LATENT SEMANTIC INDEXING

Taking a Holistic View

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.

Ignorance is Bliss

We mentioned the difficulty of teaching a computer to organize data into concepts and demonstrate understanding. One great advantage of LSI is that it is a strictly mathematical approach, with no insight into the meaning of the documents or words it analyzes. This makes it a powerful, generic technique able to index any cohesive document collection in any language. It can be used in conjunction with a regular keyword search, or in place of one, with good results.

Before we discuss the theoretical underpinnings of LSI, it's worth citing a few actual searches from some sample document collections. In each search, a red title or astrisk indicates that the document doesn't contain the search string, while a blue title or astrisk informs the viewer that the search string is present.

- In an AP news wire database, a search for `Saddam Hussein` returns articles on the Gulf War, UN sanctions, the oil embargo, and documents on Iraq that do not contain the Iraqi president's name at all.
- Looking for articles about `Tiger Woods` in the same database brings up many stories about the golfer, followed by articles about major golf tournaments that don't mention his name. Constraining the search to days when no articles were written about Tiger Woods still brings up stories about golf tournaments and well-known players.
- In an image database that uses LSI indexing, a search on `Normandy invasion` shows images of the Bayeux tapestry - the famous tapestry depicting the Norman invasion of England in 1066, the town of Bayeux, followed by photographs of the English invasion of Normandy in 1944.

In all these cases LSI is 'smart' enough to see that `Saddam Hussein` is somehow closely related to `Iraq` and the `Gulf War`, that `Tiger Woods` plays `golf`, and that `Bayeux` has close semantic ties to `invasions` and `England`. As we will see in our exposition, all of these apparently intelligent connections are artifacts of word use patterns that already exist in our document collection.

HOW LSI WORKS

The Search for Content

We mentioned that latent semantic indexing looks at patterns of word distribution (specifically, word **co-occurence**) 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 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.

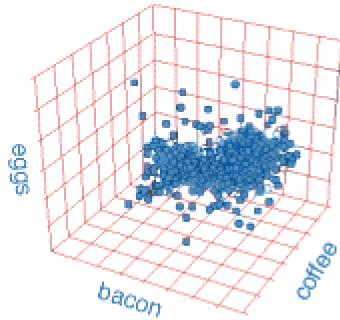
Notice that our arrangement is binary - a square in our grid either contains an X, or it doesn't. This big grid is the visual equivalent of a generic keyword search, which looks for exact matches between documents and keywords. If we replace blanks and X-es with zeroes and ones, we get a numerical **matrix** 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 (a rigorous treatment is available here), but we can get an intuitive grasp of what SVD does by thinking of the process spatially. An analogy will help.

Breakfast in Hyperspace

Imagine that you are curious about what people typically order for breakfast down at your local diner, and you want to display this information in visual form. You decide to examine all the breakfast orders from a busy weekend day, and record how many times the words `bacon`, `eggs` and `coffee` occur in each order.

You can graph the results of your survey by setting up a chart with three orthogonal axes - one for each keyword. The choice of direction is arbitrary - perhaps a `bacon` axis in the x direction, an `eggs` axis in the y direction, and the all-important `coffee` axis in the z direction. To plot a particular breakfast order, you count the occurence of each keyword, and then take the appropriate number of steps along the axis for that word. When you are finished, you get a cloud of points in three-dimensional space, representing all of that day's breakfast orders.



If you draw a line from the origin of the graph to each of these points, you obtain a set of **vectors** in 'bacon-eggs-and-coffee' space. The size and direction of each vector tells you how many of the three key items were in any particular order, and the set of all the vectors taken together tells you something about the kind of breakfast people favor on a Saturday morning.

What your graph shows is called a **term space**. Each breakfast order forms a vector in that space, with its direction and magnitude determined by how many times the three keywords appear in it. Each keyword corresponds to a separate spatial direction, perpendicular to all the others. Because our example uses three keywords, the resulting term space has three dimensions, making it possible for us to visualize it. It is easy to see that this space could have any number of dimensions, depending on how many keywords we chose to use. If we were to go back through the orders and also record occurences of `sausage`, `muffin`, and `bagel`, we would end up with a six-dimensional term space, and six-dimensional document vectors.

Applying this procedure to a real document collection, where we note each use of a content word, results in a term space with many thousands of dimensions. Each document in our collection is a vector with as many components as there are content words. Although we can't possibly visualize such a space, it is built in the exact same way as the whimsical breakfast space we just described. Documents in such a space that have many words in common will have vectors that are near to each other, while documents with few shared words will have vectors that are far apart.

