Implementing Information Retrieval

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

http://www.cs.uni.edu/~okane/
http://threadsafebooks.com/
Table of Contents

1 Introduction ........................................................................................................................................... 7
  1.1 What is Information Retrieval? ........................................................................................................ 7
  1.2 Text Indexing and Retrieval ............................................................................................................. 8

2 Software Overview ................................................................................................................................. 9
  2.1 Indexing and Retrieval Modules ........................................................................................................ 9

3 Database Software for Document Indexing ........................................................................................... 12
  3.1 Overview ........................................................................................................................................... 12
  3.2 What is Mumps? .................................................................................................................................. 13
  3.3 Mumps Global Array Database Overview ......................................................................................... 16

4 The Experimental Document Collection ............................................................................................... 23
  4.1 Modified OHSUMED File ................................................................................................................ 23
  4.2 Document Preprocessing .................................................................................................................. 26

5 Boolean Searching and Inverted Files ...................................................................................................... 30
  5.1 Simple Boolean Searching Using grep ............................................................................................ 32
    5.1.1 Regular Expression Searching with grep .................................................................................. 32
    5.1.2 Boolean Searching with Regular Expressions ........................................................................... 33
  5.2 A Boolean Search in Mumps ............................................................................................................. 38

6 Basic Indexing Concepts ........................................................................................................................ 41
  6.1 Indexing Models ............................................................................................................................... 41
  6.2 Indexing Vocabularies ....................................................................................................................... 42
    6.2.1 Controlled Vocabularies ............................................................................................................ 42
    6.2.2 Uncontrolled Vocabularies ........................................................................................................ 44

7 Vector Space Model ................................................................................................................................ 46
  7.1 Overview ........................................................................................................................................... 46
  7.2 Basic Similarity Functions ............................................................................................................... 49
  7.3 Document Indexing Flow Chart ....................................................................................................... 51

8 Representing the Vector Space Model in Mumps .................................................................................... 54
  8.1 Implementing the Vector Space Model in Mumps ........................................................................... 54
  8.2 Vector Space Global Arrays ............................................................................................................ 55
  8.3 Result Files ........................................................................................................................................ 56
  8.4 Run Time Settings and Defaults ..................................................................................................... 57

9 Identifying Good Indexing Terms ........................................................................................................... 59
  9.1 Stop Lists ......................................................................................................................................... 59
  9.2 Building an Enhanced Stop List ...................................................................................................... 61
  9.3 Basic Word Dictionary ..................................................................................................................... 63
  9.4 Zipf’s Law ......................................................................................................................................... 66
  9.5 KWIC/KWOC Indices ...................................................................................................................... 67
  9.6 Reducing Words to Stems ................................................................................................................. 69

10 Document-Term Matrix ......................................................................................................................... 72
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]

Information retrieval systems are generally composed of two parts:

1. An indexing engine that analyses digitized collections of information, often text but also images, videos, audio, scientific and clinical data, for the purposes of:
   (a) identifying and extracting tokens that are indicative of the content, and
   (b) organizing and storing these tokens in a structured, searchable semantic knowledge base.

2. An interactive search engine that is used to:
   (a) construct and formulate queries in terms of the semantic structures built by the indexing engine,
   (b) execute these, and,
   (c) meaningfully report, rank, visualize and interpret the results.

Users interact with IR systems in many ways. These include text queries as well as a wide variety of graphical interfaces that facilitate navigation, visualization and interpretation of retrieved content.

An IR system matches queries with objects based on partial matches and probabilistic analysis of the similarities between queries and objects.

Thus, for example, a query to a text-based IR system working on a collection of medical text that seeks information on the broad search key antibiotics will identify many documents. Some of these will deal with antibiotic usage while many others will deal with related topics such as research and development, side effect and so on. Many items retrieved may not actually contain the search term itself but be related to the topic by semantically similar terms such as infection control, antiseptic, antibacterial, germicidal or as specific antibiotic agents such as penicillin, amoxicillin, ampicillin, erythromycin and so on. Results will normally be presented based on a ranking to indicate which documents the IR system's algorithms determine most likely best satisfied the user's query.

While there are indexing and retrieval systems for many areas such as video, images, audio and so forth, in this book, we will concentrate on the basic algorithms and procedures associated with text indexing and retrieval.
1.2 Text Indexing and Retrieval

Perhaps the most common form of information retrieval involves text retrieval. For most people, this 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. Most queries, however, are just a sequence of one or more words. In some cases, however, queries are in the form of natural language sentences.

A basic query might be consist of a single overly broad term such as *aviation*. Typically, the results returned would include many references. These would include articles on 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 usually present results in a ranked or ordered arrangement as determined by algorithms that estimate relevancy of each result to the query. While 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 someone whose profile is that of a grade school student will be significantly different than those returned for a someone whose profile is that of 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.
2 Software Overview

2.1 Indexing and Retrieval Modules

A document indexing system is organized much like an assembly line. Each
document (or query in the case of retrieval) is processed by a series of steps to
produce a final result. For each of these steps, there are several different algorithms
that can be used, as we will see.

However, not all algorithms work equally well on all types of document
collections. Some algorithms perform well on collections with well defined
vocabularies, for example medicine and science, while others work better on
collections of ordinary prose. An algorithm that works well on one collection type may
work poorly on another.

Another overall concern is computational complexity. In some cases there are
algorithms that perform well but are some computationally intensive that they are
impractical when applied to large collections and thus impractical.

Thus, there is no one, correct way to build a document indexing or retrieval
system. Instead, effective systems are built by selecting those functional building
blocks that work best for the document collection at hand.

The optimal way to select the best combination of algorithms is by iterative
experimentation where the designer examines the effects of different combinations of
procedures and tuning parameters on overall performance.

Thus, in order to explore the effects of different combinations of indexing and
retrieval algorithms, these have been implemented as collection of interoperable,
modular programs that accompany this book.

Overall, the software modules implement the Vector Space Model first proposed
by Salton [Salton 1968] which is discussed in detail below.

The accompanying package includes:

1. a set of 65 indexing and retrieval programs written in Mumps and/or Bash
   script;
2. a corpus of nearly 300,00 medical journal abstracts; and
3. a Mumps language/database interpreter.

These are available for download at:

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

The modules in the package 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 the following data structures:

1. document-term matrix,  
2. term-document matrix,  
3. term-term matrix  
4. document-document matrix,  
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 also 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 MeSH¹,

¹ National Library of Medicine Medical Subject Headings
2. KWIC/KWOC\textsuperscript{2} indices,
3. n-grams [Manning 1999] and

Selection of which programs to run, in what order and with what parameters is determined by a large Bash script file known as \textit{index.script} (see Chapter 21 on page 156).
3 Database Software for Document Indexing

3.1 Overview

When working with a document collection of any meaningful scope, vectors, matrices and file structures can quickly grow to enormous size. The information retrieval system was tested using a corpus of documents concerning computer science subjects.

Implementing the Vector Space model requires several very large vectors and sparse matrices. By large, we mean matrices whose row count is the number of documents in the collection and whose column count is the number of words in the vocabulary.

As we will see in greater detail below, one of the central data structures used in the Vector Space model is called the document-term matrix. In this matrix, each row represents a document and each column represents a word in the vocabulary. Initially, the elements of the matrix indicate the frequency of occurrence of a word in a document. This is one of several multi-dimensional matrices and vectors that are fundamental to the Vector Space model.

In the case of the relatively small test collection provided with the software distribution, where there are approximately 300,000 abstracts and a candidate vocabulary of about 64,000 where each matrix element would require 4 bytes, this would mean a document-term matrix of approximately 300,000* 64,000 or 19,200,000,000 elements with a potential storage requirement of 76,800,000,000 bytes and this is only one matrix (usually several others are required) for a relatively small collection of documents. However, the matrices are also very sparse. Each row usually only has a tiny number of actual elements, ten to twenty, in many cases.

Another issue concerns matrix access. While the rows are document numbers, the columns are words. Thus, the matrices and vectors must be able to be addressed not only by row or column number, but also by text.

Thus, when planning an implementation strategy for the Vector Space model, it is very useful that the programming language used have built-in support for very large, text indexed, sparse matrices as well as strong string manipulation capabilities.

From a database point of view, there have been only a basic few choices available to implement a large, sparse matrix that give reasonable performance. Among those that have been used in the past are:

Coded Tuples

In this approach is to represent each row (document) as a collection of tuples each of which consists of a word token and a frequency. The word token identifies the vocabulary term and the frequency gives the frequency of the term in the document. In this scheme, a minimum of eight bytes would be required for each tuple (four bytes to represent a number identifying the term and four bytes to represent the frequency).

In order for the file to be quickly accessed, either each document row needs to be a fixed length record or, if the records are of variable length, a separate inverted index file, probably B-tree based, needs to be constructed.
Allowing for up to 30 terms per row in a fixed length record format, this would require a 72,000,000 byte file to represent the test collection (300,000*20*8). In a variable record format, the savings in row length would be offset by the size of the inverted index.

It is possible, however, at least during the document indexing phase, to process records sequentially (as was done in the earliest Salton experiments) in which case variable length records could be used without the need for an inverted index. This, however, would not be a very fast option.

**Bit Maps**

Alternatively, a bit mapping model represents documents as positional binary vectors with a "1" indicating that a given term occurs in a document and a "0" indicating that it does not. While this is done to conserve space and improve vector access time, it also precludes the storage of important information concerning the relative weight or strength of a term in a document. This approach was used in some early systems where the vocabulary was limited but, when using the test data set, a positional binary vector representation of each document would be 8,000 bytes (64,000 bits) in length for a total of 2,400,000,000 bytes for the collection as a whole.

**Relational Database Tables**

A relational database representation would be essentially similar to coded tuples. In a relational implementation, the document-term matrix would be mapped to a three column table with one column being the document number, one the word token, and one the frequency. The size requirements would vary by RDBMS chosen but can be expected to be larger than the coded tuples as the document number is redundantly repeated for each vocabulary word. Access to, and manipulation of, records would be by means of SQL queries, some of which are quite complex (see Chapter 16) and, depending on the system, time consuming.

While all of the storage methods listed above can be accessed through programs written in the usual languages, C, C++, Java, and so on, programs written in none of these languages structurally visualizes the database as multi-dimensional matrices as used by the model.

In this text, however, we use the Mumps language as the implementation language and contend that it is uniquely suited for Vector Space model indexing and retrieval implementations.

**3.2 What is Mumps?**

A quick tutorial and interpreter, both free, for the language are available here:

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

The link above also has a link to a full book on the Mumps language. The reader is advised to look through the tutorial.

---

3 This is information conveyed by the record number in the fixed length record tuple scheme, or by the inverted index in the variable length record version.
The interpreter is mainly intended to work with Linux but it will run on Windows if Cygwin is installed. A better solution if you are using Windows or an Apple system is to install Linux Mint in a virtual machine such as Oracle's Virtual Box (free).

Mumps is a relatively simple language and most people familiar with programming languages should be able to read it with minimal effort. However, as it's syntax is somewhat rigid, writing programs requires some training.

In the following, we survey the history of the language and highlight its main database features. There will be additional details in subsequent chapters as well.

Mumps (Massachusetts General Hospital Utility Multi-programming System) is a general purpose programming language environment that provides ACID (Atomic, Consistent, Isolated, and Durable) database access by means of program level subscripted arrays and variables. The Mumps database allows schema-less, key-value access to disk resident data organized as trees that may also be viewed as large, sparse multi-dimensional arrays.

Beginning in 1966, Mumps (also referred to as M), was developed by Neil Pappalardo and others in Octo Barnett's lab at the Massachusetts General Hospital (MGH) on a PDP-7, the same architecture on which Unix was being implemented at approximately the same time.

Initial experience with Mumps was very positive and it soon was ported to a number of other architectures including the PDP-11, the VAX, Data General, Prime, and, eventually, Intel x86 based systems, among others. It quickly became the basis for many early applications of computers to the health sciences.

When Mumps was originally designed, there were very few general database systems in existence. The origin of the term 'database' itself dates from this period. Such systems as existed, were mainly ad hoc application specific implementations that were neither portable nor extensible. The notion of a general purpose database design was just developing.

One of the first attempts to implement of a general purpose database system was GE/Honeywell's initial IDS - Integrated Data Store - which was developed in the mid-60s. Experience with this system lead to the subsequent organization and creation of the CODASYL DBTG (Committee on Data Systems Languages - DataBase Task Group) whose Network Model database was proposed (1969).

The Network Model was very complex and was implemented, in varying degrees, by only a few vendors. All of these were mainframe based. Most notable of these were GE/Honeywell's IDS/2, Cullinet's Integrated Database Management System (IDMS), Univac's DMS-1100, and Digital Equipment Corporation's DEC-10 based DBMS32.

At about the same time, IBM's IMS (Information Management System), was being developed in connection with the NASA Apollo program. It was first placed into service in 1968 running on IBM 360 mainframes. IMS, which is still in use today, is, like Mumps, hierarchical in structure. IMS is reputed to be IBM's highest revenue software product.

The table based relational database model was initially proposed by E. F. Codd in 1970 but it wasn't until 1974 that IBM began to develop System R, the first system to
utilize the model, as a research project. The first commercially available relational database system was released by Oracle in 1979.

In late 1960s mini-computers, although expensive, were becoming more widely available but they were still mainly confined to dedicated, mostly laboratory, applications. The operating systems available on these systems were primitive and, for the most part, single user. On many, the user was the operating system, flipping switches to manually install boot code, and loading compilers, linkers and programs from paper or magnetic tape.

DEC's RSX-11, the first commercial multi-user system on the PDP-11, was introduced in 1972. RSTS/E, a time sharing system mainly used for time-shared BASIC language programming, was implemented in 1970. Language support was likewise limited to a small set of languages such as FORTRAN, BASIC and Assembly Language. Although Unix existed at this time, it was not available outside AT&T until 1984.

Thus, in 1966 when the PDP-7 arrived at MGH, it had very little in the way of software, operating system or database support. So, as there were few options available, they started from scratch and designed Mumps to be not only a multi-user operating system, but also a language, and a database all in one.

While in those early days, Mumps, out of necessity, was its own standalone operating system, this is not the case today where Mumps programs run in Unix, Linux, OS/X, and Windows based environments.

Because of its simplicity, low cost and ease of use, Mumps quickly became popular and established itself as the basis for many medical applications. COSTAR (COmputer-STored Ambulatory Record), for example, was a medical record, fiscal management and reporting system, developed in the mid-1970s for use in ambulatory care settings and it was widely used worldwide.

Today, Mumps programs are employed extensively in financial and clinical applications. If you've been to a doctor, been seen at a hospital, or used an ATM machine, your data has probably been processed by a Mumps program.

Mumps programs are the basis of the U.S. Veterans Administration's computerized medical record system VistA (Veterans Health Information Systems and Technology Architecture), the largest of its kind in the world. VistA is a collection of 80 different software subsystems that support the largest medical records system in the United States. It supports the medical records of over 8 million veterans, is used by 180,000 medical staff at 163 hospitals, more than 800 clinics, and 135 nursing homes.

Mumps is used by many health care organizations including Allscripts, Epic, Coventry Healthcare, EMIS, Partners HealthCare (including Massachusetts General Hospital), MEDITECH, GE Healthcare (formerly IDX Systems and Centricity), Sunquest Information Systems, DASA, Quest Diagnostics, and Dynacare, among others.

Some of the largest and most well known hospitals use Mumps based electronic health records systems. These include Kaiser Permanente, Cleveland Clinic, Johns Hopkins Medicine Hospitals, UCLA Health, Texas Health Resources, Massachusetts General Hospital, Mount Sinai Health System in New York City and the Duke University Health System.
Among financial institutions, it is used by Ameritrade, the Bank of England, and Barclays Bank, as well as others.

For its database design, Mumps originally used a hierarchical model because this closely matched the tree structured format of medical records. To represent database trees in the language, they decided to use array references where each successive array index was part of a path description from the root of the array to intermediate and terminal nodes. They called these disk resident structures *global arrays*.

While in those days, Mumps, out of necessity, was its own standalone operating system, this is not the case today where Mumps programs run in Unix, Linux, OS/X, and Windows based environments.

Today, Mumps programs are employed extensively in financial and clinical applications. If you've been to a doctor, been seen at a hospital, or used an ATM machine, your data has probably been processed by a Mumps program.

Mumps programs are the basis of the U.S. Veterans Administration's computerized medical record system VistA (*Veterans Health Information Systems and Technology Architecture*), the largest of its kind in the world. VistA is a collection of 80 different software subsystems that support the largest medical records system in the United States. It supports the medical records of over 8 million veterans, is used by 180,000 medical staff at 163 hospitals, more than 800 clinics, and 135 nursing homes.

Mumps is used by many health care organizations including Allscripts, Epic, Coventry Healthcare, EMIS, Partners HealthCare (including Massachusetts General Hospital), MEDITECH, GE Healthcare (formerly IDX Systems and Centricity), Sunquest Information Systems, DASA, Quest Diagnostics, and Dynacare, among others.

Some of the largest and most well known hospitals use Mumps based electronic health records systems. These include Kaiser Permanente, Cleveland Clinic, Johns Hopkins Medicine Hospitals, UCLA Health, Texas Health Resources, Massachusetts General Hospital, Mount Sinai Health System in New York City and the Duke University Health System.

Among financial, institutions it is used by Ameritrade, the Bank of England and Barclays Bank, as well as others.

