1. **INTRODUCTION**

**1.1 MOTIVATION**

Latent Semantic Indexing (LSI) has been applied to a wide variety of learning tasks involving textual data. LSI is often applied for tasks such as search and retrieval, classification and retrieval. The vast amount of textual information available today is useless unless it can be effectively and efficiently searched. Information retrieval works in match user information requests, or queries, with relevant information items, or documents. LSI is a dimensionality reduction approach for modelling documents. It was originally thought to bring out the ‘latent semantics’ within a corpus of documents. However, LSI does not derive ‘semantic’ information per se. It does captures higher order term co-occurrence, and we prefer to state that LSI captures ‘term relationship’ information, rather than ‘latent semantic’ information. The LSI algorithm is easy to understand and implement (see Section 2), and theoretical work on LSI is ongoing. This theoretical work is important, because LSI does not provide improved retrieval performance across all copra .A natural result of this theoretical work is the development of algorithms which use just the term relationship information from LSI to improve retrieval performance. There some issues that remain with LSI:

1. The number of dimensions needed is typically large(100 - 300 dimensions). This has significant implicationsfor indexing run time, query run time, and theamount of memory required.

2. The number of dimensions needed must be determinedfor each collection; therefore training data, in theform of standardized queries with known truth sets, isneeded for each collection. This training data oftendoes not exist for large collections.

3. LSI performs worse than traditional vector space retrievalfor some collections.

**Lsi Timeline**

* **Mid-1960s** – Factor analysis technique first described and tested (H. Borko and M. Bernick)
* **1988** – Seminal paper on LSI technique published
* **1989** – Original patent granted
* **1992** – First use of LSI to assign articles to reviewers[]](https://en.wikipedia.org/wiki/Latent_semantic_indexing#cite_note-12)
* **1994** – Patent granted for the cross-lingual application of LSI (Landauer et al.)
* **1995** – First use of LSI for grading essays
* **1999** – First implementation of LSI technology for intelligence community for analyzing unstructured text ([SAIC](https://en.wikipedia.org/wiki/Science_Applications_International_Corporation)).
* **2002** – LSI-based product offering to intelligence-based government agencies (SAIC)
* **2005** – First vertical-specific application – publishing – EDB

**Mathematics of Lsi**

LSI uses common linear algebra techniques to learn the conceptual correlations in a collection of text. In general, the process involves constructing a weighted term-document matrix, performing a **Singular Value Decomposition** on the matrix, and using the matrix to identify the concepts contained in the text.

**1.2 PROBLEM DEFINTION**

**Latent semantic indexing** (**LSI**) is an indexing and retrieval method that uses a mathematical technique called singular value decomposition (SVD) to identify patterns in the relationships between the terms and concepts contained in an unstructured collection of text. LSI is based on the principle that words that are used in the same contexts tend to have similar meanings. A key feature of LSI is its ability to extract the conceptual content of a body of text by establishing associations between those terms that occur in similar contexts.

LSI is also an application of correspondence analysis, a multivariate statistical technique developed by Jean-Paul Benzécri in the early 1970s, to a contingency table built from word counts in documents.Called Latent Semantic Indexing because of its ability to correlate semantically related terms that are latent in a collection of text, it was first applied to text at Bellcore in the late 1980s. The method, also called latent semantic analysis (LSA), uncovers the underlying latent semantic structure in the usage of words in a body of text and how it can be used to extract the meaning of the text in response to user queries, commonly referred to as concept searches. Queries, or concept searches, against a set of documents that have undergone LSI will return results that are conceptually similar in meaning to the search criteria even if the results don’t share a specific word or words with the search criteria.

Regular keyword searches approach a document collection with a kind of accountant mentality: a document contains a given word or it doesn't, with no middle ground. We create a result set by looking through each document in turn for certain keywords and phrases, tossing aside any documents that don't contain them, and ordering the rest based on some ranking system. Each document stands alone in judgement before the search algorithm - there is no interdependence of any kind between documents, which are evaluated solely on their contents.

Latent semantic indexing adds an important step to the document indexing process. In addition to recording which keywords a document contains, the method examines the document collection as a whole, to see which other documents contain some of those same words. LSI considers documents that have many words in common to be semantically close, and ones with few words in common to be semantically distant. This simple method correlates surprisingly well with how a human being, looking at content, might classify a document collection. Although the LSI algorithm doesn't understand anything about what the words *mean*, the patterns it notices can make it seem astonishingly intelligent.

When you search an LSI-indexed database, the search engine looks at similarity values it has calculated for every content word, and returns the documents that it thinks best fit the query. Because two documents may be semantically very close even if they do not share a particular keyword, LSI does not require an exact match to return useful results. Where a plain keyword search will fail if there is no exact match, LSI will often return relevant documents that don't contain the keyword at all.

To use an earlier example, let's say we use LSI to index our collection of mathematical articles. If the words n-dimensional, manifold and topology appear together in enough articles, the search algorithm will notice that the three terms are semantically close. A search for n-dimensional manifolds will therefore return a set of articles containing that phrase (the same result we would get with a regular search), but also articles that contain just the word topology. The search engine understands nothing about mathematics, but examining a sufficient number of documents teaches it that the three terms are related. It then uses that information to provide an expanded set of results with better recall than a plain keyword search.

**Benefits of LSI**

LSI overcomes two of the most problematic constraints of Boolean keyword queries: multiple words that have similar meanings (synonymy) and words that have more than one meaning (polysemy). Synonymy is often the cause of mismatches in the vocabulary used by the authors of documents and the users of information retrieval systems.As a result, Boolean or keyword queries often return irrelevant results and miss information that is relevant.

LSI is also used to perform automated document categorization. In fact, several experiments have demonstrated that there are a number of correlations between the way LSI and humans process and categorize text. Document categorization is the assignment of documents to one or more predefined categories based on their similarity to the conceptual content of the categories. LSI uses *example* documents to establish the conceptual basis for each category. During categorization processing, the concepts contained in the documents being categorized are compared to the concepts contained in the example items, and a category (or categories) is assigned to the documents based on the similarities between the concepts they contain and the concepts that are contained in the example documents.

Dynamic clustering based on the conceptual content of documents can also be accomplished using LSI. Clustering is a way to group documents based on their conceptual similarity to each other without using example documents to establish the conceptual basis for each cluster. This is very useful when dealing with an unknown collection of unstructured text.

Because it uses a strictly mathematical approach, LSI is inherently independent of language. This enables LSI to elicit the semantic content of information written in any language without requiring the use of auxiliary structures, such as dictionaries and thesauri. LSI can also perform cross-linguistic concept searching and example-based categorization. For example, queries can be made in one language, such as English, and conceptually similar results will be returned even if they are composed of an entirely different language or of multiple languages.

LSI is not restricted to working only with words. It can also process arbitrary character strings. Any object that can be expressed as text can be represented in an LSI vector space. For example, tests with MEDLINE abstracts have shown that LSI is able to effectively classify genes based on conceptual modeling of the biological information contained in the titles and abstracts of the MEDLINE citations.

LSI automatically adapts to new and changing terminology, and has been shown to be very tolerant of noise (i.e., misspelled words, typographical errors, unreadable characters, etc.). This is especially important for applications using text derived from Optical Character Recognition (OCR) and speech-to-text conversion. LSI also deals effectively with sparse, ambiguous, and contradictory data.

Text does not need to be in sentence form for LSI to be effective. It can work with lists, free-form notes, email, Web-based content, etc. As long as a collection of text contains multiple terms, LSI can be used to identify patterns in the relationships between the important terms and concepts contained in the text.

LSI has proven to be a useful solution to a number of conceptual matching problems. The technique has been shown to capture key relationship information, including causal, goal-oriented, and taxonomic information.

**1.3 OBJECTIVE OF THE PROJECT**

Regular keyword searches approach a document collection with a kind of accountant mentality: a document contains a given word or it doesn't, with no middle ground. We create a result set by looking through each document in turn for certain keywords and phrases, tossing aside any documents that don't contain them, and ordering the rest based on some ranking system. Each document stands alone in judgement before the search algorithm - there is no interdependence of any kind between documents, which are evaluated solely on their contents.

Latent semantic indexing adds an important step to the document indexing process. In addition to recording which keywords a document contains, the method examines the document collection as a whole, to see which other documents contain some of those same words. LSI considers documents that have many words in common to be semantically close, and ones with few words in common to be semantically distant. This simple method correlates surprisingly well with how a human being, looking at content, might classify a document collection. Although the LSI algorithm doesn't understand anything about what the words *mean*, the patterns it notices can make it seem astonishingly intelligent.

When you search an LSI-indexed database, the search engine looks at similarity values it has calculated for every content word, and returns the documents that it thinks best fit the query. Because two documents may be semantically very close even if they do not share a particular keyword, LSI does not require an exact match to return useful results. Where a plain keyword search will fail if there is no exact match, LSI will often return relevant documents that don't contain the keyword at all.

To use an earlier example, let's say we use LSI to index our collection of mathematical articles. If the words n-dimensional, manifold and topology appear together in enough articles, the search algorithm will notice that the three terms are semantically close. A search for n-dimensional manifolds will therefore return a set of articles containing that phrase (the same result we would get with a regular search), but also articles that contain just the word topology. The search engine understands nothing about mathematics, but examining a sufficient number of documents teaches it that the three terms are related. It then uses that information to provide an expanded set of results with better recall than a plain keyword search.

**1.4 LIMITATIONS OF THE PROJECT**

Early challenges to LSI focused on scalability and performance. LSI requires relatively high computational performance and memory in comparison to other information retrieval techniques. However, with the implementation of modern high-speed processors and the availability of inexpensive memory, these considerations have been largely overcome. Real-world applications involving more than 30 million documents that were fully processed through the matrix and SVD computations are not uncommon in some LSI applications. A fully scalable (unlimited number of documents, online training) implementation of LSI is contained in the open source gensim software package.

Another challenge to LSI has been the alleged difficulty in determining the optimal number of dimensions to use for performing the SVD. As a general rule, fewer dimensions allow for broader comparisons of the concepts contained in a collection of text, while a higher number of dimensions enable more specific (or more relevant) comparisons of concepts. The actual number of dimensions that can be used is limited by the number of documents in the collection. Research has demonstrated that around 300 dimensions will usually provide the best results with moderate-sized document collections (hundreds of thousands of documents) and perhaps 400 dimensions for larger document collections (millions of documents). However, recent studies indicate that 50-1000 dimensions are suitable depending on the size and nature of the document collection.

