Notes by Ben Horowitz

- Evaluating information retrieval effectiveness
- Signature files
  - Generating signatures
  - Accessing signatures
  - Query logic on signature files
  - Choosing signature width
  - An example: TREC
- Vector space models
  - Selecting weights
  - Similarity measures
  - A simple example
  - Implementation
  - Relevance feedback
  - Clustering
- Latent semantic indexing (LSI)
  - Singular value decomposition (SVD)
  - Using SVD for LSI
  - An example of LSI
  - Applications of LSI
  - Performance of LSI on TREC

1 Evaluating information retrieval effectiveness

Two measures are commonly used to evaluate the effectiveness of information retrieval methods: *precision* and *recall*. The *precision* of a retrieval method is the fraction of the documents retrieved that are relevant to the query:

$$\text{precision} = \frac{\text{number retrieved that are relevant}}{\text{total number retrieved}}$$

The *recall* of a retrieval method is the fraction of relevant documents that were retrieved:

$$\text{recall} = \frac{\text{number relevant that are retrieved}}{\text{total number relevant}}$$

2 Signature files

Signature files are an alternative to inverted file indexing. The main advantage of signature files is that they don't require that a lexicon be kept in memory during query processing. In fact they do not require a lexicon at all. If the vocabulary of the stored documents is rich, then the amount of space occupied by a lexicon may be a substantial fraction of the amount of space filled by the documents themselves.

Signature files are a probabilistic method for indexing documents. Each term in a document is assigned a random *signature*, which is a bit vector. These assignments are made by hashing. The *descriptor* of document is the bitwise logical OR of the signatures of its terms. As we will see, queries to signature files sometimes respond that a term is present in a document when in fact the term is absent. Such *false matches* necessitate a three-valued query logic.

There are three main issues to discuss: (1) generating signatures, (2) searching on signatures, and (3) query logic on signature files.

2.1 Generating signatures

The *width* W of each of the signatures, is the number of bits in each term's signature. Out of these bits some typically small subset of them are set to 1 and the rest are set to zero. The parameter b specifies how many are set to 1. Typically, $1{,}000 \leq W \leq 10{,}000$. (*Managing Gigabytes* seems to suggest that typically $6 \leq b \leq 20$.) The probability of false matches may be kept arbitrarily low by making W large, at the expense of increasing the lengths of the signature files.

To generate the signature of a term, we use b hash functions as follows

for $i = 1$ to b
    $signature[hash_i(term)\ \%\ w] = 1$

In practice we just have one hash function, but use i as an additional parameter.

To generate the signature of a document we just take the logical or of the term signatures. As an example, consider the list of terms from the nursery rhyme "Pease Porridge Hot" and corresponding signatures:

Term	Signature
cold	1000 0000 0010 0100
days	0010 0100 0000 1000
hot	0000 1010 0000 0000
in	0000 1001 0010 0000
it	0000 1000 1000 0010
like	0100 0010 0000 0001
nine	0010 1000 0000 0100
old	1000 1000 0100 0000
pease	0000 0101 0000 0001
porridge	0100 0100 0010 0000
pot	0000 0010 0110 0000
some	0100 0100 0000 0001
the	1010 1000 0000 0000

Note that a term may get hashed to the same location by two hash functions. In our example, the signature for *hot* has only two bits set as a result of such a collision. If the documents are the lines of the rhyme, then the document descriptors will be:

Document	Text	Descriptor
1	Pease porridge hot, pease porridge cold,	1100 1111 0010 0101
2	Pease porridge in the pot,	1110 1111 0110 0001
3	Nine days old.	1010 1100 0100 1100
4	Some like it hot, some like it cold,	1100 1110 1010 0111
5	Some like it in the pot,	1110 1111 1110 0011
6	Nine days old.	1010 1100 0100 1100

2.2 Searching on Signatures

To check whether a term T occurs in a document, check whether all the bits which are set in T's signature are also set in the document's descriptor. If not, T does not appear in the document. If so, T *probably* occurs in the document; because some combination of other term signatures might have set these bits in T's descriptor, it cannot be said with certainty whether or not T appears in the document. For example, since the next-to-last bit in the signature for *it* is set, *it* can only occur in the fourth and fifth documents, and indeed it occurs in both. The term *the* can occur in documents two, three, five, and six; but in fact, it occurs only in the second and fifth.