### 3.3 Mumps Global Array Database Overview

Mumps implements, as an integral part of the language, a hierarchical and multi-dimensional database paradigm. When viewed as trees, data nodes can addressed as path descriptions in a manner which is easy for a novice programmer to master in a relatively short time. Alternatively, the trees can be viewed as sparse n-dimensional matrices of effectively unlimited size which are especially well suited to implement aspects of the Vector Space Model.

Mumps supports built-in string manipulation operators and functions that provide programmers with access to efficient methods to accomplish complex string manipulation and pattern matching operations.

The version of Mumps used here has been specially augmented with many additional built-in functions designed for document indexing applications.
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 (^).

Syntactically, global arrays differ from local arrays only in that global array names are prefixed by an up-arrow character (^).

For example, consider an array reference of the form \(^\text{root("p2","m2","d2")}\). This could be interpreted to represent a cell in a three dimensional matrix named \(^\text{root}\) indexed by the values ("p2","m2","d2") or, alternatively, it could be interpreted as a path from the origin (\(^\text{root}\)) 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 \(^\text{root}, ^\text{root("p2")}, ^\text{root("p2","m2")}\) as well as \(^\text{root("p2","m2","d2")}\) or, at none at all.

Because Mumps arrays can have many dimensions (limited by an implementation defined maximum line length), when viewed as trees, they can be of many levels of depth and these levels of depth may differ 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 \($\text{data()}\) and \($\text{order()}\). The first of these, \($\text{data()}\), reports if a node exists, if it has data and if it has descendants. The second, \($\text{order()}\), 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 1, 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 1, 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 $order()$ 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 1 can be created with the code shown in Program 1.

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

```mumps
#!/usr/bin/mumps
set $^\text{root}(1,37)=1
set $^\text{root}(1,92,77)=2
set $^\text{root}(1,92,177)=3
set $^\text{root}(5)=4
set $^\text{root}(8,1)=5
set $^\text{root}(8,100)=6
set $^\text{root}(15)=7
set $^\text{root}(32)=8
set $^\text{root}(32,5)=9
set $^\text{root}(32,5,3)=10
set $^\text{root}(32,5,8)=11
set $^\text{root}(32,123)=12
```

4 Mumps assignment statements require the keyword set.

Figure 1 Global Array Tree
In some cases, an empty string (nothing) is stored at a leaf node of a global array because the path description itself is the actual data. For example, consider the global array containing clinical laboratory tests seen in Program 2.

In this case, the first index is a patient id number, the second is the name of a lab test, the third is the date of the test and the fourth is the test result. In actuality, no further information is needed: the indices contain all the data. The $data()$ and $order()$ functions (see below), can be used to extract the values from what, in this case, is more a four column table than a tree.

A more realistic example can be seen in the Figure 2 which shows a section of the Medical Subject Headings (MeSH) as developed by the U.S. National Library of Medicine.

In Figure 2, a sub-tree of the National Library of Medicine MeSH hierarchy is represented along with the corresponding Mumps code to create the sub-tree.

Quote marks around the numeric indices are not required except in cases where you want to preserve leading zeros as is the case in some nodes.

As noted above, data may be stored not only at fully subscripted terminal tree elements but also at other levels. For example, a three dimensional matrix named mat1, could be initialized as shown in Program 3. In this example, all the elements of a traditionally structured three dimensional matrix of 100 rows, 100 columns and 100 planes are initialized to zero. Note: the for command is the iterative loop command in Mumps. Its arguments are a loop variable, an initial value, an increment, and a final value. This matrix is similar to the C matrix shown in Program 4 although the Mumps matrix can store data values other than ints.

However, unlike other programming languages, there are other nodes of the matrix which could have been initialized such as indicated by the example in Program 15.

In effect, this means that mat1 can also be a single dimensional vector, a two dimensional matrix and a three dimensional matrix simultaneously. This is not possible in traditional languages.
Furthermore, not all elements of a matrix need exist. That is, the matrix can be sparse as is shown in Program 6.

![Figure 2 MeSH Tree as a Global Array](image)

In the example in Program 6, only index values 0, 10, 20, 30, 40, 50, 60, 70, 80, and 90 are used to create each of the dimensions of the array and, consequently, only those elements of the matrix are created. The omitted elements do not exist. This also is not possible in traditional languages.

While global arrays are unique to Mumps, as a programmer, you will work with them as though they were ordinary arrays but the system interprets them as path descriptions in the system's external data files.

Global arrays may have both string and numeric indices as shown in Program 7.

```mumps
int mat1[100][100][100], i, j, k;
for (i=0, i<101; i++)
    for (j=0; j<101; j++)
        for (k=0; k<101; k++)
            mat[i][j][k]=0
```

Program 4 C Matrix

```mumps
#!/usr/bin/mumps
for i=0:1:100 do
```
The Mumps global array facility had its origin in the early use of Mumps for medical databases which are often viewed as hierarchical in nature. The Mumps global arrays were a solution to the problem of how to represent the tree-like structure of patient data in a simple and easily manipulated structure.

Clinical data is often organized by diagnostic or problem category and each problem or diagnostic code is divided into episodes of the problem organized by onset date. For a given problem and onset, the data are divided by category (medications, lab tests, orders, notes, etc.) which are further subdivided by, in the case of lab tests, test name, date, time and result.

In the code example below, the tree is named patient. The Mumps code to populate the tree might look like the example in Program 8.

The first index of the tree is the patient's SSN or other id number. At the second level is a code number indicating is lower levels contain address information, lab data, and so forth. The codes are shown as Mumps variables but the values stored in the tree are the numbers associated with each variable.
Notice at the last line that the empty string is stored at the node. In this cases, the actual data (the lab test result) is the actual value of the final index.

Also, note that each intermediate node need not be explicitly created nor contain data: the nodes representing lab, dx, lab, and so forth are not separately created. Their creation is implicit in constructing the longer paths of which they are intermediates.

In Mumps, the database is part of the language which has caused some to ask if it is a database with a language or a language with a database? A bit of both, actually.
The Experimental Document Collection

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:


4.1 Modified OHSUMED File

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

For examples of this format, see:

The original format used many very long lines that 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 3 and the revised database format is shown in Figure 5.

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.

Mauch TJ; Tuma DJ; Sorrell MF.

The meaning of the codes used in Figure 3 are shown in Figure 4.

1 .I sequential identifier
2 .U MEDLINE identifier (UI)
3 .M Human-assigned MeSH terms (MH)
4 .T Title (TI)
5 .P Publication type (PT)
6 .W Abstract (AB)
7 .A Author (AU)
8 .S Source (SO)
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 5 are given in Figure 6.

1 MH - MeSH heading term
2 TI - title
3 AB - abstract.
4 All data fields begin in column 7 and all descriptors begin in column 1
5 Each entry begins with the text STAT- MEDLINE

This revised file is named osu.medline in the distribution.
4.2 Document Preprocessing

The *bash* script file `index.script` (see Chapter 21) contains the instructions to pre-process, index and do a retrieval test on the collection. It will be discussed in detail later but initially, we need to look at part of it the pre-processes the file in the format shown in Figure 5 to become the format shown in Figure 7 below. The file in the format of Figure 7 is the file that we will actually use as input to the document indexing software.

The script file `index.script` in Program 63 works on an input file named `$DB.medline` where `$DB` is a *bash* variable obtained from the invoking command line parameter. The default value of `$DB` is `osu` if none is supplied.

The *bash* script file `index.script` checks if file `$DB.converted` exists. If not, it is created (this can be time consuming for large input files). This is shown in Program 11.

If `$DB.converted` is to be built, it is first preprocessed by a group of `sed` commands that remove punctuation, numbers, and short words to create the file `tmp.tmp`. This file is passed to `reformat.mps` (Program 11) whose output is further processed by `sed`. The output of the `sed` actually creates `$DB.converted`.

The final format of the file `$DB.converted` is as follows:

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

Initially `map.mps`, shown in Program 9, 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. The program `map.mps` also write a file consisting of the abstract titles, truncated to 93 characters. Each title is preceded by the document number of the document in which the title appeared. These are used later in the retrieval phase.

---

5 “…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*
Input to the series of piped *sed* commands in Program 10 is initially from $DB.medline$. Each successive *sed* command sends its output to the next and, on a 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.

```
#!/usr/bin/mumps
# map.mps December 11, 2014
# Copyright 2014 Kevin C. O'Kane

set M=$zgetenv("MAXDOCS")
if M="" set M=1000000
open 1:"titles.list,new"
kil `^map

set D=0
for do if D>M quit
 . use 5
 . set o=$ztell read line
 . if `$test break
 . if $extract(line,1,2)="TI" do quit
 .. set D=D+1
 .. use 1 write D," ",$extract(line,7,100),!
 .. use 5
 .. set `^map(D)=off
 .. quit
 . if $extract(line,1,2)="MH" quit
 . if $extract(line,1,13)="STAT- MEDLINE" set off=o quit
 . if $extract(line,1,2)'="AB" quit
 . for do
 .. read line
 .. if `$test break
 .. if line="" break

close 1
```

Program 9 map.mps

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\(^6\): \[+"=%,;:()/
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

\(^6\) Note that ] and [ appear at the beginning and in that order. This is because [...] has a meaning in *sed*. 27
9. Find and remove hyphens followed by a blank
10. Find and remove hyphens preceded and followed by blanks
11. Find and remove asterisks

```bash
# 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/[[]{2,5}]/ /g' | \
  sed -e 's/ / /g' | \
  sed -e 's/ / /g' | \
  sed -e 's/ / /g' | \
  sed -e 's/ / /g' | \
  sed -e 's/ / /g' | \
  sed -e 's/ / /g' | \
  > tmp.tmp

  reformat.mps < tmp.tmp | \
  sed -e 's/\{2,6\}/ /g' | \
  sed -e 's/[a-zA-Z\-\{1,3\}\{1,3\}\{1,3\}\{1,3\}\{1,3\}]/ /g' | \
  sed -e 's/ / /g' | \
  > $DB.converted
fi
```

Program 10: Document pre-processing

The file created by the `sed` commands (`tmp.tmp`) is passed to `reformat.mps` which is shown in Program 11. 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 for display purposes to 93 characters.

Output from `reformat.mps` is also processed by several `sed` expressions that remove words of three of fewer characters and short fragments of words with hyphens.

```
#!/usr/bin/mumps
# reformat.mps November 22, 2011
# Copyright 2014 Kevin C. O'Kane

set M=$zgetenv("MAXDOCS")
if M="" set M=1000000

set D=0
```
for do if D>M quit
.. set o=$ztell read line
.. if '$test break
.. if $extract(line,1,2)="TI" do quit
.. set D=D+1
.. use 5
.. write off," ",D," ",$zlower($e(line,7,2048))," 
.. quit
.. if $extract(line,1,2)="MH" quit
.. if $extract(line,1,13)="STAT-MEDLINE" set off=o write:D'=0 ! quit
.. write $extract(line,7,2048)," 
.. for do
.. read line
.. if '$test break
.. if line="" break
.. set line=$zlower($extract(line,7,255))
.. write line," 

Program 11 reformat.mps

An example of \$DB.converted shown in Figure 7. In this example:

1. the original text begins at offset 148710\(^7\) in the osu.medline file, and
2. the document number is 130.

148710 130 chromaticity achromaticity evidence a functional differentiation visual field defects the visual field defects patients had suffered lesions the postgeniculate part the primary visual projection red-green discrimination achromatic target detection tested addition these patients were tested detection red green targets targets were presented a photopic achromatic background that red green targets differed from background both intensity in wavelength whereas achromatic target differed intensity only patients showed evidence discriminating between and green targets patients could also detect colour targets none could detect achromatic that presented the same retinal position these results imply that wavelength intensity information treated differentially suggest that these patients possess residual colour-opponent channels that subserve defective part the visual field

Figure 7 Modified OSUMED database item

When processing the documents, we begin with the full text where the only modification is that all letters have been converted to lower case. In later steps, we:

1. remove commonly occurring terms (stop list words), and
2. replace words with their semantic stems,
3. remove numbers and short words.

By semantic stem, we mean a term which is the root of several variant forms of the word. For example, compute for computer, computed, computing, compute, computes, computational and so forth. In many cases, this may simply mean a changing plural forms to singular while in others is can result in significant prefix and suffix removal. In some cases, semantic tables are used that collect many synonyms and variants into a single collective code.

\(^7\) The number of bytes from the start of the file.
5 Boolean Searching and Inverted Files

Before we examine the Vector Space model we should explore some very basic search techniques because some aspects of these are used with the Vector Space model.

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 as a set of documents. When two words are \textit{and'ed}, the sets are intersected; when two words are \textit{or'ed}, the sets are combined (duplicate identifiers are removed). When a \textit{NOT} is used, the \textit{not'ed} set is subtracted from the first set. Parentheses are used as needed to express the order of evaluation. For example:

\begin{enumerate}
  \item COMPUTERS AND MEDICINE
  \item COMPUTERS AND (ONCOLOGY OR GASTROENTEROLOGY OR CARDIOLOGY)
  \item COMPUTERS NOT ANALOG
\end{enumerate}

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

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

\begin{enumerate}
  \item the documents themselves or
  \item assigned to the documents from a controlled vocabulary or dictionary.
\end{enumerate}

A Boolean search can be conducted in two ways:

\begin{enumerate}
  \item Each document in the collection can be inspected and evaluated in terms of the Boolean search expression, or;
  \item The Boolean search expression can be evaluated by means of an inverted index file.
\end{enumerate}

An inverted file is one in which, for each word in the indexing vocabulary, there is a set of pointers to those documents that contain the words.
In a Boolean search using an inverted file system, sets of document numbers (or other referential tokens) are intersected or joined according to the logical expression. The resulting document numbers are then used to retrieve the actual text. An example is shown in Figure 8.

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 are examples which were used in the IBM STAIRS system [Blair 1996] (also 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:

ONCOLOGY=4
CARDIOLOGY=5
VIROLOGY=3
GASTROENTEROLOGY=2
THRESHOLD=6
ONCOLOGY OR CARDIOLOGY OR VIROLOGY OR GASTROENTEROLOGY

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.

5.1 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*.

5.1.1 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.

### 5.1.2 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\(^8\):

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

\(^8\) The backslash at the line end indicates that the line is continued on the next line.
could be rendered as shown in Program 12

```bash
#!/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 12 Boolean OR search with grep

The $$ figure in Program 12 is a builtin bash variable that contains the process ID of the current shell. Since no two process IDs are concurrently same\(^9\), 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` concatenates\(^{11}\) 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.

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:

\(^9\) 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.

\(^{10}\) Process IDs are ultimately re-used but the same process ID is never assigned to two processes executing at the same time.

\(^{11}\) the operator `>>` means concatenate output at the end of the target file.
(alcohol OR gambling) AND smoking

could be written as shown in Program 13. In Program 13 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.