Checking the amount of variance in the data after computing the SVD can be used to determine the optimal number of dimensions to retain. The variance contained in the data can be viewed by plotting the singular values (S) in a scree plot. Some LSI practitioners select the dimensionality associated with the knee of the curve as the cut-off point for the number of dimensions to retain. Others argue that some quantity of the variance must be retained, and the amount of variance in the data should dictate the proper dimensionality to retain. Seventy percent is often mentioned as the amount of variance in the data that should be used to select the optimal dimensionality for recomputing the SVD.

**2. LITERATURE SURVEY**

**2.1 INTRODUCTION**

**VECTOR SPACE METHOD**

**Basic Terminologies**

**Cosine Similarity**

* Distance between vectors *d*1 and *d*2 *captured* by the cosine of the angle *x* between them.
* A vector can be *normalized* (given a length of 1) by dividing each of its components by its length – here we use the *L2* norm

Capture38.JPG

* This maps vectors onto the unit sphere:
* Longer documents don’t get more weight

**Points And Space**

A point is just a list of numbers. This list of numbers, or coordinates, speciﬁes the point’s position in space. How many coordinates there are determines the dimensions of that space. For example, we can specify the position of a point on the edge of a ruler with a single coordinate.

The position of the two points 0.5cm and 1.2cm are precisely speciﬁed by single coordinates. Because we’re using a single coordinate to identify a point, we’re dealing with points in one-dimensional space, or 1-space.

Generally, space represented by more than three dimensions is called hyperspace. You’ll also see the term n-space used to talk about spaces of diﬀerent dimensionality (e.g. 1-space, 2-space, n-space).

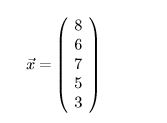
For example, if I want a succinct way of describing the amount of food I eat in a given day, I can use points in n-space to do so. Let the dimensions of this space be the following food items:

Eggs Grapes Bananas Chickens Cans of Tuna

There are ﬁve categories, so we’re dealing with points in 5-space. Thus, the interpretation of the point (3,18,2,0.5,1,) would be “three eggs, eighteen grapes, two bananas, half a chicken, one can of tuna”.

**Vectors**

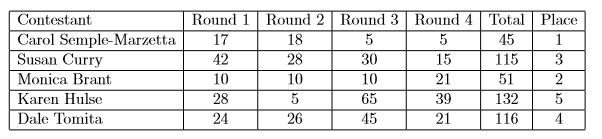
For most purposes, points and vectors are essentially the same thing1, that is, a sequence of numbers corresponding to measurements along various dimensions. Vectors are usually denoted by a lower case letter with an arrow on top, e.g. ⃗x. The numbers comprising the vector are now called components, and the number of components equals the dimensionality of the vector. We use a subscript on the vector name to refer to the component in that position. In the example below, ⃗x is a 5-dimensional vector, x1 = 8, x26, etc.



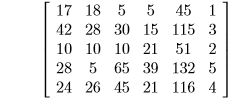
Vectors can be equivalently represented horizontally to save space, e.g. ⃗x = [8,6,7,5,3] is the same vector as above. More generally, a vector ⃗x with n-dimensions is a sequence of n numbers, and component xi represents the value of ⃗x on the ith dimension.

**Matrices**

A matrix is probably most familiar as a table of data, like , which shows the top 5 scorers on a judge’s scorecard in the 1997 Fitness International competition



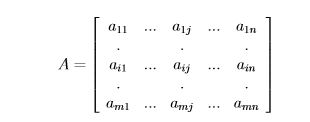
A table consists of rows (the horizontal list of scores corresponding to a contestant’s name), and columns (the vertical list of numbers corresponding to the scores for a given round). What makes this table a matrix is that it’s a rectangular array of numbers. Written as a matrix, Table 1 looks like this:



The size, or dimensions, of a matrix is given in terms of the number of rows by the number of columns. This makes the matrix above a “ﬁve by six” matrix, written 5×6 matrix. We can generalize the descriptions made so far by using variables to stand in for the actual numbers we’ve been using. Traditionally, a matrix in the abstract is named A. The maximum number of rows is assigned to the variable m, and the number of columns is called n. Matrix entries (also called elements or components) are denoted by a lower-case a, and a particular entry is referenced by its row index (labeledi) and its column index (labeled j). For example, 132 is the entry in row 4 and column 5 in the matrix above, so another way of saying that would be a45 = 132. More generally, the element in the ith row and jth column is labeledaij, and called the ij-entry or ij-component.

**Matrix Notation**

Let m, n be two integers ≥ 1. Let aij, i = 1,...,m, j = 1,...,n be mn numbers. An array of numbers



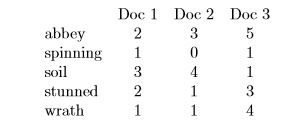
is an m×n matrix and the numbers aij are elements of A. The sequence of numbers

A(i) = (ai1,...,ain)

is the ith row of A, and the sequence of numbers

A(j) = (a1j,...,amj)

is the jth column of A. Just as the distinction between points and vectors can blur in practice, so does the distinction between vectors and matrices. A matrix is basically a collection of vectors. We can talk about row vectors or column vectors. Or a vector with n components can be considered a 1×n matrix. For example, the matrix below is a word×document matrix which shows the number of times a particular word occurs in some made-up documents. Typical accompanying description of



this kind of matrix might be something like “high dimensional vector space model”. The dimensions are the words, if we’re talking about the column vectors representing documents, or documents, if we’re talking about the row vectors which represent words. High dimensional means we have a lot of them. Thus, “hyperspace document representation” means a document is represented as a vector whose components correspond in some way to the words in it, plus there are a lot of words. This is equivalent to “a document is represented as a point in n-dimensional space.”

**Vector Terminology**

**Vector Length**

The length of a vector is found by squaring each component, adding them all together, and taking the square root of the sum. If ⃗v is a vector, its length is denoted by ⃗ |v|. Moreconsiely,

For example, if ⃗v = [4,11,8,10], then ⃗| v| = √42 + 112 + 82 + 102 = √301 = 17.35

**Vector Addition**

Adding two vectors means adding each component in v⃗1 to the component in the corresponding position in v⃗2 to get a new vector. For example [3,2,1,−2] + [2,−1,4,1] = [(3 + 2),(2−1),(1 + 4),(−2 + 1)] = [5,1,5,−1] More generally, if A = [a1,a2,...an] and B = [b1,b2,...bn], then A+B = [a1+b1,a2+b2,...an+ bn].

**Inner Product**

The inner product of two vectors (also called the dot product or scalar product) deﬁnes multiplication of vectors. It is found by multiplying each component in v⃗1 by the component

in v⃗2 in the same position and adding them all together to yield a scalar value. The inner product is only deﬁned for vectors of the same dimension. The inner product of two vectors is

denoted (v⃗1,v⃗2) or v⃗1 ·v⃗2 (the dot product). Thus,



For example, if ⃗x = [1,6,7,4] and ⃗y = [3,2,8,3], then ⃗x·⃗y = 1(3) + 6(2) + 7(8) + 3(4) = 83.

**Orthogonality**

Two vectors are orthogonal to each other if their inner product equals zero. In twodimensional space this is equivalent to saying that the vectors are perpendicular, or that the only angle between them is a 90◦ angle. For example, the vectors [2,1,−2,4] and [3,−6,4,2] are orthogonal because [2,1,−2,4]·[3,−6,4,2] = 2(3) + 1(−6)−2(4) + 4(2) = 0

**Normal Vector**

A normal vector (or unit vector) is a vector of length 1. Any vector with an initial length > 0 can be normalized by dividing each component in it by the vector’s length. For example, if ⃗v = [2,4,1,2], then

⃗| v| = √22 + 42 + 12 + 22 = √25 = 5

Then ⃗u = [2/5,4/5,1/5,1/5] is a normal vector because

⃗| u| =√(2/5)2 + (4/5)2 + (1/5)2 + (2/5)2 =√25/25 = 1

**Orthonormal Vectors**

Vectors of unit length that are orthogonal to each other are said to be orthonormal. For example,

⃗u = [2/5,1/5,−2/5,4/5] and ⃗v = [3/√65,−6/√65,4/√65,2/√65]

are orthonormal because

⃗| u| =√(2/5)2 + (1/5)2 + (−2/5)2 + (4/5)2 = 1

⃗| v| =√(3/√65)2 + (−6/√65)2 + (4/√65)2 + (2/√65)2 = 1

⃗u·⃗v = 6 5√65 − 6 5√65 − 8 5√65 + 8 5√65 = 0

**Eigenvalues and Eigenvectors**

Let A be an n × n matrix with elements being real numbers. If x is an n-dimensional vector, then the matrix-vector product Ax is well-deﬁned, and the result is again an n-dimensional vector. In general, multiplication by a matrix changes the direction of a non-zero vector x, unless the vector is special and we have that

Ax = λx,

for some scalar λ. In such a case, the multiplication by matrix A only stretches or contracts or reverses vector x, but it does not change its direction. These special vectors and their corresponding λ’s are called eigenvectors and eigenvalues of A. For diagonal matrices it is easy to spot the eigenvalues and eigenvectors. For example matrix

A =

has eigenvalues and eigenvectors

λ1 = 4 with x1=,λ2 = 3 with x2 =, and λ3 = 2 with x3 =

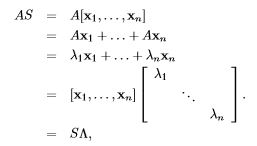
We will see that the number of eigenvalues is n for an n × n matrix. Regarding eigenvectors, if x is an eigenvector then so is ax for any scalar a. However, if we consider only one eigenvector for each ax family, then there is a 1-1 correspondence of such eigenvectors to eigenvalues. Typically, we consider eigenvectors of unit length. Diagonal matrices are simple, the eigenvalues are the entries on the diagonal, and the eigenvectors are their columns. For other matrices we ﬁnd the eigenvalues ﬁrst by reasoning as follows. If Ax = λx then (A−λI) x = 0, where I is the identity matrix. Since x is non-zero, matrix A−λI has dependent columns and thus its determinant |A − λI| must be zero. This gives us the equation |A − λI| = 0 whose solutions are the eigenvalues of A. As an example let

A =and A − λI=

Then the equation |A−λI| = 0 becomes −4 = 0 which has λ1 = 1 and λ2 = 5 as solutions. For each of these eigenvalues the equation (A − λI)x = 0 can be used to ﬁnd the corresponding eigenvectors,

e.g. A −λI=yields x1 =and A − λ2I =yields x2 =

In general, for an n × n matrix A, the determinant |A − λI| will give a polynomial of degree n which has n roots. In other words, the equation |A − λI| = 0 will give n eigenvalues. Let us create a matrix S with columns the n eigenvectors of A. We have that

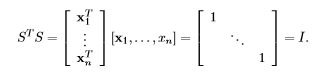


where Λ is the above diagonal matrix with the eigenvalues of A along its diagonal. Now suppose that the above n eigenvectors are linearly independent. This is true when the matrix has n distinct eigenvalues. Then matrix S is invertible and by mutiplying both sides of AS = SΛ we have

A = SΛS−1.