The question remains of how we efficiently check which documents match the signature of the term (include all its bits). Consider the following (naive) procedure: to find the descriptor strings for which the bits in a given signature are set, pull out all descriptors, and for each descriptor, check whether the appropriate bits are set for each descriptor. When the descriptors are long and there are many files, this approach will involve many disk accesses, and will therefore be too expensive.

A substantially more efficient method is to access columns of the signature file, rather than rows. This technique is called *bit-slicing*. The signature file is stored on disk in transposed form. That is, the signature file is stored in column-major order; in the naive approach, the signature file was stored in row-major order. To process a query for a single term, only b columns need to be read from disk; in the naive approach, the entire signature file needed to be read. The bitwise AND of these b columns yields the documents in which the term probably occurs. For example, to check for the presence of *porridge*, take the AND of columns two, six, and eleven, to obtain $[110110]^T$. (This query returned two false matches.)

2.3 Query logic on signature files

We have seen that collisions between term signatures can cause a word to seem present in a document when in fact the word is absent; on the other hand, words that seem absent are absent. If, for example, *pease* is absent from a document D, then the answer to "Is *pease* is D?" is No; if *pease* seems to be present, then the answer is Maybe.

For queries with negated terms (for example, "Is *pease* absent from document d?") the situation is reversed. For example, since *pease* has bits six, eight, and 16 set, it cannot occur in documents three, four, or six, so for these documents the appropriate answer is Yes. However, the other documents might contain *pease*, so for these documents the appropriate answer is Maybe.

More generally, queries in signature files need to be evaluated in three-valued logic. Atomic queries (e.g. "Is *nine* present?") have two responses, No and Maybe. More complex queries — queries built from atomic queries using NOT, AND, and OR — get evaluated using the following rules:

NOT	
N	Y
M	M
Y	N

AND	N	M	Y
N	N	N	N
M	N	M	M
Y	N	M	Y

OR	N	M	Y
N	N	M	Y
M	M	M	Y
Y	Y	Y	Y

Consider, for example, the evaluation of "(some OR NOT hot) AND pease":

Doc.	s	h	p	NOT h	s OR NOT h	(s OR NOT h) AND p
1	M	M	M	M	M	M
2	M	M	M	M	M	M
3	N	N	N	Y	Y	N
4	M	M	N	M	M	N
5	M	M	M	M	M	M
6	N	N	N	Y	Y	N

2.4 Choosing signature width

How should W be chosen so that the expected number of false matches is less than or equal to some number z? It turns out that W should be set to

$$\frac{1}{1-(1-p)^{1/B}} \tag{9}$$

where

p, the probability that an arbitrary bit in a document descriptor is set, is given by $\frac{z}{N}^{1/b}$;

B, the total number of bits set in the signature of an average document, is given by $\frac{f}{N} \cdot \frac{b}{q}$;

N is the number of documents;

f is the number of (*document, term*) pairs;

b is the number of bits per query; and

q is the number of terms in each query.

The analysis used to derive (9) can be found in Witten, Moffat, and Bell, *Managing Gigabytes*, chapter 3, Section 3.5. One assumption made in this analysis is that all documents contain roughly the same number of distinct terms. Clearly, this assumption does not always hold.

2.5 An example: TREC

As an example, consider the TREC (Text REtrieval Conference) database. The TREC database, which is used to benchmark information retrieval experiments, is composed of roughly 1,000,000 documents — more than 3Gbytes of ASCII text — drawn from a variety of sources: newspapers, journal abstracts, U.S. patents, and so on. (These figures were true of TREC in 1994. It has probably grown in size since then.) In this example, we assume that TREC contains a mere 750,000 documents, with 137 Million document-term pairs totaling more than 2Gbytes of text.

Consider making queries on a single term and assume we want at no more than 1 false match. We assume $b = 8$. To calculate W by equation (9) we have, $f = 137 * 10^6, N = .75 * 10^6, z = 1, q = 1$, which gives:

$$
\begin{aligned}
p &= \left(\frac{1}{.75 * 10^6}\right)^{1/8} = .185 \\
B &= \frac{137 * 10^6}{.75 * 10^6} * \frac{8}{1} = 1470 \\
W &= 7200
\end{aligned}
$$

Thus, the total space occupied by the signature file is $(7,200/8) \cdot 750$Kbytes = 675Mbytes (about 1/3 of the space required by the documents themselves). For $b = 8$, a query on a term will read 8 slices of 750Kbits, which is 750Kbytes (about .1% of the total database).

