>
<td>he</td>
<td>20</td>
<td>then</td>
<td>22</td>
<td>if</td>
<td>20</td>
<td>young</td>
</tr>
<tr>
<td>18</td>
<td>in</td>
<td>20</td>
<td>when</td>
<td>23</td>
<td>face</td>
<td>21</td>
<td>will</td>
</tr>
<tr>
<td>20</td>
<td>was</td>
<td>20</td>
<td>would</td>
<td>22</td>
<td>are</td>
<td>21</td>
<td>went</td>
</tr>
<tr>
<td>23</td>
<td>his</td>
<td>20</td>
<td>so</td>
<td>22</td>
<td>voice</td>
<td>21</td>
<td>time</td>
</tr>
</tbody>
</table>
24 her 21 julia 22 about 21 some 21
21 said 21 jane 22 no 21 do 21 while
21 she 21 out 22 eyes 21 this 21 stood
23 had 22 browne 22 only 21 too 21
23 that 20 who 22 go 21 think 21
21 gabriel 21 which 21 this 21 stood 21
22 i 21 what 22 its 21 see 21
22 it 21 up 22 good 21 little 21
23 with 22 asked 22 back 22 how 22
24 for 20 them 22 an 22 himself 22
24 on 21 their 22 ivors 22 again 22
24 him 21 one 22 could 21 your 21
24 at 22 into 22 come 21 upon 21
23 aunt 21 there 22 over 21 two 21
24 as 22 malins 22 know 22 table 22
23 you 22 well 22 darcy 22 snow 22
21 mr 21 like 22 after 22 man 22
21 but 22 down 22 where 22 long 22
21 not 22 did 22 room 22 hand 22
21 kate 22 o 22 never 22 before 22
21 were 22 now 23 ladies 21 wife 21
21 from 21 we 23 gretta 22 why 22
21 they 22 very 22 old 22 three 22

Figure 27 Zipf constants - The Dead

| 369 of | 403 during | 393 new | 447 treated |
| 425 in | 414 comment | 398 protein | 448 not |
| 613 the | 426 has | 403 using | 450 cases |
| 733 and | 423 comments | 403 use | 454 immunodeficiency |
| 502 a | 436 see | 405 two | 458 heart |
| 555 with | 428 or | 411 syndrome | 460 myocardial |
| 561 to | 437 effects | 417 therapy | 463 cardiac |
| 514 the | 428 in | 422 report | 465 acid |
| 529 for | 425 disease | 426 virus | 469 primary |
| 426 patients | 416 clinical | 430 growth | 473 following |
| 429 on | 399 blood | 434 children | 472 diagnosis |
| 462 by | 408 at | 439 role | 472 receptor |
| 498 is | 410 cells | 433 determine | 475 analysis |
| 432 letter | 408 effect | 433 rat | 476 management |
| 417 was | 417 acute | 422 editorial | 476 gene |
| 407 from | 423 to | 420 pulmonary | 479 carcinoma |
| 425 were | 408 associated | 423 cancer | 479 may |
| 437 an | 397 be | 425 artery | 478 we |
| 414 human | 400 case | 426 an | 482 care |
| 365 study | 400 studied | 430 ventricular | 484 patients |
| 379 we | 396 between | 435 disease | 481 liver |
| 396 that | 394 renal | 437 response | 484 [news] |
| 393 after | 389 chronic | 439 activity | 486 can |
| 386 as | 383 factor | 439 normal | 490 effect |
| 394 are | 380 patient | 435 infection | 493 bone |
| 396 have | 385 who | 437 function | 497 type |
| 370 cell | 391 this | 442 which | 497 its |
| 381 treatment | 383 coronary | 441 studies | 494 had |
| 393 been | 389 used | 444 this |

Figure 28 Zipf constants - OHSUMED
7.5 KWIC/KWOC Indices

In the early days of information retrieval, and still to this day, many indexing schemes have used several simple techniques, based mainly on titles, known as Key Word In Context (KWIC), Key Word Out of Context (KWOC) or Key Word Alongside Context (KWAC) to organize and index content.

Program 15 shows a KWIC program (this may be enabled in index.script) and the output is shown in Figure 29. Similarly, Program 16 shows a KWOC program and Figure 30 shows its output.

```
#!/usr/bin/mumps
# kwic.mps Copyright 2014 Kevin C. O'Kane

set %=$zStopInit("stop.words")

for do
  . set i=$zzinput("w")
  . if i=0 break
  . set i=i-1
  . set doc=w(0)
  . for j=1:1:i do
    .. if $l(w(j))<3 quit
    .. if $l(w(j))>24 quit
    .. if w(j)?1n.e quit
    .. set tw=$zlower(w(j))
    .. if $zStopLookup(tw) quit
    .. write tw,?26,doc,": "
    .. set x=""
    .. for k=j+1:1:i set x=x_w(k)" "
    .. set y=""
    .. for k=j-1:-1:1 set y=w(k)" "_ quit:$l(y)>25
    .. write ?(80-$l(y)),y,?80,w(j)," ",x,!
```

Program 15 KWIC index

```
269: serotonin S2 and thromboxane A2 prostaglandin H2 receptor activation activation
976: leukocyte scanning of the abdomen Analysis of its value for diagnosis and management
118: Nonspecific acute abdominal pain letter
426: Eruptive abdominal pain Chylomicronemia
482: Abdominal and pelvic pain
562: Recurrent abdominal pain as the sole manifestation of hereditary angioedema
779: Anatomy of the abductor muscles of the hip as studied by computed tomography
563: apolipoprotein B48 in two cases of abetalipoproteinemia
524: function and verbal learning ability in patients with complex partial seizures of tempo
```

Figure 29 KWIC output

```
#!/usr/bin/mumps
# kwoc.mps Copyright 2014 Kevin C. O'Kane

set %=$zStopInit("stop.words")

for do
  . set i=$zzinput("w")
  . if i=0 break
  . set doc=w(0)
  . set i=i-1
  . for j=1:1:i do
    .. if $l(w(j))<3 quit
    .. if $l(w(j))>24 quit
```

Figure 30 KWOC output
Many journals publish annual KWIC/KWOC indices to help searchers locate articles. These, as shown above, are normally based solely on titles. Unfortunately, not all titles are suitable candidates for indexing as this link from Amazon.com clearly and humorously indicates:

http://www.amazon.com/Books-odd-misleading-titles/lm/1TGJCC3FZ48QY

### 7.6 Reducing Words to Stems

A step normally done in most systems, as noted above, involves reducing words to root stems. That is, finding a common semantic root term and replacing all the variants of it with that single root term. For example, *computer, computing, computers computation, computed, computational* and so forth all have *comput* as a common root.

Sometimes stemming can as simple as converting plural forms to singular, while in other cases it can involve prefix and suffix removal along with other modifications. Sometime spelling must be taken into account such a *center* and *centre, favor* and *favour* and so forth. Sometimes, several variant words without a common spelling root may be combined (such as aerial and antenna).

There are a number of methods to reduce words to their root stems and the process is sometimes error prone. Mumps has a basic builtin function known as $\text{	extasciitilde}z\text{stem()}$\textsuperscript{47}. It returns the root of a word by means of a table driven technique that seeks to remove common prefixes and suffixes. A more thorough method, however, would involve the more extensive semantic vocabulary studies such as those found in WordNet shown below (Chapter 17 on page 162).

The Mumps stem filter program used in these exercises is shown in Program 17. The program builds a global array named $^\text{stem()}$ which gives for each word stem the word or words that reduced to it. This is used in later programs to reconstruct the original terms.

\[47\text{ A better word stem program is needed. The current version has issues.}\]
The output consists of two files. The first, to stdout, resembles the input file except words have been reduced to their stems. The other file, stem-words.tmp, is a list of every stemmed word occurrence of three or more characters in length and not beginning with a digit, one word per line. As before above, this is used to build a stemmed vocabulary showing the stemmed word and the the number of times it occurs in the collection. As above, this is done by means of the sort and uniq programs but in this case, at the script level (see index.script):

```
head -$MAXDOCS osu.converted | stop.mps | stems.mps > initial-stemmed-docs.tmp
```

The program head sends $MAXDOCS lines (one document per line) of osu.converted to stop.mps which removes common stop words. The output of stop.mps is sent to stems.mps for conversion of the remaining words to stems. The final result is written to stemmed-docs.tmp.

On a subsequent line:

```
sort < stem-words.tmp | uniq -c > stemmed-vocabulary.tmp
```

The file is sorted alphabetically and the result passed to uniq which counts the number of duplicate occurrences and writes out the counts and the words.

---

48 Note that using multiple Mumps programs in a common pipe may cause problems if you are using the single user database and if both Mumps procedures attempt to access the global array database in read-write mode. Only one will gain access and the other will wait and result in a deadlock. There is no problem in this example because neither of the Mumps programs accesses the global array database.

49 This line could have been a continuation of the line above. Both lines, separately or combined, invite significant parallel processing.
Recalculation of dictionaries may be appropriate at this point as word frequencies may change significantly once the terms have been reduced to stems.

### 7.6.1 Words Stems

Program 17 creates the global array ^stem which gives stems and the words that reduced to the stems. The first index of the two dimensional array is a word stem and the second is a source text word that reduced to the stem. For example, both *computing* and *computed* reduce to the stem *compute*. In the global array ^stem, *compute* would be the first index and *computed* and *computing* would be second level indices.

After all the words have been processed, a file containing the word stems and their associated terms is printed to file *stemList.txt* as shown in Program 18. Each line in *stemList.txt* begins with a word stem for llowed by the root words separated from one another by a blank.
Later in the execution of the script file `index.script`, after the database is re-initialized, the stems are read back in by `loadStems.mps` which is shown in Program

```plaintext
#!/usr/bin/mumps
# loadStems.mps Dec 23, 2014
# convert data base to word stems
# Copyright 2014 Kevin C. O'Kane

    set c=0

    for do
      . read a
      . if '$test break
      . set w=$piece(a," ",1)
      . set "stem(w)=""
      . set c=c+1
      . for i=2:1 do
        . set x=$piece(a," ",i)
        . if x="" break
        . set ^stem(w,x)=""

    write c,!
```

Program 31 `loadStems.mps`
8 Vector Space Model

8.1 Overview

One popular approach to automatic document indexing, the vector space model, views computer generated document vectors as describing a hyperspace in which the number of dimensions (axes) is equal to the number of indexing terms. This approach was originally proposed by G. Salton [Salton 1968, 1971, 1983, 1992].

Each document vector is a point in that space defined by the distance along the axis associated with each document term proportional to the term's importance or significance in the document being represented. Queries are also portrayed as vectors that define points in the document hyperspace. Documents whose points in the hyperspace lie within an adjustable envelope of distance from the query vector point are retrieved. The information storage and retrieval process involves converting user typed queries to query vectors and correlating these with document vectors in order to select and rank documents for presentation to the user.

Documents are viewed as points in a hyperspace whose axes are the terms used in the document vectors. The location of a document in the space is determined by the degree to which the terms are present in a document. Some terms occur several times while other occur not at all. Terms also have weights associated with their content indicating strength and this is factored into the equation as well.

![Figure 32 Vector space model](image)

Figure 32 Vector space model
Queries are converted to Query vectors and also treated as points in the hyperspace and the documents that lie within a set distance of the query are determined to satisfy the query.

Figure 33 Vector space queries

Figure 34 Vector space clustering
Another technique involved recognizing when documents are similar to one another through clustering. Clustering involves identifying groupings of documents and constructing a cluster centroid vector to speed information storage and retrieval. Hierarchies of clusters can also be constructed.

![Figure 35 Calculating vector space similarities](image)

### 8.2 Basic Similarity Functions

There are several popular formulae to calculate the distance between points in the hyperspace. One of the better known is the Cosine function illustrated in the figure above (from Salton 1983). In this formula, the Cosine between points is used to measure the distance. Some of the common formulae are:

\[
\text{Sim}_1(Doc_i, Doc_j) = \frac{2 \left[ \sum_{k=1}^{t} (\text{Term}_{ik} \cdot \text{Term}_{jk}) \right]}{\sum_{k=1}^{t} \text{Term}_{ik} + \sum_{k=1}^{t} \text{Term}_{jk}}
\]

\[
\text{Sim}_2(Doc_i, Doc_j) = \frac{\sum_{k=1}^{t} \text{Term}_{ik} \cdot \text{Term}_{jk}}{\sum_{k=1}^{t} \text{Term}_{ik} + \sum_{k=1}^{t} \text{Term}_{jk} - \sum_{k=1}^{t} (\text{Term}_{ik} \cdot \text{Term}_{jk})}
\]
Sim$_3$(Doc$_i$, Doc$_j$) = \[
\frac{\sum_{k=1}^{t} (\text{Term}_{ik} \cdot \text{Term}_{jk})}{\sqrt{\sum_{k=1}^{t} (\text{Term}_{ik})^2 \cdot \sum_{k=1}^{t} (\text{Term}_{jk})^2}}
\]

Sim$_4$(Doc$_i$, Doc$_j$) = \[
\frac{\sum_{k=1}^{t} (\text{Term}_{ik} \cdot \text{Term}_{jk})}{\min(\sum_{k=1}^{t} \text{Term}_{ik}, \sum_{k=1}^{t} \text{Term}_{jk})}
\]

Sim$_5$(Doc$_i$, Doc$_j$) = \[
\frac{\sum_{k=1}^{t} \min(\text{Term}_{ik}, \text{Term}_{jk})}{\sum_{k=1}^{t} \text{Term}_{ik}}
\]

Figure 36 Similarity functions

In the above from [Salton 1983], the Cosine is formula 3. These formulae calculate the similarity between Doc$_i$ and Doc$_j$ by examining the relationships between \text{term}_{i,k} and \text{term}_{j,k} where \text{term}_{i,k} is the weight of term k in document i and \text{term}_{j,k} is the weight of term k in document j. Sim$_1$ is known as the Dice coefficient and Sim$_2$ is known as the Jaccard coefficient (see: Jaccard 1912, "The distribution of the flora of the alpine zone", New Phytologist 11:37-50).

The example in Figure 37 illustrate the application of the above\textsuperscript{50}.

\[
\begin{align*}
\text{Doc}_i &= (3,2,1,0,0,0,1,1) \\
\text{Doc}_j &= (1,1,1,0,0,1,0,0) \\
\text{Sim}_1(\text{Doc}_i, \text{Doc}_j) &= (2*6)/(8+4) \rightarrow 1 \\
\text{Sim}_2(\text{Doc}_i, \text{Doc}_j) &= (6)/(8+4-6) \rightarrow 1 \\
\text{Sim}_3(\text{Doc}_i, \text{Doc}_j) &= (6)/\sqrt{16*4} \rightarrow 0.75 \\
\text{Sim}_4(\text{Doc}_i, \text{Doc}_j) &= 6/4 \rightarrow 1.5 \\
\text{Sim}_5(\text{Doc}_i, \text{Doc}_j) &= 3/8 \rightarrow 0.375
\end{align*}
\]

Figure 37 Example similarity coefficient calculations

8.3 Implementing the Vector Space Model in Mumps

\textsuperscript{50} Salton 1983, pg 202-203.
Most IS&R systems have been implemented in C, Pascal and C++, although these languages provide little native support for the hyperspace model. Similarly, popular off-the-shelf legacy relational data base systems are inadequate to efficiently represent or manipulate sparse document vectors in the manner needed to effectively implement ISR systems.

While the global arrays were originally seen as a way of mapping an essentially hierarchical patient record to a database, they can also be viewed as string indexed sparse multidimensional arrays.

As many of the vector space model data structures make extensive use of one, two and three dimensional string indexed arrays, the Mumps global array facility along with the many built-in string handling functions make it a useful choice for quickly implementing many algorithms.

### 8.4 Vector Space Global Arrays

The following is a list of the main global arrays used to implement the vector space model:

1. $^\text{dict}(\text{word})$

   A vector indexed by a vocabulary word giving the number of times the word occurs in the collection.

2. $^\text{df}(\text{word})$

   A vector indexed by a vocabulary word giving the number of documents in the collection the word appears in.

3. $^\text{idf}(\text{word})$

   A vector indexed by a vocabulary word giving the inverse document frequency weight of the word in the collections as a whole.

4. $^\text{mca}(\text{word})$

   A vector indexed by a vocabulary word giving the discrimination coefficient of the word in the collection as a whole.

5. $^\text{doc}(d,\text{word},pos)$

   A matrix known as the document-term matrix giving at the $^\text{doc}(d,\text{word})$ level nodes where $d$ is a document number and $\text{word}$ is a word appearing in document $d$ either (1) the number of times the word occurs in the document or, (2) the weight of the word in the document. Initially, the values for this matrix are word counts. In later steps, these are converted to weights.

   The third index, $pos$, gives the position(s) of each word occurrence relative to the start of the document. No value is stored at the third level. The third index
is the relevant data.

6. \( ^\text{index}(\text{word}, d) \)

The transpose of \( ^\text{doc}(d, \text{word}) \) portion of \( ^\text{doc}(d, \text{word}, \text{pos}) \). The position indicator is not stored in this matrix.

7. \( ^\text{dd}(d1, d2) \)

The \textit{document-document} matrix giving a value indicating the similarity between \( d1 \) and \( d2 \).

8. \( ^\text{tt}(w1, w2) \)

The \textit{term-term} correlation matrix giving the an indicator of similarity or usage between word \( w1 \) and word \( w2 \).

9. \( ^\text{title}(d) \)

The text title of document \( d \) (possibly truncated).

10. \( ^\text{ttp}(w1, w2, 1) \)

The proximity term-term correlation matrix where \( ^\text{ttp}(w1, w2) \) is the sum of the absolute values of the term proximities between word \( w1 \) and \( w2 \) and \( ^\text{ttp}(w1, w2, 1) \) contains the signed sum of the proximities. The proximity between two words is the difference between their document relative position numbers as stored in the third index of \( ^\text{doc}(d, \text{w1}, \text{pos}) \). A high sum of the absolute values means that the words appear closely often. The sum of the values indicates the preferred order of occurrence (which of the words in normally first).

**8.5 Result Files**

Figure 38 is a list of the output files created by \textit{index.script} and a description of each:

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deleted-words-and-docs-for-idf.txt</td>
<td>An alphabetic list of words and their IDF weights that were deleted as a result of falling below the minimum IDF cutoff. As a result of word deletion, some documents may have lost too many words and were also deleted. These are listed as well.</td>
</tr>
<tr>
<td>deleted-words-for-frequency.txt</td>
<td>A list of words that were deleted because their frequency of occurrence was too high or too low.</td>
</tr>
<tr>
<td>discrimination-coefficients.txt</td>
<td>A list of words and their discrimination coefficients sorted by discrimination coefficient.</td>
</tr>
<tr>
<td>document-clusters-by-title.txt</td>
<td>A list of documents titles and the document clusters to which they belong.</td>
</tr>
<tr>
<td>File Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>document-clusters.txt</td>
<td>A list of document clusters and the document numbers and titles of the documents in the clusters.</td>
</tr>
<tr>
<td>document-document-matrix.txt</td>
<td>A list of document numbers and the cosine of similarities between them and those documents whose similarities exceed a threshold. A printout of the $^\text{dd}(d_1,d_2)$ global array.</td>
</tr>
<tr>
<td>document-term-matrix-weighted.txt</td>
<td>A list of documents and, for each, a list of the words occurring in the document and the number of occurrences of each word.</td>
</tr>
<tr>
<td>idf-weights.sorted.txt</td>
<td>A list of words and their Inverse Document Frequency weights sorted by IDF weight from high to low.</td>
</tr>
<tr>
<td>term-clusters.txt</td>
<td>A list of word clusters.</td>
</tr>
<tr>
<td>term-frequencies.txt</td>
<td>A list of words along with the number of documents each occurs in and the total number of occurrences of each.</td>
</tr>
<tr>
<td>term-term-cohesion-similarities.txt</td>
<td>A list or terms and those other related terms whose cohesion score exceeds a threshold.</td>
</tr>
<tr>
<td>term-term-jaccard-similarities.sorted.txt</td>
<td>A list of terms and those other related terms whose jaccard similarity exceeds a threshold.</td>
</tr>
<tr>
<td>term-term-matrix.sorted.txt</td>
<td>A list of terms and those other related terms whose similarity exceeds a threshold</td>
</tr>
<tr>
<td>term-term-proximity-similarities.txt</td>
<td>A list of terms and those other terms whose proximity similarity score exceeds a threshold.</td>
</tr>
<tr>
<td>term-term-term.txt</td>
<td>A list of second order term connections.</td>
</tr>
</tbody>
</table>

Figure 38 Output Files

8.6 Run Time Settings and Defaults

Figure 39 lists the environment variables internal to index.script that control thresholds used by various programs.

<table>
<thead>
<tr>
<th>Command line argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(MAXDOCS)</td>
<td>Number of documents to process.</td>
</tr>
<tr>
<td>MIN_DIVISOR=500</td>
<td>Word must occur once per 500 docs.</td>
</tr>
<tr>
<td>MIN_WORD_FREQ=$\left(\frac{\text{MAXDOCS}}{\text{MIN_DIVISOR}}\right)$</td>
<td>Words deleted if less than this number.</td>
</tr>
<tr>
<td>MAXFRACTION=$\left(\frac{100}{\text{MAX_DIVISOR}}\right)$</td>
<td>Words deleted if they appear in more than this percentage of documents (on average).</td>
</tr>
<tr>
<td>MIN_DOC_VECTOR_WORDS=3</td>
<td>Kill docs with fewer this number of words</td>
</tr>
<tr>
<td>MINIDF=4.5</td>
<td>Kill words with IDF value below this number</td>
</tr>
<tr>
<td>DDWGT=5</td>
<td>Documents must have this number of words in common to be included in document-document matrix</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MINTT=<code>mintt.mps $MAXDOCS</code></td>
<td>Terms must have this number of documents in common to be included in term-term matrix</td>
</tr>
<tr>
<td>JMIN=0.001</td>
<td>Discard Jaccard similarities less than this</td>
</tr>
<tr>
<td>MINDD=0.7</td>
<td>Minimum document-document cosine to be included in a document cluster-document matrix</td>
</tr>
<tr>
<td>HYPMIN=.4</td>
<td>Minimum cluster centroid cosine similarity for hyperclusters</td>
</tr>
<tr>
<td>HYPWGT=.4</td>
<td>Cosmological constant.</td>
</tr>
<tr>
<td>TTPMIN=$MINTT</td>
<td>Minimum proximity score to be included in term-term matrix</td>
</tr>
</tbody>
</table>

Figure 39 Run-time settings
9 Document-Term Matrix

9.1 Building a Document-Term Matrix

The basic data structure of the Vector Space model is the document-term matrix. That is, a matrix each of whose rows represents a document and each of whose columns represents a word or term in the vocabulary. The elements of the matrix are either the raw number of times a word occurs in a document or a weighted word occurrence value. The example in Program 19 builds a basic word-count document-term matrix from the osu.converted file (see Figure 14 on page 29). The program constructs the following disk resident global arrays:

1. the document-term matrix \( ^{\text{doc}}(\text{docNbr},\text{term},i) \) giving the number of times the term \( \text{term} \) occurs in document \( \text{docNbr} \) and the location of the term \( i \) relative to the start of the document.

2. a dictionary vector \( ^{\text{dict}}(\text{term}) \) giving the number of times term \( \text{word} \) occurs in the total collection.

3. \( ^{\text{index}}(\text{term},\text{docNbr}) \) giving, for each term, the \( \text{docNbrs} \) containing the term and the number of times the term occurs in \( \text{docNbr} \). \( ^{\text{index}}(\text{term},\text{docNbr}) \) is the transpose of \( ^{\text{doc}}(\text{docNbr},\text{term}) \). It does not, however, contain an element giving the relative position of the term in the document. Elements in the \( ^{\text{index}}(\text{term},\text{docNbr}) \) matrix at this point consist only of counts. These counts are used in \textit{discrim.mps} to calculate discrimination coefficients. \( ^{\text{index}}(\text{term},\text{docNbr}) \) will be recalculated by \textit{weight.mps} in which case the counts will become weights.

4. a document frequency vector \( ^{\text{df}}(\text{word}) \) giving the number of documents term \( \text{word} \) occurs in the collection.