```
#!/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-z]*" $db.converted > /tmp/$pid.1.tmp
15 grep "gambl[a-z]*" $db.converted >> /tmp/$pid.1.tmp
16 sort /tmp/$pid.1.tmp | uniq > /tmp/$pid.2.tmp
17 grep "smoking" /tmp/$pid.2.tmp | wc -l; rm /tmp/$pid.*.tmp
```

Program 13 Boolean AND and OR search with grep

An alternative grep expression for line 6 Program 13 might be:

grep "alcohol[a-z]*\\|gambl[a-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\(^\text{12}\), this modification actually
resulted in slower execution\(^\text{13}\). Not all intuitive solutions work! Always test.

Program 13 could be modified to improve performance by piping the output of
line 8 directly as input to the grep in line 9. This would reduce the number of
intermediate files.

Another variation of Program 13 would involve parallel execution of the grep
functions on lines 6 and 7. On a multi-core processor, this should improve execution
substantially. The modified code is given in Program 14. In this program, the 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 sort function on line 9. The bash
wait\(^\text{14}\) command on line 8 causes the script to pause until all background processes in
the current shell have completed.

\(^\text{12}\) See boolean-grep-02b.script

\(^\text{13}\) On a 6 core AMD processor under Linux Mint LMDE in April, 2014. Your mileage may
vary.

\(^\text{14}\) Without arguments, wait waits for all background processes to complete. If you
specify one or more process IDs as arguments to wait, wait waits only for those you
specify. The process ID of the most recent command is in the bash builtin variable $!
which can be assigned to a shell variable for later reference.
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 15:

Final output (from `egrep`) will appear on stdout. `Egrep` is an extended version of `grep` that permits additional operators, the `+` operator in this case. In the above, the `egrep` receives the output of the final `grep` and matches the first two numeric sequences on each line. The `^` operator means 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).

To display the titles from the above, use Programs 16 and 17.

15 The `^` operator means beginning of line, the `$` operator means end of line.
#!/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-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/$pid.2.tmp | egrep -o "^[0-9]+ [0-9]+ " | get-docs.mps
rm /tmp/$pid.*.tmp

Program 16 Display titles from grep search

where get-docs.mps is given in Program 17.

#!/usr/bin/mumpsRO
# boolean.mps Feb 14, 2014
# Copyright 2014 Kevin C. O'Kane

set i=0
for do
    read line
    if '$test break
    set docnbr=$p(line," ",2)
    if '$data(^title(docnbr)) quit
    write docnbr,?10,$e(^title(docnbr),1,80),!
write "Only titles in ^title() displayed.",!
halt

Program 17 Program to retrieve and display titles

As an example, the query corresponding to (alcohol OR gambling) AND smoking yields the results in Figure 9. based on the first 5,000 abstracts (long lines truncated).

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1461</td>
<td>Health and economic implications of a tobacco-free society.</td>
</tr>
<tr>
<td>1749</td>
<td>Hip fracture and the use of estrogens in postmenopausal women. The Framingham</td>
</tr>
<tr>
<td>3012</td>
<td>Pulmonary impairment in a cotton textile factory in Nigeria: is lifetime alcohol</td>
</tr>
<tr>
<td>9</td>
<td>Alcohol and the elderly: relationships to illness and smoking.</td>
</tr>
</tbody>
</table>

Figure 9 Example Boolean search with grep/egrep

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

---

16 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.
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.

### 5.2 A Boolean Search in Mumps

The code in Program 18 is a simple full-text, sequential, boolean search of the OSUMED collection written in Mumps. Program 18 produces output as shown in Figure 10. It operates on the modified OHSUMED database *osu.converted* an example of which was shown in Figure 7 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:

- 
-  
-  

where term1,...,term5 are words or terms from the vocabulary.

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

-  

internally becomes:

-  

Once a user query is converted to a Mumps expression, the syntax of the expression is tested. The expression is then applied to consecutive input lines from *osu.converted* in a loop.

Each line from *osu.converted* read into the variable *line* contains the full text of an original abstract, converted to lower case, stemmed, and devoid of punctuation.

The loop executes the expression in *exp* in an *if* statement. If the expression results in *true*, the program extracts from *line* the offset of the document in the original file.
and the document number. Based on the document number, the title\textsuperscript{17} is displayed along with the document number. The abstract could also be displayed by using the offset of the original document in the original file.

17 We assume the $^\text{title(docnbr)}$ has been previously built and contains the text of the titles for all document numbers docnbr.
The program in Program 18 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")

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

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")

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

Figure 10 Boolean search results from Program 18
6 Basic Indexing Concepts

6.1 Indexing Models

Information retrieval is the 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, as determined by Google’s relevancy algorithm, 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.

Determining the similarity between queries and documents is usually not done directly. Instead, both queries and documents are mapped into an internal representation or indexing model upon which the similarity functions can be directly calculated as diagrammed in Figure 11. The results from a query are determined by calculating the similarity between the internal representation of query and the internal representation of the documents in the context of the indexing model. Basically, this mean that queries become small documents.

![Figure 11 Overview of Indexing](image)

In this model, documents are preprocessed into the internal representation, a process called indexing. This can be a very time consuming procedure when the documents in the collection may number in the millions. Once the main collection has been indexed, however, newer documents may be added quickly.

When a query is received, it too is indexed but the cost is nominal.
6.2 Indexing Vocabularies

Traditionally, indexing was performed by experts in a subject area who read each document and classified it according to content. However, in recent years, manual indexing has been overtaken by automated indexing, of the kind performed by search engines such as Bing, Google and other online 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 indexing system is one that derives the terms, categories and hierarchies directly from the text.

6.2.1 Controlled Vocabularies

In a controlled vocabulary system, topics are described using the same preferred term or terms 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 12.

Numerous other examples abound, especially in technical disciplines where nomenclature tends to be more 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


   Mental disorders:
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 12 ACM classification system

4. International Union for Pure and Applied Chemistry Names (IUPAC Names) for chemistry Diagnostic and Statistical Manual IV nomenclature:
5. Library of Congress Classification System for a broad classification system for all works:

   http://www.loc.gov/catdir/cpso/lcco/lcco.html

6. Structural Classification of Proteins:
   http://scop.mrc-lmb.cam.ac.uk/scop/

7. Open Directory Project:
   http://www.dmoz.org/

8. For a very long list, see American Society of Indexers Thesauri Online:
   http://www.asindexing.org/site/thesonet.shtml

   In a manually indexed collection that uses a controlled vocabulary, experts trained not only in the field but also in 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 only by trained personnel who intermediated a user's queries 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.

6.2.2 Uncontrolled Vocabularies

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.

   Systems based on controlled vocabularies have many limitations, most prominent of these is the amount of manual effort to maintain them. While this investment of time
may be justified in limited areas such as medicine, it is not practical in most fields of interest.

Computer based uncontrolled vocabularies can be constructed through statistical analysis of word usage in the collection as a whole. Building systems based on derived vocabularies is the main theme of this book. In Chapter 15 on 130 we will examine indexing the OSUMEDLINE collection with the controlled MeSH vocabulary.
7 Vector Space Model

7.1 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 13 Vector space model](image-url)
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 14 Vector space queries
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.
7.2 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} (Term_{ik} \cdot Term_{jk})}{\sqrt{\sum_{k=1}^{t} (Term_{ik})^2 \cdot \sum_{k=1}^{t} (Term_{jk})^2}} \\
Sim\(_4\)(Doc\(_i\), Doc\(_j\)) = \frac{\sum_{k=1}^{t} (Term_{ik} \cdot Term_{jk})}{\min(\sum_{k=1}^{t} Term_{ik}, \sum_{k=1}^{t} Term_{jk})} \\
Sim\(_5\)(Doc\(_i\), Doc\(_j\)) = \frac{\sum_{k=1}^{t} \min(Term_{ik}, Term_{jk})}{\sum_{k=1}^{t} Term_{ik}}

Figure 17 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 term\(_i,k\) and term\(_j,k\) where term\(_i,k\) is the weight of term k in document i and 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 18 illustrate the application of the above\(^{18}\).

<table>
<thead>
<tr>
<th>(\text{Doc}_1)</th>
<th>(\text{Doc}_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3, 2, 1, 0, 0, 0, 1, 1</td>
<td>1, 1, 1, 0, 0, 1, 0, 0</td>
</tr>
</tbody>
</table>

\(\text{Sim}_1(\text{Doc}_1, \text{Doc}_2) = (2 \times 6) / (8 + 4) \rightarrow 1\) 
\(\text{Sim}_2(\text{Doc}_1, \text{Doc}_2) = 6 / (8 + 4 - 6) \rightarrow 1\) 
\(\text{Sim}_3(\text{Doc}_1, \text{Doc}_2) = 6 / \sqrt{16 \times 4} \rightarrow 0.75\) 
\(\text{Sim}_4(\text{Doc}_1, \text{Doc}_2) = 6 / 4 \rightarrow 1.5\) 
\(\text{Sim}_5(\text{Doc}_1, \text{Doc}_2) = 3 / 8 \rightarrow 0.375\)

Figure 18 Example similarity coefficient calculations

\(^{18}\) Salton 1983, pg 202-203.
Figure 19 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 19 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\(^{19}\) file access to the documents.

9. Various term weights are calculated including the inverse document frequency weight and the discrimination coefficients.

\(^{19}\) That is, access to the documents by word rather than by document number.
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 term-term matrix and a term-term proximity matrix are calculated identifying terms whose usage and proximity are similar for possible synonym identification and query supplementation.

12. A 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.
8 Representing the Vector Space Model in Mumps

As mentioned above, one of the main data structures for the Vector State model is the document-term matrix. From this, we derive many other structures such as the transposed term-document matrix, the term-term matrix, the document-document matrix and so forth.

In Mumps, suppose we have a document, say document number 555 in the collection, with three words, the document vector is:

<Chickens (3), Ducks (2), Robins (6)>

Where Word1, Word2 and Word3 are the words themselves and freq1, freq2 and freq3 are the frequencies of occurrence of the words in the document.

In Mumps we would store the document vector with the code:

```mumps
set ^docTerm(555,"Chickens")=3
set ^docTerm(555,"Ducks")=2
set ^docTerm(555,"Robins")=6
```

That is, for row 555 and columns Chickens, Ducks and Robins we store the respective frequencies of occurrence. A very simple and easily grasped visualization of the document-term matrix.

In Mumps, if an element of a matrix does not exist, it takes no space. So, while the domain of the second dimension in the document-term matrix may encompass 64,000 words, this vector only has three and takes only three units of space in the database.

But the ^docTerm matrix could be more detailed. In addition to storing the count, it could also store the relative location of each word in a document.

For example, if the word Chickens appear as the 5th, 22nd and 55th word in the document, we could represent this information with:

```mumps
set ^docTerm(555,"Chickens")=3
set ^docTerm(555,"Chickens",5)="
set ^docTerm(555,"Chickens",22)="
set ^docTerm(555,"Chickens",55)="
```

The third index gives the relative location of each instance of the word. The relative location is all that is needed so we store the empty string at these nodes. Now the matrix is both two dimensional and three dimensional, depending upon whether we wish to access location information or not.

8.1 Implementing the Vector Space Model in Mumps

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.

### 8.2 Vector Space Global Arrays

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

1. \(^{dict}(word)\)
   
   A vector indexed by a vocabulary word giving the number of times the word occurs in the collection.

2. \(^{df}(word)\)
   
   A vector indexed by a vocabulary word giving the number of documents in the collection the word appears in.

3. \(^{idf}(word)\)
   
   A vector indexed by a vocabulary word giving the inverse document frequency weight of the word in the collections as a whole.

4. \(^{mca}(word)\)
   
   A vector indexed by a vocabulary word giving the discrimination coefficient of the word in the collection as a whole.

5. \(^{doc}(d,word,pos)\)
   
   A matrix known as the document-term matrix giving at the \(^{doc}(d,word)\) level nodes where \(d\) is a document number and \(word\) 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. \(^{index}(word,d)\)
   
   The transpose of \(^{doc}(d,word)\) portion of \(^{doc}(d,word,pos)\). The position indicator is not stored in this matrix.

7. \(^{dd}(d1,d2)\)
   
   The document-document matrix giving a value indicating the similarity between \(d1\) and \(d2\).
8. $^tt(w_1,w_2)$

The term-term correlation matrix giving the an indicator of similarity or usage between word $w_1$ and word $w_2$.

9. $^tttitle(d)$

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

10. $^ttp(w_1,w_2,1)$

The proximity term-term correlation matrix where $^ttp(w_1,w_2)$ is the sum of the absolute values of the term proximities between word $w_1$ and $w_2$ and $^ttp(w_1,w_2,1)$ 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 $^doc(d,w_1,pos)$. 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).

8.3 Result Files

Figure 20 is a list of the output files created by 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>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 $^dd(d_1,d_2)$ 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</td>
</tr>
<tr>
<td>File Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</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 of 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 20 Output Files

#### 8.4 Run Time Settings and Defaults

Figure 21 lists the environment variables internal to `index.script` that control thresholds used by various programs.

<table>
<thead>
<tr>
<th>Command line argument (MAXDOCS)</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=${(MAXDOCS/MIN_DIVISOR)}</td>
<td>Words deleted if less than this number.</td>
</tr>
<tr>
<td>MAXFRACTION=${(100/MAX_DIVISOR)}</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>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 21 Run-time settings
9 Identifying Good Indexing Terms

Information retrieval pioneer Hans Luhn [Luhn 1958] believed that the resolving power of terms in a collection of text would be greatest in the middle-frequency range (see Figure 22). In this context, 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.

9.1 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 stop list. A stop list is a list of words which are not used for indexing (sometimes referred to as a 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: the, and, 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 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 19. 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 of the array $w(j)$ except for elements zero and one which contain, respectively, the byte offset and document number.
The program in Figure 19 reads each input line where each line is a document (a very long line in most cases) and writes each word to stdout if it is not in the stop list. After all the words in a line have been processed, a 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 19 Stop list removal

9.2 Building an Enhanced Stop List

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

For example, while the word *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, (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 23 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. The most frequently occurring word occurred about 180,000 times. As rank increases, frequency of occurrence drops dramatically.

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

---

20 Removal of common suffixes.

21 The most frequent word is rank 1, the next, rank 2 and so forth.
whose length was less than three or greater than 25, and rejection of words beginning with digits.

Figure 24 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.

![Figure 23 Frequency of top 75 OSUMED words](image)

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 that 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 24 Frequency of top 75 Wikipedia words](image)

**Figure 24** Frequency of top 75 Wikipedia words

### 9.3 Basic Word Dictionary

A simple Mumps program to build a basic dictionary of terms using one of the versions\(^\text{22}\) of *osu.converted* is shown in Program 20. The first two words on each line are ignored. These are the byte offset and document number.

```
1 #!/usr/bin/mumps
2 # dictionary.mps February 14, 2014
3 # Copyright 2014 Kevin C. O'Kane
4
5 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.
```

---

\(^{22}\) 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.
At the end, program 20 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 21.

Program 21 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 20, Program 21 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\(^{23}\).

After the file is sorted, it is read in and ^dict() is initialized. The function $zblanks()$ removes extra blanks that uniq inserts. On balance, this procedure is substantially faster that incrementing individual elements of ^dict(). The sort and 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.

\(^{23}\) 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.
<table>
<thead>
<tr>
<th>Count</th>
<th>Word 1</th>
<th>Count</th>
<th>Word 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>58760</td>
<td>of</td>
<td>1998</td>
<td>study</td>
</tr>
<tr>
<td>50093</td>
<td>the</td>
<td>1943</td>
<td>which</td>
</tr>
<tr>
<td>39368</td>
<td>and</td>
<td>1937</td>
<td>group</td>
</tr>
<tr>
<td>35125</td>
<td>in</td>
<td>1894</td>
<td>normal</td>
</tr>
<tr>
<td>20814</td>
<td>to</td>
<td>1874</td>
<td>no</td>
</tr>
<tr>
<td>17939</td>
<td>with</td>
<td>1846</td>
<td>increased</td>
</tr>
<tr>
<td>17396</td>
<td>a</td>
<td>1824</td>
<td>blood</td>
</tr>
<tr>
<td>14042</td>
<td>was</td>
<td>1814</td>
<td>p</td>
</tr>
<tr>
<td>12065</td>
<td>were</td>
<td>1425</td>
<td>has</td>
</tr>
<tr>
<td>11083</td>
<td>patients</td>
<td>1780</td>
<td>two</td>
</tr>
<tr>
<td>10257</td>
<td>The</td>
<td>1703</td>
<td>control</td>
</tr>
<tr>
<td>10078</td>
<td>for</td>
<td>1770</td>
<td>cells</td>
</tr>
<tr>
<td>8147</td>
<td>by</td>
<td>1721</td>
<td></td>
</tr>
<tr>
<td>7502</td>
<td>that</td>
<td>1726</td>
<td>We</td>
</tr>
<tr>
<td>6636</td>
<td>is</td>
<td>1405</td>
<td>clinical</td>
</tr>
<tr>
<td>6147</td>
<td>or</td>
<td>1398</td>
<td>years</td>
</tr>
<tr>
<td>5740</td>
<td>than</td>
<td>1398</td>
<td>cases</td>
</tr>
<tr>
<td>5701</td>
<td>from</td>
<td>1380</td>
<td>associated</td>
</tr>
<tr>
<td>5198</td>
<td>on</td>
<td>1364</td>
<td>10</td>
</tr>
<tr>
<td>4762</td>
<td>at</td>
<td>1355</td>
<td>found</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1350</td>
<td>per</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1349</td>
<td>activity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1345</td>
<td>3</td>
</tr>
</tbody>
</table>

The final result is `dictionary.sorted` consisting of lines containing (1) a count and, (2) a word. Some sample output is shown in Figure 25. As can be seen, the word `of` was the most frequently occurring term with the word `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.

```mumps
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=$zzInput("w")
9   . if i=0 break
10  . set k=i-1
11  . use 1
12  . for j=2:1:k write w(j),!
13 close 1
14
15 shell sort < dictionary.unsorted | uniq -c | sort -nr > dictionary.sorted
16
17 kill ^dict
18 open 1:"dictionary.sorted,old"
19 use 1
20 for  do
21   . read a
22   . if "$test break
23   . set a=$zblanks(a)
24   . set ^dict($piece(a," "),2)=$piece(a," ",1)
25 close 1
26 halt
```

Program 21 Modified dictionary program
9.4 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:

\[ \text{Frequency} \times \text{rank} \approx \text{constant} \]

where \( \text{Frequency} \) 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 \( \text{Frequency} \). That is, the most frequently occurring term is rank 1, the second most frequently occurring term is rank 2 and so forth. Program 22 calculates Zipf's constants. The input to Program 22 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 26 and the OHSUMED collection in Figure 27. 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.

