Latent Semantic Analysis (LSA), also known as Latent Semantic Indexing (LSI) literally means analyzing documents to find the underlying meaning or concepts of those documents. If each word only meant one concept, and each concept was only described by one word, then LSA would be easy since there is a simple mapping from words to concepts.

Unfortunately, this problem is difficult because English has different words that mean the same thing (synonyms), words with multiple meanings, and all sorts of ambiguities that obscure the concepts to the point where even people can have a hard time understanding.

For example, the word bank when used together with mortgage, loans, and rates probably means a financial institution. However, the word bank when used together with lures, casting, and fish probably means a stream or river bank.
How Latent Semantic Analysis Works

Latent Semantic Analysis arose from the problem of how to find relevant documents from search words. The fundamental difficulty arises when we compare words to find relevant documents, because what we really want to do is compare the meanings or concepts behind the words. LSA attempts to solve this problem by mapping both words and documents into a "concept" space and doing the comparison in this space.

Since authors have a wide choice of words available when they write, the concepts can be obscured due to different word choices from different authors. This essentially random choice of words introduces noise into the word-concept relationship. Latent Semantic Analysis filters out some of this noise and also attempts to find the smallest set of concepts that spans all the documents.

In order to make this difficult problem solvable, LSA introduces some dramatic simplifications.

1. Documents are represented as "bags of words", where the order of the words in a document is not important, only how many times each word appears in a document.
2. Concepts are represented as patterns of words that usually appear together in documents. For example "leash", "treat", and "obey" might usually appear in documents about dog training.
3. Words are assumed to have only one meaning. This is clearly not the case (banks could be river banks or financial banks) but it makes the problem tractable.

To see a small example of LSA, take a look at the next section.

A Small Example

As a small example, I searched for books using the word “investing" at Amazon.com and took
the top 10 book titles that appeared. One of these titles was dropped because it had only one index word in common with the other titles. An index word is any word that:

- appears in 2 or more titles, and
- is not a very common word such as “and”, “the”, and so on (known as stop words). These words are not included because they do not contribute much (if any) meaning.

In this example we have removed the following stop words: “and”, “edition”, “for”, “in”, “little”, “of”, “the”, “to”.

Here are the 9 remaining titles. The index words (words that appear in 2 or more titles and are not stop words) are underlined.

1. The Neatest Little Guide to Stock Market Investing
2. Investing For Dummies, 4th Edition
3. The Little Book of Common Sense Investing: The Only Way to Guarantee Your Fair Share of Stock Market Returns
4. The Little Book of Value Investing
5. Value Investing: From Graham to Buffett and Beyond
7. Investing in Real Estate, 5th Edition
8. Stock Investing For Dummies

Once Latent Semantic Analysis has been run on this example, we can plot the index words and titles on an XY graph and identify clusters of titles. The 9 titles are plotted with blue circles and the 11 index words are plotted with red squares. Not only can we spot clusters of titles, but since index words can be plotted along with titles, we can label the clusters. For example, the blue cluster, containing titles T7 and T9, is about real estate. The green cluster, with titles T2, T4, T5, and T8, is about value investing, and finally the red cluster, with titles T1 and T3, is about the stock market. The T6 title is an outlier, off on its own.
Latent Semantic Analysis (LSA) Tutorial

In the next few sections, we'll go through all steps needed to run Latent Semantic Analysis on this example.

**Part 1 - Creating the Count Matrix**

The first step in Latent Semantic Analysis is to create the word by title (or document) matrix. In this matrix, each index word is a row and each title is a column. Each cell contains the number of times that word occurs in that title. For example, the word “book” appears one time in title T3 and one time in title T4, whereas “investing” appears one time in every title. In general, the matrices built during LSA tend to be very large, but also very sparse (most cells contain 0). That is because each title or document usually contains only a small number of all the possible words. This sparseness can be taken advantage of in both memory and time by more sophisticated LSA implementations.