5. a one element vector \( ^{\text{DocCount}}(1) \) giving the total number of documents in the collection.

1 #!/usr/bin/mumps
2 # DocumentTermMatrix.mps Feb 14, 2014
3 # Copyright 2014 Kevin C. O'Kane
4
5   kill `df
6   kill `dict
7   kill `doc
8
9 if `$data(1)` write "Missing parameter",! halt
10
11   set wmin=$1
12
13   if wmin="" set wmin=2
14
15   set d=0
16
17   open 1:"dict.tmp,new"
18   open 2:"doc.tmp,new"
for do
. use 5 set i=zzInput("w")
. if i=0 break
. set off=w(0),doc=w(1),k=i-1,d=d+1,`doc(doc)=off
. for j=2:1:k do
.. set `doc(doc,w(j),j)="
.. use 2 write doc," ",w(j),!
.. use 1 write w(j),!
use 5
open 1:"dict.sorted.tmp,old"
use 1
for do
. read a
. if 'test break
. set a=$zblanks(a)
. set `dict($piece(a," ",2))=$piece(a," ",1)
use 5
shell sort < dict.tmp | uniq -c > dict.sorted.tmp
open 1:"df.tmp,new"
use 1
for d=$order(`doc(d)) do
. for w=$order(`doc(d,w)) do
.. write w,!
close 1

shell sort < df.tmp | uniq -c > df.sorted.tmp
open 1:"doc.sorted.tmp,old"
use 1
for do
. read a
. if 'test break
. set a=$zblanks(a)
. set c=$p(a," ",1)
. set a2=$piece(a," ",2),a3=$piece(a," ",3)
. set `doc(a2,a3)=c
. set `index(a3,a2)=c
use 5
close 1

shell sort < doc.tmp | uniq -c > doc.sorted.tmp
open 1:"df.sorted.tmp,old"
use 1
for do
. read a
. if 'test break
. set a=$zblanks(a)
. set c=$p(a," ",1)
. set a2=$piece(a," ",2),a3=$piece(a," ",3)
. set `doc(a2,a3)=c
. set `index(a3,a2)=c
use 5
close 1

shell sort < df.tmp | uniq -c > df.sorted.tmp
open 1:"df.sorted.tmp,old"
use 1
for do
. read a
. if 'test break
. set a=$zblanks(a)
In the example in Program 19, the *stemmed.txt* file is read. Each line begins with the offset in the original OSUMED file followed by the document number. The program checks if *word* is already in the row for document *doc* and increments the count if it is, instantiates it to 1 otherwise. Similarly, the count of the total number of times the word has occurred in the collection is stored in $^\text{dict}(\text{word})$.

After the input file has been processed, the total number of times each word has occurred is examined. Words that are too few or many are deleted and their references in the $^\text{doc}()$ matrix are also deleted.

The total number of documents is stored in $^\text{DocCount}(1)$ (line 28) for future reference.

Next, the number of documents each word occurs in is calculated and stored in the vector $^\text{df}(\text{word})$. Then the results are written to three output files.
9.2 Normalized Word Frequencies

In the examples used here, the size of abstracts, with some exceptions, are approximately the same. That is, they have about the same number of words.

If, however, they differed significantly in length, it would be necessary to normalize the frequencies of word occurrence.

For example, if document 1 has 1,000 words and word A occurs 10 times and document 2 has 100 words and word A also occurs 10 times, the frequency of occurrence of word A needs to be adjusted so that it reflects its relative importance in both documents.

The simplest way to do this is to determine the average word length in documents in the collection and multiply, as appropriate, the frequencies of actual word occurrences accordingly.

This, from the example above, if the average document contains 500 words, the frequency for word A in document 1 should become 5 and the frequency for word A in document 2 should become 50.

9.3 Assigning Word Weights

Words used for indexing vary in their ability to indicate content and, thus, their importance as indexing terms. Some words, such as the, and, was and so forth are worthless as content indications and we eliminated them from consideration immediately. Other words occur so infrequently that they are also unlikely to be useful as indexing terms. These were also eliminated. Other words, however, with a middle frequency of occurrence, are candidates to be indexing terms.

However, not all words are equally good indexing terms. For example, the word computer in a collection of computer science articles conveys very little information useful to indexing the documents since so many, if not all, the documents contain the word. The goal is to determine a metric indicating the ability of a word to convey information.

In the example in Figure 40, several weighting schemes are compared. In Figure 40:

1. \(^{doc}(i,w)\) is the number of times term \(w\) occurs in document \(i\);
2. \(^{dict}(w)\) is the number of times term \(w\) occurs in the collection as a whole;
3. \(^{df}(w)\) is the number of documents term \(w\) occurs in;
4. \(NbrDocs\) is the total number of documents in the collection; and
5. Function \(zlog()\) is the natural logarithm;
6. The operation \(\div\) is integer division.

51 Median is another possibility.
7. Wgt1 is the number of times a word occurs in a document versus the number of time it occurs in a typical document. For example, if a word occurs twice as many times in a document than it typically does, Wgt1 will be 2.

8. Wgt2 uses what is known as the Inverse Document Frequency Weight (IDF) (discussed below). It is a measure of how widely distributed a word is in the collection. The IDF value of the word in the collection is multiplied by the number of times the word occurs in a document for the final weight of the term in a particular document.

9. Wgt3 combines Wgt1 and Wgt2. Words that occur more often in an individual document than they do in a typical document and are not widely distributed are given higher scores than words that are widely distributed and occur less frequently in a particular document than in the collection as a whole.

10. The MCA weight is the Modified Centroid Algorithm calculation method to calculate the Term Discrimination weight and will be discussed separately below.

In Figure 40 the document vectors for 20 documents (out of 1000) from computer science trade publications of the mid-80s are shown. Several term weighting schemes are shown.

While some knowledge of the state of the literature in the mid-80s would be useful, it should be clear that the word *computer* is not a good indexing term in a collection of articles about computers while a word such as *database* (databases were new then) is highly ranked.

<table>
<thead>
<tr>
<th>Word</th>
<th>^doc(i,w)</th>
<th>^dict(w)</th>
<th>^df(w)</th>
<th>Wgt1</th>
<th>Wgt2</th>
<th>Wgt3</th>
<th>MCA</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] Death of a cult. (Apple Computer needs to alter its strategy) (column)</td>
<td>4</td>
<td>261</td>
<td>112</td>
<td>1.716</td>
<td>9.757</td>
<td>17</td>
<td>-1.1625</td>
</tr>
<tr>
<td>apple</td>
<td>4</td>
<td>706</td>
<td>358</td>
<td>2.028</td>
<td>5.109</td>
<td>10</td>
<td>-19.4405</td>
</tr>
<tr>
<td>computer</td>
<td>2</td>
<td>146</td>
<td>71</td>
<td>0.973</td>
<td>6.290</td>
<td>6</td>
<td>-0.0256</td>
</tr>
<tr>
<td>mac</td>
<td>4</td>
<td>210</td>
<td>107</td>
<td>2.038</td>
<td>9.940</td>
<td>20</td>
<td>-0.5855</td>
</tr>
<tr>
<td>macintosh</td>
<td>2</td>
<td>79</td>
<td>67</td>
<td>1.696</td>
<td>6.406</td>
<td>11</td>
<td>-0.0592</td>
</tr>
<tr>
<td>strategy</td>
<td>3</td>
<td>114</td>
<td>68</td>
<td>1.789</td>
<td>9.065</td>
<td>16</td>
<td>0.0054</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Word</th>
<th>Num</th>
<th>City</th>
<th>State</th>
<th>Value1</th>
<th>Value2</th>
<th>Value3</th>
<th>Value4</th>
<th>Value5</th>
</tr>
</thead>
<tbody>
<tr>
<td>edit</td>
<td>2</td>
<td>111</td>
<td>77</td>
<td>1.387</td>
<td>6.128</td>
<td>8</td>
<td>-0.0961</td>
<td></td>
</tr>
<tr>
<td>frame</td>
<td>2</td>
<td>9</td>
<td>7</td>
<td>1.556</td>
<td>10.924</td>
<td>17</td>
<td>0.0131</td>
<td></td>
</tr>
<tr>
<td>import</td>
<td>2</td>
<td>29</td>
<td>19</td>
<td>1.310</td>
<td>8.927</td>
<td>12</td>
<td>0.0998</td>
<td></td>
</tr>
<tr>
<td>macintosh</td>
<td>3</td>
<td>210</td>
<td>107</td>
<td>1.529</td>
<td>7.705</td>
<td>12</td>
<td>-0.5855</td>
<td></td>
</tr>
<tr>
<td>macro</td>
<td>3</td>
<td>38</td>
<td>24</td>
<td>1.895</td>
<td>12.189</td>
<td>23</td>
<td>0.1075</td>
<td></td>
</tr>
<tr>
<td>outstand</td>
<td>1</td>
<td>100</td>
<td>9</td>
<td>0.900</td>
<td>5.711</td>
<td>5</td>
<td>0.0168</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td>4</td>
<td>861</td>
<td>435</td>
<td>2.021</td>
<td>4.330</td>
<td>9</td>
<td>-26.8094</td>
<td></td>
</tr>
<tr>
<td>wordperfect</td>
<td>8</td>
<td>24</td>
<td>8</td>
<td>2.667</td>
<td>39.627</td>
<td>106</td>
<td>0.1747</td>
<td></td>
</tr>
</tbody>
</table>

[4] Radius Pivot for Built-In Video an Radius Color Pivot. (Hardware Review) (new Mac monitors)(includes related article on design of

<table>
<thead>
<tr>
<th>Word</th>
<th>Num</th>
<th>City</th>
<th>State</th>
<th>Value1</th>
<th>Value2</th>
<th>Value3</th>
<th>Value4</th>
<th>Value5</th>
</tr>
</thead>
<tbody>
<tr>
<td>built-in</td>
<td>3</td>
<td>35</td>
<td>29</td>
<td>2.486</td>
<td>11.621</td>
<td>29</td>
<td>0.0678</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>3</td>
<td>81</td>
<td>47</td>
<td>1.741</td>
<td>10.173</td>
<td>18</td>
<td>0.0809</td>
<td></td>
</tr>
<tr>
<td>mac</td>
<td>2</td>
<td>146</td>
<td>71</td>
<td>0.973</td>
<td>6.290</td>
<td>6</td>
<td>-0.0256</td>
<td></td>
</tr>
<tr>
<td>monitor</td>
<td>6</td>
<td>88</td>
<td>52</td>
<td>3.545</td>
<td>18.739</td>
<td>66</td>
<td>0.0946</td>
<td></td>
</tr>
<tr>
<td>resolution</td>
<td>2</td>
<td>50</td>
<td>32</td>
<td>1.280</td>
<td>7.884</td>
<td>10</td>
<td>0.0288</td>
<td></td>
</tr>
<tr>
<td>screen</td>
<td>2</td>
<td>92</td>
<td>62</td>
<td>1.348</td>
<td>6.561</td>
<td>9</td>
<td>0.0199</td>
<td></td>
</tr>
<tr>
<td>video</td>
<td>4</td>
<td>106</td>
<td>61</td>
<td>2.302</td>
<td>12.188</td>
<td>28</td>
<td>0.0187</td>
<td></td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Word</th>
<th>Num</th>
<th>City</th>
<th>State</th>
<th>Value1</th>
<th>Value2</th>
<th>Value3</th>
<th>Value4</th>
<th>Value5</th>
</tr>
</thead>
<tbody>
<tr>
<td>desk</td>
<td>2</td>
<td>127</td>
<td>76</td>
<td>1.197</td>
<td>6.154</td>
<td>7</td>
<td>-0.1062</td>
<td></td>
</tr>
<tr>
<td>engine</td>
<td>1</td>
<td>15</td>
<td>13</td>
<td>0.867</td>
<td>5.343</td>
<td>5</td>
<td>0.0282</td>
<td></td>
</tr>
<tr>
<td>font</td>
<td>4</td>
<td>111</td>
<td>37</td>
<td>1.333</td>
<td>14.187</td>
<td>19</td>
<td>0.6350</td>
<td></td>
</tr>
<tr>
<td>laser</td>
<td>3</td>
<td>61</td>
<td>27</td>
<td>1.328</td>
<td>11.836</td>
<td>16</td>
<td>0.2562</td>
<td></td>
</tr>
<tr>
<td>print</td>
<td>3</td>
<td>140</td>
<td>66</td>
<td>1.414</td>
<td>9.154</td>
<td>13</td>
<td>0.0509</td>
<td></td>
</tr>
</tbody>
</table>

[6] 4D Write, 4D Calc, 4D XREF. (Software Review) (add-ins for Acius' Fourth Dimension database software) (evaluation)

<table>
<thead>
<tr>
<th>Word</th>
<th>Num</th>
<th>City</th>
<th>State</th>
<th>Value1</th>
<th>Value2</th>
<th>Value3</th>
<th>Value4</th>
<th>Value5</th>
</tr>
</thead>
<tbody>
<tr>
<td>add-in</td>
<td>2</td>
<td>97</td>
<td>38</td>
<td>0.784</td>
<td>7.540</td>
<td>6</td>
<td>0.5551</td>
<td></td>
</tr>
<tr>
<td>analysis</td>
<td>2</td>
<td>179</td>
<td>139</td>
<td>1.553</td>
<td>4.947</td>
<td>8</td>
<td>-0.8492</td>
<td></td>
</tr>
<tr>
<td>database</td>
<td>5</td>
<td>138</td>
<td>67</td>
<td>2.428</td>
<td>14.515</td>
<td>35</td>
<td>0.1832</td>
<td></td>
</tr>
<tr>
<td>midrange</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>0.857</td>
<td>6.116</td>
<td>5</td>
<td>0.0218</td>
<td></td>
</tr>
<tr>
<td>spreadsheet</td>
<td>2</td>
<td>75</td>
<td>44</td>
<td>1.173</td>
<td>7.247</td>
<td>9</td>
<td>0.1707</td>
<td></td>
</tr>
<tr>
<td>vary</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>0.857</td>
<td>6.116</td>
<td>5</td>
<td>0.0107</td>
<td></td>
</tr>
</tbody>
</table>

[7] ConvertIt! (Software Review) (utility for converting HyperCard stacks to IBM PC format) (evaluation)

<table>
<thead>
<tr>
<th>Word</th>
<th>Num</th>
<th>City</th>
<th>State</th>
<th>Value1</th>
<th>Value2</th>
<th>Value3</th>
<th>Value4</th>
<th>Value5</th>
</tr>
</thead>
<tbody>
<tr>
<td>converter</td>
<td>2</td>
<td>24</td>
<td>13</td>
<td>1.083</td>
<td>9.686</td>
<td>10</td>
<td>0.0698</td>
<td></td>
</tr>
<tr>
<td>doe</td>
<td>5</td>
<td>97</td>
<td>84</td>
<td>4.330</td>
<td>13.385</td>
<td>58</td>
<td>-0.1139</td>
<td></td>
</tr>
<tr>
<td>graphical</td>
<td>2</td>
<td>307</td>
<td>171</td>
<td>1.114</td>
<td>4.532</td>
<td>5</td>
<td>-2.4079</td>
<td></td>
</tr>
<tr>
<td>hypercard</td>
<td>4</td>
<td>25</td>
<td>13</td>
<td>2.080</td>
<td>18.371</td>
<td>38</td>
<td>0.1517</td>
<td></td>
</tr>
<tr>
<td>mac</td>
<td>2</td>
<td>146</td>
<td>71</td>
<td>0.973</td>
<td>6.290</td>
<td>6</td>
<td>-0.0256</td>
<td></td>
</tr>
<tr>
<td>map</td>
<td>2</td>
<td>17</td>
<td>10</td>
<td>1.176</td>
<td>10.210</td>
<td>12</td>
<td>0.1180</td>
<td></td>
</tr>
<tr>
<td>program</td>
<td>4</td>
<td>670</td>
<td>334</td>
<td>1.994</td>
<td>5.386</td>
<td>11</td>
<td>-15.4832</td>
<td></td>
</tr>
<tr>
<td>script</td>
<td>3</td>
<td>54</td>
<td>32</td>
<td>1.778</td>
<td>11.326</td>
<td>20</td>
<td>0.1239</td>
<td></td>
</tr>
<tr>
<td>software</td>
<td>3</td>
<td>913</td>
<td>449</td>
<td>1.475</td>
<td>3.402</td>
<td>5</td>
<td>-30.7596</td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td>5</td>
<td>15</td>
<td>8</td>
<td>2.667</td>
<td>25.142</td>
<td>67</td>
<td>0.0700</td>
<td></td>
</tr>
</tbody>
</table>

[8] Reports 2.0. (Software Review) (Nine To Five Software Reports 2.0 report generator for HyperCard 2.0) (evaluation)

<table>
<thead>
<tr>
<th>Word</th>
<th>Num</th>
<th>City</th>
<th>State</th>
<th>Value1</th>
<th>Value2</th>
<th>Value3</th>
<th>Value4</th>
<th>Value5</th>
</tr>
</thead>
<tbody>
<tr>
<td>hypercard</td>
<td>5</td>
<td>25</td>
<td>13</td>
<td>2.600</td>
<td>22.714</td>
<td>59</td>
<td>0.1517</td>
<td></td>
</tr>
<tr>
<td>print</td>
<td>3</td>
<td>140</td>
<td>66</td>
<td>1.414</td>
<td>9.154</td>
<td>13</td>
<td>0.0509</td>
<td></td>
</tr>
</tbody>
</table>
[9] Project-scheduling tools. (FastTrack Schedule, MacSchedule) (Software Review) (evaluation)

manage
2 318 174 1.094 4.497 5 -2.4884

[10] Digital Darkroom. (Software Review) (new version of image-processing software) (evaluation)

apply 1 17 15 0.882 5.200 5 0.0317
digital 4 90 52 2.311 12.826 30 -0.0042
image 4 107 58 2.168 12.389 27 0.1422
palette 2 18 12 1.333 9.846 13 0.0660
portion 2 17 15 1.765 9.399 17 0.0295
software 4 913 449 1.967 4.203 8 -30.7596
text 2 55 46 1.673 7.158 12 0.0304
user 5 861 435 2.526 5.162 13 -26.8094


accessory 2 14 10 1.429 10.210 15 0.0540
desk 2 127 76 1.197 6.154 7 -0.1062
display 2 106 78 1.472 6.102 9 -0.1278
program 3 670 334 1.496 4.290 6 -15.4832
sound 2 14 8 1.143 10.657 12 0.1172
user 3 861 435 1.516 3.497 5 -26.8094

[12] DisplayServer II-DPD. (Hardware Review) (DisplayServer II video card for using VGA monitor with Macintosh) (evaluation)

apple 4 261 112 1.716 9.757 17 -1.1625
card 2 99 56 1.131 6.765 8 0.0790
display 2 106 78 1.472 6.102 9 -0.1278
macintosh 3 210 107 1.529 7.705 12 -0.5855
monitor 6 88 52 3.545 18.739 66 0.0946
vga 2 91 62 1.363 6.561 9 0.0104
video 2 106 61 1.151 6.594 8 0.0187

capture 2 14 11 1.571 10.020 16 0.0271
image 3 107 58 1.626 9.542 16 0.1422
software 3 913 449 1.475 3.402 5 -30.7596
window 4 417 159 1.525 3.497 5 -26.8094

[14] Studio Vision. (Software Review) (Lehrman, Paul D.) (evaluation) Lehrman, Paul D.
audio 1 8 6 0.750 6.116 5 0.0161
disk 3 234 121 1.551 7.336 11 -1.1468
edit 3 111 77 2.081 8.692 18 -0.0961
operate 2 269 184 1.368 4.386 6 -2.1852
portion 1 17 15 0.882 5.200 5 0.0295
requirement 2 87 76 1.747 6.154 11 -0.1203
sound 6 14 8 3.429 29.970 103 0.1172
user 3 861 435 1.516 3.497 5 -26.8094

[15] 70 things you need to know about System 7.0. (includes related articles on past reports about System 7.0, Adobe Type 1 fonts,
Data on the run. (Hardware Review) (palmtop organizers)(includes related article describing the WristMac from Microseeds)

cache: 1 10 9 0.900 5.711 5 0.0127

cache: 8 49 30 4.898 29.052 142 0.1613

card: 6 99 56 3.394 18.294 62 0.0790

chip: 2 117 67 1.145 6.406 7 -0.1153

high-speed: 2 18 14 1.556 9.537 15 0.0352

memory: 3 142 98 2.070 7.968 16 -0.2999

ram: 2 145 93 1.283 5.750 7 -0.3992

High-speed, low-cost IIci cache cards. (includes related article on caching for other Mac models) (buyers guide)

cach: 1 10 9 0.900 5.711 5 0.0127

cache: 8 49 30 4.898 29.052 142 0.1613

card: 6 99 56 3.394 18.294 62 0.0790

chip: 2 117 67 1.145 6.406 7 -0.1153

high-speed: 2 18 14 1.556 9.537 15 0.0352

memory: 3 142 98 2.070 7.968 16 -0.2999

ram: 2 145 93 1.283 5.750 7 -0.3992

Mac, DOS and VAX file servers. (multiplatform file servers)(includes related articles on optimizing server

add-on: 1 17 15 0.882 5.200 5 0.0374

apple: 2 261 112 0.858 5.379 5 -1.1625

file: 10 271 151 5.572 19.905 111 -1.3982

lan: 2 98 51 1.041 6.952 7 0.0366

mac: 4 146 71 1.945 11.580 23 -0.0256

macintosh: 6 210 107 3.057 14.410 44 -0.5855

ms_dos: 2 98 65 1.327 6.467 9 0.0481

netware: 2 60 28 0.933 8.151 8 0.2314


ratio: 1 18 16 0.889 5.135 5 0.0154

server: 12 162 75 5.556 32.083 178 -0.1592

software: 3 913 449 1.475 3.402 5 -30.7596

unix-based: 1 15 13 0.867 5.343 5 0.0376

user: 3 861 435 1.516 3.497 5 -26.8094

vax: 2 28 14 1.000 9.537 10 0.1692
9.4 Inverse Document Frequency Weight

One of the simplest word weighting schemes to implement is the Inverse Document Frequency weight. The IDF weight is a measure of how widely distributed a term is in a collection. Low IDF weights mean that the term is widely used while high weights indicate that the usage is more concentrated.

An IDF weight measures the weight of a term in the collection as a whole, rather than the weight of a term in a particular document.

In individual document vectors, the normalized frequency of occurrence of each term is multiplied by the IDF to give a weight for the term in the particular document. Thus, a term with a high frequency but a low IDF weight could still be a highly weighted term in a particular document, and, on the other hand, a term with a low frequency but a high IDF weight could also be an important term in a given document. The IDF weight for a term \( W \) in a collection of \( N \) documents is:

\[
\log_2 \left( \frac{N}{\text{DocFreq}_w} \right)
\]

where \( \text{DocFreq}_w \) is the number of documents in which term \( W \) occurs.

9.4.1 OSU MEDLINE Data Base IDF Weights

The IDF weights for the OSU MEDLINE collection were calculated based on the \(^{df()}\) vector created in Program 19 on page 81.

Note that because of tuning parameters that set thresholds for the construction of the stop list and other factors, different runs on the document collection will produce variations in the values displayed. The IDF weights can range from lows such as:

<table>
<thead>
<tr>
<th>Term</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>human</td>
<td>0.189135</td>
</tr>
<tr>
<td>and</td>
<td>0.288966</td>
</tr>
<tr>
<td>the</td>
<td>0.300320</td>
</tr>
</tbody>
</table>
to highs such as:

- actinomycetoma
- actinomycetomata
- actinomycoma
- actinomyosine
- actinoplane
- actinopterygii
- actinoxanthin
- actisomide
- activ
- activationin

Note: for a given IDF value, the words are presented alphabetically. The OSU MEDLINE collection has many terms that appear only once and, consequently, the IDF weights for the above, all with a document frequency of one, are the same.

### 9.4.2 Calculating IDF Weights

To calculate IDF weights we first build a document-term matrix $^{doc(i,w)}$, as shown above, where $i$ is the document number and $w$ is a term. Each cell in the matrix contains the count of the number of times a term occurs in the document.

Next, from the document-term matrix, we construct a document frequency vector $^{df(w)}$ where each element gives the number documents in which the term $w$ occurs.