3 Vector space models

Boolean queries are useful for detecting boolean combinations of the presence and absence of terms in documents. However, Boolean queries never yield more information than a Yes or No answer. In contrast, vector space models allow search engines to quantify the *degree* of similarity between a query and a set of documents. The uses of vector space models include:

Ranked keyword searches, in which the search engine generates a list of documents which are ranked according to their relevance to a query.

Relevance feedback, where the user specifies a query, the search engine returns a set of documents; the user then tells the search engine which documents among the set are relevant, and the search engine returns a new set of documents. This process continues until the user is satisfied.

Semantic indexing, in which search engines are able to return a set of documents whose "meaning" is similar to the meanings of terms in a user's query.

In vector space models, documents are treated as vectors in which each term is a separate dimension. Queries are also modeled as vectors, typically 0-1 vectors. Vector space models are often used in conjunction with clustering to accelerate searches; see Section (3.6) below.

3.1 Selecting weights

In vector space models, documents are modeled by vectors, with separate entries for each distinct term in the lexicon. But how are the entries in these vectors obtained? One approach would be to let the entry $w_{d,t}$, the weight of term t in document d, be 1 if t occurs in d and 0 otherwise. This approach, however, does not distinguish between a document containing one occurrence of *elephant* and a document containing fifty occurrences of *elephant*. A better approach would be to let $w_{d,t}$ be $f_{d,t}$, the number of times t occurs in document d. However, it seems that five occurrences of a word shouldn't lead to a weight that is five times as heavy, and that the first occurrence of a term should count for more than subsequent occurrences. Thus, the following rule for setting weights is often used: set $w_{d,t}$ to $\log_2(1 + f_{d,t})$. Under this rule, an order of magnitude increase in frequency leads to a constant increase in weight.

These heuristics for setting $f_{d,t}$ all fail to take account of the "information content" of a term. If *supernova* appears less frequently than *star*, then intuitively *supernova* conveys more information. Borrowing from information theory, we say that the weight w_t of a term in a set of documents is $\log_2(N/f_t)$, where N is the number of documents and f_t is the number of documents in which the term appears.

One way to represent the weight of a term t in document d is by combining these ideas:

$$w_{d,t} = \log_2(N/f_t)\log_2(1 + f_{d,t})$$

It should be emphasized that this rule for setting $w_{d,t}$ is one among many proposed heuristics.

3.2 Similarity measures

To score a document according to its relevance to a query, we need a way to measure the similarity between a query vector v_q and a document vector v_d. One similarity measure is inverse Euclidean distance, $1/\|v_q - v_d\|$. This measure discriminates against longer documents, since v_d which are distant from the origin are likely to be farther away from typical v_q. Another is the dot product $v_q \cdot v_d$. This second measure unfairly favors longer documents. For example, if $v_{d_1} = 2v_{d_2}$, then $v_q \cdot v_{d_1} = 2v_q \cdot v_{d_2}$. One solution to the problems with both these measures is to normalize the lengths of the vectors in the dot product, thereby obtaining the following similarity measure:

$$\frac{w_q \cdot v_d}{\|v_q\|\|v_d\|}$$

This similarity measure is called the *cosine measure*, since if θ is the angle between vectors $\vec{x}$ and $\vec{y}$, then $\cos\theta = \vec{x} \cdot \vec{y}/(\|\vec{x}\|\|\vec{y}\|)$.

3.3 A simple example

Suppose we have the set of documents listed in the following table. These documents give rise to the frequency matrix $[f_{d,t}]$, frequencies f_t of terms, and informational contents $\log_2(N/f_t)$ in the table:

Doc. no.	Document	Frequency matrix $[f_{d,t}]$				
		a	b	c	d	e
1	apple balloon balloon elephant apple apple	3	2	0	0	1
2	chocolate balloon balloon chocolate apple chocolate duck	1	2	3	1	0
3	balloon balloon balloon balloon elephant balloon	0	5	0	0	1
4	chocolate balloon elephant	0	1	1	0	1
5	balloon apple chocolate balloon	1	2	1	0	0
6	elephant elephant elephant chocolate elephant	0	0	1	0	4
f_t		3	5	4	1	4
$\log_2(N/f_t)$		1.00	0.26	0.58	2.58	0.58