In the following matrix, we have left out the 0’s to reduce clutter.

<table>
<thead>
<tr>
<th>Index Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>book</td>
</tr>
<tr>
<td>dads</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Titles</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

- XY Plot of Words and Titles
Download the python code [here](#).

Throughout this article, we'll give Python code that implements all the steps necessary for doing Latent Semantic Analysis. We'll go through the code section by section and explain everything. The Python code used in this article can be downloaded [here](#) and then run in Python. You need to have already installed the Python NumPy and SciPy libraries.

**Python - Import Functions**

First we need to import a few functions from Python libraries to handle some of the math we need to do. NumPy is the Python numerical library, and we'll import zeros, a function that creates a matrix of zeros that we use when building our words by titles matrix. From the linear algebra part of the scientific package (scipy.linalg) we import the svd function that actually does the singular value decomposition, which is the heart of LSA.

```python
from numpy import zeros
from scipy.linalg import svd
```
**Python - Define Data**

Next, we define the data that we are using. Titles holds the 9 book titles that we have gathered, stopwords holds the 8 common words that we are going to ignore when we count the words in each title, and ignorechars has all the punctuation characters that we will remove from words. We use Python's triple quoted strings, so there are actually only 4 punctuation symbols we are removing: comma (,), colon (:), apostrophe ('), and exclamation point (!).

```python
titles = [
    "Investing For Dummies, 4th Edition",
    "The Little Book of Common Sense Investing: The Only Way to Guarantee Your Fair Share of Stock Market Returns",
    "The Little Book of Value Investing",
    "Value Investing: From Graham to Buffett and Beyond",
    "Rich Dad's Guide to Investing: What the Rich Invest in, That the Poor and the Middle Class Do Not!",
    "Investing in Real Estate, 5th Edition",
    "Stock Investing For Dummies",
    "Rich Dad's Advisors: The ABC's of Real Estate Investing: The Secrets of Finding Hidden Profits Most Investors Miss"
]
stopwords = ['and', 'edition', 'for', 'in', 'little', 'of', 'the', 'to']
ignorechars = "',:;!"
```

**Python - Define LSA Class**

The LSA class has methods for initialization, parsing documents, building the matrix of word counts, and calculating. The first method is the `__init__` method, which is called whenever an instance of the LSA class is created. It stores the stopwords and ignorechars so they can be used later, and then initializes the word dictionary and the document count variables.

```python
class LSA(object):
    def __init__(self, stopwords, ignorechars):
        self.stopwords = stopwords
        self.ignorechars = ignorechars
        self.wdict = {}
        self.dcount = 0
```
Python - Parse Documents

The parse method takes a document, splits it into words, removes the ignored characters and turns everything into lowercase so the words can be compared to the stop words. If the word is a stop word, it is ignored and we move on to the next word. If it is not a stop word, we put the word in the dictionary, and also append the current document number to keep track of which documents the word appears in.

The documents that each word appears in are kept in a list associated with that word in the dictionary. For example, since the word book appears in titles 3 and 4, we would have self.wdict['book'] = [3, 4] after all titles are parsed.

After processing all words from the current document, we increase the document count in preparation for the next document to be parsed.

```python
    def parse(self, doc):
        words = doc.split();
        for w in words:
            w = w.lower().translate(None, self.ignorechars)
            if w in self.stopwords:
                continue
            elif w in self.wdict:
                self.wdict[w].append(self.dcount)
            else:
                self.wdict[w] = [self.dcount]
            self.dcount += 1
```

Python - Build the Count Matrix

Once all documents are parsed, all the words (dictionary keys) that are in more than 1 document are extracted and sorted, and a matrix is built with the number of rows equal to the number of words (keys), and the number of columns equal to the document count. Finally, for each word (key) and document pair the corresponding matrix cell is incremented.

```python
    def build(self):
        self.keys = [k for k in self.wdict.keys() if len(self.wdict[k]) > 1]
        self.keys.sort()
        self.A = zeros([len(self.keys), self.dcount])
        for i, k in enumerate(self.keys):
            for d in self.wdict[k]:
                self.A[i,d] += 1
```
**Python - Print the Count Matrix**