A basic program to calculate IDF weights is shown in Program 20. It assumes that the global arrays $^{doc()}$ and $^{df()}$ have been calculated (see Program 19 on page 81).

```
#!/usr/bin/mumps
# idf.mps Feb 14, 2014
# Copyright 2014 Kevin C. O'Kane
set doc=^DocCount(1)
# calculate IDF weights
for w=$order(^df(w)) do
  set x=$zlog(doc/^df(w))
  set ^idf(w)=$justify(x,1,2)
halt
```

Program 20 IDF calculation

If a word has a very low or high IDF value, it may be a candidate for removal from the collection. Program 21 deletes words whose IDF value falls below a specified threshold. In these examples, we retain all high IDF words but this can be easily changed. Very high IDF values indicate a very low frequency of occurrence. Note that Program 21 also deletes documents if they have fewer than a predetermined (from a
parameter) number of words in their vectors. This can also lead to words being deleted if, as a result, they are no longer contained in any document.

Program 21 Deletion of words with low IDF scores

9.5 Discrimination Coefficients

Discrimination coefficients are another way to determine the importance of a term in the collection. Discrimination coefficients [Willet 1985, Crouch 1988, Salton 1983] measure the ability of terms to differentiate one document from another. They are
calculated based on the effect a term has on overall hyperspace density with and without a given term.

If the space density is greater after a term is removed, that means the term was making documents look less like one another (a good discriminator) while a term whose removal decreases the density, thus making documents look more like one another, is a poor discriminator.

The discrimination values for a set of terms in a given document collection are comparable to the values for the IDF weights but not exactly.

The basic procedure to calculate discrimination coefficients involves calculating the average of pair-wise similarities between all documents in the space. This is the documents space density.

Then, for each word, the average pair-wise similarities of all the documents is recalculated without the word. The difference in the averages is the term discrimination value for the term in question.

In practice, this is a very expensive weight to calculate unless speed-up techniques are used.

The centroid algorithm approaches [Crouch 1988], are an attempt to improve the speed of calculation. The exact calculation, where all pairwise similarity values are calculated each time, is of complexity on the order of \((N)(N-1)(w)(W)\) where \(N\) is the number of documents, \(W\) is the number of words in the collection and \(w\) is the average number of terms per document vector.

Crouch [Crouch 1988] discusses several methods to improve the speed of this calculation. The first of these, the centroid approximate algorithm, involves calculating the similarities of the documents with a centroid vector representing the collection as a whole rather than all pair-wise similarities.

In the centroid approximation, a centroid vector for the entire collection is first calculated. A centroid vector is the average of all the document vectors and, by analogy, represents a point at the center of the hyperspace.

When we use a centroid vector, rather than calculate all the pair-wise similarities, we only calculate the similarities between each document and the centroid vector. The average of these similarities is taken to be the space density which, although not exactly the same as the full pairwise scheme, has been shown to be very similar [Crouch 1987].

The prime advantage of this method is thus, rather than calculating \(N^2\) document-document similarities, we calculate only \(N\) similarities. This improves the complexity to be on the order of \((N)(w)(W)\) which represents a significant improvement in performance.
A further modification, called the Modified Centroid Algorithm (MCA), is possible. It is based on the centroid approximation from above but further reduces the overall complexity.

In the MCA, we calculate an initial space density in the same manner as the centroid approximation from above. However, in the MCA approach, we retain and store the individual contribution of each document to the initial total document space density. This requires additional space on the order of $N$ to store the original $N$ contributions.

Subsequently, when calculating the effect of a some term $i$'s removal on the document space density, the algorithm first subtracts from the total space density the original contribution of each document that contains term $i$ (using the stored values) and then it adds to the space density the similarity between each document containing term $i$ and the centroid vector calculated without term $i$.

So, for example, if we have a collection of 10,000 documents. First we calculate a centroid vector which is the average of all 10,000 vectors. Next we calculate the sum of the similarities between each document and the centroid vector. This is the initial space density $D$. We also, for each document, store its individual contribution to $D$.

In the end, if $D$ is large, it means that the documents were very similar to one another. However, if $D$ is small, it means the documents were not very similar to one another.

Next, for each term we calculate a new space density without the term.

For example, for some term $i$ we copy the initial space density to a variable $D_i$ which will be the space density without term $i$.

If some term $i$ occurs in ten documents, we find the ten values that those documents originally contributed to the initial space density and subtract them from $D_i$.

Now, for each of the ten documents, we calculate ten new similarities between the document vectors and the centroid vector but without any contribution from term $i$. These we add to $D_i$. The difference between $D$ and $D_i$ is the effect on the space density of term $i$.

If the $D_i$ is less than $D$, it means that term $i$ made the documents more like one another and is thus not a good discriminator. On the other hand, if $D_i$ is larger than $D$, it means that term $i$ was making the documents look less like one another and thus a good discriminator.

So, if $(D_i - D)$ is negative, term $i$ is a poor discriminator. If it is positive, term $i$ is a good discriminator.

Complexity is on the order of $N + (DF)(w)(W)$ where $DF$ is the average number of documents in which an average term occurs, $W$ is the total number of words in the collection and $w$ is the average number of terms per document vector.
This method increases the amount of space required in two ways. First, a vector of length N of original contributions is required.

Second, an inverted list giving, for each word, those documents containing the word is needed. The list will contain W entries and the amount of data stored will be proportional to the average number of documents in which an average word appears. Such an inverted file, however, is normally present at this stage of indexing in the form of the term-document matrix.

The advantage of using an inverted term-document matrix is that it quickly identifies those documents containing terms of interest rather than scanning, for each term, through the entire document-term matrix looking for documents that contain the term.

While the MCA method yields values that are only an approximation of the exact method, the values are very similar to the exact method while the savings in time to calculate the coefficients are very significant.

Crouch [Crouch 1988] reports that the MCA method was on the order of 527 times faster than the exact method on relatively small data sets. Larger data sets yield even greater savings as the time required for the exact method grows with the square of the number of documents while the MCA method grows linearly.

The basic MCA algorithm is given in Program 22. This program operates on a copy (copy2) of the database and runs in parallel with other routines. It uses unweighted (term counts only) copies of the ^doc and ^index matrices.

```mumps
1 #!/usr/bin/mumps
2 # discrim.mps Feb 14, 2014
3 # Copyright 2014 Kevin C. O'Kane
4
5    database "copy2"
6
7    set D=^DocCount(1) // number of documents
8    set sq=0
9    kill ^mca
10
11   # calculate centroid vector ^c() for entire collection and
12   # the sum of the squares (needed in cos calc but should only be done once)
13   #++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
14   #++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
15   #++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
16    for w=$order(^dict(w)) do
17       . set ^c(w)=^dict(w)/D // centroid is composed of avg word usage
18       . set sq=^c(w)**2+sq // The sum of the squares is needed below.
19
20   # Calculate total similarity of doc for all words (T) space by
21   # calculating the sum of the similarities of each document with the centroid.
22   # Remember and store contribution of each document in `dc(dn).
23   #++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
24   #++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
25   #++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
26    set T=0
27    for i=$order(^doc(i)) do
28       . set x=0
29       . set y=0
30```

92
Program 22 Modified centroid algorithm

In Program 22, the density of the space without word i is:
where T is the original similarity between all documents and the centroid vector with no terms removed, T_{i1} is the sum of original similarity contributions to T of those documents in which term i appeared, and T_{i2} is the newly calculated sum of the contributions of those same documents without word i. Note that T_{i1} and T_{i2} only refer to the contribution to T by those documents which originally contained term i.

That is, we subtract the original contribution (T_{i1}) from T and then add in the replacement contribution T_{i2} calculated without the word under consideration.

The difference between the original density T and the density without word i is:

\[ T - (T - T_{i1} + T_{i2}) \]

which reduces to

\[ (T_{i1} - T_{i2}) \]

A larger value (higher space density) of either T_{i1} or T_{i2} means documents are more like one another while a smaller value (lower space density) means they are less like one another.

Thus, if T_{i1} is larger than T_{i2}, it means that for word i, removing it from the collection made the documents look less like one another. Consequently, term i makes documents look more like one another and it is a poor discriminator.

On the other hand, if T_{i1} is smaller than T_{i2}, it means that removing word i made the documents look more like one another and therefore term i is a good discriminator. Word i is a discriminator.

In order to give positive values to good discriminators and negative values to poor discriminators, in Program 22 we reverse the order of the subtraction to become:

\[ (T_{i2} - T_{i1}) \]

MCA discrimination co-efficients range from very large negative numbers (poor discriminators) to small positive numbers (good discriminators). Good discriminators are small positive values because they tend to be words used infrequently in few documents and, consequently, their effect on the space density is minimal. On the other hand, poor discriminators, by definition, occur in many documents and thus their contribution to the document space density is large.

Because overall the numbers tend to be very small, we multiply the results by an arbitrary factor to bring them into a more easily managed range and (by the integer division operator \( \backslash \)), we truncate fractional parts.

Figure 41 gives some sample good discriminators and Figure 42 gives some poor discriminators. The discrimination co-efficient is the first number, followed by the word
followed by the inverse document frequency weight. In the good discriminator list, some technical words are not shown.

Note that the inverse document frequency weights often differ with discrimination co-efficients. For example, the words:

982 phenytoin 6.66
985 atenolol 7.01
988 streptokinase 6.83
1009 betacarotene 8.05
1009 meningioma 6.89

are very near one another with regard to discrimination co-efficients (first column) but quite different with regard to inverse document frequency weights (third column).

Also note that in the list of poor discriminators, very high frequency and stop list words had already been eliminated at earlier stages and thus do not appear here. Also note that the discrimination co-efficients and the inverse document frequencies are in closer agreement.

Also, while in most texts words like platelet, breast, glucose, laser, bladd[er], diabetic and cholesterol would be important terms, in this specialized medical database, they are not.

956 restenosis 6.81
965 valvuloplasty 6.52
972 gerbil 8.05
973 folate 7.11
977 cpr 6.97
980 sufentanil 7.49
982 phenytoin 6.66
985 atenolol 7.01
988 streptokinase 6.83
1009 betacarotene 8.05
1009 meningioma 6.89
1010 splenectomy 6.56
1025 ribavirin 7.39
1029 dobutamine 7.18
1031 hemangioma 6.78
1034 acetaldehyde 7.35
1056 lithium 6.89
1083 selenium 7.24
1083 taurine 7.56
1097 coarct 6.97
1098 digoxin 6.93
1116 hypoglycemia 6.81
1124 islet 6.78
1134 nalbuphine 8.05
1140 oxalate 7.21
1169 pseudoaneurysm 7.11
1187 methadone 7.64
1187 thymoma 7.24
1221 amiodarone 6.54
1226 mesothelioma 7.08
1230 propafenone 8.42
1236 surfactant 6.89
1299 pseudocyst 7.52
Table 1: Some Important Medical Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ketanserin</td>
<td>7.42</td>
</tr>
<tr>
<td>nicardipine</td>
<td>7.64</td>
</tr>
<tr>
<td>pheochromocytoma</td>
<td>6.93</td>
</tr>
<tr>
<td>rilmenidine</td>
<td>8.25</td>
</tr>
<tr>
<td>sucralfate</td>
<td>7.13</td>
</tr>
<tr>
<td>caffeine</td>
<td>6.69</td>
</tr>
<tr>
<td>nabumetone</td>
<td>7.73</td>
</tr>
<tr>
<td>tgfbeta</td>
<td>8.11</td>
</tr>
</tbody>
</table>

Figure 41: Good Discriminators

<table>
<thead>
<tr>
<th>Term</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>platelet</td>
<td>4.65</td>
</tr>
<tr>
<td>breast</td>
<td>4.54</td>
</tr>
<tr>
<td>glucose</td>
<td>4.55</td>
</tr>
<tr>
<td>laser</td>
<td>4.74</td>
</tr>
<tr>
<td>bladd</td>
<td>4.63</td>
</tr>
<tr>
<td>diabetic</td>
<td>4.77</td>
</tr>
<tr>
<td>cord</td>
<td>4.52</td>
</tr>
<tr>
<td>cholesterol</td>
<td>4.75</td>
</tr>
<tr>
<td>hiv</td>
<td>4.72</td>
</tr>
<tr>
<td>alcohol</td>
<td>4.63</td>
</tr>
<tr>
<td>cervical</td>
<td>4.51</td>
</tr>
<tr>
<td>arthrit</td>
<td>4.63</td>
</tr>
<tr>
<td>beta</td>
<td>4.61</td>
</tr>
<tr>
<td>thyroid</td>
<td>4.84</td>
</tr>
<tr>
<td>intake</td>
<td>4.52</td>
</tr>
<tr>
<td>diabete</td>
<td>4.52</td>
</tr>
<tr>
<td>hemorrhage</td>
<td>4.58</td>
</tr>
<tr>
<td>neuron</td>
<td>4.71</td>
</tr>
<tr>
<td>traine</td>
<td>4.71</td>
</tr>
<tr>
<td>image</td>
<td>4.59</td>
</tr>
<tr>
<td>fiber</td>
<td>4.68</td>
</tr>
<tr>
<td>motor</td>
<td>4.54</td>
</tr>
<tr>
<td>intestinal</td>
<td>4.55</td>
</tr>
<tr>
<td>arteries</td>
<td>4.51</td>
</tr>
<tr>
<td>repair</td>
<td>4.57</td>
</tr>
<tr>
<td>radiation</td>
<td>4.55</td>
</tr>
<tr>
<td>aneurysm</td>
<td>4.87</td>
</tr>
<tr>
<td>tachycardia</td>
<td>4.88</td>
</tr>
<tr>
<td>stroke</td>
<td>4.64</td>
</tr>
<tr>
<td>bypass</td>
<td>4.63</td>
</tr>
<tr>
<td>wound</td>
<td>4.69</td>
</tr>
<tr>
<td>carotid</td>
<td>4.81</td>
</tr>
<tr>
<td>urine</td>
<td>4.55</td>
</tr>
<tr>
<td>percutane</td>
<td>4.59</td>
</tr>
<tr>
<td>birth</td>
<td>4.60</td>
</tr>
<tr>
<td>burn</td>
<td>5.08</td>
</tr>
<tr>
<td>insulin</td>
<td>4.99</td>
</tr>
<tr>
<td>visu</td>
<td>4.63</td>
</tr>
<tr>
<td>leukemia</td>
<td>4.81</td>
</tr>
<tr>
<td>antagonist</td>
<td>4.52</td>
</tr>
<tr>
<td>fragment</td>
<td>4.52</td>
</tr>
<tr>
<td>output</td>
<td>4.53</td>
</tr>
<tr>
<td>food</td>
<td>4.73</td>
</tr>
<tr>
<td>energy</td>
<td>4.67</td>
</tr>
<tr>
<td>anesthesia</td>
<td>4.62</td>
</tr>
<tr>
<td>neck</td>
<td>4.62</td>
</tr>
</tbody>
</table>

Figure 42: Poor Discriminators

96
10 Weighted Matrices

10.1 Document-Term Matrix

At this point it is now possible to re-calculate the document-term matrix such that the values for each cell reflect the weight of the word in the document rather than merely the number of occurrences or the word in the document. To do this we replace each word count by the product of the word count and the weight of the word in the collection as a whole. This is shown in Program 23.

We may use either the IDF weights, MCA weights (assuming that the MCA weights have been converted a positive range suitable for multiplication), or any other weighting scheme available such that a higher value for a weight means the term is more important. These weights, by themselves, indicate the importance of the associated term in the collection as a whole. By multiplying these weights by the number of times the term occurs in a particular document we obtain a measure of how important this term is in this document.

For example, we might have a term that is not very important in the collection as a whole but, because it occurs many times in a single document, it is important in that document. Conversely, a term highly rated in the collection as a whole may occur at a low frequency in a particular document. Thus, a term with a low collection weight can be more highly weighted in some documents than a term with a higher collection weight.

In Program 23 IDF weights are used as multipliers (see line 12).

Additionally, Program 23 builds the term-document matrix. The term-document matrix is the transpose of the document-term matrix. In it, rows represent the terms and the columns are the documents. The content of each element for a given document number and term is the same as in the document-term matrix: the frequency or weighted frequency of a term in the document.

```
1 #!/usr/bin/mumps
2 # weight.mps December 26, 2011
3
4 kill ^index
5
6 open 2:"document-term-matrix-weighted.txt,new"
7
8 for d=$order(^doc(d)) do
9   . use 2 write "doc=",d,?15
10   . use 2 write ^title(d),!,?15
11   . for w=$order(^doc(d,w)) do
12     .. set x=^idf(w)*^doc(d,w)
13     .. set ^doc(d,w)=x
14     .. set ^index(w,d)=x
15     .. write w,"(",x,") "
16     . write !
17 close 2
18
19 open 2:"term-document-matrix-weighted.txt,new"
20 use 2
```
21 for w=$order(`index(w)) do
22  . write w,?26
23  . for d=$order(`index(w,d)) do
24  .. write d,"("`index(w,d),") "
25  . write !
26 close 2

Program 23 Weighted Doc-Term and Term-Doc Matrices

10.2 Title Global Array

Next, we create a global array vector of document titles for use with subsequent retrieval examples. This is shown in Program 24. The original file of titles, stdin input to this program, was written when osu.converted was created by reformat.mps. Long titles are truncated. The input file consists of the document number followed by a blank followed by the title.

1 #!/usr/bin/mumps
2 # loadTitles.mps Feb 14, 2014
3 # Copyright 2014 Kevin C. O’Kane
4
5  for do
6  . read t
7  . if '$test halt
8  . set `title($piece(t,"",1))=$piece(t,"",2,100)

Program 24 Create global array of title text

10.3 SQL Support

Also, a file of SQL CREATE TABLE and INSERT commands for use with experiments involving relational database management systems is optionally created. This is shown in Program 25. Figure 43 shows some sample SQL SELECT for document retrieval and Figure 44 shows sample SQL INSERT, DROP, and CREATE TABLE statements. Generally speaking, RDBMS systems perform poorly in IS&R applications.

1 #!/usr/bin/mumpsRO
2 # SQLdocvectors.mps December 17, 2011
3 # Copyright 2014 Kevin C. O’Kane
4
5  write "use isr;!
6  write "drop table if exists docVect;!
7  write "create table docVect (doc int, word varchar(25), wgt float);!
8  write "drop table if exists docs;!,
9  write "create table docs (doc int, title varchar(255));!
10 for d=$order(`doc(d)) do
11  . if $order(`doc(d,""))="" quit
12  . set t=$translate($extract(`title(d),7,256),"'"
13  . use 5 write "insert into docs values ("doc(d),"",1)=
14  . for w=$order(`doc(d,w)) do
15  .. write "insert into docVect values ("doc(d,w)");!

Program 25 Creating an RDBMS Doc-Term matrix

select doc, round(sum(wgt),2) from docVect where word='mole' or word='ethylate' group by
11 Thesaurus construction

It is possible to discover connections between vocabulary terms based on their frequency of co-occurrence. Terms that co-occur frequently together are likely to be related and this can indicate that the words may be synonyms used to express a similar concepts or phrase components.

For example, the strong statistical relationship such as between the words *artificial* and *intelligence* in a computer science database is due to the phrase *artificial intelligence* which names a branch of computing. In this case, the relationship is not that of a synonym but that of a phrase. Similarly, in a medical data base, terms such as *circadian rhythm* and *vena cava* and *herpes simplex* are also concepts expressed using more than one term.

On the other hand, terms such as *synergism* and *synergistic*, *cyst* and *cystic*, *schizophrenia* and *schizophrenic*, *nasal* and *nose*, and *laryngeal* and *larynx* are examples of synonym relationships.

In other cases, the relationship is not so tight so as to be a full synonym but it may express a categorical relationship such as *anesthetic* and *halothane; analgesia* and *morphine*, *nitrogen* and *urea*; and *nurse* and *personnel.*
Regardless of the relationship, a basic thesaurus, that is, a data structure that lists words in groups of synonyms and related concepts, can be used to:

1. Augment queries with related words to improve recall. For example, if the user enters the term *halothane*, the system might add, at a lower weight, the term *anesthetic* to the query. This would broaden the query thus improving recall but at the expense of precision.

2. Combine multiple, related, infrequently occurring terms into broader, more frequently occurring terms. In many systems, low frequency terms are discarded. For example, a collection might contain a number of specific medication names all related to the same medical condition which individually occur infrequently (*e.g.*, names of medicines associated with treatment of high blood pressure, of which there are many). However, by gathering these terms into a collective term, their aggregate frequency of occurrence might be high enough to be included in the indexing vocabulary. In place of the original terms in the documents, a stand-in token for the category would replace each original word instance.

3. Create middle frequency composite terms from otherwise unusable, high frequency terms. While some words individually may have frequencies of occurrence too high to be useful as index terms (*e.g.*, computer), taken as a component of a phrase with other high frequency terms (*e.g.*, computer aided instruction), the phrase’s frequency may drop below the high frequency cutoff.

To build a thesaurus we first we construct a square term-term correlation matrix which gives the frequency of co-occurrence of terms with one another. A term-term matrix is a symmetric\(^{52}\) square matrix with terms denoting the rows and columns.

The values in the cells of the matrix indicate the number of times a a row term and a column term co-occur. The diagonal is the number of times a term occurs. Only the lower or upper triangular matrix need be computer since they are equivalent (the number of times some term A occurs with term B is the same as term B with term A). A brief example is given in Figure 45.

<table>
<thead>
<tr>
<th></th>
<th>birds</th>
<th>cats</th>
<th>dogs</th>
<th>fish</th>
</tr>
</thead>
<tbody>
<tr>
<td>birds</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>cats</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>dogs</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>fish</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>9</td>
</tr>
</tbody>
</table>

*Figure 45 Term-Term Matrix*

To build a term-term matrix we inspect the documents for words that co-occur with one another. If some term A occurs in 20 documents and if term B also occurs in these same documents, the term-term correlation matrix cell for row A and column B will have a value of 20. Since the relationship between term A and B is always the same as the relationship between term B and A, a term-term correlation matrix's lower diagonal

\(^{52}\)The matrix is equal to its transpose.
matrix is the same as the upper diagonal matrix. The diagonal itself is the number of times a term occurs with itself which is the count of the number of documents in which the term occurs.

Calculating a complete term-term correlation matrix based on all documents in a large collection can be very time consuming. In many cases, a term-term matrix potentially contains many billions of elements (the square of the number of vocabulary terms) summed over the entire collection of documents. In practice, however, it is only necessary to sample a representative part of the total collection. That is, you can calculate a representative matrix by looking at every fifth, tenth or twentieth document, etc., depending on the total size of the collection.

However, be aware that many collections may contain clusters of documents concerning specific topics. For example, a number of documents taken from the same sub-speciality journal may appear consecutively in the collection. As these would deal with a particular aspect of a discipline, the word usage might be skewed. Thus, it is better to sample across all the documents than to only process documents in an initial fraction of the collection.

As many words never occur with others, especially in specialized technical collections such as the OSU medical data base, a term-term matrix for a vocabulary rich technical discipline may be very sparse. More general topic collections, however, will probably have matrices that are less sparse.

11.1 Building a Term-Term Co-Occurrence Matrix

A term-term co-occurrence matrix, also referred to as the term connection matrix, is an \( N \times N \) square matrix, where \( N \) is the number of terms in the vocabulary. Its elements give the number of documents common to each pair of terms. From a formal point of view, it is:

\[
\text{TermTerm} = \text{DocTerm}^T \cdot \text{DocTerm}
\]

That is, the product of the document-term matrix \( \text{DocTerm} \) with its transpose.

In our \( \text{TermTerm} \) matrix, the rows and columns are term identifiers and the cells contain a count of the number of documents which contained both terms.

Because the number of times a term occurs in a document is not important in terms of calculating term co-occurrences, for purposes of calculation, we treat the elements of \( \text{DocTerm} \) as binary values where a 0 indicates the term is not present in the document and a non-zero value indicates the term is present.

The elements of the \( \text{TermTerm} \) matrix \( \text{tt}_{jk} \) for terms \( j \) and \( k \) can be calculated from the document-term matrix elements \( d_{ij} \) for document \( i \) and terms \( j \) and \( k \) where \( i \) ranges from 1 to \( N \) the total number of document as follows:

\[
\text{tt}_{jk} = d_{1j}d_{1k} + d_{2j}d_{2k} + d_{3j}d_{3k} + \ldots + d_{Nj}d_{Nk}
\]

53 For example, pediatric terminology is unlikely to be common in articles about adult nutrition.
In other words, the similarity between term $i$ and term $j$ is the sum of the number of times they co-occur in a document.

For example, if you have 3 three documents entitled:

Doc$_1$ = Pseudomonas-aeruginosa cystic-fibrosis.
Doc$_2$ = Pseudomonas-aeruginosa immune response
Doc$_3$ = Immune complexes in cystic-fibrosis

Altogether we have five significant terms: *pseudomonas-aeruginosa, immune, cystic-fibrosis, response*, and *complexes* which are column headings for the matrix $DocTerm$ (rows here are document numbers) and the row headings for $DocTerm^T$ (column headings are documents). Thus row one of $DocTerm$ indicates that the document only contained the terms *pseudomonas-aeruginosa* and *cystic-fibrosis*. The first row of the $DocTerm^T$ matrix indicates that the term *pseudomonas-aeruginosa* occurs in documents 1 and 2 but not document 3.

In the $TermTerm$ matrix, both row and column headings are terms. The first row indicates that the term *pseudomonas-aeruginosa* occurs twice in the collection as a whole (diagonal element), once each with the terms *immune, cystic-fibrosis, and response*, but not at all with the term *complexes*.

\[
DocTerm = \begin{pmatrix}
1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 1 \\
\end{pmatrix} \quad DocTerm^T = \begin{pmatrix}
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 \\
\end{pmatrix}
\]

Consequently, the Term-Term matrix will thus be:

\[
TermTerm = DocTerm^T \cdot DocTerm = \begin{pmatrix}
2 & 1 & 1 & 1 & 0 \\
1 & 2 & 1 & 1 & 1 \\
1 & 1 & 2 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 \\
\end{pmatrix}
\]

A Mumps program to calculate a term-term matrix is given in Program 26. In this figure, the global array $\hat{doc}$ is assumed to be a modified document-term matrix similar to the ones used in other examples except that the values of the rows are either 0 or 1, not counts or weights. A 1 indicates that a term is present and a 0 indicates it is not.

1 #!/usr/bin/mumps
2 3 k  ^docT,^tt

102
Another way to construct the term-term correlation matrix is shown in Program 27. This program also yields a sparse matrix of term-term counts but can work with document-term matrices that contain counts rather than binary values. This is also called a term connection matrix in that its elements give the number of documents that are connected to one another through a given term.

Program proceeds by taking each document \( d \) in the doc-term matrix and selecting each term \( w \) in document \( d \). For each term \( w \), it selects those terms \( w_1 \) in \( d \) which are alphabetically greater than \( w \). For each pair of terms \( \{w,w_1\} \), it increments the co-occurrence count (or instantiates it with a value of 1 if it did not exist) of the cell in term-term matrix \(^tt\) at row \( w \) and column \( w_1 \). Effectively, this produces an upper diagonal matrix but no values on the diagonal itself are calculated.

The elements of the term-term matrix may now examined (not shown in Program 27) and those elements whose frequency of co-occurrence are below a threshold may be deleted. This threshold is adjusted according to collection size. In index.script, the threshold (\( MINTT \)) is set at 0.001*MAXDOCS if the number of documents is 5,000 or greater, 5 otherwise.

The methods in Program 26 and Program 27 can be slow. A speed up technique is shown in Programs 28 and 29. These are similar to Program 27 but it avoid the time consuming multiple database accesses in lines 10 and 11 of Program 27 by making use of system utility programs sort and uniq. The procedure is done in three steps. The first is Program 28, the second is a bash command string the processes the output of Program 28, and the third is Program 29 which reads the results of the bash script, makes final calculations and stores the results.

In Program 28, instead of incrementing the \(^tt()\) matrix for each word pair, each word pair is written to a temporary file. Each line in this file contains two co-occurring words separated by a blank.
Ultimately, the file contains a list of all co-occurring terms. The file is then sorted alphabetically. The result is a file in which all instances of co-occurring terms are grouped together on consecutive lines.

For example, if the terms *vena* and *cava* co-occur in documents of the collection twenty times, there will be twenty consecutive lines containing *vena* and *cava* in the sorted file.

Next the sorted file is processed by the system utility *uniq* which, for each consecutive duplicate line, produces one line with the count of the duplicate lines followed by the line. Thus, for the *vena* and *cava* example above, the twenty lines would be reduced to one line containing *vena* and *cava* preceded by the number 20. It does this for all word pairs. The resulting file is thus a list of co-occurring terms preceded by the number of times they co-occurred. The sort/uniq process is very fast.

The following line of code from *index.script* performs the sort and uniq operations:

```
tt1.mps copy1 | sort | uniq -c > tt.sorted.tmp 
```

Next, as shown in Program 29, the file is read and the ^tt() matrix is instantiated and a text file of the results generated. The following command from *index.script* does this:

```
tt2.mps $MINTT < tt.sorted.tmp | sort -n > term-term-matrix.sorted.txt 
```

```
#!/usr/bin/mumpsRO
# Copyright 2014 Kevin C. O'Kane
# tt1.mps December 11, 2011
5 if '$data(1) write "missing parameter",! halt
6 database 1
8 for d=order(^doc(d)) do
10 . for w=order(^doc(d,w)) do
11 .. for w1=w:order(^doc(d,w1)):"" do
12 ... if w1=w quit
13 ... write w," ",w1,!
```

Program 28 Faster Term-Term Matrix 1

The ^tt() matrix produced by Program 29 goes beyond that shown in Program 27 which produced a matrix with gave containing the raw count of the number of co-occurrences between terms.

Program 29 deletes term pairs whose frequency of co-occurrence is below a given threshold (*MINTT* mentioned above). Rather than store the simple count of co-occurrences, it calculates a value for each pair of terms in the range {0,1} where zero indicates no association and one indicates a complete association between the terms. That is, if two terms always co-occur together and never separately, their association is 1. Alternatively, if two terms never co-occur, their association is zero.

---

54 In fact, a value of zero is never be stored if a non-zero minimum co-occurrence threshold is provided to the program.
An example of the output of Program 29 is shown in Figure 46. In this example, 10,000 documents were processed.

Several lines from the start of the file as well as the end are shown. Words are shown in stem form. Note that the terms magnetic and resonance have the highest association (0.945). This is because they each occur 106 times and 103 of them they occur together in the same document. The association is significant.

Similarly, the lowest association is for the terms catheter and median (0.030). They occur 166 and 1172 times, respectively but only 10 times together in the same document. The association is not significant.
Salton suggested two metrics for term-term association.

In the first formula [Salton 1983], which was used in Program 29, is shown below using Salton's original notation. In Salton's original, the $t_{ij}$ are the weights of term $j$ in document $i$. In Program 29, the values of $t_{ij}$ are either zero (indicating that term $j$ did not occur in document $i$) or one, indicating that it did. The weight of a term is not very relevant when computing co-occurrences. Either the terms do co-occur or they do not.

\[
\text{SIMILAR}(\text{TERM}_k, \text{TERM}_h) = \frac{\sum_{i=1}^{n} t_{i,k} t_{i,h}}{\sum_{i=1}^{n} (t_{i,k})^2 + \sum_{i=1}^{n} (t_{i,h})^2 - \sum_{i=1}^{n} t_{i,k} t_{i,h}}
\]

If the $t_{ij}$ are taken to be counts, the figure

\[
\sum_{i=1}^{n} (t_{i,k})^2 + \sum_{i=1}^{n} (t_{i,h})^2
\]

becomes the product of the document frequencies of terms $k$ and $h$ as used in Program 29 because when the $t_{ik}$ either zero or one

\[
\sum_{i=1}^{n} (t_{i,k})^2
\]

becomes the number of documents that term $k$ appeared in. Similarly,

\[
\sum_{i=1}^{n} t_{i,k} t_{i,h}
\]

is the count of the number of the co-occurrences between terms $k$ and $h$.

Salton also proposed a second formula [Salton 1988] the $d_{ij}$ are the weight of term $j$ in document $i$: 
In Figure 47, the frequency of co-occurrence (number of times two words co-occur together, vertical axis) for the entire OSU collection is plotted against frequency of co-occurrence rank (horizontal axis). With regard to the horizontal axis, the term pair that co-occur the most frequently (a bit less than 1200 times) is rank number one (appears first), the term pair co-occurring next most frequently next, and so on.

As can be seen, the frequency of co-occurrence drops off rapidly to a nearly constant. Thus, only a few term pairs in this vocabulary, roughly the top 300, stand out as significantly more likely to co-occur than the remainder of the possible combinations due to chance alone.
11.2 Term-Term-Term Connection Matrix

A term-term connection matrix gives second order term connections. That is, if some term \(A\) is related to some term \(B\) and term \(B\) is related to term \(C\), there may be a relationship between terms \(A\) and \(C\) if the connections are sufficiently strong. It is calculated as follows:

\[
\text{tt}_{km}^2 = \sum_{p=1}^{N} \sum_{i=1}^{N} \sum_{j=1}^{N} d_{ik} d_{pk} d_{pm} d_{jm}
\]

where \(p, i, j\) are document numbers and \(k\) and \(m\) are terms. Again, as above, the values of \(d_{ik}\) are zero or one indicating if term \(k\) is present in document \(i\).

For any matrix element in \(tt^2\) addressed by row term \(k\) and any column term \(m\), the value of the is the number of intermediate terms they have in common. Thus, for example, if some term \(A\) does not co-occur with term \(B\) but they both co-occur with one term \(C\), the value of the element would be 1 and there is a weak indirect connection between term \(A\) and term \(C\). Higher numbers imply stronger connections.

Program 30 shows calculation of a term-term connection matrix.

Program 30 ttt.mps

11.3 Construction of Term Phrases

Recall (at the expense of precision) can be improved if additional related terms are added to a query. Thus, a query for \(antenna\) will usually result in more hits in the database if the related term \(aerial\) is added.

An increase in recall, however, is often accompanied by a decrease in precision. As is evidenced by the fact that while \(aerial\) is a commonly used synonym for \(antenna\), as in \(television aerial\), it can also refer to a dance move, a martial arts move, skiing, various musical groups, performances, and any activity that is done at a height (e.g., aerial photography). Thus, adding it to a query with \(antenna\) has the potential to introduce many extraneous hits.
Identification of phrases, however, has the potential to increase precision. Phrases can be regarded as composite terms of high specificity - such as *television aerial* noted above. While both *television* and *aerial* individually are broad terms, the phrases *television aerial* and *television antenna* are quite specific.

When a phrase is identified, it should be added as a single term in the document vector (see below).

Phrases can be identified by both syntactic and statistical methods.

### 11.3.1 Term Cohesion

While statistical techniques may take into account term proximity as well as co-occurrence such as seen above, Salton [Salton 1983] suggests the following simpler formula for construction of term phrases (using global array notation corresponding to the arrays in the toolkit):

\[
^\text{Cohesion}(i,j) = \text{SIZE_FACTOR} \ast \left( \frac{^\text{tt}(i,j)}{^\text{dict}(i) \ast ^\text{dict}(j)} \right)
\]

That is, the sum or the co-occurrences between term \( i \) and term \( j \) divided by the product of the total number of occurrences of term \( i \) and term \( j \).

The code to perform the cohesion calculation is shown in Program 31. In the script file *index.script*, it is invoked with:

```plaintext
cohesion.mps < term-term-matrix.sorted.txt | \sort -nr > term-term-cohesion-similarities.txt
```

where the input, *term-term-matrix.sorted.txt*, is the output of the term-term correlation program *tt2.mps*.

---

**Program 31 cohesion.mps**

```plaintext
#!/usr/bin/mumps
# ~/Medline2012/cohesion.mps February 14, 2014
# Copyright 2014 Kevin C. O'Kane

# phrase construction
for do
. read a
. if "$test break
. set w1=$piece(a," ",2)
. set w2=$piece(a," ",3)
. set co=$piece(a," ",4)
. set ch=co/(^dict(w1)*^dict(w2))*100000\1
. if ch>0 write ch," ",w1," ",w2,!
```

Figure 48 gives a sample of the results.

```
1730 compute ct
1792 blot gene
```
However the user should be cautioned that this procedure can sometimes result in unwanted connections such as *Venetian blind* and *blind Venetian*. For that reason, the aggregate relative position of the terms, as shown in the proximity calculation above, are useful in deciding when two terms are consistently linked. That is, if the order is strongly in favor of one term preceding another, this indicates a probable phrase; on the other hand, if the relative order is in neither direction, this is probably not a phrase.

### 11.3.2 Jaccard Term-Term Similarity

A similar measure can be calculated using the Jaccard metric. The formula (using global array notation) is:

\[
\text{set } \ ^{jaccardtt}(i,j) = \ ^{tt}(i,j) / (\ ^{df}(i) + \ ^{df}(j) - \ ^{tt}(i,j))
\]

That is, the Jaccard similarity between two words \(i\) and \(j\) is equal to the sum of the co-occurrences between term \(i\) and term \(j\) divided by the sum of the number of documents term \(i\) and term \(j\) occur in minus the sum of the co-occurrences of the terms.

Program 32 gives an example (not, the jaccard similarities are loaded in a separate program, *jLoad.mps* as shown. A sample of the output (using word stems) is shown in Figure 49.

```mumps
#!/usr/bin/mumpsRO
# jaccard-tt.mps December 12, 2014
# Copyright 2014 Kevin C. O'Kane

if '$data(JMIN) set JMIN=0.1

# input format: wgt word1 word2 co-occur-ww-w2 ^df(w1) ^df(w2)
# output format: jaccardtt word1 word2
```
for do
  . read a
  . if '$test break
  . set w1=$piece(a," ",2)
  . set w2=$piece(a," ",3)
  . set co=$piece(a," ",4)
  . set d1=$piece(a," ",5)
  . set d2=$piece(a," ",6)
  . set jc=co/(d1+d2-co)
  . if jc<JMIN quit
  . write $justify(jc,6,3)," ",w1," ",w2,!

#!/usr/bin/mumps
# jLoad.mps December 10, 2014
# Copyright 2014 Kevin C. O'Kane
# input format: #co­occur w1 w2

for do
  . use 5
  . read a
  . if '$test break
  . set wgt=$piece(a," ",1)
  . set w1=$piece(a," ",2)
  . set w2=$piece(a," ",3)
  . set ^jtt(w1,w2)=wgt

Program 32 jaccardtt.mps and jLoad.mps

0.435 diastolic systolic
0.438 hypertensive normotensive
0.455 echocardiography two-dimension
0.462 imag resonance
0.462 lymph node
0.474 gene mrna
0.500 doppler echocardiography
0.500 endothelial endothelium
0.500 imag magnetic
0.500 myocardium reperfusion
0.533 compute tomography
0.556 dismutase superoxide
0.556 hypoxic normoxic
0.571 messenger mrna
0.588 mammography screeene
0.600 colit ulcerative
0.615 hypoxia hypoxic
0.714 dodecyl electrophoresis
0.714 fertilization oocyte
0.714 mitr valve
0.875 magnetic resonance
1.000 adenylate cyclase
1.000 aureus staphylococcus
1.000 expiratory forc

Figure 49 Jaccard Term Term Relationships

11.3.3 Term Proximity

Another modification to improve the detection of term-term relationships is shown in Program 33. It involves retaining, during initial document scanning, the relative
positions of each term in each document in the collection. Subsequently, when calculating the term-term matrix, term proximity can be taken into account.

The assumption is that terms that co-occur frequently near one another and predominantly in the same order are in fact closely related.

Recall that when we initially scanned the documents stored the position of each term relative to the start of the document in the third index:

\[ ^{doc}(DocNbr,Word,Position) \]

The value for Position indicates if the term is the first, second, third, and so forth term in the document. Thus, this records, for each term in each document, the position of each instance of each word relative to the beginning of the document.

With this information, it becomes possible to calculate the distance between co-occurring terms as well as note which term appears before the other. Terms that frequently co-occur close to one another and normally in the same order are more likely to be related than terms that co-occur at a distance and in no preferred order with respect to one another.

A revised term-term matrix calculation taking into account proximity and order is calculated as follows:

1. for each document \( k \)
2. for each term \( i \) in \( k \)
3. for each other term \( j \) in \( k \) where term \( j \) is alphabetically greater than term \( i \)
4. for each position \( m \) of term \( i \)
5. for each position \( n \) of term \( j \)
6. calculate a weight based on the distance between the terms and add this to the global array \( ^{ttp}(i,j) \)
7. add to global array \( ^{ttp}(i,j,1) \) the signed difference between the terms \( n-m \). That is, the distance in word units between the two terms.

The weight added to \( ^{ttp}(i,j) \) in step 6 is calculated with the formula:

\[
\text{set } dd = \frac{z\log(1/|m-n|)*20+1}{1}
\]

which yields, for various distances as shown in Figure 50.

Thus, in Program 33, terms immediately next to one another receive a score of 3 while terms more than eleven positions apart receive a score of zero.

For each pair \( \{i,j\} \) the third level index is the sum of signed distances between \( m \) and \( n \). After all co-occurrences have been summed, a positive or negative value for this term indicates a preference as to which term appears first most often. Values of zero or near zero indicate that the terms appear in no specific order relative to one another.

55 Line 25, Program 19, page 81.
In the script file `index.script` the program `proximity.mps` works on a copy of the database so that it may be executed in parallel. It's output (to `stdout`) is subsequently sorted into a final result file.