For simplicity, suppose $w_{d,t}$ is calculated according to the rule $w_{d,t} = f_{d,t} \cdot \log_2(N/f_t)$. Then the weight matrix $[w_{d,t}]$ would be as follows (the norms of each row of the weight matrix are underneath $\|v_d\|$):

	a	b	c	d	e	$\|v_d\|$
1	3	.52	0	0	.58	3.10
2	1	.52	1.74	2.58	0	3.31
3	0	1.3	0	0	.58	1.42
4	0	.26	.58	0	.58	0.86
5	1	.52	.58	0	0	1.27
6	0	0	.58	0	2.32	2.39

Then, for the queries listed below, the cosine measure gives the following similarities:

Doc. no.	Query				
	d $\|v_q\| = 2.58$	c $\|v_q\| = 0.58$	c, d $\|v_q\| = 2.64$	a, b, e $\|v_q\| = 1.18$	a, b, c, d, e $\|v_q\| = 2.90$
1	0.00	0.00	0.00	0.95	0.39
2	0.78	0.53	0.88	0.29	0.92
3	0.00	0.00	0.00	0.40	0.16
4	0.00	0.67	0.15	0.40	0.30
5	0.00	0.46	0.10	0.76	0.40
6	0.00	0.24	0.05	0.48	0.24

Note that in the second query, the fourth document beats the second because the fourth is shorter overall. For each of the other queries, there is a single document with a very high similarity to the query.

3.4 Implementation of cosine measures using inverted lists

Directly computing the dot product of a query vector and all document vector s is too expensive, given the fact that both vectors are likely to be sparse. A more economical alternative is to keep (document, weight) pairs in the posting lists, so that a typical inverted file entry would look like

$$\langle t; [(d_{t,1}, w_{d_{t,1},t}), (d_{t,2}, w_{d_{t,2},t}), \ldots, (d_{t,m}, w_{d_{t,m},t})]\rangle.$$

Here, t is a term, $d_{t,i}$ is a pointer to the ith document containing term t, and $w_{d_{t,i},t}$ is weight of t in document $d_{t,i}$. (Heuristics for calculating $w_{d,t}$ were given in Section 3.1.) For example, the inverted file entry for *apple* might be

$$\langle \text{apple}; [(3,3), (6,1), (29,6)]\rangle.$$

Fortunately, the weights $w_{d_{t,i},t}$ can typically be compressed at least as well as the distances $d_{t,i+1} - d_{t,i}$ in the inverted file entry.

These posting lists help quicken the search for the documents which are most similar to a query vector. In the following algorithm Q is a list of query terms, which we assume are unweighted; $A = \{a_d \mid a_d$ is the score so far for document $d\}$ is a set of accumulators; and w_d is the weight of document d, which we assume has been precomputed. This algorithm returns the k documents which are most relevant to the query.

```
Search(Q)
For each term t ∈ Q
    ⟨t; P_t⟩ = Search lexicon for t
    P_t = Uncompress(P_t)
    For each (d, w_{d,t}) in P_t
        If a_d ∈ A
            a_d = a_d + w_{d,t}
        Else
            a_d = w_{d,t}
            A = A ∪ {a_d}
For each a_d ∈ A
    a_d = a_d/W_d
Return the k documents with the highest a_d.
```

In this algorithm, the inverted file entries for every term $t \in Q$ are processed in full. Each document d that appears in some such inverted file entry adds a cosine contribution to the accumulator a_d. At the end, the accumulator values are normalized by the weights W_d. Note that there is no need to normalize by the weight w_q of the query, as w_q is constant for any particular query.

The size of A can grow very large, so some search engines place an *a priori* bound on the number of accumulators a_d in A. One might think this approach would result in poor retrieval effectiveness, since the most relevant documents may be found by the last terms in the query. However, experiments with TREC and a standard collection of queries have shown that 5,000 accumulators suffice to extract the top 1,000 documents.