```
#!/usr/bin/mumps
# zipf.mps Copyright 2014 Kevin C. O'Kane
# input is from stdin
# output is to stdout

if '$data(%) set d=1000
else set d=%1

write $zd," Zipf Table Rank*Freq/d",!!
for i=1:1 do
  . read a
  . if '$test break
  . set a=$zblanks(a)
  . set f=$piece(a," ",1)
  . set w=$piece(a," ",2)
  . set t=i*f/d
  . write $justify(t,6,0)," ",w,!
```

Program 22 Zipf's Law example

<table>
<thead>
<tr>
<th>9 the</th>
<th>21 miss</th>
<th>22 or</th>
<th>22 my</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 and</td>
<td>22 is</td>
<td>22 freddy</td>
<td>23 been</td>
</tr>
<tr>
<td>12 of</td>
<td>22 all</td>
<td>23 by</td>
<td>22 mrs</td>
</tr>
<tr>
<td>15 to</td>
<td>21 be</td>
<td>22 have</td>
<td>22 came</td>
</tr>
<tr>
<td>17 a</td>
<td>20 mary</td>
<td>22 me</td>
<td>21 still</td>
</tr>
<tr>
<td>17 he</td>
<td>20 then</td>
<td>22 if</td>
<td>20 young</td>
</tr>
<tr>
<td>18 in</td>
<td>20 when</td>
<td>23 face</td>
<td>21 will</td>
</tr>
<tr>
<td>20 was</td>
<td>20 would</td>
<td>22 are</td>
<td>21 went</td>
</tr>
<tr>
<td>23 his</td>
<td>20 so</td>
<td>22 voice</td>
<td>21 time</td>
</tr>
<tr>
<td>24 her</td>
<td>21 julia</td>
<td>22 about</td>
<td>21 some</td>
</tr>
<tr>
<td>21 said</td>
<td>21 jake</td>
<td>22 no</td>
<td>21 do</td>
</tr>
<tr>
<td>21 she</td>
<td>21 out</td>
<td>22 eyes</td>
<td>21 while</td>
</tr>
<tr>
<td>23 had</td>
<td>22 browne</td>
<td>22 only</td>
<td>21 too</td>
</tr>
<tr>
<td>23 that</td>
<td>20 who</td>
<td>22 go</td>
<td>21 think</td>
</tr>
<tr>
<td>21 gabriel</td>
<td>21 which</td>
<td>21 this</td>
<td>21 stood</td>
</tr>
<tr>
<td>22 i</td>
<td>21 what</td>
<td>22 its</td>
<td>21 see</td>
</tr>
<tr>
<td>22 it</td>
<td>21 up</td>
<td>22 good</td>
<td>21 little</td>
</tr>
<tr>
<td>23 with</td>
<td>22 asked</td>
<td>22 back</td>
<td>22 how</td>
</tr>
</tbody>
</table>
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 23 shows a KWIC program (this may be enabled in index.script) and the output is shown in Figure 28. Similarly, Program 24 shows a KWOC program and Figure 29 shows its output.
#!/usr/bin/mumps
# kwic.mps Copyright 2014 Kevin C. O’Kane

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

for do
  . set i=$zinput("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 $zStopLookup(tw) quit
    .. write tw,?26,doc,": 
    .. set x="
    .. for k=j+1:1:i set x=x_w(k)_" 
    .. set y="
    .. for k=j-1:1:1 set y=w(k)_" _y quit:$l(y)>25
    .. write ?(80-$l(y)),y,?80,w(j)," ",

Program 23 KWIC index

abdominal* serotonin S2 and thromboxane A2prostaglandin H2 receptor activation activation
leukocyte scanning of the abdomen Analysis of its value for diagnosis and managemen
Non-specific acute abdominal pain letter
Eruptive abdominal pain Chylomicronemia
Abdominal and pelvic pain
Recurrent abdominal pain as the sole manifestation of hereditary angioedema
Anatomy of the abductor muscles of the hip as studied by computed tomography
Apolipoprotein B48 in two cases of abetalipoproteinemia
Function and verbal learning ability in patients with complex partial seizures of tempo

Figure 28 KWIC output

abdominal* 482: Abdominal and pelvic pain
abdominal* 562: Recurrent abdominal pain as the sole manifestation of hereditary angioedema in
abductor* 779: Anatomy of the abductor muscles of the hip as studied by computed tomography
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

9.6 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 built-in function known as $zstem()^{24}$. 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 19 on page 153).

The Mumps stem filter program used in these exercises is shown in Program 25. The program builds a global array named $^*stem()$ 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.

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

---

24 A better word stem program is needed. The current version has issues.
is sent to *stem.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.

```mumps
#!/usr/bin/mumps
# stems.mps Nov 11, 2014
# convert data base to word stems
# Copyright 2014 Kevin C. O'Kane

# first parameter is MINIMUMWORDLENGTH
# words must be longer than this value
# default is 3

open 1:"stem-words.tmp,new"

kill ^stem

for do
  .set i=$zzInput("w")
  .if i=0 break
  .set off=w(0)
  .set doc=w(1)
  .write off," ",doc," 
  .set i=i-1
  .for j=2:1:i do
    ..set x=$zstem(w(j))
    ..set ^stem(x,w(j))=
    ..use 5 write x," 
    ..use 1 write x,!
  ..use 5 write !
  .use 5 write !

close 1
```

Recalculation of dictionaries may be appropriate at this point as word frequencies may change significantly once the terms have been reduced to stems.

Program 25 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`,

25 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.

26 This line could have been a continuation of the line above. Both lines, separately or combined, invite significant parallel processing.
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 26. Each line in stemList.txt begins with a word stem followed by the root words separated from one another by a blank.

```
#!usr/bin/mumps
# showStems.mps Nov 11, 2014
# convert data base to word stems
# Copyright 2014 Kevin C. O'Kane

for w=$order(^stem(w)) do
  . write w
  for s=$order(^stem(w,s)) write "  ",s
  . write !
```

Program 26 showStems.mps

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 27.

```
#!/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 27 loadStems.mps
10 Document-Term Matrix

10.1 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 28 builds a basic word-count document-term matrix from the `osu.converted` file (see Figure 7 on page 29). The program constructs the following disk resident global arrays:

1. the document-term matrix \(^{\text{doc}(\text{docNbr},term,i)}\) giving the number of times the term \(term\) occurs in document \(docNbr\) 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 \(word\) occurs in the total collection.
3. \(^{\text{index}(\text{term},docNbr)}\) giving, for each \(term\), the \(docNbrs\) containing the term and the number of times the term occurs in \(docNbr\): \(^{\text{index}(\text{term},docNbr)}\) is the transpose of \(^{\text{doc}(\text{docNbr},term)}\). It does not, however, contain an element giving the relative position of the term in the document. Elements in the \(^{\text{index}(\text{term},docNbr)}\) matrix at this point consist only of counts. These counts are used in `discrim.mps` to calculate discrimination coefficients. \(^{\text{index}(\text{term},docNbr)}\) will be recalculated by `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 \(word\) occurs in the collection.
5. a one element vector \(^{\text{DocCount}(1)}\) giving the total number of documents in the collection.

```
#!/usr/bin/mumps
# DocumentTermMatrix.mps Feb 14, 2014
# Copyright 2014 Kevin C. O'Kane

kill ^df
kill ^dict
kill ^doc

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

set wmin=%1
set wmin=2
if wmin="" set wmin=2
set d=0
open 1:"dict.tmp,new"
open 2:"doc.tmp,new"
```

72
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),!
   endfor
use 5
close 1
close 2
shell sort < dict.tmp | uniq -c > dict.sorted.tmp

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
close 1
shell sort < doc.tmp | uniq -c > doc.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=$piece(a," ",3)
   . set `doc(a2,a3)=c
   . set `index(a3,a2)=c
use 5
close 1
open 1:"df.tmp,new"
use 1
for d=$order(`doc(d)) do
   . for w=$order(`doc(d,w)) do
      .. write w,!
endfor
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
In the example in Program 28, 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 `^dict(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 `^doc()` matrix are also deleted.

The total number of documents is stored in `^DocCount(1)` (line 28) for future reference.

Next, the number of documents each word occurs in is calculated and stored in the vector `^df(word)`. Then the results are written to three output files.
10.2 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 number of words for 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.

10.3 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 30, several weighting schemes are compared. The terms in Figure 30 are:

1. $^\text{doc}(i,w)$ is the number of times term $w$ occurs in document $i$;
2. $^\text{dict}(w)$ is the number of times term $w$ occurs in the collection as a whole;
3. $^\text{df}(w)$ is the number of documents term $w$ occurs in;
4. $\text{NbrDocs}$ is the total number of documents in the collection; and
5. Function $\text{zlog}()$ is the natural logarithm;
6. The operation "/" is integer division.
7. $\text{Wgt1}$ 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, $\text{Wgt1}$ will be 2.

---

27 Median is another possibility.
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 30 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>apple</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>computer</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>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>
</tr>
<tr>
<td>macintosh</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>strategy</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>
</tbody>
</table>

<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>document</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>
<tr>
<td>operate</td>
<td>3</td>
<td>269</td>
<td>184</td>
<td>2.052</td>
<td>6.078</td>
<td>12</td>
<td>-2.1852</td>
</tr>
</tbody>
</table>

<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>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>
</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>
</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>
</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>
</tr>
</tbody>
</table>


1000 documents; 29942 word instances, 563 distinct words

^doc(i,w) Number times word w used in document i
^dict(w) Number times word w used in total collection
^df(w) Number of documents word w appears in
Wgt1 ^doc(i,w)/(^dict(w)/^df(w))
Wgt2 ^doc(i,w)*$zlog(NbrDocs/^df(w))+1
Wgt3 Wgt1*Wgt2+0.5\1
### [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></th>
<th>Quality</th>
<th>Reliability</th>
<th>Performance</th>
<th>Price</th>
<th>Ease of Use</th>
<th>Features</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>0.0678</td>
</tr>
<tr>
<td>color</td>
<td>3</td>
<td>81</td>
<td>47</td>
<td>1.741</td>
<td>10.173</td>
<td>0.0809</td>
</tr>
<tr>
<td>mac</td>
<td>2</td>
<td>146</td>
<td>71</td>
<td>0.973</td>
<td>6.290</td>
<td>-0.0256</td>
</tr>
<tr>
<td>monitor</td>
<td>6</td>
<td>88</td>
<td>52</td>
<td>3.545</td>
<td>18.739</td>
<td>0.0946</td>
</tr>
<tr>
<td>resolution</td>
<td>2</td>
<td>50</td>
<td>32</td>
<td>1.280</td>
<td>7.884</td>
<td>0.0288</td>
</tr>
<tr>
<td>screen</td>
<td>2</td>
<td>92</td>
<td>62</td>
<td>1.348</td>
<td>6.561</td>
<td>0.0199</td>
</tr>
<tr>
<td>video</td>
<td>4</td>
<td>106</td>
<td>61</td>
<td>2.302</td>
<td>12.188</td>
<td>0.0187</td>
</tr>
</tbody>
</table>

### [5] CrystalPrint Express. (Software Review) (high-speed desktop laser printer) (evaluation)

<table>
<thead>
<tr>
<th></th>
<th>Quality</th>
<th>Reliability</th>
<th>Performance</th>
<th>Price</th>
<th>Ease of Use</th>
<th>Features</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>-0.1062</td>
</tr>
<tr>
<td>engine</td>
<td>1</td>
<td>15</td>
<td>13</td>
<td>0.867</td>
<td>5.343</td>
<td>0.0282</td>
</tr>
<tr>
<td>font</td>
<td>4</td>
<td>111</td>
<td>37</td>
<td>1.333</td>
<td>14.187</td>
<td>0.6350</td>
</tr>
<tr>
<td>laser</td>
<td>3</td>
<td>61</td>
<td>27</td>
<td>1.328</td>
<td>11.836</td>
<td>0.2562</td>
</tr>
<tr>
<td>print</td>
<td>3</td>
<td>140</td>
<td>66</td>
<td>1.414</td>
<td>9.154</td>
<td>0.0509</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></th>
<th>Quality</th>
<th>Reliability</th>
<th>Performance</th>
<th>Price</th>
<th>Ease of Use</th>
<th>Features</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>0.5551</td>
</tr>
<tr>
<td>analysis</td>
<td>2</td>
<td>179</td>
<td>139</td>
<td>1.553</td>
<td>4.947</td>
<td>-0.8492</td>
</tr>
<tr>
<td>database</td>
<td>5</td>
<td>138</td>
<td>67</td>
<td>2.428</td>
<td>14.515</td>
<td>0.1832</td>
</tr>
<tr>
<td>midrange</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>0.857</td>
<td>6.116</td>
<td>0.0218</td>
</tr>
<tr>
<td>spreadsheet</td>
<td>2</td>
<td>75</td>
<td>44</td>
<td>1.173</td>
<td>7.247</td>
<td>0.1707</td>
</tr>
<tr>
<td>vary</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>0.857</td>
<td>6.116</td>
<td>0.0107</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Quality</th>
<th>Reliability</th>
<th>Performance</th>
<th>Price</th>
<th>Ease of Use</th>
<th>Features</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>0.0698</td>
</tr>
<tr>
<td>doe</td>
<td>5</td>
<td>97</td>
<td>84</td>
<td>4.330</td>
<td>13.385</td>
<td>-0.1139</td>
</tr>
<tr>
<td>graphical</td>
<td>2</td>
<td>307</td>
<td>171</td>
<td>1.114</td>
<td>4.532</td>
<td>-2.4079</td>
</tr>
<tr>
<td>hypercard</td>
<td>4</td>
<td>25</td>
<td>13</td>
<td>2.080</td>
<td>18.371</td>
<td>0.1517</td>
</tr>
<tr>
<td>mac</td>
<td>2</td>
<td>146</td>
<td>71</td>
<td>0.973</td>
<td>6.290</td>
<td>-0.0256</td>
</tr>
<tr>
<td>map</td>
<td>2</td>
<td>17</td>
<td>10</td>
<td>1.176</td>
<td>10.210</td>
<td>0.1180</td>
</tr>
<tr>
<td>program</td>
<td>4</td>
<td>670</td>
<td>334</td>
<td>1.994</td>
<td>5.386</td>
<td>-15.4832</td>
</tr>
<tr>
<td>script</td>
<td>3</td>
<td>54</td>
<td>32</td>
<td>1.778</td>
<td>11.326</td>
<td>0.1239</td>
</tr>
<tr>
<td>software</td>
<td>3</td>
<td>913</td>
<td>449</td>
<td>1.475</td>
<td>3.402</td>
<td>-30.7596</td>
</tr>
<tr>
<td>stack</td>
<td>5</td>
<td>15</td>
<td>8</td>
<td>2.667</td>
<td>25.142</td>
<td>0.0700</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></th>
<th>Quality</th>
<th>Reliability</th>
<th>Performance</th>
<th>Price</th>
<th>Ease of Use</th>
<th>Features</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>0.1517</td>
</tr>
<tr>
<td>print</td>
<td>3</td>
<td>140</td>
<td>66</td>
<td>1.414</td>
<td>9.154</td>
<td>0.0509</td>
</tr>
<tr>
<td>software</td>
<td>3</td>
<td>913</td>
<td>449</td>
<td>1.475</td>
<td>3.402</td>
<td>-30.7596</td>
</tr>
<tr>
<td>stack</td>
<td>2</td>
<td>15</td>
<td>8</td>
<td>1.067</td>
<td>10.657</td>
<td>0.0700</td>
</tr>
</tbody>
</table>

### [9] Project-scheduling tools. (FastTrack Schedule, MacSchedule) (Software Review) (evaluation)
<table>
<thead>
<tr>
<th>manage</th>
<th>2</th>
<th>318</th>
<th>174</th>
<th>1.094</th>
<th>4.497</th>
<th>5</th>
<th>-2.4884</th>
</tr>
</thead>
</table>

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

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


<table>
<thead>
<tr>
<th>accessory</th>
<th>2</th>
<th>14</th>
<th>10</th>
<th>1.429</th>
<th>10.210</th>
<th>15</th>
<th>0.0540</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>
</tr>
<tr>
<td>display</td>
<td>2</td>
<td>106</td>
<td>78</td>
<td>1.472</td>
<td>6.102</td>
<td>9</td>
<td>-0.1278</td>
</tr>
<tr>
<td>program</td>
<td>3</td>
<td>670</td>
<td>334</td>
<td>1.496</td>
<td>4.290</td>
<td>6</td>
<td>-15.4832</td>
</tr>
<tr>
<td>sound</td>
<td>2</td>
<td>14</td>
<td>8</td>
<td>1.143</td>
<td>10.657</td>
<td>12</td>
<td>0.1172</td>
</tr>
<tr>
<td>user</td>
<td>3</td>
<td>861</td>
<td>435</td>
<td>1.516</td>
<td>3.497</td>
<td>5</td>
<td>-26.8094</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>apple</th>
<th>4</th>
<th>261</th>
<th>112</th>
<th>1.716</th>
<th>9.757</th>
<th>17</th>
<th>-1.1625</th>
</tr>
</thead>
<tbody>
<tr>
<td>card</td>
<td>2</td>
<td>99</td>
<td>56</td>
<td>1.131</td>
<td>6.765</td>
<td>8</td>
<td>0.0790</td>
</tr>
<tr>
<td>display</td>
<td>2</td>
<td>106</td>
<td>78</td>
<td>1.472</td>
<td>6.102</td>
<td>9</td>
<td>-0.1278</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>
</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>
</tr>
<tr>
<td>vga</td>
<td>2</td>
<td>91</td>
<td>62</td>
<td>1.363</td>
<td>6.561</td>
<td>9</td>
<td>0.0104</td>
</tr>
<tr>
<td>video</td>
<td>2</td>
<td>106</td>
<td>61</td>
<td>1.151</td>
<td>6.594</td>
<td>8</td>
<td>0.0187</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>capture</th>
<th>2</th>
<th>14</th>
<th>11</th>
<th>1.571</th>
<th>10.020</th>
<th>16</th>
<th>0.0271</th>
</tr>
</thead>
<tbody>
<tr>
<td>image</td>
<td>3</td>
<td>107</td>
<td>58</td>
<td>1.626</td>
<td>9.542</td>
<td>16</td>
<td>0.1422</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>
</tr>
<tr>
<td>window</td>
<td>4</td>
<td>417</td>
<td>159</td>
<td>1.525</td>
<td>8.355</td>
<td>13</td>
<td>-3.4780</td>
</tr>
</tbody>
</table>

[14] Studio Vision. (Software Review) (Lehrman, Paul D.) (evaluation) Lehrman, Paul D.