```
$zabs(m-n)=1  result  3
$zabs(m-n)=2  result  2
$zabs(m-n)=3  result  2
$zabs(m-n)=4  result  1
$zabs(m-n)=5  result  1
$zabs(m-n)=6  result  1
$zabs(m-n)=7  result  1
$zabs(m-n)=8  result  1
$zabs(m-n)=9  result  1
$zabs(m-n)=10 result  1
$zabs(m-n)=11 result  1
$zabs(m-n)=12 result  0
$zabs(m-n)=13 result  0
$zabs(m-n)=14 result  0
$zabs(m-n)=15 result  0
```

Figure 50 Proximity calculations
#!/usr/bin/mumps
#
proximity.mps Feb 7, 2014
#
Copyright 2014 Kevin C. O'Kane
#	ttx term-term correlation matrix
#
calculate term-term proximity coefficients within env words

database "copy0"

if '$data(%1) write "Missing parameter",! halt

kill ^ttp /* delete any old term-term correlation matrix

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# for each document k, sum the co-occurrences of words i and j
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# for each document k

set k=""
for do
  set k=$order(^doc(k))
  if k="" break

# for each term i in p k

  set i=""
  for do
    set i=$order(^doc(k,i))
    if i="" break

  # for each other term j in doc k
  .. set j=i
  .. for do
    .. set j=$order(^doc(k,j))
    .. if j="" break

  # for each position m of term i in doc k
  .... set m=""
  .... for do
    .... set m=$order(^doc(k,i,m))
    .... if m="" break

  # for each position n of term j in doc k
  ..... set n=""
  ..... for do
    ..... set n=$order(^doc(k,j,n))
    ..... if n="" break

# calculate and store weight based on proximity

  ..... set dd=$zlog(1/$zabs(m-n)*20+1)
  ..... if dd<1 quit
  ..... if '$data(^ttp(i,j)) set ^ttp(i,j)=dd,^ttp(i,j,1)=n-m
  ..... else set ^ttp(i,j)+dd,^ttp(i,j,1)+=(n-m)

for i=$order(^ttp(i)) do
  for j=$order (^ttp(i,j)) do
    .. if ^ttp(i,j)<%1 kill ^ttp(i,j) quit
Program 33 *proximity.mps*

Figure 51 contains some of the top ranking proximity based term-term proximity based correlation calculation from Program 33. All terms are shown in stem form.

In Figure 51 the first column gives the sum of the proximity weight scores while the last column gives net difference in the positions. The terms on each line are presented in alphabetic order. However the last column indicates which term tends to precede which. A large positive number favors the order shown while a large negative number favors the reverse order. A number near zero indicates that the terms are likely to appear in either order.

<table>
<thead>
<tr>
<th>Position</th>
<th>Term</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1721</td>
<td>infarct myocardial</td>
<td>-465</td>
</tr>
<tr>
<td>1445</td>
<td>dens lipoprotein</td>
<td>73</td>
</tr>
<tr>
<td>1184</td>
<td>abstract word</td>
<td>1184</td>
</tr>
<tr>
<td>1128</td>
<td>magnetic resonance</td>
<td>376</td>
</tr>
<tr>
<td>1095</td>
<td>cord spin</td>
<td>-309</td>
</tr>
<tr>
<td>1057</td>
<td>lymph node</td>
<td>313</td>
</tr>
<tr>
<td>964</td>
<td>arthrit rheumatoid</td>
<td>-307</td>
</tr>
<tr>
<td>936</td>
<td>female male</td>
<td>-43</td>
</tr>
<tr>
<td>878</td>
<td>cholesterol lipoprotein</td>
<td>43</td>
</tr>
<tr>
<td>872</td>
<td>blind double</td>
<td>-304</td>
</tr>
<tr>
<td>859</td>
<td>compute tomography</td>
<td>255</td>
</tr>
<tr>
<td>811</td>
<td>imag resonance</td>
<td>-189</td>
</tr>
<tr>
<td>762</td>
<td>carry out</td>
<td>262</td>
</tr>
<tr>
<td>742</td>
<td>determine whether</td>
<td>321</td>
</tr>
<tr>
<td>733</td>
<td>cholesterol dens</td>
<td>-137</td>
</tr>
<tr>
<td>686</td>
<td>mitr valve</td>
<td>114</td>
</tr>
<tr>
<td>660</td>
<td>state unit</td>
<td>-240</td>
</tr>
<tr>
<td>649</td>
<td>centre nerv</td>
<td>221</td>
</tr>
<tr>
<td>639</td>
<td>amino sequence</td>
<td>-7</td>
</tr>
<tr>
<td>638</td>
<td>aortic valve</td>
<td>194</td>
</tr>
<tr>
<td>633</td>
<td>guinea pig</td>
<td>32</td>
</tr>
<tr>
<td>629</td>
<td>care health</td>
<td>-97</td>
</tr>
<tr>
<td>625</td>
<td>imag magnetic</td>
<td>-439</td>
</tr>
<tr>
<td>610</td>
<td>cystic fibrosis</td>
<td>179</td>
</tr>
<tr>
<td>587</td>
<td>head neck</td>
<td>149</td>
</tr>
<tr>
<td>566</td>
<td>death sudden</td>
<td>-191</td>
</tr>
<tr>
<td>563</td>
<td>biopsy specimen</td>
<td>230</td>
</tr>
<tr>
<td>558</td>
<td>electron microscopy</td>
<td>111</td>
</tr>
<tr>
<td>548</td>
<td>marrow transplant</td>
<td>113</td>
</tr>
<tr>
<td>544</td>
<td>tract urin</td>
<td>-156</td>
</tr>
<tr>
<td>544</td>
<td>diabete mellitus</td>
<td>161</td>
</tr>
<tr>
<td>531</td>
<td>erratum publish</td>
<td>-177</td>
</tr>
<tr>
<td>530</td>
<td>sensit specificity</td>
<td>210</td>
</tr>
<tr>
<td>530</td>
<td>morbid mort</td>
<td>110</td>
</tr>
<tr>
<td>522</td>
<td>resistance vascular</td>
<td>-151</td>
</tr>
<tr>
<td>522</td>
<td>excretion urin</td>
<td>-283</td>
</tr>
<tr>
<td>522</td>
<td>ejection fraction</td>
<td>130</td>
</tr>
<tr>
<td>519</td>
<td>cerebrospinal fluid</td>
<td>106</td>
</tr>
<tr>
<td>517</td>
<td>coli escherichia</td>
<td>-246</td>
</tr>
<tr>
<td>513</td>
<td>diastolic end</td>
<td>-58</td>
</tr>
<tr>
<td>508</td>
<td>man old</td>
<td>-207</td>
</tr>
<tr>
<td>500</td>
<td>anti monoclon</td>
<td>-97</td>
</tr>
<tr>
<td>475</td>
<td>state steady</td>
<td>-174</td>
</tr>
<tr>
<td>470</td>
<td>erythematosus lupu</td>
<td>-134</td>
</tr>
<tr>
<td>469</td>
<td>randomize trial</td>
<td>143</td>
</tr>
</tbody>
</table>
A proximity based term-term matrix can yield substantially better results although it is considerably more expensive to calculate in terms of time and space. The example was calculated on the first 10,000 documents of the data base. Again, it should be noted, that uncovering term-term relationships need not involve analysis of the entire collection, just a representative sample of it.

11.3.4 Adding Phrases to the Collection

Phrases may be added to the document-term matrix, the term-document matrix and to augment search queries. Several methods of detecting term-term relationships were discussed above. Of these, we will use the phrases based on word proximity.
determined by *proximity.mps* shown in Program 33 shown on page 115 rather than to other methods discussed above. Phrases are added to the collection in *proxLoad.mps* as shown in Program 34.

Program 34 creates two new global array matrices $^ttp()$ and $^phrase()$. It also adds phrase entry terms to $^doc()$, $^index()$, $^df()$ and $^dict()$.

### 11.4 Term-Term clustering

Related terms can be grouped into clusters using the results of the term-term matrix. The program to do this is shown in Program 35. This program implements a simple single link clustering algorithm. That is, a term is added to a cluster if it is related to at least one term already in the cluster.

The input to this program is the term-term relationships contained in $^tt()$ calculated in Program 29 above. The other term-term relationships may be tried as well although the proximity based relations give poor results.

The program works as follows. A temporary file is created whose lines consist of the term-term correlation score followed by the two correlated words, blank separated. Words whose correlation scores are below a threshold are not written. The file is sorted according to the correlation score in reverse order (the highest score is at the beginning of the file, the lowest at the end).

The file is read line by line. Initially, the highest scoring word pair are placed in a cluster of their own.

Next successive lines are read. The words on each line compared to the words in the existing clusters. If one of the two words is in a cluster, the other word is added to the cluster. If neither word is in an existing cluster, a new cluster is created containing the two words.

At the end, the clusters are printed.

1 #!/usr/bin/mumps
2 # clustertt.mps February 14, 2014
3 # Copyright 2014 Kevin C. O'Kane
4
5 kill `clstr
6 kill `x
7
8 if `$data(1) set min=.1
9 else set min=1
10
11 open 1:"TTtmp.tmp,new"
12 use 1
13 for w1=$order(`tt(w1)) do
14 . for w2=w1:$order(`tt(w1,w2)):"" do
15 .. if `jtt(w1,w2)<min quit
16 .. write `tt(w1,w2)," ",w1," ",w2,
17 close 1
18
19 shell sort -n -r < TTtmp.tmp > TTtmp.sorted.tmp
20 open 1:"TTtmp.sorted.tmp,old"
21
set c=1
for do
  . use 1
  . read a // correlation word1 word2
  . if '$test break
  . set score=$p(a," ",1)
  . set w1=$p(a," ",2)
  . set w2=$p(a," ",3)
  . if w1=w2 quit
  . set f=1

# ^x() is a two dimensional array that contains, at the second level,
# a list of clusters to which the word (w1) belongs
# ^cluster() is the cluster matrix. Each row (s) is a cluster
# numbered 1,2,3 ... The second level is a list of the words
# in the cluster.

# The following
# code runs thru all the clusters first for w1 (w1) and
# adds w2 to those clusters w1 belongs to. It
# repeats the process for w2. If a word pair are not
# assigned to some cluster (f=1), they are assigned to a new
# cluster and the cluster number is incremented (c)

  . if $d(^x(w1)) for s=$order(^x(w1,s)) do
    .. set ^clstr(s,w2)=""
    .. set ^x(w2,s)=""
    .. set f=0

  . if $d(^x(w2)) for s=$order(^x(w2,s)) do
    .. set ^clstr(s,w1)=""
    .. set ^x(w1,s)=""
    .. set f=0

  . if f do
    .. set ^clstr(c,w1)="" set ^x(w1,c)=""
    .. set ^clstr(c,w2)="" set ^x(w2,c)="
    .. set c=c+1

# print the clusters
  close 1
  use 5
  write "number of clusters: ",c,!!
  for cx=$order(^clstr(cx)) do
    . write "cluster: ",cx,!
    . for w1=$order(^clstr(cx,w1)) do
      .. use 5 write w1," 
  . write !!
  halt

kill ^dt
kill ^dtt
for d=$order(^doc(d)) do
  . for w=$order(^doc(d,w)) do
    .. if $data(^x(w)) do
      ... for c=$order(^x(w,c)) do
        .... if $data(^dt(d,c)) set ^dt(d,c)=^dt(d,c)+1
        .... else set ^dt(d,c)=1
Figure 52 gives a sample of the output of the above. In this example, there 438 clusters were found.

164 cluster
adenocarcinoma adenoma colonic colorectal crohn hypercalcemia hyperparathyroidism metastasis parathyroid parathyroidectomy pth

165 cluster
anemia aplastic ferritin iron overload sickle transferrin

166 cluster
extension femore flexion insertion invasion motion rotation

167 cluster
angina anginal angiographic anomaly arrest arrhythmia atrioventricular atrium axis cardiomyopathy catheterization cava congestive echocardiographic echocardiography ejection glob hypertrophy radionuclide sept shortene ventriculography

168 cluster
abscess arteriovenous clos closure cosmetic fistula flap incision suture

169 cluster
ace acetylcholine ach adenosine adherence adrenergic angina anginal angiographic angioplasty angiotensin antagonize arrest arrhythmia atherosclerotic atrioventricular attenuate autonomic balloon beat blocker canine capill cardiomyopathy circumflex compete congestive conscious contractile contractility contraction creatine cyclase descend dihydropyridine diuretic dog ejection endothelial glob glomerular guinea inflation inotropic intra isometric junction lad mongrel myocardium narrow nephron norepinephrine occlude ouabain patent perfus phen tolamine pig postischemic potency pressor propranolol pump radionuclide relaxation reperfus reperfusion restenosis resuscitare revascularization shortene stenos strip sympathetic transluminal treadmill vaso constrictor ventriculography wire yohimbine

170 cluster
acting activate activator adenosine adenylate adrenergic amp autoimmune band blot camp carboxyl cdna complementary construct cyclase cyclic deduce deletion dodecyl domain dot enhancer epitope exon family fibroblast fibronectin forskolin glycoprotein granulocyte gt11 homolog homology hybridization killer kilobase lambda lectin libr mammalian mononuclear monophosphate mutant northern nucleotide oligonucleotide perfus phosphodiesterase polypeptide precursor residue restriction screen sera substitution sulfate synthesize terminu tran transcription transmembrane vector western

171 cluster
allele allogeneic basement bear biologic blot carrier cd3 cd4 cd8 comple haplotype helper histocompatible hybridize immunofluorescence l3t4 lambda leu linkage lyt mitogen mitogenic monocyte phenotype phytohemagglutinin polypeptide rearrange restriction southern spleen subset
11.4.1 Adding Clusters to the Collection

It is possible (but not done in the toolkit) to replace original terms in the documents that occur in one or more clusters with a token indicating the cluster(s) to which they belong and then using these tokens as though they were terms. This approach can increase recall considerably but at the expense to precision. Alternatively, words from clusters can be added to search vectors.
12 Document-Document Matrix

It is also possible to construct a Document-Document Matrix giving the correlation between those documents which have significant similarities with one another as shown in the programs listed in Program 36. Such a matrix can be used to generate document clusters and for purposes of document browsing by permitting the user to navigate related documents to the one being viewed. That is, if a user finds one of the retrieved articles of particular interest, a Document-Document matrix can be used to quickly identify other documents related to the document of interest.

The code in Program 36 only calculates the cosines between documents that share at least one term in common rather than calculating the cosines between all possible documents. If two documents do not share at least one term in common, they have a cosine of zero.

To improve speed, the process is divided into two parts: docdoc1.mps and docdoc2.mps. Program docdoc1.mps runs in parallel with other tasks and operates on a copy of the database. It uses bash shell commands to count the number of document-document co-occurrences. Program docdoc1.mps writes its output to a sequential file.

Subsequently, docdoc2.mps reads this file, stores the results in a global array and formats a display of the related documents. These steps cannot be done by the docdoc1.mps since it is operating only on a temporary copy of the master database.

```mumps
1 /usr/bin/mumpsRO
2 # docdoc1.mps Feb 7, 2014
3 # Copyright 2014 Kevin C. O'Kane
4
5 if '%$data(%1) write "Missing parameter",! halt
6
database "copy1"
7
8 set min=%1
9 if min="" set min=5
10
11 open 1:"dd.tmp,new"
12
13 use 1
14 for w=$order(^index(w)) do
15 . for d1=$order(^index(w,d1)) do
16 .. for d2=d1:$order(^index(w,d2)):"" do
17 ... if d1=d2 quit
18 ... write d1," ",d2,!
19
20 close 1
21
22 shell sort < dd.tmp | uniq -c | sort -rn > dd1.tmp
23
24 open 1:"dd1.tmp,old"
25
26 # stop reading when threshold encountered (break)
27
28 for do
29 . use 1 read a
30 . if '%$test break
```
12.1 Document Clustering

The program `clusterdd.mps` in Program 37 uses a single link clustering technique similar to that used in the term clustering discussed above (Section 11.4 on page 117).

The program first generates and then reads a file of document-document correlations sorted in reverse (highest to lowest) correlation order. There is a filter on line 19 which ignores document-document correlation cosines that are below a given threshold (`min`, a command line parameter).
open 1:"ddtmp.tmp,new"
use 1
for d1=$order(^dd(d1)) do
  for d2=d1:$order("dd(d1,d2)"":"" do
.. if "dd(d1,d2)<min quit
.. write "dd(d1,d2)",",d1,"",d2,!
close 1
shell sort -n -r < ddtmp.tmp > ddtmp.sorted.tmp
open 1:"ddtmp.sorted.tmp,old"
set c=1
for do
  use 1
  read a // correlation doc1 doc2
  if "$test break
  set score=$p(a,"",1)
  set seq1=$p(a,"",2)
  set seq2=$p(a,"",3)
  if seq1=seq2 quit
  set f=1

  if $d(^x(seq1)) for s="":$order(^x(seq1,s)):"" do
    set ^clstr(s,seq2)="
    set ^x(seq2,s)="
    set f=0
  end
  if $d(^x(seq2)) for s="":$order(^x(seq2,s)):"" do
    set ^clstr(s,seq1)="
    set ^x(seq1,s)="
    set f=0
  end
  if f do
    set ^clstr(c,seq1)="
    set ^x(seq1,c)="
    set ^clstr(c,seq2)="
    set ^x(seq2,c)="
    set c=c+1
  end

# print the clusters
close 1
use 5
for cx=$order(^clstr(cx)) do
  write "cluster ",cx,!
  for seq1=$order(^clstr(cx,seq1)) do
    use 5 write seq1,?8,^title(seq1),!
    set ^ct(cx,seq1)="
    if $data(^dc(seq1,cx)) set ^dc(seq1,cx)=^dc(seq1,cx)+1
    else set ^dc(seq1,cx)=1
  end
  write !
open 1:"document-clusters-by-title.txt,new"
use 1
Figure 53 gives a sample output from Program 37.

Document clusters can be used during retrieval to display documents that might be related to documents retrieved as a result of a user query.

A program to generate clusters of clusters is given in Program 38. This program can be time consuming on a large collection. This program builds, for each cluster, a centroid vector and then clusters the centroid vectors.
#!/usr/bin/mumps
# hyperCluster.mps Feb 14, 2014
# Copyright 2014 Kevin C. O'Kane

kill ^hc
set c=0,k=0

if '$data(%1) write "Missing parameter",! halt
if '$data(%2) write "Missing parameter",! halt

set min=%1
set wgt=%2

if min="" set min=0.4
if wgt="" set wgt=.4

# read the level one clusters and build
# centroid vectors

for do
  . read a
  . if '$test break
  . if a="" quit
  . set t=$p(a," ",1)
  . if t="cluster" do quit
  .. for w=$order(^hc(c,w)) set ^hc(c,w)=^hc(c,w)/k
  .. set c=c+1,k=0
  .. quit
  . for w=$order(^doc(t,w)) do
    .. if $data(^hc(c,w)) set ^hc(c,w)=^hc(c,w)+^doc(t,w)
    .. else set ^hc(c,w)=^doc(t,w)
    .. set k=k+1
  .. quit

for i=1:1:c for w=$order(^hc(i,w)) if ^hc(i,w)<wgt kill ^hc(i,w)

# write centroid vectors
write !,"Centroid vectors",!
for i=1:1:c do
  . write i," 
  . for w=$order(^hc(i,w)) write w," (",j(^hc(i,w),3,2),") 
  . write !

open 1:"hypertmp.tmp,new"

# calculate cluster similarities
write !,"Cluster similarities:",!
for i=1:1:c do
  . for j=i+1:1:c do
    .. s x=$zzCosine(^hc(i),^hc(j))
    .. if x<min quit
    .. use 5 write i," ",j," ",x!,
    .. use 1 write x," ",i," ",j,!
  .. quit
use 5
close 1

kill ^clstr
kill ^x
shell sort -n -r < hypertmp.tmp > hypertmp.sorted.tmp

open 1:"hypertmp.sorted.tmp,old"
set c=1
for do
  . use 1
  . read a // correlation doc1 doc2
  . if '$test break
  . set score=${p(a," ",1)}
  . set seq1=${p(a," ",2)}
  . set seq2=${p(a," ",3)}
  . if seq1=seq2 quit
  . set f=1
.
# ^x() is a two dimensional array that contains, at the second level,
# a list of clusters to which the document number (seq1) belongs
# ^cluster() is the cluster matrix. Each row (s) is a cluster
# numbered 1,2,3 ... The second level is a list of the document
# numbers of those documents in the cluster. The following
# code runs thru all the clusters first for doc1 (seq1) and
# adds seq2 (doc2) to those clusters doc1 belongs to. It
# repeats the process for seq2 (doc2). If a doc pair are not
# assigned to some cluster (f=1), they are assigned to a new
# cluster and the cluster number is incremented (c)
.
  . if $d(^x(seq1)) for s=${order(^x(seq1,s))} do
    .. set ^clstr(s,seq2)="
    .. set ^x(seq2,s)="
    . set f=0
  . if $d(^x(seq2)) for s=${order(^x(seq2,s))} do
    .. set ^clstr(s,seq1)="
    .. set ^x(seq1,s)="
    . set f=0
  . if f do
    .. set ^clstr(c,seq1)=" set ^x(seq1,c)="
    .. set ^clstr(c,seq2)=" set ^x(seq2,c)="
  . set c=c+1
.
# print the clusters

close 1
use 5
write !!,"Number of clusters: ",c-1,!!
for cx=${order(^clstr(cx))} do
  . write "cluster ",cx,!
  . for seq1=${order(^clstr(cx,seq1))} do
    .. write "base cluster ",seq1,!
  . for cz=${order(^ct(seq1,cz))} do
    ... write seq1,?8,^title(cz),!
  . write !
13 Retrieval

Having constructed the basic matrices and vectors, we now explore some simple retrieval methods.

13.1 Retrieval Using the Doc-Term Matrix

A very simple program to scan the document-term matrix looking for documents that have terms from a query vector is given in Program 39. In this program, a line of query terms is read. The terms are converted to lower case and stemmed. Next, the documents are scanned sequentially and, if found to have any one of the query terms, the title and document number are printed. If a document contains more than one query vector term, it will be displayed twice. This program may operate in read-only mode as it does not change the database.

```
#!/usr/bin/mumpsRO
# query1.mps Copyright 2014 Kevin C. O'Kane

write "Enter search terms:"
set i=$zzInput("w")-1
for j=0:1:i do
  set a=$zlower(w(j))
  set a=$zstem(a)
  set query(a)="

set t1=$zd1
for i=$order(^doc(i)) do
  for j=$order(query(j)) do
    .. if $data(^doc(i,j)) write i,?8,"",^title(i),! break

write !,"Time elapsed: ",$zd1­t1,
halt
```

Enter search terms:
epithelial fibrosis
1228 Energy expenditure of patients with cystic fibrosis.
1345 The occurrence of vasculitis in perianeurysmal fibrosis.
1486 Cellular and non-cellular compositions of crescents in human glomerulonephritis.
1504 Epithelial ovarian tumor in a phenotypic male.
1560 Neonatal long lines for intravenous antibiotic therapy in cystic fibrosis [letter]
1871 Intraperitoneal cis-platinum as salvage therapy for refractory epithelial ovarian
carcinoma.
1881 The effect of a collagen bandage lens on corneal wound healing: a preliminary
1882 Removal of lens epithelial cells by ultrasound in endocapsular cataract surgery.
1886 Oncocytic carcinoma of the plica semilunaris with orbital extension.
1904 Retinal pigment epithelial cells release inhibitors of neovascularization.

Program 39 Simple retrieval program

13.2 Retrieval Using the Term-Doc Matrix

Unfortunately, Program 39 is very slow because all documents must be inspected. Program 40, on the other hand, is a simple program that uses the term-document matrix to more quickly search for documents that contain one or more search terms.

---

56 Note: titles may not necessarily contain the search terms.
In this program, only those documents that actually contain a query term are inspected. When a document containing one of the search terms is found, its document number and title are printed to stdout. However, as with Program 39, if a document contains more than one query term, its title will be printed twice.

However, this can be fixed by routing the output through sort and uniq which will eliminate duplicates as well as give a count, for each title, of the number of times the title appeared. This number is also the number of terms from the query vector found in each document. This can be done with a command such as:

```
query2.mps < input_words.txt | sort | uniq -c | sort -nr > results
```

Where input_words.txt contains the search terms (one per line). The output from query2.mps is sorted so that any duplicate document number / title combinations will be on consecutive lines. The number of times a document number / title combination line appears indicates the number of keywords found in the document.

The uniq program removes duplicate lines and the -c option causes the remaining line to be prefixed by a count of the number of times the line appeared in the input file.

The final sort sorts the results based on the first field on each line which is now the count of the number of times the title appeared. The sort is such that those titles with the highest counts appear first and those with lower counts appear last (the -n option causes a numeric sort and the -r option reverses the order so that results display with the titles with higher counts appear first).

This is a primitive form of weighted results. Those documents with more search terms in common with the query are shown first. Many other weighting schemes are possible.

```
#!/usr/bin/mumps
#query2.mps Copyright 2014 Kevin C. O'Kane
write "Enter search terms:","!
set i=$zzInput("w")-1
for j=0:1:i do
 . set a=$zlower(w(j))
 . set a=$zstem(a)
 . set query(a)="

set t1=$zd1
for w=$order(query(w)) do
 . for d=$order(^index(w,d)) do
 .. write d,?8,"",^title(d),!
write !,"Time elapsed: ",$zd1-t1,!
```

Enter search terms:
epithelial fibrosis
1486 Cellular and non-cellular compositions of crescents in human glomerulonephritis.
1504 Epithelial ovarian tumor in a phenotypic male.
1871 Intraperitoneal cis-platinum as salvage therapy for refractory epithelial ovarian

Note: the order of the titles will normally be different than in Program 39.
1881  The effect of a collagen bandage lens on corneal wound healing: a preliminary
1882  Removal of lens epithelial cells by ultrasound in endocapsular cataract surgery.
1886  Oncocytic carcinoma of the plica semilunaris with orbital extension.
1904  Retinal pigment epithelial cells release inhibitors of neovascularization.
1924  Ocular pathologic findings in neonatal adrenoleukodystrophy.
2019  Metastatic adeno or undifferentiated carcinoma from an unknown primary site—natural

Elapsed time: 0

Program 40 Term–Doc matrix search

13.3 Weighted Retrieval using the Term–Doc Matrix

Program 41 is similar to Program 40 but rather than print the title of any document
which contains any search term, it creates a primitive score for each document to
indicate how similar the document is to the query vector. It does this by summing the
weights of those terms from each document vector which are in common with the
terms in the query vector. It then prints the titles of the 10 documents with the highest
scores.

This is another form of primitive search weighting. If a title has several terms in
common with the query vector and if these are highly weighted terms, the title is
deemed to be a better match with the query that if it had only a few low weighted
terms in common.

Note that in Program 41 that the $job builtin variable returns the process id of the
currently running program. This is unique number and it is used to name the
temporary file that contains the unsorted results.

Program 41 creates a temporary global array vector (^tmp) whose indices are the
document numbers of those documents that have one or more terms in common with
the query vector. The value of an element of the ^tmp() vector is the sum of the
weights of the query terms in common with the document.

For example, if document 10 has three terms in common with the query and the
terms have weights 2, 3 and 5, respectively, the value in ^tmp(10) will be 10. The
weights in this example are assumed to be IDF weights which were discussed above
although other weighting schemes may be used.

Program 41, like Program 40, iterates through the query vector and, for each word in
the query vector, finds all those documents that contain the term. The weights of the
query vector term in the document are added to the ^tmp() vector for the document.

After the totals are calculated in ^tmp, the scores (from ^tmp) and the
corresponding titles are written to the temporary file. This is then sorted by score and
the top ten are printed.

1 #!/usr/bin/mumps
2 # query3.mps Copyright 2014 Kevin C. O'Kane
3
4 kill ^tmp
5
6 write "Enter search terms:","!
7 set i=$zzInput("w")-1
8 for j=0:1:i do
9 . set a=$zlower(w(j))
10 . set a=$zstem(a)
11 . set query(a)="
12
13 if $order(query(""))="" halt
14
15 set t1=$zd1
16
17 for w=$order(query(w)) do
18 . for i=$order(^index(w,i)) do
19 .. if $data(^tmp(i)) set ^tmp(i)=^tmp(i)+^index(w,i)
20 .. else set ^tmp(i)=^index(w,i)
21
22 set fn=$job_",new"
23 open 1:fn
24 use 1
25 for i=$order(^tmp(i)) do
26 . use 1 write $justify(^tmp(i),6,2)," ",^title(i),!
27 close 1
28 use 5
29 set cmd="sort -n "<_$job_" | tail -10 ; rm "<_$job
30 shell &~cmd~
31
32 write !,"Elapsed time: ",zd1-t1,!
33 halt

Enter search terms:
zinc

23.64 Effect of maternal zinc supply on blood and tissue metallothionein I concentrations
23.64 Observations of serum trace elements in chronic lymphocytic leukemia.
23.64 Zinc absorption in humans from meals based on rye, barley, oatmeal, triticale and
35.46 Cadmium-induced immunopathology is prevented by zinc administration in mice.
41.37 Magnesium and zinc status during the menstrual cycle.
59.10 The relationship between cadmium, zinc, and birth weight in pregnant women who smoke.
76.83 Iron and zinc concentrations and 59Fe retention in developing fetuses of zinc-defic
82.74 The effect of smoking on placental and fetal zinc status.
82.74 Zinc deficiency increases the osmotic fragility of rat erythrocytes.
100.47 Dietary conditions influencing relative zinc availability from foods to the rat and

Program 41 Weighted Term-Doc matrix search

### 13.4 Simple Cosine Weighted Retrieval Part 1

The previous programs used relatively simple weighting schemes based on counts
and weight sums. These can be useful but fail deal with issues such as query and
document vector lengths, among others.

More sophisticated methods take into account the number of terms in the search
vector, the number of terms in the document vector and the number of terms in the
intersection of the query and document vector.

Program 42 uses a more sensitive similarity metric. It reads a set of query words
into a query vector then calculates the cosines between the query and each
document vector and, for those documents where the cosine is greater than zero, it
writes the cosine and title to a temporary file. When all documents have been

58 See Basic Similarity Functions on page 73.
processed, it sorts the results in the temporary file based on their cosine scores and
prints the titles of the 10 documents with the highest scores. A shell script is used to
sort the result. Output from the shell script is written directly to the user’s console.

1 #!/usr/bin/mumps
2 # simpleRetrieval.mps Copyright 2014 Kevin C. O’Kane
3
4 kill ^query
5
6 write "Enter search terms: ", !
7
8 set i=$zzInput("w")-1
9 for j=0:1:i do
10 . set a=$zlower(w(j))
11 . set a=$zstem(a)
12 . set ^query(a)=1
13
14 if $order(^query(""))="" halt
15
16 write "Query is: ",
17 for w=$order(^query(w)) write w," ",
18 write !
19
20 set time0=$zd1
21
22 set fn=$job_",new"
23 open 1:fn
24 use 1
25
26 set x=0
27 for i=$order(^doc(i)) do // calculate cosine between query and each doc
28 . if i="" break
29 . set c=$zzCosine(^doc(i),^query)
30
31 # If cosine is > zero, put it and the doc offset (^doc(i)) into an answer vector.
32 # Make the cosine a right justified string of length 5 with 3 digits to the
33 # right of the decimal point. This will force numeric ordering on the first key.
34
35 . if c>0 write $justify(c,5,3),",^title(i),!, set x=x+1
36
37 use 5
38 close 1
39
40 write x," documents found - showing top 10 results:"!, !
41
42 set cmd="sort -n <"_.job_" | tail -10 ; rm "_.job
43 shell &~cmd
44
45 write !,"Time used: ",$zd1-time0," seconds",!
46
47 halt

Enter search terms:
zinc
Query is: zinc
27 documents found - showing top 10 results:

0.438 Observations of serum trace elements in chronic lymphocytic leukemia.
0.482 Pulmonary involvement in zinc fume fever.
0.511 Magnesium and zinc status during the menstrual cycle.
13.5 Simple Cosine Weighted Retrieval Part 2

As before, program 42 is slow because it calculates the cosines between all documents and the query vector. In fact, most documents contain no words in common with the query vector and, consequently, their cosines are zero. Thus, a possible speedup technique would be to only calculate the cosines between the query and those documents that contain at least one term in common with the query vector.

We do this by first constructing a vector containing document numbers of those documents with at least one term in common with the query. This is shown in Program 43.

For each query term \( w \), the document numbers \( d \) from row \(^{\text{index}}(w)\) are used as indices to create elements of \(^{\text{tmp}}(d)\). No value is stored at the nodes as the index (the document number) is the information in which we are interested. The \(^{\text{tmp}}()\) vector a list of document numbers of those documents containing at least one query term. If a document has several query terms, it still has only one element in \(^{\text{tmp}}()\). The number of elements of \(^{\text{tmp}}\) will normally be considerably smaller than the total number of documents\(^{59}\).

When all query words have been processed, the temporary vector \(^{\text{tmp}}\) indices are those document numbers of documents that contain at least one query term. The cosines are then calculated between the query vector and only those documents whose document numbers are indices of \(^{\text{tmp}}()\).

Program 43 yields the same results as above but takes less than 1 second.

59 Unless the query vector consists of a large list of terms that are widely distributed among the document collection. This would be a very non-specific query, to say the least!
. set a=$zlower(w(j))
.set a=$zstem(a)
.set `query(a)=1

if $order(`query`(""))=" halt
write "Query is: "
for w=$order(`query(w)) write w," "
write !

set time0=$zd1

# Find documents containing one or more query terms.
for w=$order(`query(w)) do
  for d=$order(`index(w,d)) set `tmp(d)="" // retain doc id
use 1
set x=0
for i=$order(`tmp(i)) do // calculate cosine between query and each doc
  set c=$zzCosine(`doc(i),`query) // MDH cosine calculation
  # If cosine is > zero, put it and the doc offset (`doc(i)) into an answer vector.
  # Make the cosine a right justified string of length 5 with 3 digits to the
  # right of the decimal point. This will force numeric ordering on the first key.
  if c>0 write $justify(c,5,3),"",^title(i),! set x=x+1
use 5
close 1
write x," documents found - showing top 10 results:","!
set cmd="sort -n <"$_job_" | tail -10 ; rm "$_job_"
shell &~cmd~
write !,"Time used: ",zd1-time0," seconds","!
halt

Enter search terms:
zinc
Query is: zinc
27 documents found - showing top 10 results:

0.438 Observations of serum trace elements in chronic lymphocytic leukemia.
0.482 Pulmonary involvement in zinc fume fever.
0.511 Magnesium and zinc status during the menstrual cycle.
0.515 Multiple sclerosis and the workplace: report of an industry-based cluster.
0.526 Iron and zinc concentrations and 59Fe retention in developing fetuses of zinc-defic
0.565 Diagnostic value of zinc levels in pleural effusions [letter]
0.628 Zinc deficiency increases the osmotic fragility of rat erythrocytes.
0.641 The relationship between cadmium, zinc, and birth weight in pregnant women who smoke.
0.693 The effect of smoking on placental and fetal zinc status.
0.810 Dietary conditions influencing relative zinc availability from foods to the rat and

Time used: 0 seconds

Program 43 Faster simple retrieval
13.6 Retrieval Enhancements

The code from Program 43 may be enhanced as shown in Program 44. This version, for each query term:

1. gives any other terms in the word stem group for the term,
2. augments the query with related terms, and
3. gives a list of terms that may sound similar to the query term.

The terms added to the query are based on term proximity. The list of sound-alike terms is based on Soundex codes.