The printA() method is very simple, it just prints out the matrix that we have built so it can be checked.

```python
def printA(self):    print self.A
```

**Python - Test the LSA Class**

After defining the LSA class, it's time to try it out on our 9 book titles. First we create an instance of LSA, called mylsa, and pass it the stopwords and ignorechars that we defined. During creation, the __init__ method is called which stores the stopwords and ignorechars and initializes the word dictionary and document count.

Next, we call the parse method on each title. This method extracts the words in each title, strips out punctuation characters, converts each word to lower case, throws out stop words, and stores remaining words in a dictionary along with what title number they came from.

Finally we call the build() method to create the matrix of word by title counts. This extracts all the words we have seen so far, throws out words that occur in less than 2 titles, sorts them, builds a zero matrix of the right size, and then increments the proper cell whenever a word appears in a title.

```python
mylsa = LSA(stopwords, ignorechars)
for t in titles:    mylsa.parse(t)  mylsa.build()
mylsa.printA()
```

Here is the raw output produced by printA(). As you can see, it's the same as the matrix that we showed earlier.

```
[[ 0.  0.  1.  0.  0.  0.  0.  0. ]
 [ 0.  0.  0.  0.  1.  0.  0.  1. ]
 [ 0.  1.  0.  0.  0.  0.  1.  0. ]
 [ 0.  0.  0.  0.  1.  0.  1.  0. ]
 [ 1.  0.  0.  0.  1.  0.  0.  0. ]
```
In sophisticated Latent Semantic Analysis systems, the raw matrix counts are usually modified so that rare words are weighted more heavily than common words. For example, a word that occurs in only 5% of the documents should probably be weighted more heavily than a word that occurs in 90% of the documents. The most popular weighting is TFIDF (Term Frequency - Inverse Document Frequency). Under this method, the count in each cell is replaced by the following formula.

\[
\text{TFIDF}_{i,j} = \left( \frac{N_{i,j}}{N_{*,j}} \right) \times \log \left( \frac{D}{D_i} \right)
\]

where

- \(N_{i,j}\) = the number of times word i appears in document j (the original cell count).
- \(N_{*,j}\) = the number of total words in document j (just add the counts in column j).
- \(D\) = the number of documents (the number of columns).
- \(D_i\) = the number of documents in which word i appears (the number of non-zero columns in row i).

In this formula, words that concentrate in certain documents are emphasized (by the \(N_{i,j} / N_{*,j}\) ratio) and words that only appear in a few documents are also emphasized (by the \(\log(D / D_i)\) term).

Since we have such a small example, we will skip this step and move on to the heart of LSA, doing the singular value decomposition of our matrix of counts. However, if we did want to add TFIDF to our LSA class we could add the following two lines at the beginning of our python file to import the log, asarray, and sum functions.

```python
from math import log
from numpy import asarray, sum
```
Then we would add the following TFIDF method to our LSA class. WordsPerDoc \((N_{i,j})\) just holds the sum of each column, which is the total number of index words in each document.

DocsPerWord \((D_{i})\) uses asarray to create an array of what would be True and False values, depending on whether the cell value is greater than 0 or not, but the ‘i’ argument turns it into 1’s and 0’s instead. Then each row is summed up which tells us how many documents each word appears in. Finally, we just step through each cell and apply the formula. We do have to change cols (which is the number of documents) into a float to prevent integer division.

```python
def TFIDF(self):
    WordsPerDoc = sum(self.A, axis=0)
    DocsPerWord = sum(asarray(self.A > 0, 'i'), axis=1)
    rows, cols = self.A.shape
    for i in range(rows):
        for j in range(cols):
```

### Part 3 - Using the Singular Value Decomposition

Once we have built our (words by titles) matrix, we call upon a powerful but little known technique called Singular Value Decomposition or SVD to analyze the matrix for us. The [Singular Value Decomposition Tutorial](#) is a gentle introduction for readers that want to learn more about this powerful and useful algorithm.