<table>
<thead>
<tr>
<th>audio</th>
<th>1</th>
<th>8</th>
<th>6</th>
<th>0.750</th>
<th>6.116</th>
<th>5</th>
<th>0.0161</th>
</tr>
</thead>
<tbody>
<tr>
<td>disk</td>
<td>3</td>
<td>234</td>
<td>121</td>
<td>1.551</td>
<td>7.336</td>
<td>11</td>
<td>-1.1468</td>
</tr>
<tr>
<td>edit</td>
<td>3</td>
<td>111</td>
<td>77</td>
<td>2.081</td>
<td>8.692</td>
<td>18</td>
<td>-0.0961</td>
</tr>
<tr>
<td>operate</td>
<td>2</td>
<td>269</td>
<td>184</td>
<td>1.368</td>
<td>4.386</td>
<td>6</td>
<td>-2.1852</td>
</tr>
<tr>
<td>portion</td>
<td>1</td>
<td>17</td>
<td>15</td>
<td>0.882</td>
<td>5.200</td>
<td>5</td>
<td>0.0295</td>
</tr>
<tr>
<td>requirement</td>
<td>2</td>
<td>87</td>
<td>76</td>
<td>1.747</td>
<td>6.154</td>
<td>11</td>
<td>-0.1203</td>
</tr>
<tr>
<td>sound</td>
<td>6</td>
<td>14</td>
<td>8</td>
<td>3.429</td>
<td>29.970</td>
<td>103</td>
<td>0.1172</td>
</tr>
<tr>
<td>user</td>
<td>3</td>
<td>861</td>
<td>435</td>
<td>1.516</td>
<td>3.497</td>
<td>5</td>
<td>-26.8094</td>
</tr>
</tbody>
</table>

[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,

<table>
<thead>
<tr>
<th>apple</th>
<th>3</th>
<th>261</th>
<th>112</th>
<th>1.287</th>
<th>7.568</th>
<th>10</th>
<th>-1.1625</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication</td>
<td>2</td>
<td>199</td>
<td>110</td>
<td>1.106</td>
<td>5.415</td>
<td>6</td>
<td>-0.6984</td>
</tr>
<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>
</tr>
<tr>
<td>disk</td>
<td>2</td>
<td>234</td>
<td>121</td>
<td>1.034</td>
<td>5.224</td>
<td>5</td>
<td>-1.1468</td>
</tr>
</tbody>
</table>
[16] Data on the run. (Hardware Review) (palmtop organizers)(includes related article describing the WristMac from Microseeds)

duplicate 10 9 0.900 5.711 5 0.0143
file 271 151 1.672 6.671 11 -1.3982
font 37 111 0.667 7.594 5 0.6350
memory 98 142 2.761 10.291 28 -0.2999
tip 6 8 0.750 6.116 5 0.0335
user 435 861 2.021 4.330 9 -26.8094
virtual 15 17 1.765 9.399 17 0.0424

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

cach 10 9 0.900 5.711 5 0.0143
cache 30 49 4.898 29.052 142 0.1613
card 56 99 3.394 18.294 62 0.0790
chip 67 117 1.459 6.406 7 -0.1153
high-speed 14 18 1.556 9.537 15 0.0352
memory 98 142 2.070 5.750 7 -0.3992
ram 93 145 1.283 6.298 5 0.0216
review 238 265 1.796 3.871 7 -2.4234
rom 17 19 0.895 5.075 5 0.0589
software 449 913 1.967 4.203 8 -30.7596
transfer 10 44 1.333 7.247 10 0.0918

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

add-on 15 17 0.882 5.200 5 0.0374
apple 112 261 0.858 5.379 5 -1.1625
file 151 271 5.572 19.905 111 -1.3982
lan 51 98 1.041 6.952 7 0.0366
mac 71 146 1.945 11.580 23 -0.0256
macintosh 44 210 3.057 14.410 44 -0.5855
ms_dos 65 98 1.327 6.467 9 0.0481
network 28 60 0.933 8.151 8 0.2314
network 222 571 2.333 10.030 23 -9.4287
ratio 16 18 0.889 5.135 5 0.0154
server 75 162 5.556 32.083 178 -0.1592
software 449 913 1.475 3.402 5 -30.7596
unix-based 13 15 0.867 5.343 5 0.0376
user 435 861 1.516 3.497 5 -26.8094
vax 14 28 1.000 9.537 10 0.1692

[19] Is it time for CD-ROM? (guide to 16 CD-ROM drives)(includes related articles on using IBM-compatible CD-ROMs with the Mac,

audio 6 8 0.750 6.116 5 0.0161
10.4 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.

10.4.1 OSU MEDLINE Data Base IDF Weights

The IDF weights for the OSU MEDLINE collection were calculated based on the \(^{df()}\) vector created in Program 28 on page 74.

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>\text{DocFreq}_w</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>
<tr>
<td>with</td>
<td>0.542811</td>
</tr>
<tr>
<td>for</td>
<td>0.737224</td>
</tr>
<tr>
<td>was</td>
<td>0.793466</td>
</tr>
<tr>
<td>were</td>
<td>0.867298</td>
</tr>
</tbody>
</table>

to highs such as:

<table>
<thead>
<tr>
<th>Term</th>
<th>\text{DocFreq}_w</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>
<tr>
<td>with</td>
<td>0.542811</td>
</tr>
<tr>
<td>for</td>
<td>0.737224</td>
</tr>
<tr>
<td>was</td>
<td>0.793466</td>
</tr>
<tr>
<td>were</td>
<td>0.867298</td>
</tr>
</tbody>
</table>
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.

### 10.4.2 Calculating IDF Weights

To calculate IDF weights we first build a document-term matrix $^\text{doc}(i,w)$, 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 $^\text{df}(w)$ where each element gives the number documents in which the term $w$ occurs.

A basic program to calculate IDF weights is shown in Program 29. It assumes that the global arrays $^\text{doc}()$ and $^\text{df}()$ have been calculated (see Program 28 on page 74).

```mumps
#!/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 29 IDF calculation

If a word has a very low or high IDF value, it may be a candidate for removal from the collection. Program 30 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 30 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.

```mumps
#!/usr/bin/mumps
# IdfCutoff.mps Feb 14, 2014
# Copyright 2014 Kevin C. O'Kane

if '$data(%1) write "Missing parameter",! halt
if '$data(%2) write "Missing parameter",! halt
```
10.5 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 re-calculated 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 \(N*N\) 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 31. 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 \textit{doc} and \textit{index} matrices.

```mumps
#!/usr/bin/mumps
discrim.mps Feb 14, 2014
# Copyright 2014 Kevin C. O'Kane
database "copy2"
set D=\text{\textasciitilde DocCount}(1) // number of documents
set sq=0
kill \text{\textasciitilde mca}

# calculate centroid vector \text{\textasciitilde c()} for entire collection and
# the sum of the squares (needed in cos calc but should only be done once)

for w=\text{\textasciitilde order}(\text{\textasciitilde dict}(w)) do
  set \text{\textasciitilde c}(w)=\text{\textasciitilde dict}(w)/D // centroid is composed of avg word usage
  set sq=\text{\textasciitilde c}(w)**2+sq // The sum of the squares is needed below.
# Calculate total similarity of doc for all words (T) space by
# calculating the sum of the similarities of each document with the centroid.
# Remember and store contribution of each document in \text{\textasciitilde dc}(dn).

set T=0
for i=\text{\textasciitilde order}(\text{\textasciitilde doc}(i)) do
  set x=0
  set y=0
  for w=\text{\textasciitilde order}(\text{\textasciitilde doc}(i,w)) do
    set d=\text{\textasciitilde doc}(i,w)
    set x=d*\text{\textasciitilde c}(w)+x // numerator of cos(c,doc) calc
    set y=d*d+y // part of denominator

  if y=0 quit
  set \text{\textasciitilde dc}(i)=x$/sqrt{sq*y}$ // cos(c,doc(i))
  set T=\text{\textasciitilde dc}(i)+T // sum the cosines
# calculate similarity of doc space with words removed
# For each document containing \text{\textasciitilde W}, calculate sum of the contribution
# of the cosines of these documents to the total (T). \text{\textasciitilde dc}(i) is
# the original contribution of doc i. Sum of contributions is stored in T1.

for W=\text{\textasciitilde order}(\text{\textasciitilde dict}(W)) do
```

85
In Program 31, the density of the space without word $i$ is:

$$(T - T_{i1} + T_{i2})$$

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 likely to be similar while a smaller value (lower space density) means they are less likely to be similar.
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 31 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 \), we truncate fractional parts.

Figure 31 gives some sample good discriminators and Figure 32 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:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>982</td>
<td>phenytoin</td>
<td>6.66</td>
</tr>
<tr>
<td>985</td>
<td>atenolol</td>
<td>7.01</td>
</tr>
<tr>
<td>988</td>
<td>streptokinase</td>
<td>6.83</td>
</tr>
<tr>
<td>1009</td>
<td>betacarotene</td>
<td>8.05</td>
</tr>
<tr>
<td>1009</td>
<td>meningioma</td>
<td>6.89</td>
</tr>
</tbody>
</table>

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, bladder, diabetic and cholesterol would be important terms, in this specialized medical database, they are not.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>956</td>
<td>restenosis</td>
<td>6.81</td>
</tr>
<tr>
<td>965</td>
<td>valvuloplasty</td>
<td>6.52</td>
</tr>
<tr>
<td>972</td>
<td>gerbil</td>
<td>8.05</td>
</tr>
<tr>
<td>973</td>
<td>folate</td>
<td>7.11</td>
</tr>
<tr>
<td>977</td>
<td>cpr</td>
<td>6.97</td>
</tr>
<tr>
<td>980</td>
<td>sufentanil</td>
<td>7.49</td>
</tr>
<tr>
<td>982</td>
<td>phenytoin</td>
<td>6.66</td>
</tr>
<tr>
<td>985</td>
<td>atenolol</td>
<td>7.01</td>
</tr>
<tr>
<td>Code</td>
<td>Term</td>
<td>Score</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>988</td>
<td>streptokinase</td>
<td>6.83</td>
</tr>
<tr>
<td>1009</td>
<td>betacarotene</td>
<td>8.05</td>
</tr>
<tr>
<td>1009</td>
<td>meningioma</td>
<td>6.89</td>
</tr>
<tr>
<td>1010</td>
<td>splenectomy</td>
<td>6.56</td>
</tr>
<tr>
<td>1025</td>
<td>ribavirin</td>
<td>7.39</td>
</tr>
<tr>
<td>1029</td>
<td>dobutamine</td>
<td>7.18</td>
</tr>
<tr>
<td>1031</td>
<td>hemangioma</td>
<td>6.78</td>
</tr>
<tr>
<td>1034</td>
<td>acetaldehyde</td>
<td>7.35</td>
</tr>
<tr>
<td>1056</td>
<td>lithium</td>
<td>6.89</td>
</tr>
<tr>
<td>1083</td>
<td>selenium</td>
<td>7.24</td>
</tr>
<tr>
<td>1083</td>
<td>taurine</td>
<td>7.56</td>
</tr>
<tr>
<td>1097</td>
<td>coarct</td>
<td>6.97</td>
</tr>
<tr>
<td>1098</td>
<td>digoxin</td>
<td>6.93</td>
</tr>
<tr>
<td>1116</td>
<td>hypoglycemia</td>
<td>6.81</td>
</tr>
<tr>
<td>1124</td>
<td>islet</td>
<td>6.78</td>
</tr>
<tr>
<td>1134</td>
<td>nalbuphine</td>
<td>8.05</td>
</tr>
<tr>
<td>1140</td>
<td>oxalate</td>
<td>7.21</td>
</tr>
<tr>
<td>1149</td>
<td>pseudoaneurysm</td>
<td>7.11</td>
</tr>
<tr>
<td>1187</td>
<td>methadone</td>
<td>7.64</td>
</tr>
<tr>
<td>1187</td>
<td>thymoma</td>
<td>7.24</td>
</tr>
<tr>
<td>1221</td>
<td>amiodarone</td>
<td>6.54</td>
</tr>
<tr>
<td>1226</td>
<td>mesothelioma</td>
<td>7.08</td>
</tr>
<tr>
<td>1230</td>
<td>propafenone</td>
<td>8.42</td>
</tr>
<tr>
<td>1236</td>
<td>surfactant</td>
<td>6.89</td>
</tr>
<tr>
<td>1299</td>
<td>pseudocyst</td>
<td>7.52</td>
</tr>
<tr>
<td>1305</td>
<td>ketanserin</td>
<td>7.42</td>
</tr>
<tr>
<td>1309</td>
<td>nicardipine</td>
<td>7.64</td>
</tr>
<tr>
<td>1312</td>
<td>pheochromocytoma</td>
<td>6.93</td>
</tr>
<tr>
<td>1319</td>
<td>rilmenidine</td>
<td>8.25</td>
</tr>
<tr>
<td>1322</td>
<td>sucralfate</td>
<td>7.13</td>
</tr>
<tr>
<td>1339</td>
<td>caffeine</td>
<td>6.69</td>
</tr>
<tr>
<td>1558</td>
<td>nabumetone</td>
<td>7.73</td>
</tr>
<tr>
<td>1875</td>
<td>tgfbeta</td>
<td>8.11</td>
</tr>
</tbody>
</table>

Figure 31 Good Discriminators

-107205 platelet 4.65
-86790 breast 4.54
-81255 glucose 4.55
-74248 laser 4.74
-70996 bladd 4.63
-62966 diabetic 4.77
-62359 cord 4.52
-61372 cholesterol 4.75
-59672 hiv 4.72
-57247 alcohol 4.63
-56692 cervical 4.51
-55539 arthrit 4.63
-52538 beta 4.61
-51880 thyroid 4.84
-51143 intake 4.52
-50648 diabete 4.52
-50302 hemorrhage 4.58
-48706 neuron 4.71
-48199 traine 4.71
-46889 image 4.59
-46835 fiber 4.68
-46793 motor 4.54
-46636 intestinal 4.55
Figure 32 Poor Discriminators
11 Weighted Matrices

11.1 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 32.

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 32 IDF weights are used as multipliers (see line 12).

Additionally, Program 32 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.

```
#!/usr/bin/mumps
# weight.mps December 26, 2011
4 kill ^index
6 open 2: "document-term-matrix-weighted.txt,new"
8 for d=$order(^doc(d)) do
  . use 2 write "doc=",d,!,?15
  . use 2 write ^title(d),!,?15
  . for w=$order(^doc(d,w)) do
    .. set x=^idf(w)*^doc(d,w)
    .. set ^doc(d,w)=x
    .. set ^index(w,d)=x
    .. write w,"("x,") "
18 close 2
19 open 2: "term-document-matrix-weighted.txt,new"
```
Next, we create a global array vector of document titles for use with subsequent retrieval examples. This is shown in Program 33. 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.

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

for do
   . read t
   . if ’$test halt
   . set ^title($piece(t,””,1))=$piece(t,””,2,100)
```

Program 33 Create global array of title text

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 34. Figure 33 shows some sample SQL SELECT for document retrieval and Figure 34 shows sample SQL INSERT, DROP, and CREATE TABLE statements. Generally speaking, RDBMS systems perform poorly in IS&R applications.

```
#!/usr/bin/mumpsRO
# SQLdocvectors.mps December 17, 2011
# Copyright 2014 Kevin C. O’Kane

write "use isr;",!
write "drop table if exists docVect;",!
write "create table docVect (doc int, word varchar(25), wgt float);",!
write "drop table if exists docs;",!
write "create table docs (doc int, title varchar(255));",!
for d=$order("doc(d)") do
   . if $order("doc(d,"")="" quit
   . set t=$translate($extract("title(d),7,256),"")
   . use 5 write "insert into docs values (",d,"","",t,"",",2,100)
   . for w=$order("doc(d,w)") do
      .. write "insert into docVect values (",d,"","",w,"",","",",doc(d,w),")
```

Program 34 Creating an RDBMS Doc-Term matrix

```
select doc, round(sum(wgt),2) from docVect where word='mole' or word='ethylate' group by
```
```
doc order by wgt desc;
select doc, round(sum(wgt),2) as wgts from docVect where word='mole' or word='ethylate'
group by doc order by wgt desc;
```

Figure 33 SQL Select statements

```
use isr;
drop table if exists docVect;
create table docVect (doc int, word varchar(25), wgt float);
drop table if exists docs;
create table docs (doc int, title varchar(255));
insert into docs values (1, 'The binding of acetaldehyde to the active site of ribonuclease alterations in catalytic activ');
insert into docVect values (1, 'absent', 4.84);
insert into docVect values (1, 'acetaldehyde', 35.1);
insert into docVect values (1, 'adduct', 14.38);
insert into docVect values (1, 'alpha', 4.18);
insert into docVect values (1, 'alpha- amino', 3.806662);
insert into docVect values (1, 'alpha-stable', 3.761200);
insert into docVect values (1, 'alteration', 4.43);
insert into docVect values (1, ' amino', 18.16);
insert into docVect values (1, ' amino-residues', 4.828314);
insert into docVect values (1, 'analyse', 5.68);
insert into docVect values (1, 'aris', 11.04);
insert into docVect values (1, 'bulk', 6.88);
insert into docVect values (1, 'catalytic', 13.12);
insert into docVect values (1, 'compaare', 4.35);
insert into docVect values (1, 'covalent', 13.12);
insert into docVect values (1, 'diminish', 9.76);
insert into docVect values (1, 'enrich', 6.35);
insert into docVect values (1, 'enzymatic', 5.65);
```

Figure 34 SQL commands
12 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\(^{28}\) 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

---

\(^{28}\) The matrix is equal to its transpose.
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 35.