```mumps
#! /usr/bin/mumps
# ~/Medline2012/medlineRetrieve.mps Nov 12, 2014
# Copyright 2014 Kevin C. O'Kane

kill ^query
kill ^ans
kill ^tmp

for do // extract query words to query vector
    set w=$zzscanalnum
    if w="" break
# terms
    set w1=$zstem(w)
# unknown ?
    if '$data(^dict(w1)) write ?4,"unknown word: ",w,! quit
    write !,"query term: ",w,!
# stems
    if $data(^stem(w1)) do
        write !,"Words in this stem group are: ",w
        do stems(w1)
    else write ?4,w," has no stem related words"
# related
    write !,"adding related phrase words: ",w
    if '$data(^phrase(w1)) do
        write "none",!
    else do
        set f=1
        for prs=$order(^phrase(w1,prs)) do
            write prs," ",w
            set f=0
        if f write !
        write !
# soundex
    set s=$zzsoundex(w1)
    write ?4,"words that may sound similar to ",w," ": ",w
    if '$data(^sndx(s)) write "none."
    else do
        set f=1
        for ws=$order(^sndx(s,ws)) do
            if $e(w1,1,5)=$e(ws,1,5)&(w1'=ws) do
                set f=0
                do stems(ws)
            if f write "none"
        if f write !
        if $x'=26 write !
# query word
    set ^query(w)=3
```
An example of the output of Program 44 is given in Figure 54 where the input query was:

staphylococcus aureus antibiotics
An asterisk in the *matched keys* section indicates the term is not an expanded word stem. Absence of an asterisk means the term shown is one of the terms that reduced to a matching word stem.

query term: staphylococcus

Words in this stem group are: staphylococcus
adding related phrase words: isolate-staphylococcus methicillin-resistant-staphylococcus
organism-staphylococcus staphylococcus-aureus
staphylococcus-endocarditis staphylococcus-epidermidis
staphylococcus-infections staphylococcus-strain

words that may sound similar to staphylococcus: staphylococcal
staphylococci

query term: aureus

Words in this stem group are: aureus
adding related phrase words: aureus-endocarditis aureus-epidermidis
aureus-infections aureus-strain fibronectin-aureus
isolate-aureus methicillin-aureus methicillin-resistant-aureus
organism-aureus s-aureus staphylococcus-aureus

words that may sound similar to aureus: none

query term: antibiotics

Words in this stem group are: antibiotic antibiotics
adding related phrase words: abscess-antibiotic aminoglycoside-antibiotic
antibiotic-bacteremia antibiotic-beta-lactam
antibiotic-catheter antibiotic-efficacies
antibiotic-infections antibiotic-organism
antibiotic-parenteral antibiotic-prophylaxis
antibiotic-regimen antibiotic-resistant
antibiotic-solution antibiotic-topical
appropriate-antibiotic bacterial-antibiotic
cephalosporin-antibiotic debride-antibiotic
drainage-antibiotic endocarditis-antibiotic
fever-antibiotic isolate-antibiotic
prophylactic-antibiotic strain-antibiotic
susceptibilities-antibiotic susceptibility-antibiotic
valve-antibiotic vancomycin-antibiotic
wound-antibiotic

words that may sound similar to antibiotics: antibacterial
antibacterials antibody-coated antibody-dependent
antibody-mediated antibody-positive

155 documents found.

Doc Rel | Title
---|---
18728 0.308 | Intrinsic methicillin resistance and phage complex of Staphylococcus aureus
matched keys: aureus *methicillin-aureus staphylococcus *staphylococcus-aureus

21989 0.302 | Toxic shock syndrome associated with Staphylococcus aureus enterocolitis
matched keys: aureus staphylococcus *staphylococcus-aureus

7048 0.299 | Evolution of the hyperimmunoglobulin E and recurrent infection HIE  JOB's syndrome in a youn
matched keys: aureus *s-aureus staphylococcus *staphylococcus-aureus
High rate of methicillin resistance of Staphylococcus aureus isolated from hospitalized nursin
matched keys: aureus *aureus-strain *isolate-aureus
*isolate-staphylococcus *methicillin-aureus
*methicillin-resistant-aureus *methicillin-resistant-staphylococcus staphylococcus *staphylococcus-aureus
*staphylococcus-strain

Single and combination-antibiotic therapy for experimental endocarditis caused by methicillin-
matched keys: aureus *methicillin-resistant-aureus
*methicillin-resistant-staphylococcus staphylococcus *staphylococcus-aureus

Role of beta-lactamase in expression of resistance by methicillin-resistant Staphylococcus aur
matched keys: *antibiotic-beta-lactam antibiotic-resistant aureus
*aureus-strain
*isolate-antibiotic *isolate-aureus
*isolate-staphylococcus *methicillin-resistant-aureus
*methicillin-resistant-staphylococcus staphylococcus *staphylococcus-aureus
*staphylococcus-strain *strain-antibiotic

Phenotypic expression and genetic heterogeneity of lincosamide inactivation in Staphylococcus
matched keys: aureus *aureus-strain *isolate-aureus
*isolate-staphylococcus *s-aureus staphylococcus
*staphylococcus-aureus
*staphylococcus-strain

IgE antibodies to Staphylococcus aureus in the hypereosinophilic syndrome
matched keys: aureus staphylococcus *staphylococcus-aureus

Antibiotic treatment of Staphylococcus aureus endocarditis A review of cases
matched keys: *aminoglycoside-antibiotic *antibiotic-beta-lactam
*antibiotic-regimen aureus *aureus-strain staphylococcus *staphylococcus-aureus
*staphylococcus-strain *strain-antibiotic

Acute crescentic glomerulonephritis as a complication of a Staphylococcus aureus abscess of hi
matched keys: aureus staphylococcus *staphylococcus-aureus

*Time used: 0 seconds

Figure 54 Enhanced retrieval output
14 Controlled Vocabulary Indexing Example

Hierarchical indexing schemes such as MeSH (Medical Subject Headings), the Library of Congress Classification System, the ACM's Computing Classification System, the Open Directory Project, and many others are widely used to organize and access information. Thus, we begin with some techniques to manipulate hierarchies. For the most part, these begin with a controlled vocabulary maintained by experts.

14.1 The Medical Subject Headings (MeSH)

MeSH (Medical Subject Headings) is a controlled vocabulary hierarchical indexing and classification system developed by the National Library of Medicine (NLM). The MeSH codes are used to code medical records and literature as part of an ongoing research project at the NLM.

The following examples make use of the 2003 MeSH Tree Hierarchy. Newer versions, essentially similar to these, are available from NLM.

Note: for clinical purposes, the copy of the MeSH hierarchy used here is out of date and should not be employed for clinical decision making. It is used here purely as an example to illustrate a hierarchical index.

The 2003 MeSH file contains approximately 40,000 entries. Each line consists of text along with hierarchical codes describing the subject heading. Figure 55 contains a sample from the 2003 MeSH file.

Body Regions;A01
Abdomen;A01.047
Abdominal Cavity;A01.047.025
Peritoneum;A01.047.025.600
Douglas' Pouch;A01.047.025.600.025
Mesentery;A01.047.025.600.451
Mesocolon;A01.047.025.600.451.535
Omentum;A01.047.025.600.573
Peritoneal Cavity;A01.047.025.600.678
Retroperitoneal Space;A01.047.025.750
Abdominal Wall;A01.047.050
Groin;A01.047.365
Inguinal Canal;A01.047.412
Umbilicus;A01.047.849
Back;A01.176
Lumbosacral Region;A01.176.519
Sacrococcygeal Region;A01.176.780
Breast;A01.236
Nipples;A01.236.500
Extremities;A01.378
Amputation Stumps;A01.378.100

Figure 55 Sample MeSH Hierarchy

60 Warning language required by NLM.
The format of the MeSH table is:

1. a short text description
2. a semi-colon, and
3. a sequence of decimal point separated codes.

An example of the tree structure thus defined can be seen in Figure 57. The hierarchical codes classify and categorize the data. For example, A01 and its sub codes are all body regions; A02 and its descendants concern the Musculoskeletal System; A03 is the top level descriptor for the Digestive System and so forth. The codes beneath each of these sub-divide the top level descriptor. Thus, for A01 we have second level codes such as A01.047 the Abdomen division; A01.176 for the Back; A01.236 for the Breast and so forth. Each of these is further subdivided to a finer levels of detail.

The MeSH codes are an example of a controlled vocabulary. That is, a collection of indexing terms that are preselected, defined and authorized by an authoritative source.

14.2 Building a MeSH Structured Global Array

The goal here is to write a program that will build a global array tree whose structure corresponds to the MeSH hierarchy. In this tree, each successive index in a global array reference will be a successive code from the 2003 MeSH hierarchy. The text part of each MeSH entry will be stored as the global array data value.

To do this, we want to write a program consisting of Mumps assignment statements similar to the fragment shown in Figure 56. In this example, the code identifiers from the MeSH hierarchy become global array indices and the corresponding text becomes assigned values.

```
set `mesh("A01")="Body Regions"
set `mesh("A01","047")="Abdomen"
set `mesh("A01","047","025")="Abdomenal Cavity"
set `mesh("A01","047","025","600")="Peritoneum"
.
.
.
set `mesh("A01","047","365")="Groin"
.
.
.
Figure 56 Global Array Commands
```

A graphical representation of a portion of the MeSH hierarchy can be seen in Figure 57 which depicts the MeSH tree and the corresponding Mumps assignment statements used to create the global array corresponding to the diagram.

A program to build a MeSH tree is shown in Program 45. However, rather than being a program consisting of several thousand Mumps assignment statements, instead we use the Mumps indirection facility to write a short Mumps program that reads the MeSH file and dynamically generates and executes several thousand assignment statements.
The program in Program 45, in a loop (lines 7 through 37), reads a line from the file mesh2003.txt (line 9). On lines 11 and 12 the part of the MeSH entry prior to and following the semi-colon are extracted into the strings key and code, respectively. The loop on lines 15 through 17 extracts each decimal point separated element of code into successively numbered elements of the local array x. On line 21 a string is assigned to the variable z. This will be the initial portion of the global array reference to be constructed.

On line 28 elements of the array x are concatenated onto z with encompassing quotes and separating commas. On line 29 the final element of array x is added along with a closing parenthesis, an assignment operator and the value of key and the text are prepended with a Mumps set command.

Now the contents of z look like a Mumps assignment statement and this is executed on line 37 thus creating an entry in the database. The xecute command in Mumps causes the string passed to it to be executed as Mumps code.

Note that to embed a double-quote character (") into a string, you place two immediately adjacent double-quote characters into the string. Thus: """" means a string of length one containing a single double-quote character.

Also note that line 13 uses the OR operator (!) to test if either key or code is the empty string. Observe that parentheses are needed in this expression since expressions in Mumps are executed left-to-right without precedence. Without parentheses, the predicate would evaluate as if it had been written as:

\[
(((key="")!code)="")
\]

which would yield a completely different result!

Line 28 uses the concatenation operator (_) on the local array \( x(j) \). Local arrays should be used as little as possible as access to them through the Mumps run-time symbol table which can be slow especially if there are a large number of variables or array elements in the current program.

The close command on line 39 releases the file associated with unit 1 and makes unit 1 available for re-use. Closing a file opened for input is not strictly needed unless you want to reuse the unit number. Closing a file open for output, however, is desirable in order to flush the internal system buffers to disk. If the program crashes before an output file is closed, it is possible to lose data.
#!/usr/bin/mumps

# BuildMeshTree.mps

kill ^mesh
open 1:"mtrees2003.txt,old"
if '$test write "mtrees2003.txt not found",! halt
for do
  . use 1
  . read a
  if '$test break
  . set key=$piece(a,";",1) // text description
  . set code=$piece(a,";",2) // everything else
  if key=""!(code="") break
  . for i=1:1 do
    .. set x(i)=$piece(code,".",i) // extract code numbers
    .. if x(i)="" break
  . set i=i-1
  . use 5
  . set z="^mesh("        // begin building a global reference
  for j=1:1:i-1 set z=z_""_x(j)_"","
  set z="set _z_""_x(i)_"",")=""_key_""
  for j=1:1:i-1 set z=z_""_x(j)_"","
  . write z,!
  . execute z
close 1
use 5
write "done",!
halt

Program 45 MeSH Structured Global Array

The output of Program 45 is shown in Figure 58. Line 36 writes the text of the created mumps set command. These are the commands executed by the execute command on line 37.

14.3 Displaying the MeSH Global Array Part I

Now that the MeSH global array has been created, the question is, how to print it properly indented so as to show the tree structure of the data.

Program 46 gives one way to print the global array and the results are shown in Figure 59. In this example there are nested loops to print data at lower levels. When data at a given level is printed, it is indented by 0, 5, 10, and 15 spaces depending on the level of the data.
On Line 4\(^6\) the process begins by finding successive values of the first index of \(^{\text{mesh}}\). Each iteration of this outermost loop will yield, in alphabetic order, a new top level value until there are none remaining. These are placed in the local variable \(\text{lev1}\).

\begin{verbatim}
1 set ^mesh('A01')="Body Regions"
2 set ^mesh('A01','047')="Abdomen"
3 set ^mesh('A01','047','025')="Abdominal Cavity"
4 set ^mesh('A01','047','025','600')="Peritoneum"
5 set ^mesh('A01','047','025','600','225')="Douglas' Pouch"
6 set ^mesh('A01','047','025','600','451')="Mesentery"
7 set ^mesh('A01','047','025','600','451','535')="Mesocolon"
8 set ^mesh('A01','047','025','600','573')="Omentum"
9 set ^mesh('A01','047','025','600','678')="Peritoneal Cavity"
10 set ^mesh('A01','047','025','750')="Retroperitoneal Space"
11 set ^mesh('A01','047','050')="Abdominal Wall"
12 set ^mesh('A01','047','365')="Groin"
13 set ^mesh('A01','047','412')="Inguinal Canal"
14 set ^mesh('A01','047','849')="Umbilicus"
15 set ^mesh('A01','176')="Back"
16 set ^mesh('A01','176','519')="Lumbosacral Region"
17 set ^mesh('A01','176','780')="Sacrococcygeal Region"
18 set ^mesh('A01','236')="Breast"
19 set ^mesh('A01','236','500')="Nipples"
20 set ^mesh('A01','378')="Extremities"
21 set ^mesh('A01','378','100')="Amputation Stumps"
22 set ^mesh('A01','378','610')="Lower Extremity"
23 set ^mesh('A01','378','610','100')="Buttocks"
24 set ^mesh('A01','378','610','250')="Foot"
25 set ^mesh('A01','378','610','250','149')="Ankle"
26 set ^mesh('A01','378','610','250','300')="Forefoot, Human"
27 set ^mesh('A01','378','610','250','300','480')="Metatarsus"
28 
Figure 58 Creating the MeSH tree
\end{verbatim}

The program then advances to line 6 which yields successive values of all second level codes subordinate to the value of the current top level code (\(\text{lev1}\)). Each of these is placed in \(\text{lev2}\). The second level codes are printed on line 7 indented by 5 spaces.

\begin{verbatim}
1 #!/usr/bin/mumps
2 # BasicMtreePrint.mps
3
4   for lev1=$order(^mesh(lev1)) do
5     . write lev1," ",^mesh(lev1),!
6   for lev2=$order(^mesh(lev1,lev2)) do
7     .. write ?5,lev2," ",^mesh(lev1,lev2),!
8   for lev3=$order(^mesh(lev1,lev2,lev3)) do
9     ... write ?10,lev3," ",^mesh(lev1,lev2,lev3),!
10    ... for lev4=$order(^mesh(lev1,lev2,lev3,lev4)) do
11      .... write ?15,lev4," ",^mesh(lev1,lev2,lev3,lev4),!
\end{verbatim}

The process continues for levels 3 and 4. If there are no codes at a given level, the loop at that level terminates immediately and flow is returned to the outer loop and any more deeply nested loops are not executed.

\(^6\) The format of this \textit{for} command uses a non-standard Mumps syntax extension.
A01 Body Regions
    047 Abdomen
        025 Abdominal Cavity
            600 Peritoneum
                750 Retroperitoneal Space
        050 Abdominal Wall
        365 Groin
        412 Inguinal Canal
        849 Umbilicus
    176 Back
        519 Lumbosacral Region
        780 Sacrococcygeal Region
    236 Breast
        500 Nipples
    378 Extremities
        100 Amputation Stumps
        610 Lower Extremity
            100 Buttocks
            250 Foot
            400 Hip
            450 Knee
            500 Leg
            750 Thigh
        800 Upper Extremity
            075 Arm
            090 Axilla
            420 Elbow
            585 Forearm
            667 Hand
            750 Shoulder
    456 Head
        313 Ear
        505 Face
            173 Cheek
            259 Chin
            420 Eye
            580 Forehead
            631 Mouth
            733 Nose
            750 Parotid Region
    810 Scalp
    830 Skull Base
        150 Cranial Fossa, Anterior
        165 Cranial Fossa, Middle
        200 Cranial Fossa, Posterior
    598 Neck
    673 Pelvis
        600 Pelvic Floor
    719 Perineum
    911 Thorax
        800 Thoracic Cavity
            500 Mediastinum
            650 Pleural Cavity
        850 Thoracic Wall
    960 Viscera
A02 Musculoskeletal System
    165 Cartilage
        165 Cartilage, Articular
        207 Ear Cartilages
        410 Intervertebral Disk
        507 Laryngeal Cartilages
083 Arytenoid Cartilage
211 Cricoid Cartilage
411 Epiglottis
870 Thyroid Cartilage
590 Menisci, Tibial
639 Nasal Septum
340 Fascia
424 Fascia Lata
513 Ligaments
170 Broad Ligament
514 Ligaments, Articular
100 Anterior Cruciate Ligament
162 Collateral Ligaments
287 Ligamentum Flavum
350 Longitudinal Ligaments
475 Patellar Ligament
600 Posterior Cruciate Ligament
...

Figure 59 Printed the Mesh tree

14.4 Printing the MeSH Global Array Part II

Program 47 presents a more general function to print the \textsuperscript{mesh} hierarchy and its output is shown in Figure 60. The code in Program 47 works with trees of any depth whereas the code in program 46 needs to be extended in order to work with trees whose depth is greater than four.

1 #!/usr/bin/mumps
2 # AdvancedMtreePrint.mps
3
4    set x="^mesh"
5    for do
6 .   set x=$query(@x)
7 .   if x="" break
8 .   set i=$qlength(@x)
9 .   write ?i*2, "$qsubscript(x,i)," ",@x,?50,x,!

Program 47 Alternate MeSH tree printing program

In the example in Program 47, we first set a local variable \( x \) to \textsuperscript{mesh}, the name of the MeSH global array. In the loop on lines 5 through 9, the variable \( x \) is passed as an argument to the builtin function $query()$ which returns the next ascendant global array key in the database. These are printed in line 9 and can be seen in the right hand column output of Program 47 shown in Figure 60. The values returned by $query()$ are assigned to the variable \( x \) and used as the seed to $query()$ in the next iteration unto, finally, an empty string is returned.

In line 8 the number of subscripts in the global array reference contained in variable \( x \) is assigned to the local variable \( i \). In line 9 this number is used to indent the output by twice the number of spaces as there are subscripts (\(?\!i\!*2\)).

On line 9, after indenting, we want to print the MeSH code value for the level and the text followed by, in a separate column, the full Mumps $\textsuperscript{mesh()}$ global array reference.
We use the builtin function $qsubscript()$ to extract from the global array reference the $i^{th}$ index element. The arguments to $qsubscript()$ are the text of the global (or local) array $x$ and an integer $i$ where $i$ is the current level of the tree. The function returns the value of the $i^{th}$ subscript.

next we print the value stored at the global array, the text of the description of the MeSH codes. Note that the expression @x evaluates the string in variable x which, since it is a global array reference, evaluates to the value stored at the global array node reference.

In a column indented to 50 we print the actual MeSH global array reference. Note that the last index of each global array reference is also printed (without quotes) at the beginning of the line. These are the values extracted from the global array references by $qsubscript()$. 
A01 Body Regions

047 Abdomen
  025 Abdominal Cavity
    600 Peritoneum
    225 Douglas' Pouch
    451 Mesentery
    535 Mesocolon
    573 Omentum
    678 Peritoneal Cavity
  750 Retroperitoneal Space

050 Abdominal Wall
  365 Groin
  412 Inguinal Canal
  849 Umbilicus

176 Back
  519 Lumbosacral Region
  780 Sacrococcygeal Region

236 Breast
  500 Nipples

378 Extremities
  100 Amputation Stumps
  610 Lower Extremity
    100 Buttocks
    250 Foot
      149 Ankle
      300 Forefoot, Human
        480 Metatarsus
        792 Toes
          380 Hallux
      510 Heel
    400 Hip
    450 Knee
      500 Leg
        750 Thigh
    800 Upper Extremity
      075 Arm
      090 Axilla
      420 Elbow
    585 Forearm
  667 Hand
    430 Fingers
    705 Thumb
    715 Wrist
      750 Shoulder

456 Head
  313 Ear
  505 Face
    173 Cheek
    259 Chin
  420 Eye
    338 Eyebrows
    504 Eyelids
      421 Eyelashes
    580 Forehead
      631 Mouth

515 Lip

Figure 60 Alternative MeSH printing output
14.5 Displaying Global Arrays in Key Order

A program to print the global array references in the global array $b$-tree database tree order is shown in Program 48. Here we use the Mumps function $query()$ to access the $b$-tree keys in the order in which they are actually stored, in sequential key order, as shown in Figure 61.

The program shown in Program 48 passes to $query()$ a string containing a global array reference. The function returns the next ascending global array reference in the database system. Eventually, it will run out of $^\text{mesh}$ references and return an empty string which causes program termination.