3.5 Relevance feedback

In the query systems discussed so far, the user poses a query v_{q_0}, the search engine answers it, and the process ends. But suppose the user has the ability to mark a set of documents R_0 as relevant, and a set I_0 as irrelevant. The search engine then modifies v_{q_0} to obtain v_{q_1} and fetches a set of documents relevant to v_{q_1}. This is called *relevance feedback*, and continues until the user is satisfied. It requires the initial query to be adapted, emphasizing some terms, de-emphasizing others, and perhaps introducing entirely new terms. One proposed strategy for updating the query is the *Dec Hi* strategy:

$$v_{q_{i+1}} = v_{q_i} + \left(\sum_{d \in R_i} v_d \right) - v_n.$$

To obtain the $(i+1)$th query, the vectors for the most relevant documents R_i (chosen by the user) are added to the ith query vector, and the vector v_n of the least relevant document is subtracted. A more general update rule is

$$v_{q_{i+1}} = \pi v_{q_0} + \omega v_{q_i} + \alpha \sum_{d \in R_i} v_d + \beta \sum_{d \in I_i} v_d,$$

where π, ω, α, and β are weighting constants, with $\beta \leq 0$.

One potential problem with these update rules is that the query vectors vq_i for $i \geq 1$ can have far more terms than the initial query v_{q_0}; thus, these subsequent queries may be too expensive to evaluate. One solution is to sort the terms in the relevant documents by decreasing weight, and select a subset of them to influence $v_{q_{i+1}}$. Another solution is to use clustering, which is described in the next section.

Experiments indicate that one round of relevance feedback improves responses from search engines, and two rounds yield a small additional improvement.

3.6 Clustering

Clustering may be used to speed up searches for complicated queries, and to find documents which are similar to each other. The idea is to represent a group of documents which are all close to each other by a single vector, for example the centroid of the group. Then, instead of calculating the relevance of each document to a query, we calculate the relevance of each *cluster* vector to a query. Then, once the most relevant cluster is found, the documents inside this cluster may be ranked against the query. This approach may be extended into a hierarchy of clusters; see figure 3.6.

There are many techniques for clustering, as well as many other applications. Clustering will be discussed in the November 21 lecture.

4 Latent semantic indexing (LSI)

All of the methods we have seen so far to search a collection of documents have matched words in users' queries to words in documents. These approaches all have two drawbacks. First,

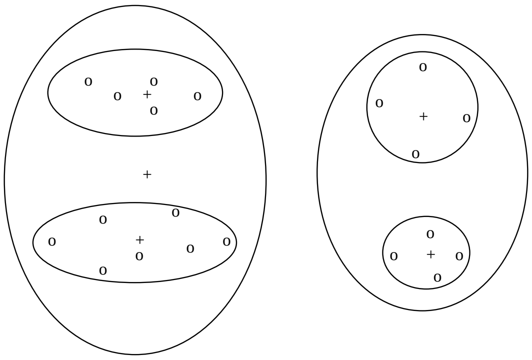

Figure 140: A hierarchical clustering approach

since there are usually many ways to express a given concept, there may be no document that matches the terms in a query even if there is a document that matches the meaning of the query. Second, since a given word may mean many things, a term in a query may retrieve irrelevant documents. In contrast, latent semantic indexing allows users to retrieve information on the basis of the conceptual content or meaning of a document. For example, the query *automobile* will pick up documents that do not contain *automobile*, but that do contain *car* or perhaps *driver*.

4.1 Singular value decomposition (SVD)

4.1.1 Definition

LSI makes heavy use of the singular value decomposition. Given an $m \times n$ matrix A with $m \geq n$, the singular value decomposition of A (in symbols, SVD(A)), is $A = U\Sigma V^T$, where:

1. $U^TU = V^TV = I_n$, the $n \times n$ identity matrix.

2. $\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_n)$, the matrix with all 0's except for the σ_i's along the diagonal.

If Σ is arranged so that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$, then the SVD of A is unique (except for the possibility of equal σ). Further, if r denotes rank(A), then $\sigma_i > 0$ for $1 \leq i \leq r$, and $\sigma_j = 0$ for $j \geq r+1$. Recall that the rank of a matrix is the number of independent rows (or columns).

We will use U_k to denote the first k columns of U, V_k the first k columns of V, and Σ_k the first k columns and rows of Σ. For latent semantic indexing, the key property of the SVD is given by the following theorem:

Theorem: Let $A = U\Sigma V^T$ be the singular value decomposition of the $m \times n$ matrix A. Define

$$A_k = U_k \Sigma_k V_k^T.$$

Define the distance between two $m \times n$ matrices A and B to be

$$\sum_{i=1}^{m}\sum_{j=1}^{n}(a_{ij} - b_{ij})^2.$$

Then A_k is the matrix of rank k with minimum distance to A.