So, we were able to “diagonalize” matrix A in terms of the diagonal matrix Λ spelling the eigenvalues of A along its diagonal. This was possible because matrix S was invertible. When there are fewer than n eigenvalues then it might happen that the diagonalization is not possible. In such a case the matrix is “defective” having too few eigenvectors. In this tutorial, for reasons to be clear soon we will be interested in symmetric matrices (A = AT). For n × n

symmetric matrices it has been shown that that they always have real eigenvalues and their eigenvectors are perpendicular. As such we have that



In other words, for symmetric matrices, S−1 is ST and we have

A = SΛST .

### 2.2 EXISTING SYSTEM

### Term-Document Matrix

LSI begins by constructing a term-document matrix, A, to identify the occurrences of the m unique terms within a collection of n documents. In a term-document matrix, each term is represented by a row, and each document is represented by a column, with each matrix cell, a_{ij}, initially representing the number of times the associated term appears in the indicated document, \mathrm{tf_{ij}}. This matrix is usually very large and very sparse.

Once a term-document matrix is constructed, local and global weighting functions can be applied to it to condition the data. The weighting functions transform each cell, a_{ij} of A, to be the product of a local term weight, l_{ij}, which describes the relative frequency of a term in a document, and a global weight, g_i, which describes the relative frequency of the term within the entire collection of documents.

Some common local weighting functions [[13]](https://en.wikipedia.org/wiki/Latent_semantic_indexing#cite_note-13) are defined in the following table.

|  |  |  |
| --- | --- | --- |
| **Binary** |  | l_{ij} = 1 if the term exists in the document, or else 0 |
| **TermFrequency** |  | l_{ij} = \mathrm{tf}_{ij}, the number of occurrences of term i in document j |
| **Log** |  | l_{ij} = \log(\mathrm{tf}_{ij} + 1) |
| **Augnorm** |  | l_{ij} = \frac{\Big(\frac{\mathrm{tf}_{ij}}{\max_i(\mathrm{tf}_{ij})}\Big) + 1}{2} |

**2.3 DISADVANTAGES OF EXISTING SYSTEM**

The vector space model has the following limitations:

1. Long documents are poorly represented because they have poor similarity values
2. Search keywords must precisely match document terms; word [substrings](https://en.wikipedia.org/wiki/Substring) might result in a "[false positive](https://en.wikipedia.org/wiki/False_positive) match"
3. Semantic sensitivity; documents with similar context but different term vocabulary won't be associated, resulting in a "[false negative](https://en.wikipedia.org/wiki/False_negative) match".
4. The order in which the terms appear in the document is lost in the vector space representation.
5. Theoretically assumes terms are statistically independent.
6. Weighting is intuitive but not very formal.

Many of these difficulties can, however, be overcome by the integration of various tools, including mathematical techniques such as [singular value decomposition](https://en.wikipedia.org/wiki/Singular_value_decomposition) and [lexical databases](https://en.wikipedia.org/wiki/Lexical_database) such as [WordNet](https://en.wikipedia.org/wiki/WordNet).

**2.4 PROPOSED SYSTEM**

LSA is a fully automatic mathematical/statistical technique for extracting and inferring relations of expected contextual usage of words in passages of discourse. It is not a traditional natural language processing or artificial intelligence program; it uses no humanly constructed dictionaries, knowledge bases, semantic networks, grammars, syntactic parsers, or morphologies, or the like, and takes as its input only raw text parsed into words defined as unique character strings and separated into meaningful passages or samples suchas sentences or paragraphs.

The first step is to represent the text as a matrix in which each row stands for a unique word and each column stands for a text passage or other context. Each cell contains the frequency with which the word of its row appears in the passage denoted by its column. Next, the cell entries are subjected to a preliminary transformation, whose details we will describe later, in which each cell frequency is weighted by a function that expresses both the word’s importance in the particular passage and the degree to which the word type carries information in the domain of discourse in general.

Next, LSA applies singular value decomposition (SVD) to the matrix. This is a form of factor analysis, or more properly the mathematical generalization of which factor analysis is a special case. In SVD, a rectangular matrix is decomposed into the product of three other matrices. One component matrix describes the original row entities as vectors of derived orthogonal factor values, another describes the original column entities in the same way, and the third is a diagonal matrix containing scaling values such that when the three components are matrix-multiplied, the original matrix is reconstructed. There is a mathematical proof that any matrix can be so decomposed perfectly, using no more factorsthan the smallest dimension of the original matrix. When fewer than the necessary numberof factors are used, the reconstructed matrix is a least-squares best fit. One can reduce thedimensionality of the solution simply by deleting coefficients in the diagonal matrix,ordinarily starting with the smallest.

Here is a small example that gives the flavor of the analysis and demonstrates what the technique accomplishes. This example uses as text passages the titles of nine technical memoranda, five about human computer interaction (HCI), and four about mathematical graph theory, topics that are conceptually rather disjoint. Thus the original matrix has nine columns, and we have given it 12 rows, each corresponding to a content word used in at least two of the titles. The titles, with the extracted terms italicized, and the corresponding word-by-document matrix . We will discuss the highlighted partsof the tables in due course.

The linear decomposition is shown next except for rounding errors, its multiplication perfectly reconstructs the original Next we show a reconstruction based on just two dimensions that approximates the original matrix. This uses vector elements only from the first two,shaded, columns of the three matrices shown in the previous figure Each value in this new representation has been computed as a linear combination of values on the two retained dimensions, which in turn were computed as linear combinations of the original cell values. Note, therefore, that if we were to change the entry in any one cell of the original, the values in the reconstruction with reduced dimensions might be changed everywhere; this is the mathematical sense in which LSA performs inference or induction.

1. **ANALYSIS**

The document collection is represented by an m 3 n term document matrix where m is the number of terms and n is the number of documents. Typically this matrix has fewer than 1% nonzero entries. Queries are represented as m-vectors, and a matrix-vector product produces an n-vector of scores that is used to rank the documents in relevance. LSI is based on the vector space method, but the m 3 n term-document matrix is replaced with a low-rank approximation generated by the truncated singular-value decomposition (SVD). The truncated SVD approximation is the sum of k rank-1 outer products of m-vectors ui with n-vectors via, weighted by scalars

**SINGULAR VALUED DECOMPOSTION**

Singular value decomposition (SVD) can be looked at from three mutually compatible points of view. On the one hand, we can see it as a method for transforming correlated variables into a set of uncorrelated ones that better expose the various relationships among the original data items. At the same time, SVD is a method for identifying and ordering the dimensions b along which data points exhibit the most variation. This ties in to the third way of viewing SVD, which is that once we have identifed where the most variation is, it's possible to \_nd the best approximation of the original data points using fewer dimensions. Hence, SVD can be seen as a method for data reduction .As an illustration of these ideas, consider the 2-dimensional data points in Figure 1.the regression line running through them shows the best approximation of the original data with a 1-dimensional object (a line). It is the best approximation in the sense that it is the line that minimizes the distance between each original point and the line. If we drew a perpendicular line from each point to the regression line, and took the intersection of those lines as the approximation of the original datapoint, we would have a reduced representation of the original data that captures as much of the original variation as possible. Notice that there is a second regression line, perpendicular to the \_rst, shown in Figure 2. This line captures as much of the variation as possible along the second dimension of the original data set. It does a poorer job of approximating the orginal data because it corresponds to a dimension exhibiting less variation to begin with. It is possible to use these regression lines to generate a set of uncorrelated data points that will show subgroupings in the original data not necessarily visible at \_rst glance.

These are the basic ideas behind SVD: taking a high dimensional, highly variable set of data points and reducing it to a lower dimensional space that exposes the substructure of the original data more clearly and orders it from most variation to the least. What makes SVD

practical for NLP applications is that you can simply ignore variation below a particular

threshold to massively reduce your data but be assured that the main relationships of

interest have been preserved.

**Example of Full Singular Value Decomposition**

SVD is based on a theorem from linear algebra which says that a rectangular matrix *A* can be broken down into the product of three matrices - an orthogonal matrix *U*, a diagonal

matrix *S*, and the transpose of an orthogonal matrix *V* . The theorem is usually presented

something like this:

*Amn* = *UmmSmn V T*

*nn*

where *UTU* = *I, V TV* = *I*; the columns of *U* are orthonormal eigenvectors of *AAT* , the

columns of *V* are orthonormal eigenvectors of *ATA*, and *S* is a diagonal matrix containing

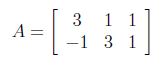
the square roots of eigenvalues from *U* or *V* in descending order.

The following example merely applies this de\_nition to a small matrix in order to compute

its SVD. In the next section, I attempt to interpret the application of SVD to document

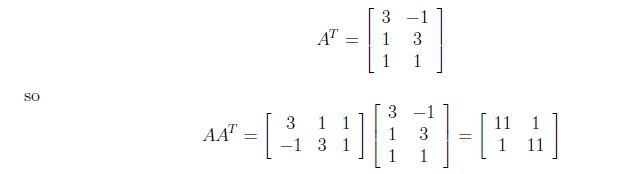
classification.

Start with the matrix



Regression line along second dimension captures less variation in original data.