Latent semantic indexing works by projecting this large, multidimensional space down into a smaller number of dimensions. In doing so, keywords that are semantically similar will get squeezed together, and will no longer be completely distinct. This blurring of boundaries is what allows LSI to go beyond straight keyword matching. To understand how it takes place, we can use another analogy.

Singular Value Decomposition

Imagine you keep tropical fish, and are proud of your prize aquarium - so proud that you want to submit a picture of it to *Modern Aquaria* magazine, for fame and profit. To get the best possible picture, you will want to choose a good angle from which to take the photo. You want to make sure that as many of the fish as possible are visible in your picture, without being hidden by other fish in the foreground. You also won't want the fish all bunched together in a clump, but rather shot from an angle that shows them nicely distributed in the water. Since your tank is transparent on all sides, you can take a variety of pictures from above, below, and from all around the aquarium, and select the best one.

In mathematical terms, you are looking for an optimal mapping of points in 3-space (the fish) onto a plane (the film in your camera). 'Optimal' can mean many things - in this case it means 'aesthetically pleasing'. But now imagine that your goal is to preserve the relative distance between the fish as much as possible, so that fish on opposite sides of the tank don't get superimposed in the photograph to look like they are right next to each other. Here you would be doing exactly what the SVD algorithm tries to do with a much higher-dimensional space.

Instead of mapping 3-space to 2-space, however, the SVD algorithm goes to much greater extremes. A typical term space might have tens of thousands of dimensions, and be projected down into fewer than 150. Nevertheless, the principle is exactly the same. The SVD algorithm preserves as much information as possible about the relative distances between the document vectors, while collapsing them down into a much smaller set of dimensions. In this collapse, information is lost, and content words are superimposed on one another.

Information loss sounds like a bad thing, but here it is a blessing. What we are losing is noise from our original term-document matrix, revealing similarities that were latent in the document

collection. Similar things become more similar, while dissimilar things remain distinct. This reductive mapping is what gives LSI its seemingly intelligent behavior of being able to correlate semantically related terms. We are really exploiting a property of natural language, namely that words with similar meaning tend to occur together.

LSI EXAMPLE - INDEXING A DOCUMENT

Putting Theory into Practice
While a discussion of the mathematics behind singular value decomposition is beyond the scope of our article, it's worthwhile to follow the process of creating a term-document matrix in some detail, to get a feel for what goes on behind the scenes. Here we will process a sample wire story to demonstrate how real-life texts get converted into the numerical representation we use as input for our SVD algorithm.
The first step in the chain is obtaining a set of documents in electronic form. This can be the hardest thing about LSI - there are all too many interesting collections not yet available online. In our experimental database, we download wire stories from an online newspaper with an AP news feed. A script downloads each day's news stories to a local disk, where they are stored as text files.
Let's imagine we have downloaded the following sample wire story, and want to incorporate it in our collection:

O'Neill Criticizes Europe on Grants
PITTSBURGH (AP)

Treasury Secretary Paul O'Neill expressed irritation Wednesday that European countries have refused to go along with a U.S. proposal to boost the amount of direct grants rich nations offer poor countries. The Bush administration is pushing a plan to increase the amount of direct grants the World Bank provides the poorest nations to 50 percent of assistance, reducing use of loans to these nations.

The first thing we do is strip all formatting from the article, including capitalization, punctuation, and extraneous markup (like the dateline). LSI pays no attention to word order, formatting, or capitalization, so can safely discard that information. Our cleaned-up wire story looks like this:

o'neill criticizes europe on grants treasury secretary paul o'neill expressed irritation wednesday that european countries have refused to go along with a us proposal to boost the amount of direct grants rich nations offer poor countries the bush administration is pushing a plan to increase the amount of direct grants the world bank provides the poorest nations to 50 percent of assistance reducing use of loans to these nations

The next thing we want to do is pick out the content words in our article. These are the words we consider semantically significant - everything else is clutter. We do this by applying a **stop list** of commonly used English words that don't carry semantic meaning. Using a stop list greatly reduces the amount of noise in our collection, as well as eliminating a large number of words that would make the computation more difficult. Creating a stop list is something of an art - they depend very much on the nature of the data collection. You can see our full wire stories stop list here.
Here is our sample story with stop-list words highlighted:

o'neill criticizes europe on grants treasury secretary paul o'neill expressed irritation wednesday that european countries have refused to go along with a US proposal to boost the amount of direct grants rich nations offer poor countries the bush administration is pushing a plan to increase the amount of direct grants the world bank provides the poorest nations to 50 percent of assistance reducing use of loans to these nations