This basically says that the best approximation of A (for the given metric) using a mapping of a k dimensional space back to the full dimension of A is based on taking the first k columns of U and V.

In these scribe notes, we will refer to $A_k = U_k \Sigma_k V_k^T$ as the *truncated SVD* of A. In the singular value decomposition $U \Sigma V^T$, the columns of U are the orthonormal eigenvectors of AA^T and are called the *left singular vectors*. The columns of V are the orthonormal eigenvectors of $A^T A$ and are called the *right singular vectors*. The diagonal values of Σ are the nonnegative square roots of the eigenvalues of AA^T, and are called the *singular values* of A.

The calculation of the SVD uses similar techniques as for calculating eigenvectors. For dense matrices, calculating the SVD takes $O(n^3)$ time. For sparse matrices, the Lanczos algorithm is more efficient, especially when only the first k columns of U and V are required.

4.2 Using SVD for LSI

To use SVD for latent semantic indexing, first construct a term-by-document matrix A. Here, $A_{t,d} = 1$ if term t appears in document d. Since most words do not appear in most documents, the term-by-document matrix is usually quite sparse. Next, computing SVD(A), generate U, Σ, and V. Finally, retain only the first k terms of U, Σ, and V. From these we could reconstruct A_k, which is an approximation of A, but we actually plan to use the U, Σ, and V directly.

The truncated SVD A_k describes the documents in k-dimensional space rather than the n dimensional space of A (think of this space as a k dimensional hyperplane through the space defined by the original matrix A). Intuitively, since k is much smaller than the number of terms, A_k "ignores" minor differences in terminology. Since A_k is the closest matrix of rank (dimension) k to A, terms which occur in similar documents will be near each other in the k-dimensional space. In particular, documents which are similar in meaning to a user's query will be near the query in k-dimensional space, even though these documents may share no terms with the query.

Let q be the vector representation of a query. This vector gets transformed into a vector $\hat{q}$ in k-dimensional space, according to the equation

$$\hat{q} = q^T U_k \Sigma_k^{-1}.$$

The projected query vector $\hat{q}$ gets compared to the document vectors, and documents get ranked by their proximity to the $\hat{q}$. LSI search engines typically use the cosine measure as a measure of nearness, and return all documents whose cosine with the query document exceeds some threshold. (For details on the cosine measure, see Section 3.2.)

4.3 An example of LSI

Suppose we want to apply LSI to the small database of book titles given in Figure 141 (a). Note that these titles fall into two basic categories, one having to do with the mathematics of differential equations (e.g. B8 and B10) and one having to do with algorithms (e.g. B5 and B6). We hope that the LSI will separate these classes. Figure 141 (b) shows the term-by-document matrix A for the titles and for a subset of the terms (the non stop-words that appear more than once).

Now for $k = 2$ we can generate U_2, Σ_2 and V_2 using an SVD. Now suppose that we are interested in all documents that pertain to *application* and *theory*. The coordinates for the query *application theory* are computed by the rule $\hat{q} = q^T U_2 \Sigma_2^{-1}$, as follows:

$$\hat{q} = (0.0511 \quad -0.3337) = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}^T \begin{pmatrix} 0.0159 & -0.4317 \\ 0.0266 & -0.3756 \\ 0.1785 & -0.1692 \\ 0.6104 & 0.1187 \\ 0.6691 & 0.1209 \\ 0.0148 & -0.3603 \\ 0.0520 & -0.2248 \\ 0.0066 & -0.1120 \\ 0.1503 & 0.1127 \\ 0.0813 & 0.0672 \\ 0.1503 & 0.1127 \\ 0.1785 & -0.1692 \\ 0.1415 & 0.0974 \\ 0.0105 & -0.2363 \\ 0.0952 & 0.0399 \\ 0.2051 & 0.5488 \end{pmatrix} \begin{pmatrix} 4.5314 & 0 \\ 0 & 2.7582 \end{pmatrix}^{-1}$$

All documents whose cosine with $\hat{q}$ exceeds 0.9 are illustrated in the shaded region of Figure 142. Note that B5, whose subject matter is very close to the query, yet whose title contains neither *theory* nor *application*, is very close to the query vector. Also note that the mapping into 2 dimensions clearly separates the two classes of articles, as we had hoped.

4.4 Applications of LSI

As we have seen, LSI is useful for traditional information retrieval applications, such as indexing and searching. LSI has many other applications, some of which are surprising.