In order to find *U*, we have to start with *AAT* . The transpose of *A* is



Next, we have to \_nd the eigenvalues and corresponding eigenvectors of *AAT* . We know that

eigenvectors are de\_ned by the equation *A⃗v* = *λ⃗v*, and applying this to *AAT* gives us Capture5.JPG

We rewrite this as the set of equations

11*x*1 + *x*2 = *λx*1

*x*1 + 11*x*2 = *λx*2

and rearrange to get

(11 *− λ*)*x*1 + *x*2 = 0

*x*1 + (11 *− λ*)*x*2 = 0

Solve for *λ* by setting the determinant of the coefficient matrix to zero,

\_\_\_\_\_

(11 *− λ*) 1

1 (11 *− λ*)

\_\_\_\_\_

= 0

which works out as

(11 *− λ*)(11 *− λ*) *−* 1 *·* 1 = 0

(*λ −* 10)(*λ −* 12) = 0

*λ* = 10*, λ* = 12

to give us our two eigen values *λ* = 10*, λ* = 12. Plugging *λ* back in to the original equations

gives us our eigenvectors. For *λ* = 10 we get (11 *−* 10)*x*1 + *x*2 = 0

*x*1 = *−x*2

which is true for lots of values, so we'll pick *x*1 = 1 and *x*2 = *−*1 since those are small and

easier to work with. Thus, we have the eigenvector [1*,−*1] corresponding to the eigenvalue

*λ* = 10. For *λ* = 12 we have

(11 *−* 12)*x*1 + *x*2 = 0

*x*1 = *x*2

and for the same reason as before we'll take *x*1 = 1 and *x*2 = 1. Now, for *λ* = 12 we have the

eigenvector [1*,* 1]. These eigenvectors become column vectors in a matrix ordered by the size

of the corresponding eigenvalue. In other words, the eigenvector of the largest eigenvalue

is column one, the eigenvector of the next largest eigenvalue is column two, and so forth

and so on until we have the eigenvector of the smallest eigenvalue as the last column of our

matrix. In the matrix below, the eigenvector for *λ* = 12 is column one, and the eigenvector

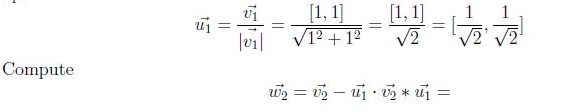
for *λ* = 10 is column two.

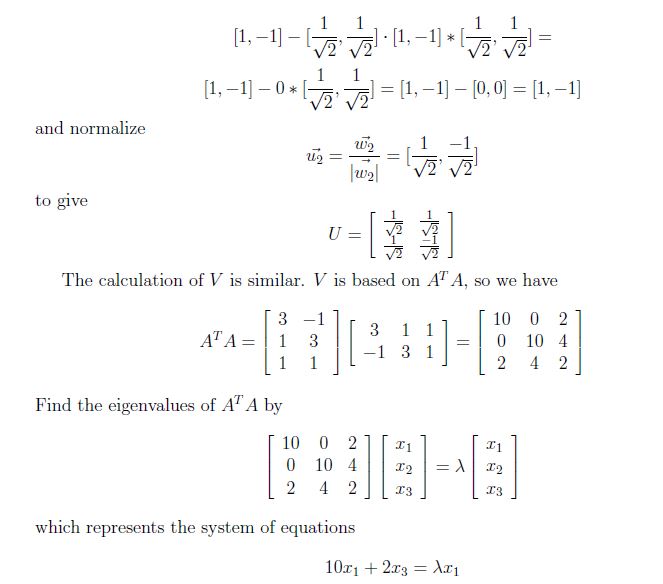
Capture6.JPG

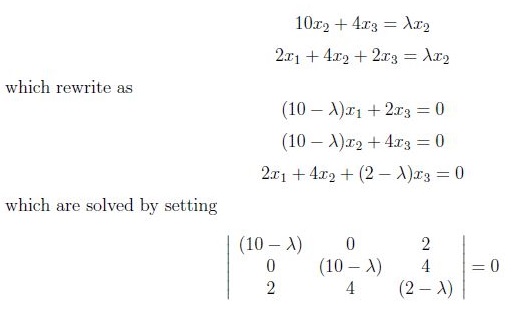
Finally, we have to convert this matrix into an orthogonal matrix which we do by applying

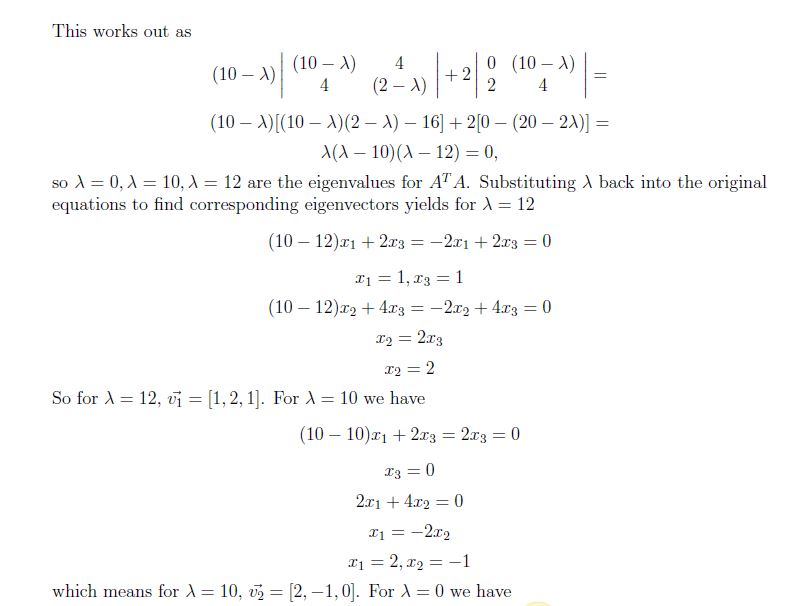
the Gram-Schmidt orthonormalization process to the column vectors. Begin by normalizing

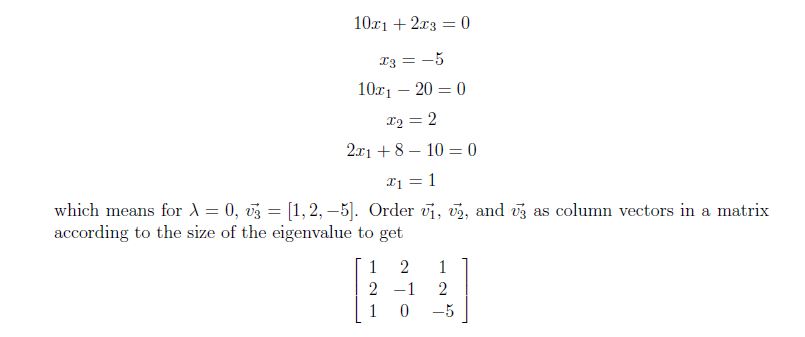
*⃗v*1.

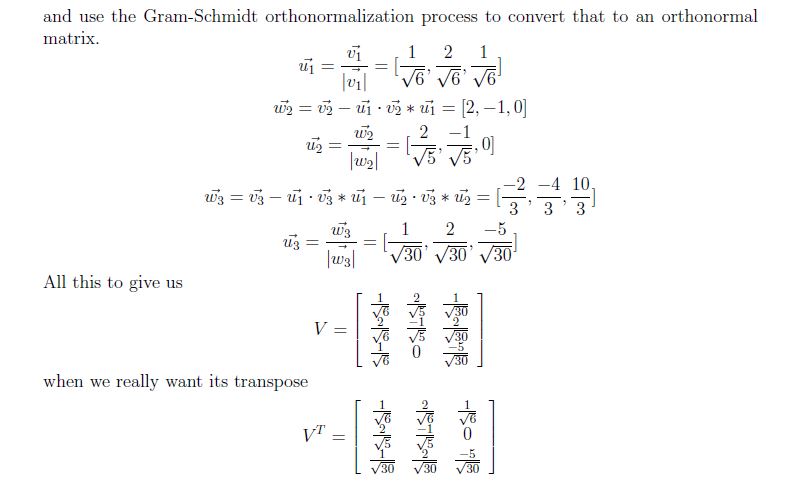












For *S* we take the square roots of the non-zero eigenvalues and populate the diagonal with

them, putting the largest in *s*11, the next largest in *s*22 and so on until the smallest value

ends up in *smm*. The non-zero eigenvalues of *U* and *V* are always the same, so that's why

it doesn't matter which one we take them from. Because we are doing full SVD, instead of

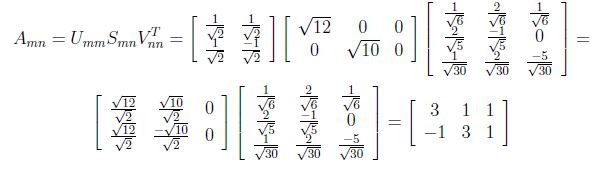
reduced SVD (next section), we have to add a zero column vector to *S* so that it is of the

proper dimensions to allow multiplication between *U* and *V* . The diagonal entries in *S* are

the singular values of *A*, the columns in *U* are called left singular vectors, and the columns

in *V* are called right singular vectors.

Capture13.JPG



**3.2 SYTEM SPECIFICATIONS**

### 3.2.1 Hardware Specifications

* System : Pentium IV 2.4 GHz.
* Hard Disk : 40 GB.
* Floppy Drive : 1.44 Mb.
* Monitor : 15 VGA Colour.
* Mouse : Logitech.
* Ram : 512 Mb.

### 

### 3.2.2 Software Requirements:

* Operating system : Windows XP/7/LINUX.
* Implementation : JAVA

**3.3 CONTENT DIAGRAM OF THE PROJECT**

We mentioned that latent semantic indexing looks at patterns of word distribution ( across a set of documents. Before we talk about the mathematical underpinnings, we should be a little more precise about what kind of words LSI looks at.

Natural language is full of redundancies, and not every word that appears in a document carries semantic meaning. In fact, the most [frequently used words in English](http://www.duboislc.org/EducationWatch/First100Words.html) are words that don't carry content at all: functional words, conjunctions, prepositions, auxilliary verbs and others. The first step in doing LSI is culling all those extraeous words from a document, leaving only content words likely to have semantic meaning. There are many ways to define a content word - here is one recipe for generating a list of content words from a document collection:

1. Make a complete list of all the words that appear anywhere in the collection
2. Discard articles, prepositions, and conjunctions
3. Discard common verbs (know, see, do, be)
4. Discard pronouns
5. Discard common adjectives (big, late, high)
6. Discard frilly words (therefore, thus, however, albeit, etc.)
7. Discard any words that appear in every document
8. Discard any words that appear in only one document

This process condenses our documents into sets of content words that we can then use to index our collection.

## Thinking Inside The Grid

Using our list of content words and documents, we can now generate a term document matrix. This is a fancy name for a very large grid, with documents listed along the horizontal axis, and content words along the vertical axis. For each content word in our list, we go across the appropriate row and put an 'X' in the column for any document where that word appears. If the word does not appear, we leave that column blank.

Doing this for every word and document in our collection gives us a mostly empty grid with a sparse scattering of X-es. This grid displays everthing that we know about our document collection. We can list all the content words in any given document by looking for X-es in the appropriate column, or we can find all the documents containing a certain content word by looking across the appropriate row.

Notice that our arrangement is binary - a square in our grid either contains an X, or it doesn't. This big grid is the visual equivalent of a generic keyword search, which looks for exact matches between documents and keywords. If we replace blanks and X-es with zeroes and ones, we get a numerical matrix containing the same information.

The key step in LSI is decomposing this matrix using a technique called singular value decomposition. The mathematics of this transformation are beyond the scope of this article but we can get an intuitive grasp of what SVD does by thinking of the process spatially. An analogy will help.