Removing these stop words leaves us with an abbreviated version of the article containing content words only:

```
o'neill criticizes europe grants treasury secretary
paul o'neill expressed irritation european countries
refused US proposal boost direct grants rich nations
poor countries bush administration pushing plan
increase amount direct grants world bank poorest
nations assistance loans nations
```

However, one more important step remains before our document is ready for indexing. Notice how many of our content words are plural nouns (`grants`, `nations`) and inflected verbs (`pushing`, `refused`). It doesn't seem very useful to have each inflected form of a content word be listed seperately in our master word list - with all the possible variants, the list would soon grow unwieldy. More troubling is that LSI might not recognize that the different variant forms were actually the same word in disguise. We solve this problem by using a **stemmer**.

Stemming

While LSI itself knows nothing about language (we saw how it deals exclusively with a mathematical vector space), some of the preparatory work needed to get documents ready for indexing is very language-specific. We have already seen the need for a stop list, which will vary entirely from language to language and to a lesser extent from document collection to document collection. Stemming is similarly language-specific, derived from the morphology of the language. For English documents, we use an algorithm called the **Porter stemmer** to remove common endings from words, leaving behind an invariant root form. Here are some examples of words before and after stemming:

information -> inform

presidency -> presid

presiding -> presid

happiness -> happi

happily -> happi

discouragement -> discourag

battles -> battl

And here is our sample story as it appears to the stemmer:

```
o'neill criticizes europe grants treasury secretary
paul o'neill expressed irritation european countries
refused US proposal boost direct grants rich nations
poor countries
bush administration pushing plan increase amount
direct grants world bank poorest nations assistance
loans nations
```

Note that at this point we have reduced the original natural-language news story to a series of word stems. All of the information carried by punctuation, grammar, and style is gone - all that remains is word order, and we will be doing away with even that by transforming our text into a word list. It is striking that so much of the meaning of text passages inheres in the number and choice of content words, and relatively little in the way they are arranged. This is very counterintuitive, considering how important grammar and writing style are to human perceptions of writing.

Having stripped, pruned, and stemmed our text, we are left with a flat list of words:

```
administrat
amount
assist
bank
boost
bush
countri (2)
direct
europ
express
grant (2)
increas
irritat
loan
nation (3)
```

```
o'neill
paul
plan
poor (2)
propos
push
refus
rich
secretar
treasuri
US
world
```

This is the information we will use to generate our term-document matrix, along with a similar word list for every document in our collection.

THE TERM-DOCUMENT MATRIX

Doing the Numbers

As we mentioned in our discussion of LSI, the **term-document matrix** is a large grid representing every document and content word in a collection. We have looked in detail at how a document is converted from its original form into a flat list of content words. We prepare a master word list by generating a similar set of words for every document in our collection, and discarding any content words that either appear in every document (such words won't let us discriminate between documents) or in only one document (such words tell us nothing about relationships across documents). With this master word list in hand, we are ready to build our TDM.

We generate our TDM by arranging our list of all content words along the vertical axis, and a similar list of all documents along the horizontal axis. These need not be in any particular order, as long as we keep track of which column and row corresponds to which keyword and document. For clarity we will show the keywords as an alphabetized list.

We fill in the TDM by going through every document and marking the grid square for all the content words that appear in it. Because any one document will contain only a tiny subset of our content word vocabulary, our matrix is very sparse (that is, it consists almost entirely of zeroes).

Here is a fragment of the actual term-document marix from our wire stories database:

Document:	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**	**q**	**r**	{ 3000 more columns }
aa	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
amotd	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
aaliyah	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
aarp	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	...
ab	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
...																			
zywicki	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	...

We can easily see if a given word appears in a given document by looking at the intersection of the appropriate row and column. In this sample matrix, we have used ones to represent document/keyword pairs. With such a binary scheme, all we can tell about any given document/keyword combination is whether the keyword appears in the document.

This approach will give acceptable results, but we can significantly improve our results by applying a kind of linguistic favoritism called **term weighting** to the value we use for each non-zero term/document pair.

Not all Words are Created Equal

Term weighting is a formalization of two common-sense insights:

1. Content words that appear several times in a document are probably more meaningful than content words that appear just once.
2. Infrequently used words are likely to be more interesting than common words.

The first of these insights applies to individual documents, and we refer to it as **local weighting**. Words that appear multiple times in a document are given a greater local weight than words that appear once. We use a formula called **logarithmic local weighting** to generate our actual value.