```
1  ^mesh("A01")
2  ^mesh("A01","047")
3  ^mesh("A01","047","025")
4  ^mesh("A01","047","025","600")
5  ^mesh("A01","047","025","600","225")
6  ^mesh("A01","047","025","600","451")
7  ^mesh("A01","047","025","600","451","535")
8  ^mesh("A01","047","025","600","573")
9  ^mesh("A01","047","025","600","678")
10 ^mesh("A01","047","025","750")
11 ^mesh("A01","047","050")
12 ^mesh("A01","047","365")
13 ^mesh("A01","047","412")
14 ^mesh("A01","047","849")
15 ^mesh("A01","176")
```

**Figure 61 MeSH global array codes**

Note that the line:

```
. write x,?50,@x,!
```

displays the global array reference in variable $x$ and then prints the contents of the node at $x$ by evaluating the global array reference (@$x$). Evaluation of a variable by the indirection operator yields the value of the variable.

```
#!/usr/bin/mumps
# AdvancedMtreePrint2.mps
4    set x="^mesh"    // build the first index
5    for do
6    . set x=$query(@x)    // get next array reference
7    . if x="" break
8    . write x,?50,@x,!
```

**Program 48 Program to print MeSH global ()**

The output from Program 48 appears in Figure 62.
Next we want to write a program that will, when given a keyword, locate all the MeSH headings containing the keyword and display the full heading, hierarchy codes, and descendants of the keywords found at this level. In effect, this program retrieves all the more specific terms related to a higher level, more general term. The program is shown in Program 49.

The program in Program 49 first reads in a keyword into a local variable `key`. It next assigns to local variable `x` the text of the initial element of the global array reference `^mesh`.

14.6 Searching the MeSH Global Array

Next we want to write a program that will, when given a keyword, locate all the MeSH headings containing the keyword and display the full heading, hierarchy codes, and descendants of the keywords found at this level. In effect, this program retrieves all the more specific terms related to a higher level, more general term. The program is shown in Program 49.

The program in Program 49 first reads in a keyword into a local variable `key`. It next assigns to local variable `x` the text of the initial element of the global array reference `^mesh`. 
It now iterates through \textit{^mesh} global array references (until there are none remaining) in the loop in lines 9 through 19.

The program examines the value stored for each \textit{^mesh} global array node in global array key order.

On line 10, the \texttt{$\text{find()}$} function determines if the value stored at the current global array node referenced by \texttt{x} (The value stored at which is @x) contains, as a substring, the value in \texttt{key}. That is, for node in the \textit{^mesh} tree, does the text value stored at the node contain \texttt{key}?

```mumps
#!/usr/bin/mumps
# FindMesh.mps December 5, 2011

read "enter keyword: ",key
write !
set x="^mesh" // build a global array ref
set x=$query(@x)
if x="" halt
for do
  . if '$\text{find(}@x,key)$ set x=$query(@x) // is key stored at this ref?
  . else do
    .. set i=$qlength(@x) // number of subscripts
    .. write x," ",@x,!
    .. for do
      ... set x=$query(@x)
      ... if x="" halt
      ... if $qlength(@x)'>i break
      ... write ?5,x," ",@x,!
  . if x="" halt
```

Program 49 Program to search MeSH global array

If \texttt{$\text{find()}$} does not detect the value in \texttt{key} in the global array node, the next global array reference is returned by \texttt{$\text{query()}$} until there are none remaining.

```
enter keyword: Skeleton
^mesh("A02","835") Skeleton
^mesh("A02","835","232") Bone and Bones
^mesh("A02","835","232","087") Bones of Upper Extremity
^mesh("A02","835","232","087","144") Carpal Bones
^mesh("A02","835","232","087","144","650") Scaphoid Bone
^mesh("A02","835","232","087","144","663") Semilunar Bone
^mesh("A02","835","232","087","227") Clavicle
^mesh("A02","835","232","087","412") Humerus
^mesh("A02","835","232","087","535") Metacarpus
^mesh("A02","835","232","087","702") Radius
^mesh("A02","835","232","087","783") Scapula
^mesh("A02","835","232","087","783","261") Acromion
^mesh("A02","835","232","087","911") Ulna
^mesh("A02","835","232","169") Diaphyses
^mesh("A02","835","232","251") Epiphyses
^mesh("A02","835","232","251","352") Growth Plate
...
```

Figure 63 MeSH keyword search results
If the key is found, however, the program prints the reference and value stored then it scans for additional references whose number of subscripts is greater than that of the found reference (that is, sub trees of the found reference) and it prints any nodes that are subordinate to the found node since these are necessarily more specific (deeper) forms of the term sought.

The function $qlength()$ returns the number of subscripts in a reference. When the number of subscripts becomes less-than-or-equal (shown as\textsuperscript{62} not-greater-than: ‘$>$’) to the number of subscripts in the found reference, printing ends and the key scan of the nodes resumes. Thus, only sub-trees of the found node will be printed.

After all the nodes of greater depth are printed, the program looks for additional instances of the search keyword.

An example is shown in Figure 63 where Skeleton was given as input.

14.7 Display OHSUMED Collection by Embedded MeSH Headings

Next, we write a program to read the MEDLINE formatted abstracts (from the modified TREC-9 data base described above\textsuperscript{63}) and write out a list of MeSH headings, the number of times each heading occurs, and the title of each abstract in which it occurs along with the byte offset of the abstract in the master file.

This is an example of an inverted index using a controlled vocabulary. That is, a mapping from a collection of pre-existing index terms, in this case the MeSH headings, to the underlying documents containing these headings. An inverted index is faster than sequentially searching each document for index terms.

First note that the lines containing MeSH headings in the OHSUMED data base all have the code \textit{MH}\textsuperscript{64} in positions 1 and 2. Note also that there is a blank line that signals the end of each abstract and he beginning of the next one (or the end of file).

Creation of the inverted index proceeds as follows: first, we locate and extract the MeSH terms in the OHSUMED file. Then, for each instance of a MeSH term, we record the term and the offset into the OHSUMED file of the article in which the term occurred in a global array (\textasciitilde MH). Additionally, we count the number of times each term occurs. The program to do this is shown in Program 50.

Finally, after the entire OHSUMED file has been processed, we write out each MeSH heading, the number of times it occurs and a list of the titles and their offsets in which it occurred. An example of the output can be seen in Figure 64.

Program 50 opens the input file (line 5), captures the initial file byte offset (line 12) and then loops reading lines from the input file designated as unit 1. The loop ends when there is no more input on unit 1.

\textsuperscript{62} Mumps does not have greater-than-or-equal or less-than-or-equal operators so we use not-less-than and not-greater-than, respectively, instead.

\textsuperscript{63} Chapter Error: Reference source not found Error: Reference source not found on page Error: Reference source not found

\textsuperscript{64} See Figure 13 MEDLINE Codes on page 25.
If an empty line is detected (line 21), the offset is recorded in the local variable \(i\), the abstract count \(x\) is incremented. Note that the value returned by \$ftell()\ on line 21 is the byte offset of the line we are about to be read, not the one most recently read. Thus, the value in variable \(i\) is the address of the first line of the next abstract.
Abdominal Injuries occurs in 13 documents
1650173 Percutaneous transcatheater steel-coil embolization of a large proximal pos
1678059 Features of 164 bladder ruptures.
2523966 Injuries to the abdominal vascular system: how much does aggressive resusc
3436121 Triple-contrast computed tomography in the evaluation of penetrating poste
4624903 Correlations of injury, toxicology, and cause of death to Galaxy Flight 20
4901771 Selective management of blunt abdominal trauma in children--the triage rol
4913645 Percutaneous peritoneal lavage using the Veress needle: a preliminary repo
6713150 The seat-belt syndrome.
7019763 Early diagnosis of shock due to pericardial tamponade using transcunaneous
7885247 The incidence of severe trauma in small rural hospitals.
8189154 Intussusception following abdominal trauma.
8808690 Hepatic and splenic injury in children: role of CT in the decision for lap
8961708 Peritoneal lavage and the surgical resident.

Abdominal Neoplasms occurs in 6 documents
10033669 Current spectrum of intestinal obstruction.
10399042 Diagnosis of metastases from testicular germ cell tumours using fine needl
116380 Intracystic injection of OK-432: a new sclerosing therapy for cystic hygro
5804499 Pheochromocytoma, polycythemia, and venous thrombosis.
8983032 Malignant epithelioid peripheral nerve sheath tumor arising in a benign sc
8991187 DTIC therapy in patients with malignant intra-abdominal neuroendocrine tum

Abdominal Wall occurs in 11 documents
10291646 Structure of abdominal muscles in the hamster: effect of elastase-induced
2142543 Surgical incision for cesarean section.
2230059 Exstrophy, epispadias, and cloacal and urogenital sinus abnormalities.
2963791 Adductor tendinitis and musculus rectus abdominis tendopathy.
5426490 Postpartum sit-ups [letter]
5438957 Bilateral upper-quadrant (intercostal) flaps: the value of protective sens
6012451 Anterior rectus sheath repair for inguinal hernia.
6557458 Effects of upper or lower abdominal surgery on diaphragmatic function.
8946400 Patterns of muscular activity during movement in patients with chronic low
8947451 Trunk muscle balance and muscular force.
9892904 Venous plasma (total) bupivacaine concentrations following lower abdominal

Figure 64 Titles organized by MeSH code

Line 23 checks to see if a line contains the code \textit{MH}. If it does, it extracts the portion of the line from position 7 up to, but not including, any / character (we ignore any text following the / character). If no / character is present, we extract to the end of line. The line length limit of 255 is overly generous as no line is that long. The actual length of the MeSH heading stored in local variable \texttt{a} is determined by the line length, not 255.

Next the MeSH heading, a pound sign and the offset of where the abstract containing the MeSH term are written to standard output.

For each MeSH heading detected, an instance of the global array \texttt{^MH} is instantiated if it does not exists and the count stored at \texttt{^MH} is incremented or set to one (lines 29 and 30). The offset of the document containing it is also recorded in \texttt{^MH} as second level index value (line 34). Thus the global array \texttt{^MH} contains at the first indexing level the MeSH headings and, at the second level, the offset addresses of thos documents that contain the term.

When the input is exhausted, the program prints for each heading the number of documents it appeared in along with a list of the documents. A sample of the output is
given in Figure 64. This form of display is called a *concordance* - a list of words and an indication of their location and context. This form of display is called a *concordance* - a list of words and an indication of their location and context.

```mumps
#!/usr/bin/mumps
# MeshTitles.mps
open 1:"osu.medline,old"
if '$test write "file open error",! halt
set x="^mesh(0)"
for do
  set x=$query(x)
  if x="" break
  set i=$qlength(x)
  write ?i*2,"$qsubscript(x,i),",@x,?50,x,!
  set z=i$zlower(@x)
  if $data(^MH(z)) do
    .. write !,?i*2+5,z, occurs in "^MH(z)," documents",!
    .. for off=$order(^MH(z,off)) do
      ... use 1
      ... do $zseek(off)
      .... read a
      .... if $extract(a,1,3)'="TI " quit
      .... use 5
      .... write ?i*2+5,"$extract(a,7,80),!
      .... break
      .. write !
```

Program 51 Hierarchical MeSH concordance

### 14.8 Display of OHSUMED Documents by MeSH Hierarchy

Now we combine the programs from Program 48 (page 149) and Program 50 into a single program that displays the titles integrated into the overall tree structure of the MeSH hierarchy. The code is shown in Program 51 and a sample of the output in Figure 65.

<table>
<thead>
<tr>
<th>MeSH Term</th>
<th>Mesh Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abdominal Cavity</td>
<td><code>mesh(&quot;A01&quot;,&quot;047&quot;,&quot;025&quot;)</code></td>
</tr>
<tr>
<td>Peritoneum</td>
<td><code>mesh(&quot;A01&quot;,&quot;047&quot;,&quot;025&quot;,&quot;600&quot;)</code></td>
</tr>
</tbody>
</table>

Peritoneum occurs in 4 documents

- Systems of membranes involved in peritoneal dialysis.
- Suppression of lymphocyte reactivity in vitro by supernatants of explants
- An evaluation of the Gore-Tex surgical membrane for the prevention of postoperative anastomotic leakage
- The morphologic effect of short-term medical therapy of endometriosis.

<table>
<thead>
<tr>
<th>MeSH Term</th>
<th>Mesh Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Douglas' Pouch</td>
<td><code>mesh(&quot;A01&quot;,&quot;047&quot;,&quot;025&quot;,&quot;600&quot;,&quot;225&quot;)</code></td>
</tr>
<tr>
<td>Mesentery</td>
<td><code>mesh(&quot;A01&quot;,&quot;047&quot;,&quot;025&quot;,&quot;600&quot;,&quot;451&quot;)</code></td>
</tr>
</tbody>
</table>

Mesentery occurs in 3 documents

- Cellular localization of angiotensinogen gene expression in brown adipose tissue
- Technique of mesenteric lengthening in ileal reservoir-anal anastomosis
- Detection of mesenteric involvement in sarcoidosis using computed tomography

<table>
<thead>
<tr>
<th>MeSH Term</th>
<th>Mesh Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesocolon</td>
<td><code>mesh(&quot;A01&quot;,&quot;047&quot;,&quot;025&quot;,&quot;600&quot;,&quot;451&quot;,&quot;535&quot;)</code></td>
</tr>
</tbody>
</table>

Note that MeSH terms are assigned to abstracts by human indexers and the actual abstracts, although they contain the concept indicated by the MeSH term, may not contain the actual term.
14.9 Searching By MeSH Terms

An inverted file is organized as a set of keys. For each key there is a set of pointers to those documents in the main document file which are indexed by the key.

For example, consider the program in Program 50 above. In this case the keywords are terms from the MeSH hierarchy found in documents in the OHSUMED collection. These are stored in the global array $^M^H(term)$ where term is an individual MeSH term contained in one or more of the OHSUMED documents.

For each MeSH term entry in $^M^H(term)$, there are one or more file offset pointers at the second level of indexing of the $^M^H$ global array that point to a document in the collection indexed by the term.

Retrieval based on MeSH keyword involves locating the term at the first level of $^M^H$ and then fetching and displaying each of the corresponding documents whose offsets are at the second level. An example of this along with output is shown in Program 52.

The program in Program 52 reads in a keyword from the user (line 11), prints the number of documents the keyword appears in and then, for each offset recorded at the second level of global array $^M^H$, begins reading the original abstract from the osu.medline file beginning at the offset stored in $^M^H$. The program reads through the first through the first few lines of the abstract until it locates the title line which it then prints along with the file offset of the beginning of the document. The total time taken by the program is measured in milliseconds despite the fact that the file being 'searched' is 336 million bytes in length.
Enter a MeSH keyword: Acetylcholinesterase

Acetylcholinesterase occurs in 6 documents

141739 The slow channel syndrome. Two new cases.
2758782 The diagnostic value of acetylcholinesterase/butyrylcholinesterase ratio i
3643396 Ultrastructural analysis of murine megakaryocyte maturation in vitro: comp
5479094 Long-term neuropathological and neurochemical effects of nucleus basalis l
6687870 Cholinesterase activities in cerebrospinal fluid of patients with senile d
8444730 Increased skeletal muscle acetylcholinesterase activity in porcine maligna

Program 52 Inverted search by MeSH term

Clearly, the inverted lookup in Program 52 is preferable to scanning each document looking for instances of the MeSH term Acetylcholinesterase!
15 Evaluation of Retrieval Systems

In order to compare the many approaches and algorithms for information retrieval, we need a way to evaluate the results. Two of the more widely used metrics are discussed next: precision and recall.

In many experimental systems a set of trial queries is developed. Then experienced researchers inspect the collection of documents and develop for each query a list of documents they feel most correctly answer the query. The results from the computer system for each query are then compared with these and scores are developed. These scores are presented as precision/recall graphs.

15.1 Precision and Recall

Two important metrics of information storage and retrieval system performance are precision and recall. Precision measures the degree to which the documents retrieved are relevant and recall measures the degree to which the system can retrieve all relevant documents.

For example, if a system responds to a query by retrieving 10 documents from the collection and of these, 8 are relevant and 2 are irrelevant and if the collection actually has 16 relevant documents, we say that the recall is 50% and the precision is 80%. That is, only 50% of the relevant documents were recalled but of those presented, 80% were correct.

For example, suppose there were 10 relevant documents in the collection and the top ten ranked results of a query are shown in Figure 66.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Relevant?</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>yes</td>
<td>0.1</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>yes</td>
<td>0.2</td>
<td>1.0</td>
</tr>
<tr>
<td>3</td>
<td>no</td>
<td>0.2</td>
<td>0.67</td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>0.3</td>
<td>0.75</td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td>0.4</td>
<td>0.80</td>
</tr>
<tr>
<td>6</td>
<td>no</td>
<td>0.4</td>
<td>0.67</td>
</tr>
<tr>
<td>7</td>
<td>no</td>
<td>0.4</td>
<td>0.57</td>
</tr>
<tr>
<td>8</td>
<td>yes</td>
<td>0.5</td>
<td>0.63</td>
</tr>
<tr>
<td>9</td>
<td>no</td>
<td>0.5</td>
<td>0.56</td>
</tr>
<tr>
<td>10</td>
<td>yes</td>
<td>0.6</td>
<td>0.60</td>
</tr>
</tbody>
</table>

Figure 66 Precision/recall example

In general, as recall increases, precision declines. For example, in the query mentioned in the previous paragraph, if by setting thresholds lower the system responds with 20 documents instead of 10 and if 12 of these are relevant but 8 are not, the recall has increased to 75% but the precision has fallen to 60%.

In most systems, as you lower thresholds and more documents are retrieved, the recall will rise but the precision will decline. In an ideal system, however, as thresholds are lowered, recall increases but precision remains 100%.
Indexing terms effect the precision/recall results. Generally speaking, terms of low frequency tend to increase the precision of a system's responses at the expense of recall as these tend to reference more narrowly defined concepts. On the other hand, terms of high frequency tend to increase recall at the expense of precision as these are often more broadly defined. Identifying those terms which strike a balance is a major goal of any system.

Salton [Salton, 1971] used precision-recall graphs similar to the one shown in Figure 67 in order to compare the results of different retrieval experiments. Those experiments which resulted in a slower drop off in precision as recall increases represent improvement in technique.

Figure 67 Precision/recall graph
16 Alternative Text Encodings

Documents and queries may be re-encoded in other ways which may also be used for indexing. These are a few that have been used over the years.

16.1 Truncation

Salton's early work was done on IBM 7090 series computers. These were six-bit, character based machines with a word size of 36 bits (6 six-bit characters per word).

When using 6 bit characters, there are only 64 possible character codes available. Consequently, these machines normally only handled upper case letters.

The 7090 series machines, as was the case with most machines of that era, had word addressable memories. That is, the smallest unit of storage that could be addressed was a 36 bit word. This simplified the system addressing circuitry considerably by reducing the number of address lines that needed to be implemented and calculated. Also, these early machines had relatively small (by today's standards) expensive magnetic core memory.

To save space and increase program efficiency, many of Salton's early experiments were conducted using only the first six characters from each word in a document. In practice, however, this worked relatively well in that much of the information content of English words is in the first few syllables.

16.2 Soundex Encoding

Soundex is a technique (patent number 1,261,167 on April 2, 1918) to convert words that sound like one another into common codes. It was originally (and still is) used for telephone directory assistance to permit operators to quickly access phone numbers based on the sound of a name rather than on a detailed spelling of the name. In this system, the operator would generate a alphanumeric code based on what the name sounded like and use this to locate a page in the directory likely to contain the name. The operator would then ask the caller the first name and then confirm the identity with the address ('do you mean Jane Smith on Willow Street?'). This works in many cases but not all.

Mumps has a builtin function $zzSoundex()$ which returns a Soundex code for a string passed as its argument. Unlike traditional Soundex, the code can be longer or shorter than four characters, depending on the size of the string passed. This returned code, can, of course, be truncated or padded with zeros as needed.

It is possible to recode a document collection to fixed length Soundex codes and base all the indexing discussed above on 'words' consisting of Soundex codes. The results can be very good.

Program 53 gives an example of converting the dictionary in ^dict to Soundex codes a sample of which are shown in Figure 68.

---

66 It wasn't until the mid-60s and the introduction of the IBM 360 series that the eight bit byte became widely available.
During World War II, *n-grams* were developed by cryptographers to break substitution ciphers [Salton 1983]. An n-gram is fixed length strings of *n* characters from the text (numbers, punctuation and blanks omitted).

Applying n-grams to indexing, the text, stripped of non-alphabetic characters, is treated as a continuous stream of data that is segmented into non-overlapping fixed length words. These words can then form the basis of the indexing vocabulary often with good results. Queries terms are converted to n-grams consisting of all possible overlapping strings of length *n*.
One problem confronting all retrieval systems is the ambiguous nature of natural language. While in scientific disciplines there is usually a non-ambiguous, precise vocabulary, in general text, many words have different meanings depending upon context. When we use words to index documents it would be desirable, if possible, to indicate the meaning of the term.

For example, the word base elicits the results shown in Figure 44 from WordNet. Even for a simple term, term are many possible meanings, depending on context.

Ideally, when processing documents, terms near the term being examined could be used to disambiguate this context. For example, the sample sentences shown in Figure 44 could provide related terms that could help select the correct sense of the term.

WordNet can be automatically installed in Linux through the Synaptic Package manager. In command line mode, words can be retrieved as shown in Figure 43.