<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 35 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 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-specialty 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 a 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\(^{29}\). More general topic collections, however, will probably have matrices that are less sparse.

### 12.1 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:

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

\(^{29}\) For example, pediatric terminology is unlikely to be common in articles about adult nutrition.
That is, the product of the document-term matrix $DocTerm$ with its transpose.

In our $TermTerm$ 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 $DocTerm$ 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 $TermTerm$ matrix $tt_{jk}$ 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:

$$
 tt_{jk} = d_{1j}d_{1k} + d_{2j}d_{2k} + d_{3j}d_{3k} ... + d_{Nj}d_{Nk}
$$

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, is 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.

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

Consequently, the Term-Term matrix will thus be:
A Mumps program to calculate a term-term matrix is given in Program 35. In this figure, the global array \(^\text{doc}\) 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.

```
#!/usr/bin/mumps
k \^docT,\^tt
do $zzTranspose(\^doc,\^docT)
set k=$zzmultiply(\^docT,\^doc,\^tt)
```

Program 35 Term-term matrix

Another way to construct the term-term correlation matrix is shown in Program Error: Reference source not found. 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.

```
#!/usr/bin/mumps
# ~/Medline2012/ttBasic.mps December 22, 2011
kill ^tt
for d=$order(\^doc(d)) do
  for w=$order(\^doc(d,w)) do
    . for w1=w:$order(\^doc(d,w1)):"" do
      ... if w1=w quit
      ... if $data(\^tt(w,w1)) set ^tt(w,w1)=^tt(w,w1)+1
    else set ^tt(w,w1)=1
```

Program 36 Basic Term-Term matrix construction

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 \(w1\) in \(d\) which are alphabetically greater than \(w\). For each pair of terms \(\{w,w1\}\), 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 \(^\text{tt}\) at row \(w\) and column \(w1\). 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 Error: Reference source not found) and those elements whose frequency of co-occurrence are below a threshold may be deleted. This threshold is adjusted according to collection size. In \texttt{index.script}, the threshold (\texttt{MINTT}) is set at \(0.001*MAXDOCS\) if the number of documents is 5,000 or greater, 5 otherwise.
The methods in Program 35 and Program Error: Reference source not found can be slow. A speed up technique is shown in Programs 37 and 38. These are similar to Program Error: Reference source not found but it avoid the time consuming multiple database accesses in lines 10 and 11 of Program Error: Reference source not found by making use of system utility programs sort and uniq. The procedure is done in three steps. The first is Program 37, the second is a bash command string the processes the output of Program 37, and the third is Program 38 which reads the results of the bash script, makes final calculations and stores the results.

In Program 37, 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:

\[
\text{tt1.mps copy1 | sort | uniq -c > tt.sorted.tmp &}
\]

Next, as shown in Program 38, 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:

\[
\text{tt2.mps $MINTT < tt.sorted.tmp | sort -n > term-term-matrix.sorted.txt}
\]

1 #!/usr/bin/mumpsRO
2 # Copyright 2014 Kevin C. O’Kane
3 # tt1.mps December 11, 2011
4
5 if '$data(%1) write "missing parameter",! halt
6
7 database %1
8
9 for d=$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 37 Faster Term–Term Matrix 1

The \( ^{tt()} \) matrix produced by Program 38 goes beyond that shown in Program Error: Reference source not found which produced a matrix with gave containing the raw count of the number of co-occurrences between terms.
Program 38 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 6, if two terms never co-occur, their association is zero.

Program 38 Faster Term-Term Matrix 2

An example of the output of Program 38 is shown in Figure 36. 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.

| 0.030 catheter median 10 166 172 |
| 0.030 recovery versus 10 170 169 |
| 0.031 area damage 10 163 170 |
| 0.031 basi defect 10 172 164 |
| 0.031 cerebre recovery 10 167 170 |
| 0.031 daily moderate 10 168 164 |
| 0.031 moderate performance 10 164 168 |
| 0.031 prolonge separate 10 172 157 |
| 0.032 abil capacity 10 173 150 |
| 0.032 aortic calcium 10 163 163 |
| 0.032 aortic vessel 10 163 161 |

In fact, a value of zero is never be stored if a non-zero minimum co-occurrence threshold is provided to the program.
Salton suggested two metrics for term-term association.

In the first formula [Salton 1983], which was used in Program 38, 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 38, 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 38 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 $h$. 

\[99\]
Salton also proposed a second formula [Salton 1988] the $d_{i,j}$ are the weight of term $j$ in document $i$:

$$\text{Sim}(T_j, T_k) = \frac{\sum_{i=1}^{n} (d_{i,j})^2}{\sqrt{\sum_{i=1}^{n} (d_{i,j})^2 \cdot \sum_{i=1}^{n} (d_{i,k})^2}}$$

Figure 37 Frequency of term co-occurrences

In Figure 37, 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.
12.2 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:

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

where \( p, i, \) and \( j \) 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 39 shows calculation of a term-term connection matrix.

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

# ttt.mps October 22, 2014

for w1=$order(^tt(w1)) do
  . for w2=$order(^tt(w1,w2)) do
    .. for w3=$order(^tt(w2,w3)) do
      ... if '$data(^ttt(w1,w3)) set ^ttt(w1,w3)=1
      ... else set ^ttt(w1,w3)=^ttt(w1,w3)+1

open 1:"term-term-term.txt,new"
use 1
for w1=$order(^ttt(w1)) do
  . for w2=$order(^ttt(w1,w2)) do
    .. write ^ttt(w1,w2)," ,w1," ,w2,!

close 1
```

Program 39 ttt.mps

12.3 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.

12.3.1 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):

\[ ^{\text{Cohesion}}(i,j) = \text{SIZE\_FACTOR} \times (\text{tt}(i,j)/(\text{dict}(i) \times \text{dict}(j))) \]

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 40. In the script file index.script, it is invoked with:

```
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 40** cohesion.mps

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

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

Figure 38 gives a sample of the results.

| 1730 compute ct  |
| 1792 blot gene   |
| 1851 inflammate inflammatory |
| 1881 gene messenger |
| 1893 ne norepinephrine |
| 1893 painful should |
| 1923 endothelial endothelium |
| 1949 mammography screene |
| 2097 lymph node |
| 2142 imag resonance |
| 2307 imag magnetic |
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.

### 12.3.2 Jaccard Term-Term Similarity

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

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

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 41 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 39.

```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)
```

Figure 38 Term Term Cohesion Results

Figure 39 Example Output

103
Another modification to improve the detection of term-term relationships is shown in Program 42. 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 an predominately in the same order are in fact closely related.

### 12.3.3 Term Proximity

![Figure 39 Jaccard Term Term Relationships](image)

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 screene
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
Recall\textsuperscript{31} that when we initially scanned the documents $^\text{doc()}$ stored the position of each term relative to the start of the document in the third index:

$^\text{doc}(\text{DocNbr}, \text{Word}, \text{Position})$

The value for $\text{Position}$ 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 $^\text{ttp}(i,j)$
7. add to global array $^\text{ttp}(i,j,1)$ the signed difference between the terms $n-m$. That is, the distance in word units between the two terms.

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

\[
\text{set } \text{dd} = \frac{1}{\text{abs}(m-n) \times 20 + 1} \\
\]

which yields, for various distances as shown in Figure 40.

Thus, in Program 42, 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.

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.

\[
\begin{align*}
\text{abs}(m-n) &= 1 & \text{result} &= 3 \\
\text{abs}(m-n) &= 2 & \text{result} &= 2 \\
\text{abs}(m-n) &= 3 & \text{result} &= 2 \\
\text{abs}(m-n) &= 4 & \text{result} &= 1 \\
\text{abs}(m-n) &= 5 & \text{result} &= 1 \\
\text{abs}(m-n) &= 6 & \text{result} &= 1
\end{align*}
\]

\textsuperscript{31} Line 25, Program 28, page 74.
$zabs(m-n)=7 \ \text{result} \ 1
$zabs(m-n)=8 \ \text{result} \ 1
$zabs(m-n)=9 \ \text{result} \ 1
$zabs(m-n)=10 \ \text{result} \ 1
$zabs(m-n)=11 \ \text{result} \ 1
$zabs(m-n)=12 \ \text{result} \ 0
$zabs(m-n)=13 \ \text{result} \ 0
$zabs(m-n)=14 \ \text{result} \ 0
$zabs(m-n)=15 \ \text{result} \ 0

Figure 40 Proximity calculations

1  #!/usr/bin/mumps
2  # proximity.mps Feb 7, 2014
3  # Copyright 2014 Kevin C. O'Kane
4
5  # ttx term-term correlation matrix
6  # calculate term-term proximity coefficients within env words
7
8  database "copy0"
9
10 if 'data(1) write "Missing parameter",! halt
11
12 kill ^ttp  /* delete any old term-term correlation matrix
13
14  #++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
15  # for each document k, sum the co-occurrences of words i and j
16  #++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
17
18  # for each document k
19
20  set k=""
21  for do
22  . set k=$order(^doc(k))
23  . if k="" break
24
25  # for each term i in p k
26
27  . set i=""
28  . for do
29  .. set i=$order(^doc(k,i))
30  .. if i="" break
31
32  # for each other term j in doc k
33
34  .. set j=i
35  .. for do
36  ... set j=$order(^doc(k,j))
37  ... if j="" break
38
39  # for each position m of term i in doc k
40  ... set m=""
41  ... for do
42  .... set m=$order(^doc(k,i,m))
43  .... if m="" break
44
45  # for each position n of term j in doc k
46
47  .... set n=""
48  .... for do
Figure 41 contains some of the top ranking proximity based term-term proximity based correlation calculation from Program 42. All terms are shown in stem form.

In Figure 41 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>Term</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>infarct myocardial</td>
<td>-465</td>
</tr>
<tr>
<td>dens lipoprotein</td>
<td>73</td>
</tr>
<tr>
<td>abstract word</td>
<td>1184</td>
</tr>
<tr>
<td>magnetic resonance</td>
<td>376</td>
</tr>
<tr>
<td>cord spin</td>
<td>-309</td>
</tr>
<tr>
<td>lymph node</td>
<td>313</td>
</tr>
<tr>
<td>arthrit rheumatoid</td>
<td>-307</td>
</tr>
<tr>
<td>female male</td>
<td>-43</td>
</tr>
<tr>
<td>cholesterol lipoprotein</td>
<td>43</td>
</tr>
<tr>
<td>blind double</td>
<td>-304</td>
</tr>
<tr>
<td>compute tomography</td>
<td>255</td>
</tr>
<tr>
<td>imag resonance</td>
<td>-189</td>
</tr>
<tr>
<td>carry out</td>
<td>262</td>
</tr>
<tr>
<td>determine whether</td>
<td>321</td>
</tr>
<tr>
<td>cholesterol dens</td>
<td>-137</td>
</tr>
<tr>
<td>mitr valve</td>
<td>114</td>
</tr>
<tr>
<td>state unit</td>
<td>-240</td>
</tr>
<tr>
<td>centre nerv</td>
<td>221</td>
</tr>
<tr>
<td>amino sequence</td>
<td>-7</td>
</tr>
<tr>
<td>aortic valve</td>
<td>194</td>
</tr>
<tr>
<td>guinea pig</td>
<td>32</td>
</tr>
<tr>
<td>care health</td>
<td>-97</td>
</tr>
<tr>
<td>imag magnetic</td>
<td>-439</td>
</tr>
<tr>
<td>cystic fibrosis</td>
<td>179</td>
</tr>
<tr>
<td>head neck</td>
<td>149</td>
</tr>
<tr>
<td>death sudden</td>
<td>-191</td>
</tr>
<tr>
<td>biopsy specimen</td>
<td>230</td>
</tr>
<tr>
<td>electron microscopy</td>
<td>111</td>
</tr>
<tr>
<td>marrow transplant</td>
<td>113</td>
</tr>
<tr>
<td>tract urin</td>
<td>-156</td>
</tr>
<tr>
<td>diabete mellitus</td>
<td>161</td>
</tr>
<tr>
<td>erratum publish</td>
<td>-177</td>
</tr>
<tr>
<td>sensit specificity</td>
<td>210</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>530</td>
<td>morbid mort</td>
</tr>
<tr>
<td>522</td>
<td>resistance vascular</td>
</tr>
<tr>
<td>522</td>
<td>excretion urin</td>
</tr>
<tr>
<td>522</td>
<td>ejection fraction</td>
</tr>
<tr>
<td>519</td>
<td>cerebrospinal fluid</td>
</tr>
<tr>
<td>517</td>
<td>coli escherichia</td>
</tr>
<tr>
<td>513</td>
<td>diastolic end</td>
</tr>
<tr>
<td>508</td>
<td>man old</td>
</tr>
<tr>
<td>500</td>
<td>anti monoclon</td>
</tr>
<tr>
<td>475</td>
<td>state steady</td>
</tr>
<tr>
<td>470</td>
<td>erythematosus lupu</td>
</tr>
<tr>
<td>469</td>
<td>randomize trial</td>
</tr>
<tr>
<td>466</td>
<td>diastolic systolic</td>
</tr>
</tbody>
</table>

Figure 41 Ranked Proximity Correlations

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.

```mumps
#!/usr/bin/mumps
# ~/Medline2012/proxLoad.mps December 10, 2014
# Copyright 2014 Kevin C. O'Kane

open 1:"phrases.txt,new"

# input format: #co-occur w1 w2 direction

for do
  . use 5
  . read a
  . if '$test break
  . set w1=$piece(a," ",2)
  . set w2=$piece(a," ",3)
  . set wgt=$piece(a," ",1)
  . set dir=$piece(a," ",4)
  . set ^ttp(w1,w2)=wgt
  . set ^ttp(w2,w1)=wgt
  . set ^ttp(w1,w2,1)=dir
  . set ^ttp(w2,w1,1)=dir

  . for d=$order(^index(w1,d)) do
    .. if $data(^index(w2,d)) do
      ... set w1stem=w1
      ... set w2stem=w2
      ... if $data(^stem(w1)) set w1=$order(^stem(w1,""))
      ... if $data(^stem(w2)) set w2=$order(^stem(w2,""))
      ... if dir<0 set prs=w2_"-_"_w1
      ... else set prs=w1_"-_"_w2
      ... write d," ",prs,"!
      ... set ^dict(prs)=wgt
      ... set ^df(prs)=wgt
      ... set ^phrase(w1stem,prs)=""
      ... set ^phrase(w2stem,prs)=""
      ... set s=$zlog(wgt)
      ... set ^doc(d,prs)=s
      ... set ^index(prs,d)=s
```
12.3.4 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 42 shown on page 107 rather than to other methods discussed above. Phrases are added to the collection in proxLoad.mps as shown in Program 43.

Program 43 creates two new global array matrices ^ttp() and ^phrase(). It also adds phrase entry terms to ^doc(), ^index(), ^df() and ^dict().

12.4 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 44. 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 38 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 a 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
.. if ^jtt(w1,w2)<min quit
.. write ^tt(w1,w2)," ",w1," ",w2,!
close 1

shell sort -n -r < TTtmp.tmp > TTtmp.sorted.tmp
open 1:"TTtmp.sorted.tmp,old"

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
    . write w1," 
  . write !!
halt

kill `dt
Program 44 Term-Term clustering

Figure 42 gives a sample of the output of the above. In this example, there 438 clusters were found.
12.4.1 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.
13 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 45. 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 45 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.

```
1 #!/usr/bin/mumps
2 # docdoc1.mps Feb 7, 2014
3 # Copyright 2014 Kevin C. O’Kane
4
5 if '$data(%1) write "Missing parameter",! halt
6
7 database "copy1"
8
9 set min=%1
10 if min="" set min=5
11
12 open 1:"dd.tmp,new"
13
14 use 1
15 for w=$order(^index(w)) do
16   . for d1=$order(^index(w,d1)) do
17     .. for d2=d1:$order(^index(w,d2)):"" do
18       ... if d1=d2 quit
19       ... write d1," ",d2,!
20
21 close 1
22
23 shell sort < dd.tmp | uniq -c | sort -rn > dd1.tmp
24
25 open 1:"dd1.tmp,old"
26
27 # stop reading when threshold encountered (break)
28
29 for do
30   . use 1 read a
31   . if '$test break
32   . set a=$zblanks(a)
33   . set c=$piece(a," ",1)
```
Program 45 Programs to calculate a document-document matrix

13.1 Document Clustering

The program clusterdd.mps in Program 46 uses a single link clustering technique similar to that used in the term clustering discussed above (Section 12.4 on page 109).

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 (\(\text{min}\), a command line parameter).
for d2=d1:$order(\dd(d1,d2)):
.. if ^dd(d1,d2)<min quit
.. write ^dd(d1,d2),",d1,"",d2,!
close 1
shell sort -n -r < ddtmp.tmp > ddtmp.sorted.tmp
open 1:"ddtmp.sorted.tmp,old"
set c=1
for do
.. use 1
.. read a // correlation doc1 doc2
.. if 'test break
.. set score=$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
.. 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
.. if $data(^dc(seq1,c)) set ^dc(seq1,c)=^dc(seq1,c)+1
.. else set ^dc(seq1,c)=1
.. write !
close 1
use 5
for cx=$order(^clstr(cx)) do
.. write "cluster ",cx,!
.. for seq1=$order(^clstr(cx,seq1)) do
.. use 5 write seq1,?8,^title(seq1),!
.. set ^ct(cx,seq1)=" set ^dc(seq1,cx)=^dc(seq1,cx)+1
.. else set ^dc(seq1,cx)=1
.. write !
open 1:"document-clusters-by-title.txt,new"
use 1
for d=$order(^dc(d)) do
.. write d,"",^title(d),!,?10
Figure 43 gives a sample output from Program 46.