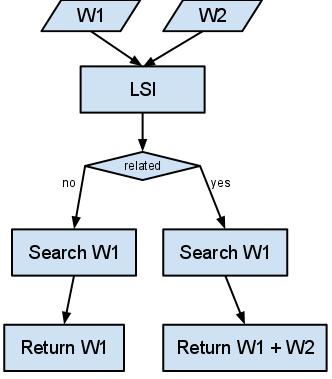
****

FIG 3.1 CONTENT DIAGRAM

**4. DESIGN**

**4.1 INTRODUCTION**

The unified modeling language allows the software engineer to express an analysis model using the modeling notation that is governed by a set of syntactic semantic and pragmatic rules.

A UML system is represented using five different views that describe the system from distinctly different perspective. Each view is defined by a set of diagram, which is as follows.

* + User Model View
    1. This view represents the system from the users perspective.
    2. The analysis representation describes a usage scenario from the end-users perspective.
  + Structural model view
    1. In this model the data and functionality are arrived from inside the system.
    2. This model view models the static structures.
* Behavioral Model View

It represents the dynamic of behavioral as parts of the system, depicting the interactions of collection between various structural elements described in the user model and structural model view.

* Implementation Model View

In this the structural and behavioral as parts of the system are represented as they are to be built.

* Environmental Model View

In this the structural and behavioral aspects of the environment in which the system is to be implemented are represented.

UML is specifically constructed through two different domains they are:

* UML Analysis modeling, this focuses on the user model and structural model views of the system.
* UML design modeling, which focuses on the behavioral modeling, implementation modeling and environmental model views.

Use case Diagrams represent the functionality of the system from a user’s point of view. Use cases are used during requirements elicitation and analysis to represent the functionality of the system. Use cases focus on the behavior of the system from external point of view.

Actors are external entities that interact with the system. Examples of actors include users like administrator, bank customer …etc., or another system like central database.

**Class Diagram**:

Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, associations and collaboration. Class diagrams basically represent the object oriented view of a system which is static in nature.Active class is used in a class diagram to represent the concurrency of the system.Class diagram represents the object orientation of a system. So it is generally used for development purpose. This is the most widely used diagram at the time of system construction.

**Object Diagram:**

Object diagrams can be described as an instance of class diagram. So these diagrams are more close to real life scenarios where we implement a systemObject diagrams are a set of objects and their relationships just like class diagrams and also represent the static view of the system. The usage of object diagrams is similar to class diagrams but they are used to build prototype of a system from practical perspective.

**Component Diagram:**

Component diagrams represent a set of components and their relationships. These components consist of classes, interfaces or collaborations. So Component diagrams represent the implementation view of a system. During design phase software artifacts (classes, interfaces etc) of a system are arranged in different groups depending upon their relationship. Now these groups are known as components.Finally, component diagrams are used to visualize the implementation.

**Deployment Diagram:**

Deployment diagrams are a set of nodes and their relationships. These nodes are physical entities where the components are deployed. Deployment diagrams are used for visualizing deployment view of a system. This is generally used by the deployment team.

**Use case Diagram:**

Use case diagrams are a set of use cases, actors and their relationships. They represent the use case view of a system. A use case represents a particular functionality of a system**.** So use case diagram is used to describe the relationships among the functionalities and their internal/external controllers. These controllers are known as actors.

**Sequence Diagram:**

A sequence diagram is an interaction diagram. From the name it is clear that the diagram deals with some sequences, which are the sequence of messages flowing from one object to another.Interaction among the components of a system is very important from implementation and execution perspective**.**So Sequence diagram is used to visualize the sequence of calls in a system to perform a specific functionality**.**

**Collaboration Diagram:**

Collaboration diagram is another form of interaction diagram. It represents the structural organization of a system and the messages sent/received. Structural organization consists of objects and links.The purpose of collaboration diagram is similar to sequence diagram. But the specific purpose of collaboration diagram is to visualize the organization of objects and their interaction.

**Statechart Diagram:**

Any real time system is expected to be reacted by some kind of internal/external events. These events are responsible for state change of the system. State chart diagram is used to

represent the event driven state change of a system. It basically describes the state change of a class, interface etc. State chart diagram is used to visualize the reaction of a system by internal/external factors.

**Activity Diagram:**

Activity diagram describes the flow of control in a system. So it consists of activities and links. The flow can be sequential, concurrent or branched. Activities are nothing but the functions of a system. Numbers of activity diagrams are prepared to capture the entire flow in a system. Activity diagrams are used to visualize the flow of controls in a system. This is prepared to have an idea of how the system will work when executed.

**4.2 UML DIAGRAMS**

**Activity Diagram :**

Activity diagram is another important diagram in UML to describe dynamic aspects of the system. Activity diagram is basically a flow chart to represent the flow form one activity to another activity. The activity can be described as an operation of the system.

So the control flow is drawn from one operation to another. This flow can be sequential, branched or concurrent. Activity diagrams deals with all type of flow control by using different elements like fork, join etc.

Purpose:

The basic purposes of activity diagrams are similar to other four diagrams. It captures the dynamic behaviour of the system. Other four diagrams are used to show the message flow from one object to another but activity diagram is used to show message flow from one activity to another.

Activity is a particular operation of the system. Activity diagrams are not only used for visualizing dynamic nature of a system but they are also used to construct the executable system by using forward and reverse engineering techniques. The only missing thing in activity diagram is the message part.

It does not show any message flow from one activity to another. Activity diagram is some time considered as the flow chart. Although the diagrams looks like a flow chart but it is not. It shows different flow like parallel, branched, concurrent and single.

So the purposes can be described as:

* Draw the activity flow of a system.
* Describe the sequence from one activity to another.
* Describe the parallel, branched and concurrent flow of the system.

**How to draw Activity Diagram?**

Activity diagrams are mainly used as a flow chart consists of activities performed by the system. But activity diagram are not exactly a flow chart as they have some additional capabilities. These additional capabilities include branching, parallel flow, swimlane etc.

Before drawing an activity diagram we must have a clear understanding about the elements used in activity diagram. The main element of an activity diagram is the activity itself. An activity is a function performed by the system. After identifying the activities we need to understand how they are associated with constraints and conditions.

So before drawing an activity diagram we should identify the following elements:

* Activities
* Association
* Conditions
* Constraints

Once the above mentioned parameters are identified we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

The following is an example of an activity diagram for order management system. In the diagram four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and mainly used by the business users.

After receiving the order request condition checks are performed to check if it is normal or special order. After the type of order is identified dispatch activity is performed and that is marked as the termination of the process.

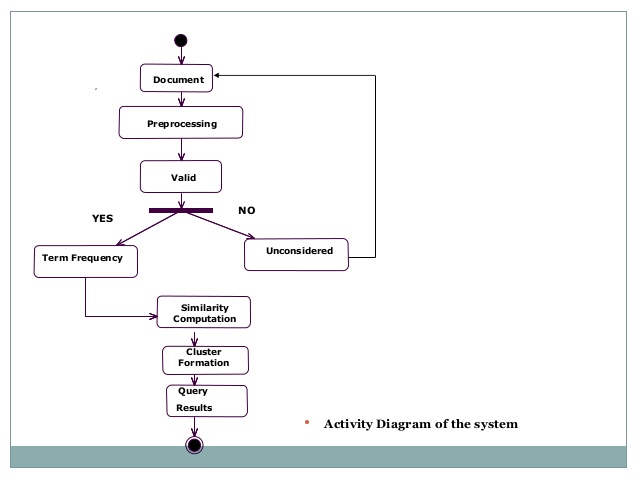


Fig 4.1 Activity Diagram

**Statechart Diagram**

The name of the diagram itself clarifies the purpose of the diagram and other details. It describes different states of a component in a system. The states are specific to a component/object of a system.

A Statechart diagram describes a state machine. Now to clarify it state machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events. Activity diagram explained in next chapter, is a special kind of a Statechart diagram. As Statechart diagram defines states it is used to model lifetime of an object.

Purpose:

Statechart diagram is one of the five UML diagrams used to model dynamic nature of a system. They define different states of an object during its lifetime. And these states are changed by events. So Statechart diagrams are useful to model reactive systems. Reactive systems can be defined as a system that responds to external or internal events.

Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. So the most important purpose of Statechart diagram is to model life time of an object from creation to termination. Statechart diagrams are also used for forward and reverse engineering of a system. But the main purpose is to model reactive system.

Following are the main purposes of using Statechart diagrams:

* To model dynamic aspect of a system.
* To model life time of a reactive system.
* To describe different states of an object during its life time.
* Define a state machine to model states of an object.

**How to draw Statechart Diagram**?

Statechart diagram is used to describe the states of different objects in its life cycle. So the emphasis is given on the state changes upon some internal or external events. These states of objects are important to analyze and implement them accurately.

Statechart diagrams are very important for describing the states. States can be identified as the condition of objects when a particular event occurs.

Before drawing a Statechart diagram we must have clarified the following points:

* Identify important objects to be analyzed.
* Identify the states.
* Identify the events.

The following is an example of a Statechart diagram where the state of Order object is analyzed.

During the life cycle of an object it goes through the following states and there may be some abnormal exists also. This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete it is considered as the complete transaction as mentioned below.

The initial and final state of an object is also shown below.

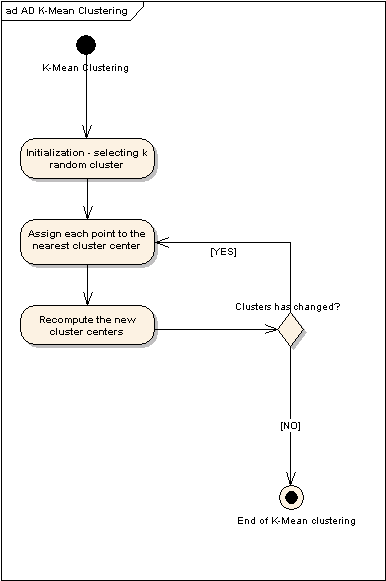


Fig 4.2 State Chart Diagram

**Use Case Diagram**

To model a system the most important aspect is to capture the dynamic behaviour. To clarify a bit in details, dynamic behaviour means the behaviour of the system when it is running /operating.

So only static behaviour is not sufficient to model a system rather dynamic behaviour is more important than static behaviour. In UML there are five diagrams available to model dynamic nature and use case diagram is one of them. Now as we have to discuss that the use case diagram is dynamic in nature there should be some internal or external factors for making the interaction.