```
#!/bin/bash
# index.script
# Copyright 2014 Kevin C. O'Kane
# Feb 13, 2014

clear
echo "Document Indexing `date`"
echo "==============================================="
echo
# To process as a command line parameter. Example:
# index.script 10000
# means to process 10,000 documents.
index.script 10000 osu
```
# means 10, docs from osu.medline database

# You MUST install Mumps using the native globals build, non-client server:
#   BuildMumpsWithGlobalsInNative.script
# for this code to work.
#---------------------------------------------------------------

if [ "$1" = "-?" ];
  then
    echo
    echo "usage: index.script nbrDocs database3"
    echo "where nbrDocs is the number of documents to process (default: 1000)"
    echo "where database is the name of the database prefix (database.medline)"
    echo
    exit
  fi

if [ -z $1 ]
  then
    echo "Max Number of Docs not supplied - default value of 1000 used..."
    MAXDOCS=1000
  else
    MAXDOCS=$1
  fi

if [ -z $2 ]
  then
    echo "Database name not given - default osu database used"
    DB="osu"
  else
    DB=$2
  fi

export $DB
echo "Database to be processed: $DB"
echo $DB > DBPREFIX

#---------------------------------------------------------------

cat /dev/null > nohup.out  # erase any prior nohup.out

# Delete any prior database instances

rm -f key.dat data.dat copy*.dat copy*.key $MumpsNativeDatabase.dat $MumpsNativeDatabase.key

rm -f *.txt *.sorted *.tmp  # delete old results
starttime.mps  # record starting time

#---------------------------------------------------------------

# Check to see if input files exist and create if not

rm $DB.converted

if [ -f "$DB.converted" ] && [ -f "titles.list" ]
  then
    echo "Not rebuilding $DB.converted - files exist"
  else
    echo "Rebuilding $DB.converted"
echo "This will take several minutes..."

sed -e 's/[]+="/ /g' osu.medline |
sed -e 's/STAT-MEDLINE/STAT-MEDLINE/' | |
sed -e 's/[0-9]*.[0-9][0-9]* / /g' | |
sed -e 's/[0-9]*./ /g' | |
sed -e 's/\.[ ]/ /g' | |
sed -e 's/\[\ ]{2,5}/ /g' | |
sed -e 's/ / /g' | |
sed -e 's/\*/ /g'
> tmp.tmp

reformat.mps < tmp.tmp |
  sed -e 's/"/"/g' \
  -e 's/^[a-zA-Z\-]{1,3}\ / /g' \
  -e 's/^[a-zA-Z\-]{1,3}\ / /g' \
  -e 's/^[a-zA-Z\-]{1,3}\$/ /g' \
  sed -e 's/\{2,6\}/ /g' \
> $DB.converted

fi

export MAXDOCS
echo "Processing $MAXDOCS number of documents."

# Zipf's Law calculation.
# Optional - uncomment if you want to run it.

# Basic.stop.words is a list of common English words.

# KWIC index
# Optional - uncomment if you want to run it.

stopcount=`wc -l stop.words`
stopcount=${stopcount:0 : `expr "$stopcount" : '0-9'`}" } # extract numeric val
echo
echo "Initial stop list word count: $stopcount"

# Step 1 - basic stop removal and stemming

# stop.mps removes stop list words as given in file stop.words
# stem.mps reduces words to stems and removes words < 3 chars
# in length and words that begin with numbers

# head sends only MAXDOCS lines where each document is a (long) line.

# Result file is one doc per line.

# $DB.converted -> remove stop words -> reduce words to stems -> initial-stemmed-docs.tmp

# words must be longer that MINWORDLENGTH

MINWORDLENGTH=4
echo "Words must be longer than $MINWORDLENGTH"

# stem-words.tmp is created by stems.mps in the above- it is a file containing
# all word instances. That is, if 'the' occurs 20,000 times, it will be in
# stem-words.tmp 20,000 times - once per line.

# Sort stem-words.tmp alphabetically so that all the instances of each word appear
# consecutively then count the number of times each words appears using uniq
# and write the result one word per line, prefixed by the number of times the
# word occurred.

# display original word and stems

# Now eliminate words that are too frequent or too infrequent.
# Calculate an estimate of the maximum and minimum word frequencies.
# These will be used to create a new stop list of high/low frequency
# words to be deleted from the text.

MIN_DIVISOR=2000  # word must occur once per 500 docs.
MIN_WORD_FREQ=$(($MAXDOCS/$MIN_DIVISOR))

MAX_DIVISOR=3
MAX_WORD_FREQ=$(($MAXDOCS/$MAX_DIVISOR))
MAXFRACTION=$((100/$MAX_DIVISOR))

echo
echo "Words whose frequencies of occurrence in the collection"
echo "as a whole are less than $MIN_WORD_FREQ (once per $MIN_DIVISOR docs) will be deleted."

echo
echo "Words whose total frequencies of occurrence in the collection are"
echo "greater than $MAX_WORD_FREQ (appearing in $MAXFRACTION% of the collection) will be deleted."

export MAX_WORD_FREQ MIN_WORD_FREQ

# Now build a new list of stop words based on frequency of occurrence.
# Words will be in the new stop.words if they are too frequent or too
# infrequent in the collection.

echo
echo "Building new stop list consisting of high/low freq words...

# FrequencyFilter.mps writes words whose frequency of occurrence is
# above or below the thresholds. A new stop list is built.

# File stemmed-vocabulary.tmp lists each word and the total number of
# times it occurs in the document collection after stemming. initial
# stoplist removal, removal of short words and words beginning with digits.

#=================================================================================================
FrequencyFilter.mps $MIN_WORD_FREQ $MAX_WORD_FREQ < stemmed-vocabulary.tmp > deleted-words-for-frequency.txt
#=================================================================================================

# Now re-process the collection to remove the new stop words.
# This deletes high/low freq words.

stopcount=`wc -l stop.words`
stopcount=${stopcount:0 : `expr "$stopcount" : '[0-9]*'` }

echo
echo "There are $stopcount words whose total frequency of occurrence"
echo "in the collection is too high or low. These will be deleted."

#=================================================================================================
stop.mps < initial-stemmed-docs.tmp > stemmed-docs.tmp  # remove new stop words
#=================================================================================================

# stemmed-docs.tmp is now the collection with high/low frequency words removed

# Now build a new dictionary.
# Write a file containing each word occurrence one per line.
WordList.mps < stemmed-docs.tmp > words.tmp

# WordListNumeric.mps < stemmed-docs.tmp > words.tmp  ## experiment with numeric codes for words
# cp NumericDocs.txt stemmed-docs.tmp   ## didn't work well

# stemmed-vocabulary.tmp is the dictionary of all the remaining unique words

sort < words.tmp | uniq -c > vocabulary.tmp

WordCount=`wc -l vocabulary.tmp`
WordCount=${WordCount:0 : `expr "$WordCount" : '[0-9]*'` }  # extract numeric val

echo "After enhanced stop list word removal and stemming, the collection contains $WordCount unique words for an average of $((WordCount/$MAXDOCS)) unique words per document."

TWordCount=`wc -l words.tmp`
TWordCount=${TWordCount:0 : `expr "$TWordCount" : '[0-9]*'` }  # extract numeric val

echo "After enhanced stop list word removal and stemming, the collection has $TWordCount word occurrences for an average of $((TWordCount/$MAXDOCS)) words per document"

# Establish processing parameters

MIN_DOC_VECTOR_WORDS=3  # kill docs with fewer this number of words
MINIDF=4.5  # kill words with IDF value below this number
DDWGT=5  # documents must have this number of words in common
          # to be included in document-document matrix
MINTT=`mintt.mps $MAXDOCS`  # terms must have this number of documents in common
          # to be included in term-term matrix
JMIN=0.001  # discard Jaccard similarities less than this
MINDD=0.7  # minimum document-document cosine to be included in
          # a document cluster-document matrix
HYPMIN=.4  # minimum cluster centroid cosine similarity for hyperclusters
HYPWGT=.4
TTPMIN=$MINTT  # minimum proximity score to be included in term-term matrix

echo "Kill docs if they have fewer than $MIN_DOC_VECTOR_WORDS words."

echo "Delete words whose IDF is less than $MINIDF."

echo "Weighting scheme in use: $WGT"

echo "Min Doc-Doc Words in common $DDWGT"

echo "Min Term-Term Docs in common $MINTT"
echo "Jaccard minimum $JMIN"
echo "Cluster minimum $MINDD"
echo "Hyper cluster min $HYPMIN"
echo "Hyper cluster wgt $HYPWGT"
echo
export MIN_DOC_VECTOR_WORDS HYPMIN HYPWGT MINIDF MINDD JMIN MINTT WGT MAXDOCS

###############################################
rm -f key.dat data.dat  # remove any prior database
###############################################
# load stems and original words that formed them
###############################################
loadStems.mps < stemList.txt > stemList2.txt
###############################################
#
# echo "Build document-term matrix"
# DocumentTermMatrix.mps $MIN_DOC_VECTOR_WORDS < stemmed-docs.tmp > document-term-matrix.txt
#
echo
echo "Document count after deletion of documents with fewer than $MIN_DOC_VECTOR_WORDS words: `dcount.mps`
#
# calculate the IDF coefficients of each word
echo
echo "Calculate Inverse Document Frequency weights of each word"
###############################################
idf.mps < stemmed-docs.tmp
###############################################
# delete words based on IDF values
echo
echo "Delete words whose IDF value is less than $MINIDF."
echo "Delete documents with fewer than $MIN_DOC_VECTOR_WORDS words"
###############################################
IdfCutoff.mps $MINIDF $MIN_DOC_VECTOR_WORDS > deleted-words-and-docs-for-idf.txt
###############################################

echo
echo "Document count after removal of low IDF words: `dcount.mps`"

###############################################
sort -k 1 -nr -k 2 -nr idfWeights.tmp > idf-weights.sorted.txt
###############################################

IDF_COUNT=`wc -l idf-weights.sorted.txt`
IDF_COUNT=${IDF_COUNT:0 : `expr "$IDF_COUNT" : '[0-9]``}
echo
echo "Number of unique words after removal of low IDF words: $IDF_COUNT"

# Some computations can be run in parallel on the document-term matrix
# at this point. Make copies of the current database.

cp key.dat copy0.key
cp data.dat copy0.dat
cp key.dat copy1.key
cp data.dat copy1.dat
cp key.dat copy2.key
cp data.dat copy2.dat

echo
echo "Calculate term-term matrix, part 1"
# tt1.mps is accesses the database in read-only mode
# tt1.mps shares database copy1
# Runs in parallel

tt1.mps copy1 | sort | uniq -c > tt.sorted.tmp &
TT=$!

echo
echo "Calculate term-term proximity matrix"
# proximity.mps accesses the database in read-write mode
# proximity.mps runs on database copy0
# Runs in parallel

proximity.mps $TTPMIN copy0 | sort -nr > term-term-proximity-similarities.txt &
PP=$!

echo
echo "Calculate document-document matrix, part 1"
# docdoc1.mps accesses the database in read-only mode
# docdoc1.mps shares copy1
# Runs in parallel

docdoc1.mps $DDWGT copy1 > docdoc1.tmp &
DD=$!

echo
echo "Calculate discrimination coefficients"
# discrim.mps accesses the database in read-write mode
# uses copy2 database
# Runs in parallel

# Discrimination Coefficients

discrim.mps copy2 | sort -n > discrimination-coefficients.txt &
DS=$!

# Create weighted doc-term and term-doc matrices based on IDF weights

echo

echo "Calculate weighted document-term and term-document matrices"

# Weighted Matrices

weight.mps

# Wait for process to finish

wait $TT

echo

echo "Calculate term-term matrix, part 2"

tt2.mps $MINTT < tt.sorted.tmp | sort -n > term-term-matrix.sorted.txt

ttcount=`wc -l term-term-matrix.sorted.txt`

ttcount=${ttcount:0 : `expr "$ttcount" : '[0-9]*'` } # extract numeric val

echo

echo "There were $ttcount term-term associations."

# Calculate term-term-term matrix

ttt.mps

# Calculate Jaccard term-term coefficients

# Runs in parallel
# Read-only database access

ejaccardtt.mps < term-term-matrix.sorted.txt | sort -n > term-term-jaccard-similarities.sorted.txt &
JJ=$!

# Calculate term-term cohesion coefficients

# Runs in parallel
# Read-only database access
cohesion.mps < term-term-matrix.sorted.txt | sort -nr > term-term-cohesion-similarities.txt
& CC=$!

wait $JJ $CC  # wait for tasks to finish

jaccard=`wc -l term-term-jaccard-similarities.sorted.txt`
jaccard=${jaccard:0 : `expr ""jaccard" : '[0-9]*'` }  # extract numeric val
echo "There were $jaccard term-term Jaccard values calculated"
echo

cohesion=`wc -l term-term-cohesion-similarities.txt`
cohesion=${cohesion:0 : `expr "$cohesion" : '[0-9]*'` }  # extract numeric val
echo "There were $cohesion term-term cohesion values calculated"
echo

echo "Calculating document-document matrix, part 2"

wait $DD  # wait for task to finish

docdoc2.mps $DDWGT < docdoc1.tmp > document-document-matrix.txt

docdoc=`grep ":" document-document-matrix.txt | wc -l`
docdoc=${docdoc:0 : `expr "$docdoc" : '[0-9]*'` }  # extract numeric val

echo "There were $docdoc documents with one or more similar documents"

# Load the document titles into the database

echo

echo "Load document titles into the database"

head -$MAXDOCS titles.list | loadTitles.mps

echo

echo "Calculating term clusters"

clustertt.mps > term-clusters.txt

termcluster=`grep "cluster:" term-clusters.txt | wc -l`
termcluster=${termcluster:0 : `expr "$termcluster" : '[0-9]*'` }  # extract numeric val

echo "There were $termcluster term clusters"

echo

echo "Calculating document clusters"
clusterdd=`grep "cluster:" document-clusters.txt | wc -l`
clusterdd=${clusterdd:0 : `expr "$clusterdd" : '0-9]*'` } # extract numeric val
echo "There were $clusterdd document clusters"
# hyper cluster min similarity -> min
# hyper cluster min centroid vector weight -> wgt
echo
echo "Calculating document hyper clusters"

hyperCluster.mps $HYPMIN $HYPWGT < document-clusters.txt > document-hyper-clusters.txt

hypercluster=`grep "cluster:" document-hyper-clusters.txt | wc -l`
hypercluster=${hypercluster:0 : `expr "$hypercluster" : '0-9]*'` } # extract numeric val
echo "There were $hypercluster document hyper clusters"

# SQLdocVects.mps creates commands to load the document vectors into an SQL database
# if you want to experiment with an SQL approach.
# uncomment if you want to use it.
# echo
# echo "Build SQL commands"
# SQLdocVects.mps > SQL.commands

echo
echo "Waiting for all processes to finish..."
echo
wait

# calculate soundex codes for vocabulary

soundex.mps > soundex.txt

proxLoad.mps < term-term-proximity-similarities.txt > proxLoad.txt
discrim=`wc -l discrimination-coefficients.txt`
discrim=${discrim:0 : `expr "$discrim" : '0-9]*'` } # extract numeric val
echo "There were $discrim term-term cohesion values calculated"

# Print summary, clean up and run test retrieval
echo
echo "------------------ File Summary ------------------"
echo
wc -l *.txt
rm copy0.* copy1.* copy2.*
echo
# retrieval test

echo
medlineRetrieve.mps < tstquery
echo

# rm -f *.tmp
echo "Docs $MAXDOCS `endtime.mps`"

Program 54 Main bash Script
References


<table>
<thead>
<tr>
<th>Alphabetical Index</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM's Computing Classification System</td>
<td>139</td>
</tr>
<tr>
<td>Adenine</td>
<td>13</td>
</tr>
<tr>
<td>American Society of Indexers</td>
<td>50</td>
</tr>
<tr>
<td>ASCII</td>
<td>45</td>
</tr>
<tr>
<td>Assigning Word Weights</td>
<td>82</td>
</tr>
<tr>
<td>Background</td>
<td>37</td>
</tr>
<tr>
<td>Bartlett's Familiar Quotations</td>
<td>50</td>
</tr>
<tr>
<td>Basic.stop.words</td>
<td>57</td>
</tr>
<tr>
<td>Bioinformatics</td>
<td>13</td>
</tr>
<tr>
<td>BLAST</td>
<td>14</td>
</tr>
<tr>
<td>Boolean logic</td>
<td>31</td>
</tr>
<tr>
<td>Boolean Searching</td>
<td>31, 33</td>
</tr>
<tr>
<td>Building an Enhanced Stop List</td>
<td>59</td>
</tr>
<tr>
<td>C/C++</td>
<td>39</td>
</tr>
<tr>
<td>Calculating IDF Weights</td>
<td>88</td>
</tr>
<tr>
<td>Centroid algorithm</td>
<td>90</td>
</tr>
<tr>
<td>Centroid approximation</td>
<td>90</td>
</tr>
<tr>
<td>Centroid vector</td>
<td>90</td>
</tr>
<tr>
<td>Clustering</td>
<td>73, 122</td>
</tr>
<tr>
<td>Cohesion</td>
<td>109</td>
</tr>
<tr>
<td>Collating sequence</td>
<td>45</td>
</tr>
<tr>
<td>Concordance</td>
<td>50</td>
</tr>
<tr>
<td>Controlled vocabulary</td>
<td>48, 140</td>
</tr>
<tr>
<td>Cosine</td>
<td>130</td>
</tr>
<tr>
<td>Cosinse</td>
<td>132</td>
</tr>
<tr>
<td>CPU utilization</td>
<td>21</td>
</tr>
<tr>
<td>Crouch</td>
<td>90</td>
</tr>
<tr>
<td>Cytosine</td>
<td>13</td>
</tr>
<tr>
<td>DBPREFIX</td>
<td>35</td>
</tr>
<tr>
<td>Deleted-words-and-docs-for-idf.txt</td>
<td>76</td>
</tr>
<tr>
<td>Deleted-words-for-frequency.txt</td>
<td>76</td>
</tr>
<tr>
<td>DIALOG</td>
<td>32</td>
</tr>
<tr>
<td>Dictionary</td>
<td>54, 62</td>
</tr>
<tr>
<td>Dictionary.sorted</td>
<td>63</td>
</tr>
<tr>
<td>Dictionary.unsorted</td>
<td>62</td>
</tr>
<tr>
<td>Discrimination coefficient</td>
<td>55</td>
</tr>
<tr>
<td>Discrimination Coefficients</td>
<td>89</td>
</tr>
<tr>
<td>Discrimination coefficient</td>
<td>75</td>
</tr>
<tr>
<td>Discrimination-coefficients.txt</td>
<td>76</td>
</tr>
<tr>
<td>Discriminator</td>
<td>90</td>
</tr>
<tr>
<td>DNA</td>
<td>13</td>
</tr>
<tr>
<td>DocCount</td>
<td>35</td>
</tr>
<tr>
<td>Document clusters</td>
<td>55</td>
</tr>
<tr>
<td>Document frequency vector</td>
<td>88</td>
</tr>
<tr>
<td>Document hyperspace</td>
<td>71</td>
</tr>
<tr>
<td>Document vectors</td>
<td>71</td>
</tr>
<tr>
<td>Document-clusters-by-title.txt</td>
<td>76</td>
</tr>
<tr>
<td>Document-clusters.txt</td>
<td>77</td>
</tr>
<tr>
<td>Document-document</td>
<td>55, 76</td>
</tr>
<tr>
<td>Document-Document</td>
<td>121</td>
</tr>
<tr>
<td>Document-document-matrix.txt</td>
<td>77</td>
</tr>
<tr>
<td>Document-term matrix</td>
<td>54, 75, 79, 88, 97</td>
</tr>
<tr>
<td>Document-term-matrix-weighted.txt</td>
<td>77</td>
</tr>
<tr>
<td>Egrep</td>
<td>33, 37</td>
</tr>
<tr>
<td>Global arrays</td>
<td>44</td>
</tr>
<tr>
<td>Grep</td>
<td>33pp, 37</td>
</tr>
<tr>
<td>Guanine</td>
<td>13</td>
</tr>
<tr>
<td>Hierarchical indexing</td>
<td>43</td>
</tr>
<tr>
<td>Hyperspace density</td>
<td>90</td>
</tr>
<tr>
<td>Term</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>IDF</td>
<td>83, 87</td>
</tr>
<tr>
<td>IDF weights</td>
<td>97</td>
</tr>
<tr>
<td>Idf-weights.sorted.txt</td>
<td>77</td>
</tr>
<tr>
<td>Index.script</td>
<td>60</td>
</tr>
<tr>
<td>Indices</td>
<td>44</td>
</tr>
<tr>
<td>Indirection</td>
<td>140</td>
</tr>
<tr>
<td>International Union for Pure and Applied Chemistry</td>
<td>49</td>
</tr>
<tr>
<td>Inverse document frequency</td>
<td>55, 75</td>
</tr>
<tr>
<td>Inverse Document Frequency</td>
<td>87</td>
</tr>
<tr>
<td>Inverse Document Frequency weight</td>
<td>77, 87</td>
</tr>
<tr>
<td>Inverse Document Frequency Weight</td>
<td>83, 87</td>
</tr>
<tr>
<td>Inverted file</td>
<td>31</td>
</tr>
<tr>
<td>Inverted index</td>
<td>42, 152</td>
</tr>
<tr>
<td>IUPAC</td>
<td>49</td>
</tr>
<tr>
<td>Jaccard</td>
<td>74</td>
</tr>
<tr>
<td>Jaccard similarity</td>
<td>77</td>
</tr>
<tr>
<td>James Joyce</td>
<td>64</td>
</tr>
<tr>
<td>JOSS</td>
<td>43</td>
</tr>
<tr>
<td>Key Word Alongside Context</td>
<td>66</td>
</tr>
<tr>
<td>Key Word In Context</td>
<td>66</td>
</tr>
<tr>
<td>Key Word Out of Context</td>
<td>66</td>
</tr>
<tr>
<td>King James Bible</td>
<td>50</td>
</tr>
<tr>
<td>KWAC</td>
<td>66</td>
</tr>
<tr>
<td>KWIC</td>
<td>66</td>
</tr>
<tr>
<td>KWOC</td>
<td>66</td>
</tr>
<tr>
<td>Library of Congress Classification System</td>
<td>50, 139</td>
</tr>
<tr>
<td>Lockheed</td>
<td>32</td>
</tr>
<tr>
<td>Logical operators</td>
<td>40</td>
</tr>
<tr>
<td>Luhn</td>
<td>57</td>
</tr>
<tr>
<td>MAXDOCS</td>
<td>35</td>
</tr>
<tr>
<td>MCA</td>
<td>83, 92</td>
</tr>
<tr>
<td>MCA weights</td>
<td>97</td>
</tr>
<tr>
<td>Medical Subject Headings</td>
<td>139</td>
</tr>
<tr>
<td>MEDLINE</td>
<td>23, 152</td>
</tr>
<tr>
<td>MeSH</td>
<td>139, 152</td>
</tr>
<tr>
<td>Microsoft SQL Server</td>
<td>43</td>
</tr>
<tr>
<td>Modified Centroid Algorithm</td>
<td>83, 91</td>
</tr>
<tr>
<td>Multi-core processor</td>
<td>37</td>
</tr>
<tr>
<td>Multidimensional</td>
<td>43</td>
</tr>
<tr>
<td>Mumps</td>
<td>43</td>
</tr>
<tr>
<td>MumpsRO</td>
<td>21</td>
</tr>
<tr>
<td>MySQL</td>
<td>43</td>
</tr>
<tr>
<td>National Center for Biotechnology Information</td>
<td>14</td>
</tr>
<tr>
<td>National Institute for Standards and Technology</td>
<td>22</td>
</tr>
<tr>
<td>National Library of Medicine</td>
<td>139</td>
</tr>
<tr>
<td>NCBI</td>
<td>14</td>
</tr>
<tr>
<td>NIST</td>
<td>22</td>
</tr>
<tr>
<td>NLM</td>
<td>139</td>
</tr>
<tr>
<td>Normalized Word Frequencies</td>
<td>82</td>
</tr>
<tr>
<td>Null dictionary</td>
<td>57</td>
</tr>
<tr>
<td>Open Directory Project</td>
<td>139</td>
</tr>
<tr>
<td>OR</td>
<td>36</td>
</tr>
<tr>
<td>Oracle RDBMS</td>
<td>43</td>
</tr>
<tr>
<td>Parallel</td>
<td>37, 39</td>
</tr>
<tr>
<td>Parallel execution</td>
<td>39, 63</td>
</tr>
<tr>
<td>Parallel processing</td>
<td>21</td>
</tr>
<tr>
<td>Perl</td>
<td>34</td>
</tr>
<tr>
<td>Phrases</td>
<td>116</td>
</tr>
<tr>
<td>Popen()</td>
<td>39</td>
</tr>
<tr>
<td>Position</td>
<td>75</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>43</td>
</tr>
</tbody>
</table>