Cross-language retrieval. This application allows queries in two languages to be performed on collections of documents in these languages. The queries may be posed in either language, and the user specifies the language of the documents that the search engine should return.

For this application, the term-by-document matrix contains documents which are composed of text in one language appended to the translation of this text in another language. For example, in one experiment, the documents were a set of abstracts that had versions

Label	Titles
B1	A Course on Integral Equations
B2	Attractors for Semigroups and Evolution Equations
B3	Automatic Differentiation of Algorithms: Theory, Implementation, and Application
B4	Geometrical Aspects of Partial Differential Equations
B5	Ideals, Varieties, and Algorithms – An Introduction to Computational Algebraic Geometry and Commutative Algebra
B6	Introduction to Hamiltonian Dynamical Systems and the N-Body Problem
B7	Knapsack Problems: Algorithms and Computer Implementations
B8	Methods of Solving Singular Systems of Ordinary Differential Equations
B9	Nonlinear Systems
B10	Ordinary Differential Equations
B11	Oscillation Theory for Neutral Differential Equations with Delay
B12	Oscillation Theory of Delay Differential Equations
B13	Pseudodifferential Operators and Nonlinear Partial Differential Equations
B14	Sinc Methods for Quadrature and Differential Equations
B15	Stability of Stochastic Differential Equations with Respect to Semi-Martingales
B16	The Boundary Integral Approach to Static and Dynamic Contact Problems
B17	The Double Mellin-Barnes Type Integrals and Their Applications to Convolution Theory

(a)

Terms	Documents																
	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15	B16	B17
algorithms	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
application	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
delay	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
differential	0	0	0	1	0	0	0	1	0	1	1	1	1	1	1	0	0
equations	1	1	0	1	0	0	0	1	0	1	1	1	1	1	1	0	0
implementation	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
integral	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
introduction	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
methods	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0
nonlinear	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
ordinary	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
oscillation	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
partial	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
problem	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0
systems	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0
theory	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1

(b)

Figure 141: An example set of document titles and the corresponding matrix A. Taken from "Using Linear Algebra for Intelligent Information Retrieval" by Berry, Dumais and O'Brien, CS-94-270, University of Tennessee.