These internal and external agents are known as actors. So use case diagrams are consists of actors, use cases and their relationships. The diagram is used to model the system/subsystem of an application. A single use case diagram captures a particular functionality of a system.

So to model the entire system numbers of use case diagrams are used.

**Purpose:**

The purpose of use case diagram is to capture the dynamic aspect of a system. But this definition is too generic to describe the purpose.

Because other four diagrams (activity, sequence, collaboration and Statechart) are also having the same purpose. So we will look into some specific purpose which will distinguish it from other four diagrams.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. So when a system is analyzed to gather its functionalities use cases are prepared and actors are identified.

Now when the initial task is complete use case diagrams are modelled to present the outside view.

So in brief, the purposes of use case diagrams can be as follows:

* Used to gather requirements of a system.
* Used to get an outside view of a system.
* Identify external and internal factors influencing the system.
* Show the interacting among the requirements are actors.

**How To Draw Use Case Diagram?**

Use case diagrams are considered for high level requirement analysis of a system. So when the requirements of a system are analyzed the functionalities are captured in use cases.

So we can say that uses cases are nothing but the system functionalities written in an organized manner. Now the second things which are relevant to the use cases are the actors. Actors can be defined as something that interacts with the system.

The actors can be human user, some internal applications or may be some external applications. So in a brief when we are planning to draw an use case diagram we should have the following items identified.

* Functionalities to be represented as an use case
* Actors
* Relationships among the use cases and actors.

Use case diagrams are drawn to capture the functional requirements of a system. So after identifying the above items we have to follow the following guidelines to draw an efficient use case diagram.

* The name of a use case is very important. So the name should be chosen in such a way so that it can identify the functionalities performed.
* Give a suitable name for actors.
* Show relationships and dependencies clearly in the diagram.
* Do not try to include all types of relationships. Because the main purpose of the diagram is to identify requirements.
* Use note when ever required to clarify some important points.

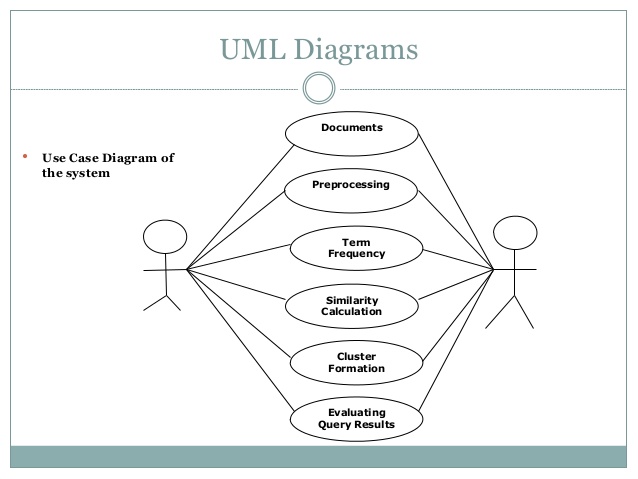


Fig 4.3 Use Case Diagram

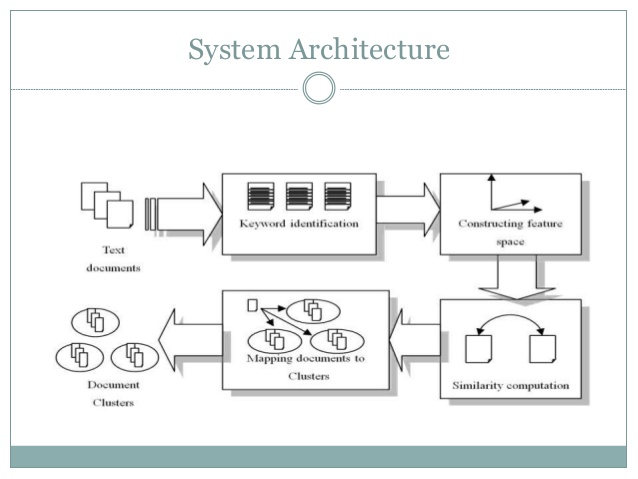


Fig 4.4 System Architecture

**4.3 MODULES AND DESIGNATION**

There are three module used in this project

1. Data collection

2. Document term matrix

3. Single Valued Decomposition

**4.3.1 Data Collection**

A “collection” consists of the following “documents”:

d1: *Shipment of gold damaged in a fire.*

d2: *Delivery of silver arrived in a silver truck.*

d3: *Shipment of gold arrived in a truck.*

Suppose that we use the term frequency as term weights and query weights. The

following document indexing rules are also used:

• stop words were not ignored

• text was tokenized and lowercased

• no stemming was used

• terms were sorted alphabetically

We wish to use this example to illustrate how LSI works.

**4.3.2 Document Term Matrix**

Use Latent Semantic Indexing (LSI) to rank these documents for the query

*gold silver truck*.

**Step 1:** Set term weights and construct the term-document matrix **A** and query matrix:

Keyword Doc1 Doc2 Doc3 query

shipmnt 1 0 1 0

gold 1 0 1 1

damaged 1 0 0 0

delvry 0 1 0 0

silver 0 2 0 1

arrived 0 1 1 0

truck 0 1 1 1

**4.3.3 Singular Value Decomposition**

Decompose matrix **A** matrix and find the **U**, **S** and **V** matrices, where

**A = USVT**

U= -0.241 0.571 0.051

-0.241 0.571 0.051

-0.067 0.311 0.667

-0.296 -0.223 0.211

-0.592 -0.447 0.421

-0.470 0.037 -0.405

-0.470 0.037 -0.405

VT= -0.192 -0.846 -0.497

0.672 -0.483 0.562

0.715 0.226 -0.661

S= 2.859 0.000 0.000

0.000 2.162 0.000

0.000 0.000 1.073

**5. IMPLEMENTATION & RESULTS**

**5.1 INTRODUCTION**

After the implementation of the singular valued decomposition method we have reduced the formed matrices. There are two in we can reduce the given matrix, it depends upon the number of documents taken in the analysis phase

### Rank-Reduced Singular Value Decomposition

A rank-reduced, singular value decomposition is performed on the matrix to determine patterns in the relationships between the terms and concepts contained in the text. The SVD forms the foundation for LSI. It computes the term and document vector spaces by approximating the single term-frequency matrix, A, into three other matrices— an ***m***by ***r*** term-concept vector matrix T, an ***r*** by ***r*** singular values matrix S, and a ***n*** by ***r*** concept-document vector matrix, D, which satisfy the following relations:

A \approx TSD^T

T^T T = I_r \quad D^T D = I_r 

S_{1,1} \geq S_{2,2} \geq \ldots \geq  S_{r,r} > 0 \quad S_{i,j} = 0 \; \text{where} \; i \neq j

In the formula, **A** is the supplied ***m*** by ***n*** weighted matrix of term frequencies in a collection of text where ***m*** is the number of unique terms, and ***n*** is the number of documents. **T** is a computed ***m*** by ***r*** matrix of term vectors where ***r*** is the rank of **A**—a measure of its unique dimensions **≤ min(*m,n*)**. **S** is a computed ***r*** by ***r*** diagonal matrix of decreasing singular values, and **D** is a computed ***n*** by ***r*** matrix of document vectors.

The SVD is then truncated to reduce the rank by keeping only the largest ***k*** « ***r*** diagonal entries in the singular value matrix **S**, where ***k*** is typically on the order 100 to 300 dimensions. This effectively reduces the term and document vector matrix sizes to ***m*** by ***k*** and ***n*** by ***k*** respectively. The SVD operation, along with this reduction, has the effect of preserving the most important semantic information in the text while reducing noise and other undesirable artifacts of the original space of **A**. This reduced set of matrices is often denoted with a modified formula such as:

A ≈ A*k* = T*k* S*k* D*k*T

The computed **T*k*** and **D*k*** matrices define the term and document vector spaces, which with the computed singular values, **S*k***, embody the conceptual information derived from the document collection. The similarity of terms or documents within these spaces is a factor of how close they are to each other in these spaces, typically computed as a function of the angle between the corresponding vectors. Efficient LSI algorithms only compute the first ***k*** singular values and term and document vectors as opposed to computing a full SVD and then truncating it.

Note that this rank reduction is essentially the same as doing Principal Component Analysis (PCA) on the matrix **A**, except that PCA subtracts off the means. PCA loses the sparseness of the **A** matrix, which can make it infeasible for large lexicons.

The same steps are used to locate the vectors representing the text of queries and new documents within the document space of an existing LSI index. By a simple transformation of the **A = T S DT** equation into the equivalent **D = AT T S−1** equation, a new vector, ***d***, for a query or for a new document can be created by computing a new column in **A** and then multiplying the new column by **T S−1**. The new column in **A** is computed using the originally derived global term weights and applying the same local weighting function to the terms in the query or in the new document.

A drawback to computing vectors in this way, when adding new searchable documents, is that terms that were not known during the SVD phase for the original index are ignored. These terms will have no impact on the global weights and learned correlations derived from the original collection of text. However, the computed vectors for the new text are still very relevant for similarity comparisons with all other document vectors.

The process of augmenting the document vector spaces for an LSI index with new documents in this manner is called *folding in*. Although the folding-in process does not account for the new semantic content of the new text, adding a substantial number of documents in this way will still provide good results for queries as long as the terms and concepts they contain are well represented within the LSI index to which they are being added. When the terms and concepts of a new set of documents need to be included in an LSI index, either the term-document matrix,and the SVD, must be recomputed or an incremental update method (such as the one described can be used.

1. if the number of documents taken are equal three then the reduced matrix should by rank two reduction

2. if the number of documents taken are greater than three the reduced matrix rank should 44 percent of the given documents.

**Applying Dimensionality Reduction To The Given Example**

Implement a Rank 2 Approximation by keeping the first two columns of **U** and **V**

and the first two columns and rows of **S**.

U= -0.241 0.571

-0.241 0.571

-0.067 0.311

-0.296 -0.223

-0.592 -0.447

-0.470 0.037

-0.470 0.037

VT= -0.192 -0.846 -0.497

0.672 -0.483 0.562

S= 2.859 0.000

0.000 2.162

**5.2 IMPLEMENTATION OF KEY FUNCTION**

**5.2.1 Multiplication**

Now we have to multiple the matrices separately first to get the the values related to words we have to multiple U and S

UxS= -0.689 1.235

-0.689 1.235

-0.192 0.672

-0.846 -0.482

-1.693 -0.966

-1.344 0.080

-1.344 0.080

And the we have to multiple the S and V transpose matrix to get the values related to document

SxVT= -0.549 -2.419 -1.421

1.453 -1.044 1.215

Now we have to find out the median or the centroid values for the query matrix

So the final value of the query matrix is

Q= [-1.241 0.116]

And values of the words in the query is

Shipment= [-0687 1.235]

Gold= [-0.689 1.235]

Damaged= [-0.192 0.672]

Delivery= [-0.846 -0.482]

Silver= [-1.693 -0.966]

Arrived= [-1.344 0.080]

Truck= [-1.344 0.080]

D1= [-0.549 1.453]

D2= [-2.419 -1.044]

D3= [-1.421 1.215]

**5.2.2 Cosine Similarity**

Rank documents in decreasing order of query-document cosine similarities. By using the cosine similarity values

Cosine similarity value for (d1, q) =0.252

Cosine similarity value for (d2, q) =0.7441

Cosine similarity value for (d3, q) =0.544

We can see that document d2 scores higher than d3 and d1. Its vector is closer to the

query vector than the other vectors.