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 47. 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.
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 $data(^hc(c,w)) set ^hc(c,w)=^hc(c,w)+^doc(t,w)
   .. else set ^hc(c,w)=^doc(t,w)
   .. set k=k+1
   for i=1:1:c for w=$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,
   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
.endif

. if $d(^x(seq2)) for s=$order(^x(seq2,s)) do
.. set `clstr(s,seq1)="
.. set `x(seq1,s)="
.. set f=0
.endif

. if f do
.. set `clstr(c,seq1)="
.. set `x(seq1,c)="
.. set `clstr(c,seq2)="
.. set `x(seq2,c)="
.. set c=c+1
.endif

# print the clusters
for cx=$order(`clstr(cx)) do
.. write "cluster ",cx,!
.. for seq1=$order(`clstr(cx,seq1)) do
.. .. write "base cluster=",seq1,!
.. .. for seq2=$order(`ct(seq1,cz)) do
.. .. .. write seq1,?8,`title(cz),!
.. .. write !
for cz=$order(`ct(seq1,cz)) do
.. write seq1,?8,`title(cz),!
.endif
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 seq2=$order(`ct(seq1,cz)) do
.. .. .. write seq1,?8,`title(cz),!
.. .. write !
write !,!!
14 Retrieval

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

14.1 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 48. 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\(^{32}\). 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 $data(^doc(i,j)) write i,?8," ",^title(i),! break
write!,"Time elapsed: ",$zd1-t1,!
halt
```

Enter search terms: 
epithelial fibrosis
energy expenditure of patients with cystic fibrosis.
the occurrence of vasculitis in perianeurysmal fibrosis.
cellular and non-cellular compositions of crescents in human glomerulonephritis.
epithelial ovarian tumor in a phenotypic male.
neonatal long lines for intravenous antibiotic therapy in cystic fibrosis [letter]
intrapерitoneal cis-platinum as salvage therapy for refractory epithelial ovarian
the effect of a collagen bandage lens on corneal wound healing: a preliminary
removal of lens epithelial cells by ultrasound in endocapsular cataract surgery.
oncocytic carcinoma of the plica semilunaris with orbital extension.
retinal pigment epithelial cells release inhibitors of neovascularization.
```

Program 48 Simple retrieval program
```

14.2 Retrieval Using the Term-Doc Matrix

Unfortunately, Program 48 is very slow because all documents must be inspected. Program 49, 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.

---

\(^{32}\) 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 48, 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:

```bash
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/mumpsRO
# 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: ",$zd1­t1,!
```

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
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.

33 Note: the order of the titles will normally be different than in Program 48.
14.3 Weighted Retrieval using the Term-Doc Matrix

Program 50 is similar to Program 49 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 50 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 50 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 ^tmo() 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 50, like Program 49, 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.
14.4 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 to 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 51 uses a more sensitive similarity metric. It reads a set of query words into a query vector then calculates the cosine 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 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.

14 set t1=$zd1
15 for w=$order(query(w)) do
16  . for i=$order(^index(w,i)) do
17   . if $data(^tmp(i)) set ^tmp(i)=^tmp(i)+^index(w,i)
18   . else set ^tmp(i)=^index(w,i)
19
20 set fn=$job","new"
21 open 1:fn
22 use 1
23 for i=$order(^tmp(i)) do
24   use 1 write $justify(^tmp(i),6,2),"",^title(i),!
25 close 1
26 use 5
27 set cmd="sort -n <"$_job_" | tail -10 ; rm "$_job
28 shell &~cmd~
29 write !,"Elapsed time: ",$zd1-t1,!
30 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
triticale and 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
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-deficien
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.
14.5 Simple Cosine Weighted Retrieval Part 2

As before, program 51 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 52.

For each query term \( w \), the document numbers \( d \) from row \( ^{index}(w) \) are used as indices to create elements of \( ^{tmp}(d) \). No value is stored at the nodes as the index (the document number) is the information in which we are interested. The \( ^{tmp()} \) 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 \( ^{tmp()} \). The number of elements of \( ^{tmp} \) will normally be considerably smaller than the total number of documents\(^{35}\).

When all query words have been processed, the temporary vector \( ^{tmp} \) 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 \( ^{tmp()} \).

Program 52 yields the same results as above but takes less than 1 second.

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

kill ^query
kill ^tmp

set fn=$job_",new"
open 1:fn

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)=1

if $order(^query(""))="" halt

write "Query is: ",
for w=$order(^query(w)) write w," ",
write !
```

\(^{35}\) 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!
# 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

14.6 Retrieval Enhancements

The code from Program 52 may be enhanced as shown in Program 53. 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 !,"unknown word: ",w,! quit
  . write !,"query term: ",w,!
# stems
  . if $data(^stem(w1)) do
  .. write !,?4,"Words in this stem group are: "]
  .. do stems(w1)
  . else write ?4,w," has no stem related words"
# related
  . write !,?4,"adding related phrase words: "]
  . if '$data(^phrase(w1)) do
  .. write "none",!
  . else do
  .. set f=1
  .. for prs=$order(^phrase(w1,prs)) do
  ... write prs," "] set f=0
  ... if $x>60 w !,?26
  ... set ^query(prs)=1
  if f write "none"
  . if $x'=26 write !
# soundex
  . set s=$zzsoundex(w1)
  . write ?4,"words that may sound similar to ",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 $x'=26 write !
# query word
  . set ^query(w)=3

# Find documents containing one or more query terms.
  . for d=$order(^index(w,d)) set ^tmp(d)="" // retain doc id
  set time0=$zd1
  write !
  set file=$j_".tmp"
  open 1:file_",new"
```
An example of the output of Program 53 is given in Figure 44 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.
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-ef ficacies
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.

<table>
<thead>
<tr>
<th>Doc Rel</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>18728</td>
<td>0.308 Intrinsic methicillin resistance and phage complex of Staphylococcus aureus matched keys: aureus *methicillin-aureus staphylococcus *staphylococcus-aureus</td>
</tr>
<tr>
<td>21989</td>
<td>0.302 Toxic shock syndrome associated with Staphylococcus aureus enterocolitis matched keys: aureus staphylococcus *staphylococcus-aureus</td>
</tr>
<tr>
<td>7048</td>
<td>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</td>
</tr>
</tbody>
</table>
| 22953  | 0.298 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 |
<p>| 5676   | 0.290 Single and combination-antibiotic therapy for experimental endocarditis caused |</p>
<table>
<thead>
<tr>
<th>Position</th>
<th>Score</th>
<th>Title</th>
<th>Matched Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>13303</td>
<td>0.276</td>
<td>Role of beta-lactamase in expression of resistance by methicillin-resistant Staphylococcus aureus</td>
<td>*antibiotic-beta-lactam antibiotic-resistant aureus *aureus-strain *isolate-aureus *isolate-staphylococcus *methylene-resistant-aureus *methylene-resistant-staphylococcus staphylococcus *staphylococcus-aureus *staphylococcus-strain *strain-antibiotic</td>
</tr>
<tr>
<td>11616</td>
<td>0.268</td>
<td>Phenotypic expression and genetic heterogeneity of lincosamide inactivation in Staphylococcus aureus</td>
<td>*aureus *aureus-strain *isolate-aureus *isolate-staphylococcus *s-aureus staphylococcus *staphylococcus-aureus *staphylococcus-strain</td>
</tr>
<tr>
<td>10351</td>
<td>0.257</td>
<td>IgE antibodies to Staphylococcus aureus in the hypereosinophilic syndrome</td>
<td>aureus staphylococcus *staphylococcus-aureus</td>
</tr>
<tr>
<td>5084</td>
<td>0.251</td>
<td>Antibiotic treatment of Staphylococcus aureus endocarditis A review of cases</td>
<td>*aminoglycoside-antibiotic *antibiotic-beta-lactam *antibiotic-regimen aureus *aureus-strain staphylococcus *staphylococcus-aureus *staphylococcus-strain *strain-antibiotic</td>
</tr>
<tr>
<td>10889</td>
<td>0.249</td>
<td>Acute crescentic glomerulonephritis as a complication of a Staphylococcus aureus abscess of hi</td>
<td>aureus staphylococcus *staphylococcus-aureus</td>
</tr>
</tbody>
</table>

*Time used: 0 seconds*
15 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.

15.1 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\(^{36}\).

The 2003 MeSH file contains approximately 40,000 entries. Each line consists of text along with hierarchical codes describing the subject heading. Figure 45 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; A01.047.025.600.225
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
Sacroccocygeal Region; A01.176.780
Breast; A01.236
Nipples; A01.236.500
Extremities; A01.378
Amputation Stumps; A01.378.100

Figure 45 Sample MeSH Hierarchy

36 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 47. 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.

### 15.2 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 46. 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")="Abdominal Cavity"
set ^mesh("A01","047","025","600")="Peritoneum"
    .
    .
    .
set ^mesh("A01","047","365")="Groin"
    .
    .
    .
```

Figure 46 Global Array Commands

A graphical representation of a portion of the MeSH hierarchy can be seen in Figure 47 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 54. 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 54, 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.
Figure 47 MeSH Tree
The output of Program 54 is shown in Figure 48. Line 36 writes the text of the created mumps set command. These are the commands executed by the xexecute command on line 37.

15.3 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 55 gives one way to print the global array and the results are shown in Figure 49. 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 437 the process begins by finding successive values of the first index of 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 lev1.

37 The format of this for command uses a non-standard Mumps syntax extension.
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 (*lev1*). Each of these is placed in *lev2*. The second level codes are printed on line 7 indented by 5 spaces.

```mumps
for lev1=$order(^mesh(lev1)) do
  write lev1," ",^mesh(lev1),
  for lev2=$order(^mesh(lev1,lev2)) do
    write ?5,lev2," ",^mesh(lev1,lev2),
    for lev3=$order(^mesh(lev1,lev2,lev3)) do
      write ?10,lev3," ",^mesh(lev1,lev2,lev3),
      for lev4=$order(^mesh(lev1,lev2,lev3,lev4)) do
        write ?15,lev4," ",^mesh(lev1,lev2,lev3,lev4),
```

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.

<table>
<thead>
<tr>
<th>A01 Body Regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>047 Abdomen</td>
</tr>
<tr>
<td>025 Abdominal Cavity</td>
</tr>
<tr>
<td>600 Peritoneum</td>
</tr>
<tr>
<td>750 Retroperitoneal Space</td>
</tr>
<tr>
<td>050 Abdominal Wall</td>
</tr>
<tr>
<td>365 Groin</td>
</tr>
<tr>
<td>412 Inguinal Canal</td>
</tr>
<tr>
<td>849 Umbilicus</td>
</tr>
<tr>
<td>176 Back</td>
</tr>
<tr>
<td>519 Lumbosacral Region</td>
</tr>
<tr>
<td>780 Sacrococcygeal Region</td>
</tr>
<tr>
<td>236 Breast</td>
</tr>
<tr>
<td>500 Nipples</td>
</tr>
<tr>
<td>378 Extremities</td>
</tr>
<tr>
<td>100 Amputation Stumps</td>
</tr>
<tr>
<td>610 Lower Extremity</td>
</tr>
<tr>
<td>100 Buttocks</td>
</tr>
</tbody>
</table>
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

136
15.4 Printing the MeSH Global Array Part II

Program 56 presents a more general function to print the \(^{\text{mesh}}\) hierarchy and its output is shown in Figure 50. The code in Program 56 works with trees of any depth whereas the code in program 55 needs to be extended in order to work with trees whose depth is greater than four.

```
#!/usr/bin/mumps
# AdvancedMtreePrint.mps

set x="^mesh"
for do
  set x=$query(@x)
  if x="" break
  set i=$qlength(@x)
  write ?i*2," $qsubscript(x,i)," ",@x,?50,x,!
```

Program 56 Alternate MeSH tree printing program

In the example in Program 56, we first set a local variable x to \(^{\text{mesh}}\), 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 56 shown in Figure 50. The values returned by $query() are assigned to the variable x and used as the seed to $query() in the next iteration until, 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 \(^{\text{mesh}}()\) global array reference.

We use the builtin function $qsubscript() to extract from the global array reference the \(i^{\text{th}}\) 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 \(i^{\text{th}}\) 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().
15.5 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 57. Here we use the Mumps function $query()$ to
access the \textit{b-tree} keys in the order in which they are actually stored, in sequential key order, as shown in Figure 51.

The program shown in Program 57 passes to $\textit{query}()$ 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 \textit{mesh} references and return an empty string which causes program termination.

\begin{verbatim}
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")
\end{verbatim}

Figure 51 MeSH global array codes

Note that the line:

\begin{verbatim}
. write x,?50,@x,!
\end{verbatim}

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.

\begin{verbatim}
1  #!/usr/bin/mumps
2  # AdvancedMtreePrint2.mps
3
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,!
\end{verbatim}

Program 57 Program to print MeSH global

The output from Program 57 appears in Figure 52.

\begin{verbatim}
^mesh("A01") Body Regions
^mesh("A01","047") Abdomen
^mesh("A01","047","025") Abdominal Cavity
^mesh("A01","047","025","600") Peritoneum
^mesh("A01","047","025","600","225") Douglas’ Pouch
^mesh("A01","047","025","600","451") Mesentery
^mesh("A01","047","025","600","451","535") Mesocolon
^mesh("A01","047","025","600","573") Omentum
^mesh("A01","047","025","600","678") Peritoneal Cavity
^mesh("A01","047","025","750") Retroperitoneal Space
^mesh("A01","047","050") Abdominal Wall
^mesh("A01","047","365") Groin
^mesh("A01","047","412") Inguinal Canal
\end{verbatim}
15.6 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 58.

The program in Program 58 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 ^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 ^mesh global array node in global array key order.

On line 10, the $find() function determines if the value stored at the current global array node referenced by x (The value stored at which is@x) contains, as a substring, the value in key. That is, for node in the ^mesh tree, does the text value stored at the node contain key?
read "enter keyword: ", key
write!
set x="^mesh" // build a global array ref
set x=$query(@x)
if x="" halt
for do
  . if '$find(@x,key) set x=$query(@x) // is key stored at this ref?
  . else do
    .. set i=$qlength(@x) // number of subscripts
    .. write x," ",i!,
    .. for do
      ... set x=$query(@x)
      ... if x="" halt
      ... if $qlength(@x)>i break
      ... write ?5,x," ",i!,
  . if x="" halt

Program 58 Program to search MeSH global array

If $find() does not detect the value in key in the global array node, the next global array reference is returned by $query() until there are none remaining.

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^38 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.

Figure 53 MeSH keyword search results

---

38 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.
An example is shown in Figure 53 where Skeleton was given as input.

### 15.7 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) 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 *MH*\(^{39}\) in positions 1 and 2. Note also that there is a blank line that signals the end of each abstract and the 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 (*^MH*). Additionally, we count the number of times each term occurs. The program to do this is shown in Program 59.

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 54.

Program 59 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.

```mumps
1 #!/usr/bin/mumps
2
3  # MeshIndex.mps
4
5  open 1:"osu.medline,old"
6  use 1
7
8  kill "MH"
9
10  set x=0 // a counter to limit the size
11
12  set i=$ztell // return the integer offset in the file
13
14  for do
15    . use 1
16    . read a
17    . if '$test break
18
19  # if a blank line, record the offset - this is the start of an abstract
20
21    . if a="" set i=$ztell set x=x+1 quit // return the offset in the file
22
23    . if $extract(a,1,3)="MH " do
24      . use 5

39 See Figure 6 MEDLINE Codes on page 25.
```

142
.. set a=$piece($extract(a,7,255),"/",1)

# create or increment entry for word
.. if $data(^MH(a)) set ^MH(a)=^MH(a)+1
.. else set ^MH(a)=1

# store the offset
.. set ^MH(a,i)="

# write for each heading the titles associated with it
use 5
set x=""
for do
.. set x=$order(^MH(x))
.. if x="" break
.. write x," occurs in ",^MH(x)," documents",!
.. for off=$order(^MH(x,off)) do
.. use 1
.. do $zseek(off)
.. for do
... read a
... if $extract(a,1,3)'="TI " quit
.. use 5
... write ?5,off,?15,$extract(a,7,80),!
.. break

Program 59 Locate instances of MeSH keywords

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
t33436121 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 transcutaneous
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.
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 \textit{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 \textit{^MH} is instantiated if it does not exists and the count stored at \textit{^MH} is incremented or \textit{set to one} (lines 29 and 30). The offset of the document containing it is also recorded in \textit{^MH} as second level index value (line 34). Thus the global array \textit{^MH} contains at the first indexing level the MeSH headings and, at the second level, the offset addresses of those 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 54. This form of display is called a \textit{concordance} - a list of words and an indication of their location and context\cite{footnote:concordance}.

40 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.
15.8 Display of OHSUMED Documents by MeSH Hierarchy

Now we combine the programs from Program 57 (page 139) and Program 59 into a single program that displays the titles integrated into the overall tree structure of the MeSH hierarchy. The code is shown in Program 60 and a sample of the output in Figure 55.