The second insight applies to the set of all documents in our collection, and is called **global term weighting**. There are many global weighting schemes; all of them reflect the fact that words that appear in a small handful of documents are likely to be more significant than words that are distributed widely across our document collection. Our own indexing system uses a scheme called **inverse document frequency** to calculate global weights.

By way of illustration, here are some sample words from our collection, with the number of documents they appear in, and their corresponding global weights.

word	count	global weight

unit	833	1.44
cost	295	2.47
project	169	3.03
tackle	40	4.47
wrestler	7	6.22

You can see that a word like `wrestler`, which appears in only seven documents, is considered twice as significant as a word like `project`, which appears in over a hundred.

There is a third and final step to weighting, called **normalization**. This is a scaling step designed to keep large documents with many keywords from overwhelming smaller documents in our result set. It is similar to handicapping in golf - smaller documents are given more importance, and larger documents are penalized, so that every document has equal significance.

These three values multiplied together - local weight, global weight, and normalization factor - determine the actual numerical value that appears in each non-zero position of our term/document matrix.

Although this step may appear language-specific, note that we are only looking at word frequencies within our collection. Unlike the stop list or stemmer, we don't need any outside source of linguistic information to calculate the various weights. While weighting isn't critical to understanding or implementing LSI, it does lead to much better results, as it takes into account the relative importance of potential search terms.

The Moment of Truth

With the weighting step done, we have done everything we need to construct a finished term-document matrix. The final step will be to run the SVD algorithm itself. Notice that this critical step will be purely mathematical - although we know that the matrix and its contents are a shorthand for certain linguistic features of our collection, the algorithm doesn't know anything about what the numbers mean. This is why we say LSI is language-agnostic - as long as you can perform the steps needed to generate a term-document matrix from your data collection, it can be in any language or format whatsoever.

You may be wondering what the large matrix of numbers we have created has to do with the term vectors and many-dimensional spaces we discussed in our earlier explanation of how LSI works. In fact, our matrix is a convenient way to represent vectors in a high-dimensional space. While we have been thinking of it as a lookup grid that shows us which terms appear in which documents, we can also think of it in spatial terms. In this interpretation, every column is a long list of coordinates that gives us the exact position of one document in a many-dimensional term space. When we applied term weighting to our matrix in the previous step, we nudged those coordinates around to make the document's position more accurate.

As the name suggests, singular value decomposition breaks our matrix down into a set of smaller components. The algorithm alters one of these components (this is where the number of dimensions gets reduced), and then recombines them into a matrix of the same shape as our original, so we can again use it as a lookup grid. The matrix we get back is an approximation of the term-document matrix we provided as input, and looks much different from the original:

	a	b	c	d	e	f	g	h	i	j	k	
aa	-0.0006	-0.0006	0.0002	0.0003	0.0001	0.0000	0.0000	-0.0001	0.0007	0.0001	0.0004	...
amotd	-0.0112	-0.0112	-0.0027	-0.0008	-0.0014	0.0001	-0.0010	0.0004	-0.0010	-0.0015	0.0012	...
aaliyah	-0.0044	-0.0044	-0.0031	-0.0008	-0.0019	0.0027	0.0004	0.0014	-0.0004	-0.0016	0.0012	...
aarp	0.0007	0.0007	0.0004	0.0008	-0.0001	-0.0003	0.0005	0.0004	0.0001	0.0025	0.0000	...
ab	-0.0038	-0.0038	0.0027	0.0024	0.0036	-0.0022	0.0013	-0.0041	0.0010	0.0019	0.0026	...
...												
zywicki	-0.0057	0.0020	0.0039	-0.0078	-0.0018	0.0017	0.0043	-0.0014	0.0050	-0.0020	-0.0011	...

Notice two interesting features in the processed data:

- The matrix contains far fewer zero values. Each document has a similarity value for most content words.
- Some of the similarity values are negative. In our original TDM, this would correspond to a document with fewer than zero occurences of a word, an impossibility. In the processed matrix, a negative value is indicative of a very large semantic distance between a term and a document.

This finished matrix is what we use to actually search our collection. Given one or more terms in a search query, we look up the values for each search term/document combination, calculate a cumulative score for every document, and rank the documents by that score, which is a measure of their similarity to the search query. In practice, we will probably assign an empirically-determined **threshold value** to serve as a cutoff between relevant and irrelevant documents, so that the query does not return every document in our collection.

The Big Picture

Now that we have looked at the details of latent semantic indexing, it is instructive to step back and examine some real-life applications of LSI. Many of these go far beyond plain search, and can assume some surprising and novel guises. Nevertheless, the underlying techniques will be the same as the ones we have outlined here.