The reason SVD is useful, is that it finds a reduced dimensional representation of our matrix that emphasizes the strongest relationships and throws away the noise. In other words, it makes the best possible reconstruction of the matrix with the least possible information. To do this, it throws out noise, which does not help, and emphasizes strong patterns and trends, which do help. The trick in using SVD is in figuring out how many dimensions or “concepts” to use when approximating the matrix. Too few dimensions and important patterns are left out, too many and noise caused by random word choices will creep back in.

The SVD algorithm is a little involved, but fortunately Python has a library function that makes it simple to use. By adding the one line method below to our LSA class, we can factor our matrix into 3 other matrices. The U matrix gives us the coordinates of each word on our “concept” space, the Vt matrix gives us the coordinates of each document in our “concept” space, and the S matrix of singular values gives us a clue as to how many dimensions or “concepts” we need to include.

```python
def calc(self):
```
In order to choose the right number of dimensions to use, we can make a histogram of the square of the singular values. This graphs the importance each singular value contributes to approximating our matrix. Here is the histogram in our example.

For large collections of documents, the number of dimensions used is in the 100 to 500 range. In our little example, since we want to graph it, we’ll use 3 dimensions, throw out the first dimension, and graph the second and third dimensions.

The reason we throw out the first dimension is interesting. For documents, the first dimension correlates with the length of the document. For words, it correlates with the number of times that word has been used in all documents. If we had centered our matrix, by subtracting the average column value from each column, then we would use the first dimension. As an analogy, consider golf scores. We don’t want to know the actual score, we want to know the score after
subtracting it from par. That tells us whether the player made a birdie, bogie, etc.

The reason we don't center the matrix when using LSA, is that we would turn a sparse matrix into a dense matrix and dramatically increase the memory and computation requirements. It's more efficient to not center the matrix and then throw out the first dimension.

Here is the complete 3 dimensional Singular Value Decomposition of our matrix. Each word has 3 numbers associated with it, one for each dimension. The first number tends to correspond to the number of times that word appears in all titles and is not as informative as the second and third dimensions, as we discussed. Similarly, each title also has 3 numbers associated with it, one for each dimension. Once again, the first dimension is not very interesting because it tends to correspond to the number of words in the title.

| book | 0.15 | -0.27 | 0.04 |
| dads | 0.24 | 0.38  | -0.09|
| dummies | 0.13 | -0.17 | 0.07 |
| estate | 0.18 | 0.19  | 0.45 |
| guide | 0.22 | 0.09  | -0.46|
| investing | 0.74 | -0.21 | 0.21 |
| market | 0.18 | -0.30 | -0.28|
| real | 0.18 | 0.19  | 0.45 |
| rich | 0.36 | 0.59  | -0.34|
| stock | 0.25 | -0.42 | -0.28|
| value | 0.12 | -0.14 | 0.23 |

* 3.91 0 0
0 2.61 0 0
0 0 2.00

T1 T2 T3 T4 T5
Part 4 - Clustering by Color

We can also turn the numbers into colors. For instance, here is a color display that corresponds to the first 3 dimensions of the Titles matrix that we showed above. It contains exactly the same information, except that blue shows negative numbers, red shows positive numbers, and numbers close to 0 are white. For example, Title 9, which is strongly positive in all 3 dimensions, is also strongly red in all 3 dimensions.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.35</td>
<td>0.22</td>
<td>0.34</td>
<td>0.26</td>
</tr>
<tr>
<td>-0.32</td>
<td>-0.15</td>
<td>-0.46</td>
<td>-0.24</td>
</tr>
<tr>
<td>-0.41</td>
<td>0.14</td>
<td>-0.16</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Latent Semantic Analysis (LSA) Tutorial

We can use these colors to cluster the titles. We ignore the first dimension for clustering because all titles are red. In the second dimension, we have the following result.