<table>
<thead>
<tr>
<th>MeSH Code</th>
<th>Term</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A01</td>
<td>Abdominal cavity</td>
<td>^mesh(&quot;A01&quot;,&quot;047&quot;,&quot;025&quot;)</td>
</tr>
<tr>
<td>025</td>
<td>Peritoneum</td>
<td>^mesh(&quot;A01&quot;,&quot;047&quot;,&quot;025&quot;,&quot;600&quot;)</td>
</tr>
<tr>
<td>600</td>
<td>Peritoneum</td>
<td>Peritoneum occurs in 4 documents</td>
</tr>
<tr>
<td>225</td>
<td>Douglas' pouch</td>
<td>^mesh(&quot;A01&quot;,&quot;047&quot;,&quot;025&quot;,&quot;600&quot;,&quot;225&quot;)</td>
</tr>
<tr>
<td>451</td>
<td>Mesentery</td>
<td>^mesh(&quot;A01&quot;,&quot;047&quot;,&quot;025&quot;,&quot;600&quot;,&quot;451&quot;)</td>
</tr>
<tr>
<td>535</td>
<td>Mesocolon</td>
<td>^mesh(&quot;A01&quot;,&quot;047&quot;,&quot;025&quot;,&quot;600&quot;,&quot;451&quot;,&quot;535&quot;)</td>
</tr>
<tr>
<td>573</td>
<td>Omentum</td>
<td>^mesh(&quot;A01&quot;,&quot;047&quot;,&quot;025&quot;,&quot;600&quot;,&quot;573&quot;)</td>
</tr>
<tr>
<td>678</td>
<td>Peritoneal cavity</td>
<td>^mesh(&quot;A01&quot;,&quot;047&quot;,&quot;025&quot;,&quot;600&quot;,&quot;678&quot;)</td>
</tr>
<tr>
<td>750</td>
<td>Retroperitoneal space</td>
<td>^mesh(&quot;A01&quot;,&quot;047&quot;,&quot;025&quot;,&quot;750&quot;)</td>
</tr>
</tbody>
</table>

Program 60 Hierarchical MeSH concordance

Figure 55 Hierarchical MeSH concordance
15.9 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 59 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 $^\text{MH}(\text{term})$ where term is an individual MeSH term contained in one or more of the OHSUMED documents.

For each MeSH term entry in $^\text{MH}(\text{term})$, there are one or more file offset pointers at the second level of indexing of the $^\text{MH}$ 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 $^\text{MH}$ 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 61.

The program in Program 61 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 $^\text{MH}$, begins reading the original abstract from the osu.medline file beginning at the offset stored in $^\text{MH}$. 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.

```mumps
#!/usr/bin/mumps
# meshword.mps
open 1:"osu.medline,old"
if '$test write "file open error",! halt

# write for each heading the titles associated with it
write "Enter a MeSH keyword: ",word
read word
write ",word," occurs in ",^MH(word)," documents",!
for off=$order(^MH(word,off)) do 
  use 1 
  do $zseek(off)
  for 
    .. read a 
    .. if $extract(a,1,3)'="TI " quit 
    .. use 5 
    .. write ?5,off,?15,$extract(a,7,80),!
  break

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
```

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 61 Inverted search by MeSH term

Clearly, the inverted lookup in Program 61 is preferable to scanning each document looking for instances of the MeSH term *Acetylcholinesterase*!
16 SQL/RDBMS Based Indexing and Retrieval
17 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 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.

17.1 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 56.

<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>

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 57 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.
18 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.

18.1 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 letters41.

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.

18.2 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 alphnumeric 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 62 gives an example of converting the dictionary in ^dict to Soundex codes a sample of which are shown in Figure 58.
# Copyright 2014 Kevin C. O'Kane
# convert data base to word stems

for w=$order(`dict(w)) do
    set sndx=$zzsoundex(w)
    set `sndx(sndx,w)=""
end

for w=$order(`sndx(w)) do
    write w,"": "
    for x=$order(`sndx(w,x)) write "",x
    write !
end

Program 62 soundex.mps

| g42:  | gels glas glass gliosis golgi |
| g421: | glossopharyngeal glycopeptide glycophorin glycoprotein glycopyrrolate |
| g422: | glasgow glucagon glucagonoma glucokinase glucosamine glucose glucose-induce glucose-infus glucose-stimulate glycogen glycogenolysis glycogenogenesis glycosaminoglycan glycosuria glycosylase glycosylate |
| g423: | galactose galactosemia galactosylate gallstone glycation |
| g424: | glycocalcin glycocalyx glycol glycolipid glycolysis glycolytic |
| g425: | glaucoma glaucomat glcnac gleason glucan gluconate gluconeogenesis gluconeogenic glycaemic glycan glycemia glycemic glycine glycine-extend glycoconjuate |
| g426: | glucocerebrosidase glucocorticoid glucocorticoid-induce glucocorticosteroid glucuronidate glucuronide glycerol |
| g43:  | glott gold golyte |

Figure 58 Soundex codes

### 18.3 N-gram encoding

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*.
19 Semantic Approaches: WordNet

WordNet can be found at:

http://wordnet.princeton.edu

It is:

"... a large lexical database of English, developed under the direction of George A. Miller. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations. The resulting network of meaningfully related words and concepts can be navigated with the browser. WordNet is also freely and publicly available for download. WordNet's structure makes it a useful tool for computational linguistics and natural language processing. ..." (from the web site)

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 34 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 34 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 33.
20 Genomic Text Indexing and 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 DNA sequence shown in Figure 59.

```
>gi|2695846|emb|Y13255.1|ABY13255 Acipenser baeri mRNA for immunoglobulin heavy chain, TGGTTACAACCTTTCTCTTTTCTAATACCACGTCGATCAAATGGGGATTTTTACAGCTCTCTGTTAATAATATACAGCTCTATCAAGTGCTGGTCTGATGTAGTGTTGACTGAGGCCAGCCAGTTATAAACGATGGAGAGTCCCATAAAA
CTGTCCTGTAACAGCTCTCTCAGACATTACACCGACCGTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTGCCGTCAAGCTCTGCTGCCTGTAAAGCCTCTGGATTCACATTCAGCAGCGCCCTACATGAGCTGGGTTCGACAAGCTCCTGGAAAGGGTG
```

Figure 59 Example DNA Sequence

The sequence in Figure 59 is that for a known and archived DNA sequence. It is shown in what is referred to as FASTA format. 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.

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.

---

42 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/

43 http://en.wikipedia.org/wiki/FASTA_format

44 Note: protein sequences are ordinarily written with an alphabet of 20 letters corresponding to the amino acids found in proteins.
There are several IR systems to do these searches but a program known as BLAST\(^{45}\) (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 59 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 59 was submitted to BLAST, a number a matches were returned. One of these is shown in Figure 60. In Figure 60 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\(^ {46}\) ..." 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.

>gb|U17058.1|LOU17058 Lepisosteus osseus Ig heavy chain V region mRNA, partial cds

| Score = 151 bits (76), Expect = 4e-33 |
| Identities = 133/152 (87%), Gaps = 0/152 (0%) |
| Strand=Plus/Plus |

| Query 242 TGGGTGGCTTTATATTCTCAGGTTGTAGTAGTAGTACTATGCCCCAGTCTGTCAGGGGA 301 |
| Sbjct 4 TGGGTGGCGTATATTCAACGCAGTGGGAGCAATACATACTATTCCCAGTCTGTCAGGGGA 63 |
| Query 302 AGATTCGCCATCTCAGAGACAGCTCCAACAGCTGTATTTCATTAAATGAAAGGCTGTG 361 |
| Sbjct 64 AGATTCACCATCTCCAGAGACAATTCCAAGAATCAGCTGTACTTACAGATGACGGCTGTG 123 |
| Query 362 AAGACTGAAGACACTGCCCCGTGTATTACTGTGAC 393 |
| Sbjct 124 AAGACTGAAGACACTGCTGCTGTATTACTGTGAC 155 |

Figure 60 Example BLAST Result

\(^{45}\) http://blast.ncbi.nlm.nih.gov/Blast.cgi

21 Bash Indexing Script

The primary *bash* script to process and index the 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*:

```bash
# 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 &

# display usage
index.script -?
```

For basic test runs, a small number such a 1,000 is useful but, due to the small sample size, the final indexing results may be poor. For full test runs, at least 10,000 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! These are discussed below.

```bash
#!/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.
```

47 The convention of using the *.script* file extension for *bash* scripts is followed throughout.
if [ "$1" = "-?" ];
    then
        echo
        echo "usage: index.script nbrDocs databases"
        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/' | |

        fi

    fi

}
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 's/\- / /g' |
| sed -e 's/\*/ /g' |
| > tmp.tmp
reformat.mps < tmp.tmp |
| sed -e "s/\'/g"
| -e 's/ \([a-zA-Z]\){1,3}\}1,3\}/ /g' |
| -e 's/\[a-zA-Z\]{1,3}\}1,3\}/ /g' |
| -e 's/ \([a-zA-Z]\){1,3}\}1,3\}$/ /g' |
| sed -e 's/ \{2,6\}/ /g' |
| > $DB.converted
fi

export MAXDOCS
echo "$MAXDOCS number of documents."
echo $MAXDOCS > MAXDOCS

# Zipf's Law calculation.
# Optional - uncomment if you want to run it.
# head -$MAXDOCS $DB.converted | WordList.mps | sort | uniq -c | sort -nr | zipf.mps
$MAXDOCS > zipfs-law.txt

# Basic.stop.words is a list of common English words.

# cp basic.stop.words stop.words # initial set of stop words

# KWIC index
# Optional - uncomment if you want to run it.
# head -$MAXDOCS titles.list | kwic.mps | sort | trunc.mps > kwic.txt
# head -$MAXDOCS titles.list | kwoc.mps | sort > kwoc.txt

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"

# 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"

head -$MAXDOCS $DB.converted | stop.mps | stems.mps $MINWORDLENGTH > initial-stemmed-docs.tmp

# 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

# sort < stem-words.tmp | uniq -c > stemmed-vocabulary.tmp  # list of unique stems with
# instance counts

WordCount=`wc -l stemmed-vocabulary.tmp`
WordCount=${WordCount:0 : `expr "$WordCount" : ' [0-9]*'` }  # extracts the numeric portion only

TWordCount=`wc -l stem-words.tmp`
TWordCount=${TWordCount:0 : `expr "$TWordCount" : ' [0-9]*'` }

159
After basic stop list word removal and stemming, the collection
has a total $TWordCount word occurrences (total word count).

# 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))

Words whose frequencies of occurrence in the collection
as a whole are less than $MIN_WORD_FREQ (once per $MIN_DIVISOR docs) will be
deleted.

Words whose total frequencies of occurrence in the collection are
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.

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]'`}

There are $stopcount words whose total frequency of occurrence
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
#
# WordList 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"
echo "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"
echo "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
echo "Kill docs if they have fewer than $MIN_DOC_VECTOR_WORDS words."
echo
echo "Delete words whose IDF is less than $MINIDF."
echo
echo "Weighting scheme in use: $WGT"
echo
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

# Load stems and original words that formed them
loadStems.mps < stemList.txt > stemList2.txt

# 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"

# ttl1.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=$!  # process id

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 $TTPMIN copy0 | sort -nr > term-term-proximity-similarities.txt &
# PP=$!  # process id

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 $DDWGT copy1 > docdoc1.tmp &
# DD=$!  # process id

#-----------------------------------------------

echo

echo "Calculate discrimination coefficients"

# discrimin.mps accesses the database in read-write mode
# uses copy2 database
# Runs in parallel

# discrimin.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"
wait $TT
# wait for process to finish

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 "There were $ttcount term-term associations."

echo
echo "Calculate jaccard term-term coefficients"

# Runs in parallel
# read-only database access

jaccardtt.mps < term-term-matrix.sorted.txt | sort -n > term-term-jaccard-similarities.sorted.txt &
JJ=$!

echo
echo "Calculate term-term cohesion coefficients"

# Runs in parallel
# read-only database access
cohession.mps < term-term-matrix.sorted.txt | sort -nr > term-term-cohesion-similarities.txt &
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
echo "There were $jaccard term-term Jaccard values calculated"

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"

# 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

# Load document titles into the database

# Load the document titles into the database

# Load the document titles into the database

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 "Calculating document clusters"

# clusterdd.mps $MINDD > document-clusters.txt

clusterdd=`grep "cluster:" document-clusters.txt | wc -l`
clusterdd=${clusterdd:0 : `expr "$clusterdd" : '[0-9]*'`} # extract numeric val
echo
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
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..."

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

cwc -l *.txt
rm copy0.* copy1.* copy2.*
echo

#########################################################################
# retrieval test
#########################################################################

echo
medlineRetrieve.mps < tstquery
echo

# rm -f *.tmp
echo "Docs $MAXDOCS `endtime.mps`"

Program 63 Main bash Script
22 Other Scrip Files

22.1.1 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.

22.1.2 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.).

22.1.3 Mesh.script

A `bash` script to index the document collection using the controlled vocabulary Medical Subjects Headings\(^{48}\).

22.2 Parallel Processing

The `bash` script in `index.script` (see chapter 21) is designed to run several tasks at once in parallel. On a multi-core machine, this greatly reduces the time required to process a large collection of documents.

Overall, as can be seen in Figure 61, the effect is to maximize CPU utilization. In this figure, all six cores of an AMD FX-6300 are actively engaged in indexing.

Because of the parallel processing code, the user should be cautious when modifying `index.script`. Some procedures must wait until results of other procedures are ready and the ordering can, as a result be complex.

\(^{48}\) https://www.nlm.nih.gov/mesh
Figure 61 Parallel processing example
23 References


Luhn 1957 Luhn, H.P, A statistical approach to mechanized encoding and searching of literary information, IBM Journal of Research and
Development, Vol 1, No 4, (1957)


Salton 1988 Salton, G., Automatic Text Processing, Addison-Wesley (Reading, 1988)


Alphabetical Index

ACID.................................................................................................................. 14
ACM’s Computing Classification System.............................................................. 130
Adenine.................................................................................................................. 154
American Society of Indexers.............................................................................. 44
ASCII.................................................................................................................... 18
Assigning Word Weights................................................................................... 75
Background.......................................................................................................... 35
Bartlett’s Familiar Quotations.............................................................................. 44
Basic.stop.words.................................................................................................. 59
Bioinformatics..................................................................................................... 154
BitMaps................................................................................................................ 13
BLAST.................................................................................................................. 155
Boolean logic....................................................................................................... 30
Boolean Searching.............................................................................................. 30, 32
Building an Enhanced Stop List........................................................................ 61
C/C++.................................................................................................................. 38
Calculating IDF Weights.................................................................................... 81
Centroid algorithm............................................................................................. 83
Centroid approximation....................................................................................... 83
Centroid vector.................................................................................................... 83
Clustering............................................................................................................ 48, 114
CODASYL........................................................................................................... 14
Coded Tuples........................................................................................................ 12
Cohesion............................................................................................................... 102
Collating sequence............................................................................................. 18
COMputer-STored Ambulatory Record.............................................................. 15
Concordance........................................................................................................ 44
Controlled vocabulary....................................................................................... 42, 131
Cosine.................................................................................................................. 122
Cosinse............................................................................................................... 124
COSTAR.............................................................................................................. 15
CPU utilization.................................................................................................... 169
Crouch................................................................................................................... 83
Cytosine............................................................................................................... 154
DBPREFIX.......................................................................................................... 34
Deleted-words-and-docs-for-idf.txt.................................................................. 56
Deleted-words-for-frequency.txt...................................................................... 56
DIALOG.............................................................................................................. 31
Dictionary........................................................................................................... 52, 63
Dictionary.sorted.................................................................................................. 65
Dictionary.unsorted............................................................................................. 64
Discrimination coefficient.................................................................................. 52
Discrimination Coefficients.............................................................................. 82
Discrimination coefficient.................................................................................. 55
Discrimination-coefficients.txt.......................................................................... 56
Discriminator....................................................................................................... 82
DNA..................................................................................................................... 154
DocCount........................................................................................................... 34
Document clusters............................................................................................. 53
Document frequency vector............................................................................... 81
Document hyperspace......................................................................................... 46
Document vectors............................................................................................... 46
Document-clusters-by-title.txt.......................................................................... 56
Document-clusters.txt....................................................................................... 56
Document-document........................................................................................ 53, 55
Document-Document........................................................................................ 113
Document-document-matrix.txt......................................................................... 56
Document-term matrix......................................................................................... 52, 55, 72, 81, 90
Document-term-matrix-weighted.txt................................................................. 56

173
Vector Space Global Arrays
Vector Space Model
Very high frequency
Very low frequency
VistA
Wait
Wc
Weighted Matrices
Wikipedia
WordNet
Xecute
Zipf's Law
Cosine
Dice
head
index.script
OSU MEDLINE Data Base
operator
^dd(d1,d2)
^df()
^df(w)
^df(word)
^dict()
^dict(term)
^dict(word)
^doc(d,word,pos)
^doc(docNbr,term,i)
^doc(i,w)
^idf(word)
^index(term,docNbr)
^index(word,d)
^mca(word)
^title(d)
^tt(w1,w2)
^tp(w1,w2,1)
$ operator
$data()
$find()
$ftell()
$MAXDOCS
$order()
$sqlenget()
$sqlscript()
$sqlquery()
$zblanks()
$zstem()
$zStopInit()
$zStopLookup()
$zsystem()
$zwparse
$zzInput()