Now after this we have plot all the words and document and a cluster is formed and this cluster will be shown as the output to the closest cluster.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# **5.3 METHOD OF IMPLEMENTATION**

# **5.3.1 Stop Words removal**

**Code:**

import java.util.\*;

import java.io.\*;

public class FileMerge

{

public static void main(String args[]) throws IOException

{

Scanner keyboard = new Scanner(System.in);

System.out.print("Enter a Directory name: ");

String dirname = keyboard.nextLine();

File dir = new File(dirname);

String[] fileList = dir.list();

String[] a=new String[10000];

int i=0,n;

for(String name:fileList)

{

a[i]=name;

System.out.println(" "+a[i]);

i++;

}

n=i;

System.out.println("No.of Files:"+i);

String[] strLine=new String[10000];

for(int k=0;k<n;k++)

{

BufferedReader br=new BufferedReader( new FileReader("Input Docs/"+a[k]));

strLine[k] = br.readLine();

System.out.println("Doc "+(k+1)+":"+strLine[k]);

}

String[] Stop = {"a", "about", "above", "above", "across", "after", "afterwards", "again", "against", "all", "almost", "alone", "along", "already", "also", "although", "always", "am", "among", "amongst", "amoungst", "amount", "an", "and", "another", "any", "anyhow", "anyone", "anything", "anyway", "anywhere", "are", "around", "as", "at", "back", "be", "became", "because", "become", "becomes", "becoming", "been", "before", "beforehand", "behind", "being", "below", "beside", "besides", "between", "beyond", "bill", "both", "bottom", "but", "by", "call", "can", "cannot", "cant", "co", "con", "could", "couldnt", "cry", "de", "describe", "detail", "do", "done", "down", "due", "during", "each", "eg", "eight", "either", "eleven", "else", "elsewhere", "empty", "enough", "etc", "even", "ever", "every", "everyone", "everything", "everywhere", "except", "few", "fifteen", "fify", "fill", "find", "fire", "first", "five", "for", "former", "formerly", "forty", "found", "four", "from", "front", "full", "further", "get", "give", "go", "had", "has", "hasnt", "have", "he", "hence", "her", "here", "hereafter", "hereby", "herein", "hereupon", "hers", "herself", "him", "himself", "his", "how", "however", "hundred", "ie", "if", "in", "inc", "indeed", "interest", "into", "is", "it", "its","did","know","itself", "keep", "last", "latter", "latterly", "least", "less", "ltd", "made", "many", "may", "me", "meanwhile", "might", "mill", "mine", "more", "moreover", "most", "mostly", "move", "much", "must", "my", "myself", "name", "namely", "neither", "never", "nevertheless", "next", "nine", "no", "nobody", "none", "noone", "nor", "not", "nothing", "now", "nowhere", "of", "off", "often", "on", "once", "one", "only", "onto", "or", "other", "others", "otherwise", "our", "ours", "ourselves", "out", "over", "own", "part", "per", "perhaps", "please", "put", "rather", "re", "same", "see", "seem", "seemed", "seeming", "seems", "serious", "several", "she", "should", "show", "side", "since", "sincere", "six", "sixty", "so", "some", "somehow", "someone", "something", "sometime", "sometimes", "somewhere", "still", "such", "system", "take", "ten", "than", "that", "the", "their", "them", "themselves", "then", "thence", "there", "thereafter", "thereby", "therefore", "therein", "thereupon", "these", "they", "thickv", "thin", "third", "this", "those", "though", "three", "through", "throughout", "thru", "thus", "to", "together", "too", "top", "toward", "towards", "twelve", "twenty", "two", "un", "under", "until", "up", "upon", "us", "very", "via", "was", "we", "well", "were", "what", "whatever", "when", "whence", "whenever", "where", "whereafter", "whereas", "whereby", "wherein", "whereupon", "wherever", "whether", "which", "while", "whither", "who", "whoever", "whole", "whom", "whose", "why", "will", "with", "within", "without", "would", "yet", "you", "your", "yours", "yourself", "yourselves", "z", "zero","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q","r","s","t","w","x","y","z",":",",","'"};

String[] arr = new String[10000];

int j=0,m,k,l,x=0;

l=Stop.length;

for(k=0;k<n;k++)

{

String line = strLine[k];

line = line.trim();

File file = new File(a[k]);

if (!file.exists())

{

file.createNewFile();

}

FileWriter fw = new FileWriter(file.getAbsoluteFile());

BufferedWriter bw = new BufferedWriter(fw);

i=0;

for (String retval: line.split(" "))

{

arr[i]=retval;

arr[i]=arr[i].toLowerCase();

System.out.println(arr[i]);

for(j=0;j<l;j++)

{

if(arr[i].equals(Stop[j]))

{

arr[i]="";

}

}

for(j=0;j<l;j++)

{

if(arr[i].equals(":"))

{

arr[i]="";

}

}

if(arr[i]!="")

{

bw.write(arr[i]);

bw.write(" ");

}

i++;

}

bw.close();

}

}

}

# **5.3.2 Key word generation**

**Code:**

import java.util.\*;

import java.io.\*;

public class KeyWord

{

public static void main(String args[]) throws IOException

{

Scanner keyboard = new Scanner(System.in);

System.out.print("Enter a Directory name: ");

String dirname = keyboard.nextLine();

File dir = new File(dirname);

String[] fileList = dir.list();

String[] a=new String[10000];

int i=0,n;

for(String name:fileList)

{

a[i]=name;

System.out.println(" "+a[i]);

i++;

}

n=i;

System.out.println("No.of Files:"+i);

String[] strLine=new String[10000];

for(int k=0;k<n;k++)

{

BufferedReader br=new BufferedReader( new FileReader("Input Docs/"+a[k]));

strLine[k] = br.readLine();

System.out.println("Doc "+(k+1)+":"+strLine[k]);

}

String[] arr = new String[10000];

int j=0,m,k;

i=0;

for(k=0;k<n;k++)

{

String line = strLine[k];

line = line.trim();

for (String retval: line.split(" "))

{

arr[i]=retval;

arr[i]=arr[i].toLowerCase();

//System.out.println(arr[i]);

i++;

}

}

m=i;

for(i=0;i<m;i++)

{

for(j=i+1;j<m;j++)

{

if(arr[i].equals(arr[j]))

{

arr[j]="";

}

}

}

try

{

File file = new File("Words.txt");

if (!file.exists())

{

file.createNewFile();

}

FileWriter fw = new FileWriter(file.getAbsoluteFile());

BufferedWriter bw = new BufferedWriter(fw);

for(i=0;i<m;i++)

{

if(!(arr[i].equals("")))

{

if(!(arr[i].equals(".")))

{

bw.write(arr[i]);

bw.write(" ");

}

}

}

bw.close();

}

catch (IOException e)

{

e.printStackTrace();

}

}

}

**5.3.3 Data Matrix**

import java.io.\*;

public class Matrix1

{

public static void main(String args[]) throws IOException

{

Scanner keyboard = new Scanner(System.in);

System.out.print("Enter a Directory name: ");

String dirname = keyboard.nextLine();

File dir = new File(dirname);

String[] fileList = dir.list();

String[] a=new String[10000];

int i=0,n;

for(String name:fileList)

{

a[i]=name;

System.out.println(" "+a[i]);

i++;

}

n=i;

System.out.println("No.of Files:"+i);

String[] strLine=new String[10000];

for(int k=0;k<n;k++)

{

BufferedReader br=new BufferedReader( new FileReader("Input Docs/"+a[k]));

strLine[k] = br.readLine();

System.out.println("Doc "+(k+1)+":"+strLine[k]);

}

Scanner keyboard1 = new Scanner(System.in);

System.out.print("Enter KeyWord file name: ");

String filename = keyboard1.nextLine();

File file1 = new File(filename);

Scanner inputFile = new Scanner(file1);

String[] arr = new String[10000];

String[] arr1 = new String[10000];

int j=0,m,k;

i=0;

while (inputFile.hasNext())

{

String line = inputFile.nextLine();

line = line.trim();

for (String retval: line.split(" "))

{

arr1[i]=retval;

arr1[i]=arr1[i].toLowerCase();

//System.out.println(arr[i]);

i++;

}

}

System.out.println("Array Size:"+i);

m=i;

int l;

int[] f=new int[1000];

int[] f1=new int[1000];

i=0;

for(k=0;k<n;k++)

{

String line = strLine[k];

line = line.trim();

f[k]=i;

for (String retval: line.split(" "))

{

arr[i]=retval;

arr[i]=arr[i].toLowerCase();

i++;

}

f1[k]=i;

}

l=0;

int[] b=new int[1000];

for(j=0;j<m;j++)

{

for(k=0;k<n;k++)

{

for(i=f[k];i<f1[k];i++)

{

if(arr1[j].equals(arr[i]))

{

b[l]++;

}

}

l++;

}

}

try

{

File file = new File("Matrix.txt");

if (!file.exists())

{

file.createNewFile();

}

FileWriter fw = new FileWriter(file.getAbsoluteFile());

BufferedWriter bw = new BufferedWriter(fw);

//bw.write("\t"+"Keyword"+"\t");

//for(int x=0;x<n;x++)

//bw.write("\t"+"Doc"+(x+1));

//System.out.println();

l=0;

for(j=0;j<m;j++)

{

//bw.write("\t"+arr1[j]+"\t");

bw.write("\t"+b[l]+"\t"+b[l+1]+"\t"+b[l+2]);

bw.write("\n");

l=l+3;

}

bw.close();

}

catch (IOException e)

{

e.printStackTrace();

}

}

**5.3.4 Matrix Multiplication**

import java.util.Scanner;

class MatrixMultiplication {

public static void main(String[] args) throws Exception

{

Scanner s = new Scanner(System.in);

System.out.print("Enter number of rows in A: ");

double rowsInA = s.nextDouble();