<table>
<thead>
<tr>
<th>Dim2</th>
<th>Titles</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>6-7, 9</td>
</tr>
<tr>
<td>blue</td>
<td>1-5, 8</td>
</tr>
</tbody>
</table>

Using the third dimension, we can split each of these groups again the same way. For example, looking at the third dimension, title 6 is blue, but title 7 and title 9 are still red. Doing this for both groups, we end up with these 4 groups.

<table>
<thead>
<tr>
<th>Dim2</th>
<th>Dim3</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>red</td>
</tr>
<tr>
<td>red</td>
<td>blue</td>
</tr>
<tr>
<td>blue</td>
<td>red</td>
</tr>
<tr>
<td>blue</td>
<td>blue</td>
</tr>
</tbody>
</table>

It's interesting to compare this table with what we get when we graph the results in the next section.

Part 5 - Clustering by Value

Leaving out the first dimension, as we discussed, let's graph the second and third dimensions using a XY graph. We'll put the second dimension on the X axis and the third dimension on the Y axis and graph each word and title. It's interesting to compare the XY graph with the table we just created that clusters the documents.

In the graph below, words are represented by red squares and titles are represented by blue circles. For example the word "book" has dimension values (0.15, -0.27, 0.04). We ignore the first dimension value 0.15 and graph "book" to position \((x = -0.27, y = 0.04)\) as can be seen in the graph. Titles are similarly graphed.
Latent Semantic Analysis (LSA) Tutorial

One advantage of this technique is that both words and titles are placed on the same graph. Not only can we identify clusters of titles, but we can also label the clusters by looking at what words are also in the cluster. For example, the lower left cluster has titles 1 and 3 which are both about stock market investing. The words “stock” and “market” are conveniently located in the cluster, making it easy to see what the cluster is about. Another example is the middle cluster which has titles 2, 4, 5, and, to a somewhat lesser extent, title 8. Titles 2, 4, and 5 are close to the words “value” and “investing” which summarizes those titles quite well.

Advantages, Disadvantages, and Applications of LSA

Latent Semantic Analysis has many nice properties that make it widely applicable to many problems.

1. First, the documents and words end up being mapped to the same concept space. In this space we can cluster documents, cluster words, and most importantly, see how these clusters coincide so we can retrieve documents based on words and vice versa.

2. Second, the concept space has vastly fewer dimensions compared to the original matrix. Not only that, but these dimensions have been chosen specifically because they contain the most information and least noise. This makes the new concept space ideal for running further algorithms such as testing different clustering algorithms.

3. Last, LSA is an inherently global algorithm that looks at trends and patterns from all documents and all words so it can find things that may not be apparent to a more locally based algorithm. It can also be usefully combined with a more local algorithm such as nearest neighbors to become more useful than either algorithm by itself.
There are a few limitations that must be considered when deciding whether to use LSA. Some of these are:

1. LSA assumes a Gaussian distribution and Frobenius norm which may not fit all problems. For example, words in documents seem to follow a Poisson distribution rather than a Gaussian distribution.

2. LSA cannot handle polysemy (words with multiple meanings) effectively. It assumes that the same word means the same concept which causes problems for words like bank that have multiple meanings depending on which contexts they appear in.

3. LSA depends heavily on SVD which is computationally intensive and hard to update as new documents appear. However recent work has led to a new efficient algorithm which can update SVD based on new documents in a theoretically exact sense.

In spite of these limitations, LSA is widely used for finding and organizing search results, grouping documents into clusters, spam filtering, speech recognition, patent searches, automated essay evaluation, etc.

As an example, iMetaSearch uses LSA to map search results and words to a “concept” space. Users can then find which results are closest to which words and vice versa. The LSA results are also used to cluster search results together so that you save time when looking for related results.

*revised on January 30, 2010*