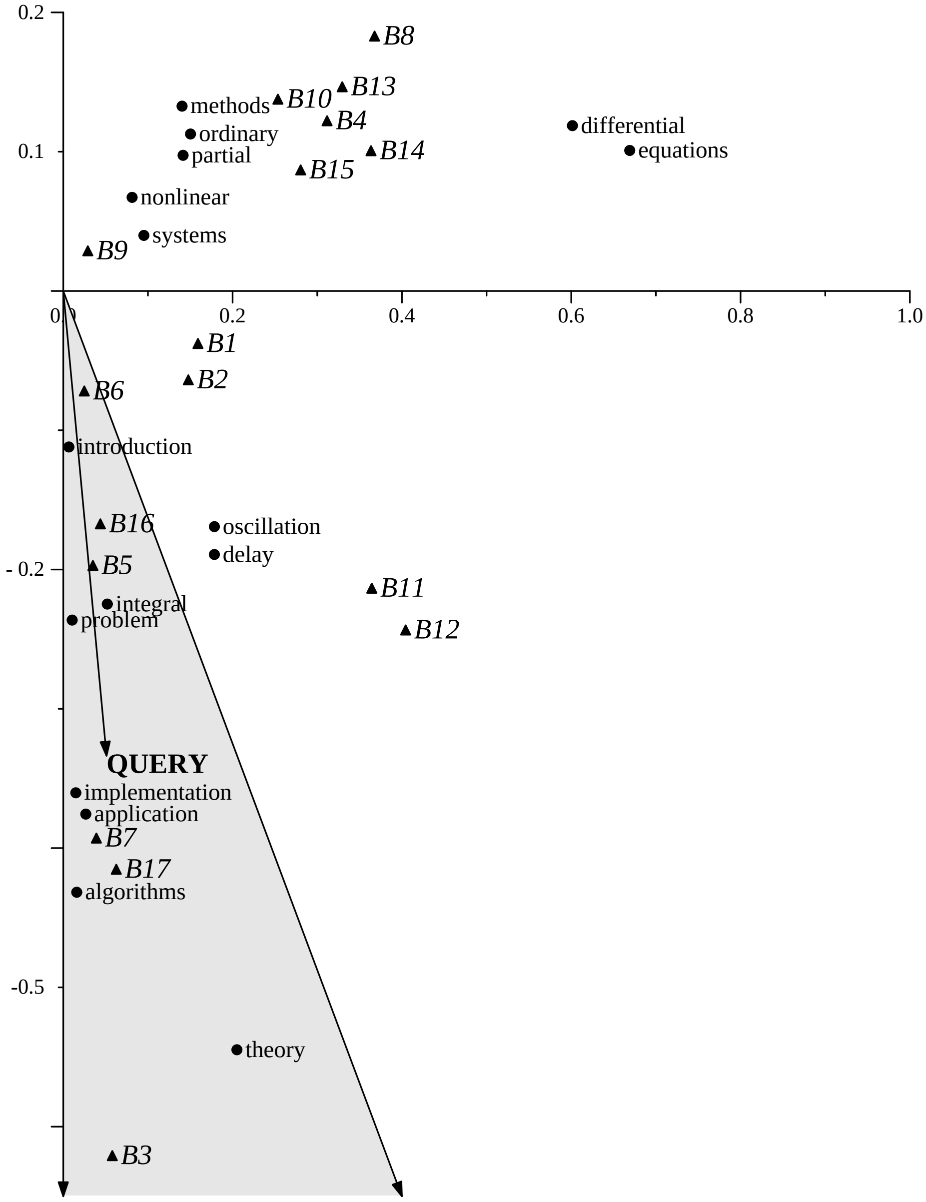

FIG. 6. *A Two-dimensional plot of terms and documents along with the query* **application theory**.

Figure 142: The 2-dimensional positioning of the documents and terms for a set of 17 articles.

in both French and English. The truncated SVD of the term-document matrix is then computed. After the SVD, monolingual documents can be "folded in" — a monolingual document will get represented as the vector sum of its constituent terms, which are already represented in the LSI space.

Experiments showed that the retrieval of French documents in response to English queries was as effective as first translating the queries into French, then performing a search on a French-only database. The cross-language retrieval method was nearly as effective for retrieving abstracts from an English-Kanji database, and for performing searches on English and Greek translations of the Bible.

Modeling human memory: LSI has been used to model some of the associative relationships in human memory, in particular, the type of memory used to recall synonyms. Indeed, LSI is often described as a method for finding synonyms — words which have similar meanings, like *cow* and *bovine*, are close to each other in LSI space, even if these words never occur in the same document. In one experiment, researchers calculated the truncated SVD of a term-by-document matrix for an encyclopedia. They then tested the truncated SVD on a synonym test, which had questions like

levied:
- (A) imposed
- (B) believed
- (C) requested
- (D) correlated

For a multiple-choice synonym test, they then computed the similarity of the first word (e.g. *levied*) to each choice, and picked the closest alternative as the synonym. The truncated SVD scored as well as the average student.

Matching people: LSI has been used to automate the assignment of reviewers to submitted conference papers. Reviewers were described by articles they had written. Submitted papers were represented by their abstracts, and then matched to the closest reviewers. In other words, reviewers were first placed in LSI space, and then people who submitted papers were matched to their closest reviewers. Analysis suggested that these automated assignments, which took less than one hour, were as good as those of human experts.

4.5 Performance of LSI on TREC

Recall that the TREC collection contains more then 1,000,000 documents, more than 3Gbytes of ASCII text. TREC also contains 200 standard benchmarking queries. A panel of human judges rates the effectiveness of a search engine by hand-scoring the documents returned by the search engine when posed with these queries. These 200 queries are quite detailed, averaging more than 50 words in length.

Because TREC queries are quite rich, a smaller advantage can be expected for any approach that involves enhancing user's queries, as LSI does. Nonetheless, when compared with the best keyword searches, LSI performed fairly well. For information retrieval tasks, LSI performed 16% better. For information filtering tasks, LSI performed 31% better. (In information filtering applications, a user has a stable profile, and new documents are matched against these long-standing interests.)

The cost of computing the truncated SVD on the full TREC collection was prohibitively expensive. Instead, a random sample of 70,000 documents and 90,000 terms was used. The resulting term-by-document matrix was quite sparse, containing only less than .002% non-zero entries. The Lanczos algorithm was used to find A_{200}; this computation required 18 hours of CPU time on a SUN SPARCstation 10.