System.out.print("Enter number of columns in A / rows in B: ");

double columnsInA = s.nextDouble();

System.out.print("Enter number of columns in B: ");

double columnsInB = s.nextDouble();

double[][] a = new double[rowsInA][columnsInA];

double[][] b = new double[columnsInA][columnsInB];

System.out.println("Enter matrix A");

for (int i = 0; i < a.length; i++) {

for (int j = 0; j < a[0].length; j++) {

a[i][j] = s.nextDouble();

}

}

System.out.println("Enter matrix B");

for (int i = 0; i < b.length; i++) {

for (int j = 0; j < b[0].length; j++) {

b[i][j] = s.nextDouble();

}

}

double[][] c = multiply(a, b);

System.out.println("Product of A and B is");

for (int i = 0; i < c.length; i++) {

for (int j = 0; j < c[0].length; j++) {

System.out.print(c[i][j] + " ");

}

System.out.println();

}

}

public static double[][] multiply(double[][] a, double[][] b) {

double rowsInA = a.length;

double columnsInA = a[0].length; // same as rows in B

double columnsInB = b[0].length;

double[][] c = new double[rowsInA][columnsInB];

for (int i = 0; i < rowsInA; i++) {

for (int j = 0; j < columnsInB; j++) {

for (int k = 0; k < columnsInA; k++) {

c[i][j] = c[i][j] + a[i][k] \* b[k][j];

}

}

}

return c;

}

}

**5.3.5 COSINE SIMILARITY**

import java.lang.Math;

import java.util.Scanner;

class cosinesimilarity {

public static void main (String [] args)throws Exception

{

Scanner s = new Scanner(System.in);

int n=s.nextInt();

double[] a = new double[n];

double[] b = new double[n];

System.out.println("enter the values of a:");

for(int i=0;i<a.length;i++)

{

a[i]=s.nextDouble();

}

System.out.println("enter the values of b:");

for(int i=0;i<b.length;i++)

{

b[i]=s.nextDouble();

}

double sum = 0;

for (int i = 0; i < a.length; i++)

{

sum += a[i] \* b[i];

}

float sum1=0;

float sum2=0;

float sum3;

System.out.println(sum);

for(int i=0;i<a.length;i++)

{

sum1+=a[i]\*a[i];

sum2+=b[i]\*b[i];

}

double d;

d=Math.sqrt(sum1)+Math.sqrt(sum2);

double c;

c=sum/d;

System.out.println("the cosinesimilarity value:"+c);

}

}

**5.3.6 SCREENSHOTS**

**Document Term Matrix**

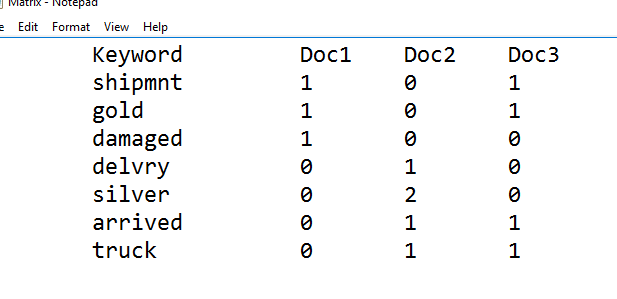
****

Fig 5.1 Document Term Matrix

**SINGULAR VALUED DECOMPOSTION**

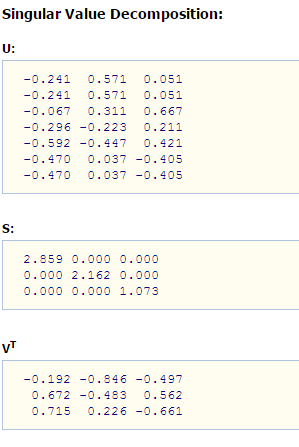
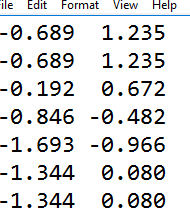
****

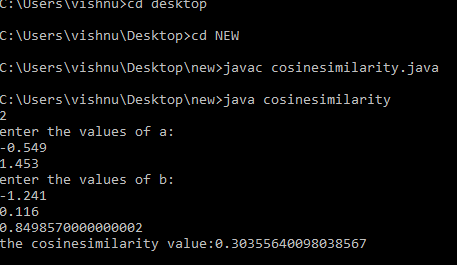
Fig 5.2 Singular Valued Decomposition

**MATRIX MULTIPLICATION**

****

5.3 Matrix Multiplication

**COSINE SIMILARITY**

****

5.4 Cosine Similarity

**5.3.7 OUTPUT**

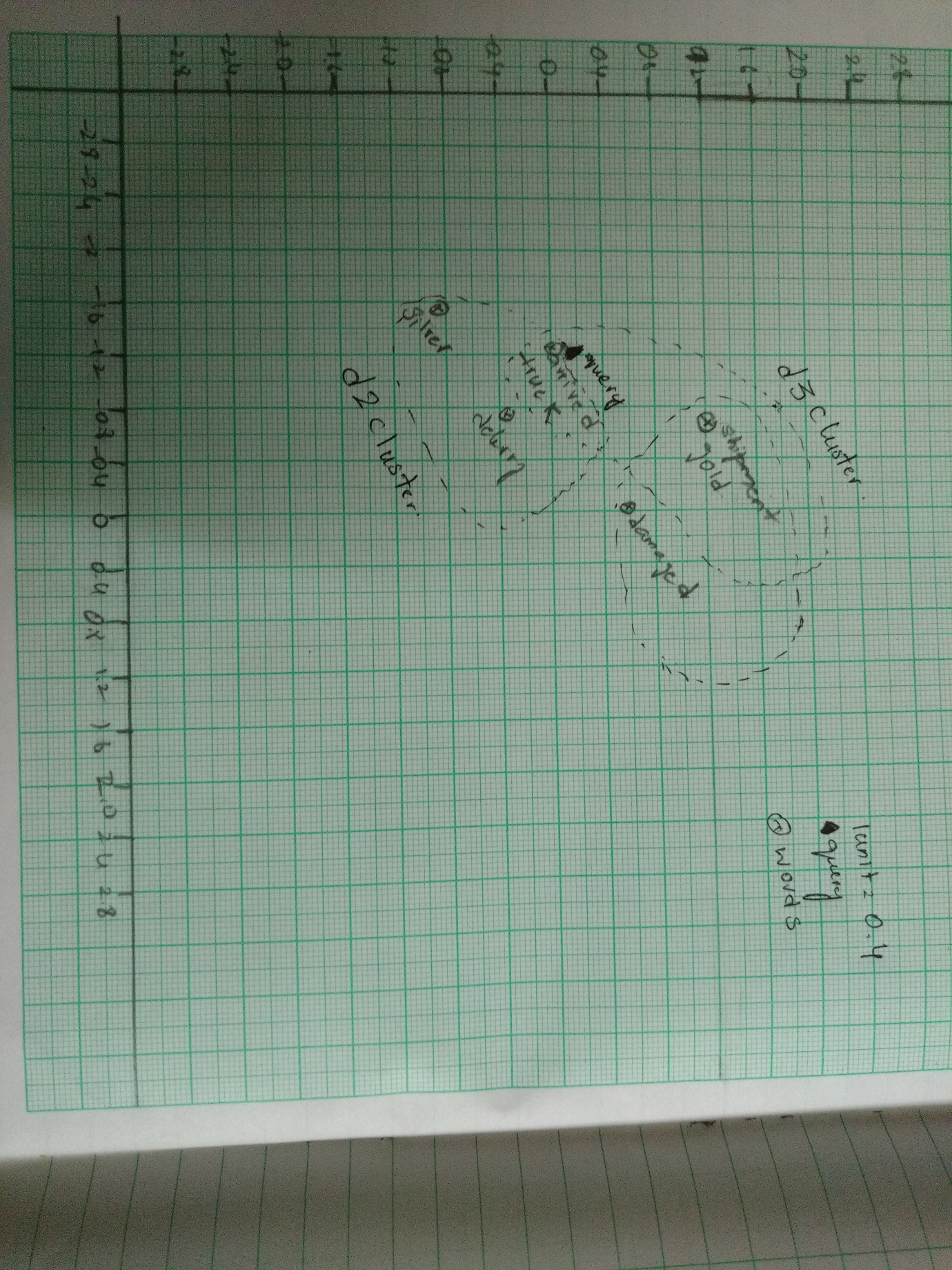
****

FIG 5.5 Result

**6. CONCLUSION**

Dumais (1993) and Dumais (1995) conducted experiments with LSI on TREC documents and tasks, using the commonly-used Lanczos algorithm to compute the SVD. At the time of their work in the early 1990's, the LSI computation on tens of thousands of documents took approximately a day on one machine. On these experiments, they achieved precision at or above that of the median TREC participant. On about 20% of TREC topics their system was the top scorer, and reportedly slightly better on average than standard vector spaces for LSI at about 350 dimensions. Here are some conclusions on LSI first suggested by their work, and subsequently verified by many other experiments.

* The computational cost of the SVD is significant; at the time of this writing, we know of no successful experiment with over one million documents. This has been the biggest obstacle to the widespread adoption to LSI. One approach to this obstacle is to build the LSI representation on a randomly sampled subset of the documents in the collection, following which the remaining documents are ``folded in'' as detailed with Equation
* As we reduce $k$, recall tends to increase, as expected.
* Most surprisingly, a value of $k$ in the low hundreds can actually *increase* precision some query benchmarks. This appears to suggest that for a suitable value of $k$, LSI addresses some of the challenges of synonymy.
* LSI works best in applications where there is little overlap between queries and documents.

The experiments also documented some modes where LSI failed to match the effectiveness of more traditional indexes and score computations. Most notably (and perhaps obviously), LSI shares two basic drawbacks of vector space retrieval: there is no good way of expressing negations (find documents that contain german but not shepherd), and no way of enforcing Boolean conditions. LSI can be viewed as *soft clustering* by interpreting each dimension of the reduced space as a cluster and the value that a document has on that dimension as its fractional membership in that cluster.

**REFERENCES**

* **http://cecs.wright.edu/~tkprasad/courses/cs707/L08VSM-tfidf.ppt**
* **www.bluebit.gr/matrix-calculator**
* **https://en.wikipedia.org/wiki/Latent\_semantic\_indexing**
* **www.cecs.wright.edu/~tkprasad/courses/cs707/L18LSI**[**.**](http://www.cecs.wright.edu/~tkprasad/courses/cs707/L18LSI.ppt)**ppt**
* **http://nlp.stanford.edu/IR-book/pdf/18lsi.pdf**