CSCI 252: Neural Networks and Graphical Models

Fall Term 2016
Prof. Levy

"Architecture" #5: Latent Semantic Analysis (Landauer & Dumais 1997)

Latent Semantic Analysis

- A.k.a. Latent Semantic Indexing
- Not a traditional neural net, but shares important features with the kinds of networks we've been studying (distributed representation)
- Of both theoretical interest ("Plato's Problem") and practical value (e.g., big-data document classification)

Words as Vectors

- Recall Zhao, Li, Kohonen (2010) trigrams: Words start out as random vectors of 0s and 1s; "meaning" of a word is sum of vectors of all words to its left and right.
- This provides a good basis for visualization (2D SOM map), but what about other, more practical tasks?
  - Authorship (who wrote this document?)
  - Translating homonyms (*The pitcher threw the ball; the pitcher was half full*.) [Elizabeth Davis '06]
  - Web search (documents likely to interest you)

The Problem in a Nutshell

- Consider two documents (e.g., web pages) A and B. Both are about baseball.
- A has the following words: *Yankees, bullpen, catcher, stadium, grand slam, …*
- *B* has the following words: *Mets, stadium, catcher, home run, foul*
- A pure word-occurrence check will return *A* and *B* for *catcher* or *stadium,* but only *A* for *grand slam*, B for *home run*, etc.
- Neither document might be returned if we search on *baseball*

The Solution in a Nutshell

- LSA solves this problem by giving us not a set of yes/no values for each document/word pair, but a relative *likelihood of finding a given word in a given document*, based on other documents, *even if the word is not actually there*.
- Hence the term latent; i.e., "hidden" (by chance or accident)
- But this still seems like magic! Let's see how it actually works ….

Step 1: Build a Co-ocurrence Matrix

	A	B	...
catcher	2	1	...
grand slam	1	0	...
home run	0	4	...
bullpen	1	0	...
foul	0	3	...
stadium	3	1	...
Yankees	1	0	
Mets	0	2	...
...	...	...	...

Step 2: Add 1 to eliminate zeros

	A	B	...
catcher	3	2	...
grand slam	2	1	...
home run	1	5	...
bullpen	2	1	...
foul	1	4	...
stadium	4	2	...
Yankees	2	1	
Mets	1	3	...
...	...	...	...

Step 3: Apply logarithm to reduce big values

	A	B	...
catcher	1.10	0.69	...
grand slam	0.69	0.00	...
home run	0.00	1.61	...
bullpen	0.69	0.00	...
foul	0.00	1.39	...
stadium	1.39	0.69	...
Yankees	0.69	0.00	
Mets	0.00	1.10	...
...	...	...	...

Step 4: Do a Singular Value Decomposition

- This is the most complicated, but most crucial step.
- Basic hack:
  - Decompose (break down) the matrix into components that when multiplied back together will yield the original matrix
  - Before multiplying the three components, zero-out some of the values in one of the components. So instead of the original matrix, we get back a “blurred” version
  - To understand this, we’ll need a little linear algebra

$%#@ Dot Product Again!

- Let's recap our linear algebra tricks thus far ...
- Dot product (single-neuron model); vector * vector = scalar

$$y = \sum_{k=1}^{n} x_k w_k \qquad \text{y = np.dot(x, w)}$$

- Vector * matrix = vector (Hopfield test method):

$$u_j = \sum_i u_i T_{ij} \qquad \text{u = np.dot(u, T)}$$

- Vector * vector = matrix; outer product (Hopfield learn):

$$T_{ij} = a_i a_j \qquad \text{T = np.outer(a,a)}$$

Vector * Matrix: Details

- The result is really just a vector of dot products
- Consider a simple example with a square matrix:

```
          d  e  f
a  b  c *  g  h  i   = ad+bg+cj  ae+bh+ck  af+bi+cl
          j  k  l
```

```
>>> u = np.asarray([1,2,3])
>>> T = np.asarray([[4,5,6],[7,8,9],[10,11,12]])
>>> u
array([1, 2, 3])
>>> T
array([[ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>> np.dot(u,T)
array([48, 54, 60])
```

Matrix * Matrix: Details

- The result is really just a matrix of dot products
- Consider a simple example with two square matrices:

```
a b c     j k l     aj+bm+cp ak+bn+cq al+bo+cr
d e f  *  m n o  =  dj+em+fp dk+en+fq dl+eo+fr
g h i     p q r     gj+hm+ip gk+hn+iq gl+ho+ir
```

```
>>> T
array([[ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>> U
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> np.dot(T,U)
array([[ 66,  81,  96],
       [102, 126, 150],
       [138, 171, 204]])
```

Singular Value Decomposition

- Given a (not necessarily square) matrix *M*, the SVD produces three matrices **U**, **Σ**, and, ***V****, such that **M = U Σ V***
- (The * in V* stands for the conjugate transpose, a complex operation that need not concern us here.)
- The "magic" happens in **Σ**, a matrix with zeros everywhere but the diagonal.

Consider the 4×5 matrix

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

A singular value decomposition of this matrix is given by $\mathbf{U\Sigma V^*}$

$$\mathbf{U} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{\Sigma} = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{V}^* = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ \sqrt{0.2} & 0 & 0 & 0 & \sqrt{0.8} \\ 0 & 0 & 0 & 1 & 0 \\ -\sqrt{0.8} & 0 & 0 & 0 & \sqrt{0.2} \end{bmatrix}$$

https://en.wikipedia.org/wiki/Singular_value_decomposition

numpy.linalg.svd()

```
import numpy as np

M = np.asarray([[1,0,0,0,2],
                [0,0,3,0,0],
                [0,0,0,0,0],
                [0,2,0,0,0]])


U, sigma, V = np.linalg.svd(M, full_matrices=False)

print(U)
print('')
print(sigma)
print('')
print(V)
print('')

Sigma = np.diag(sigma)
print(Sigma)
print('')

print(np.dot(U, np.dot(Sigma, V)))
```

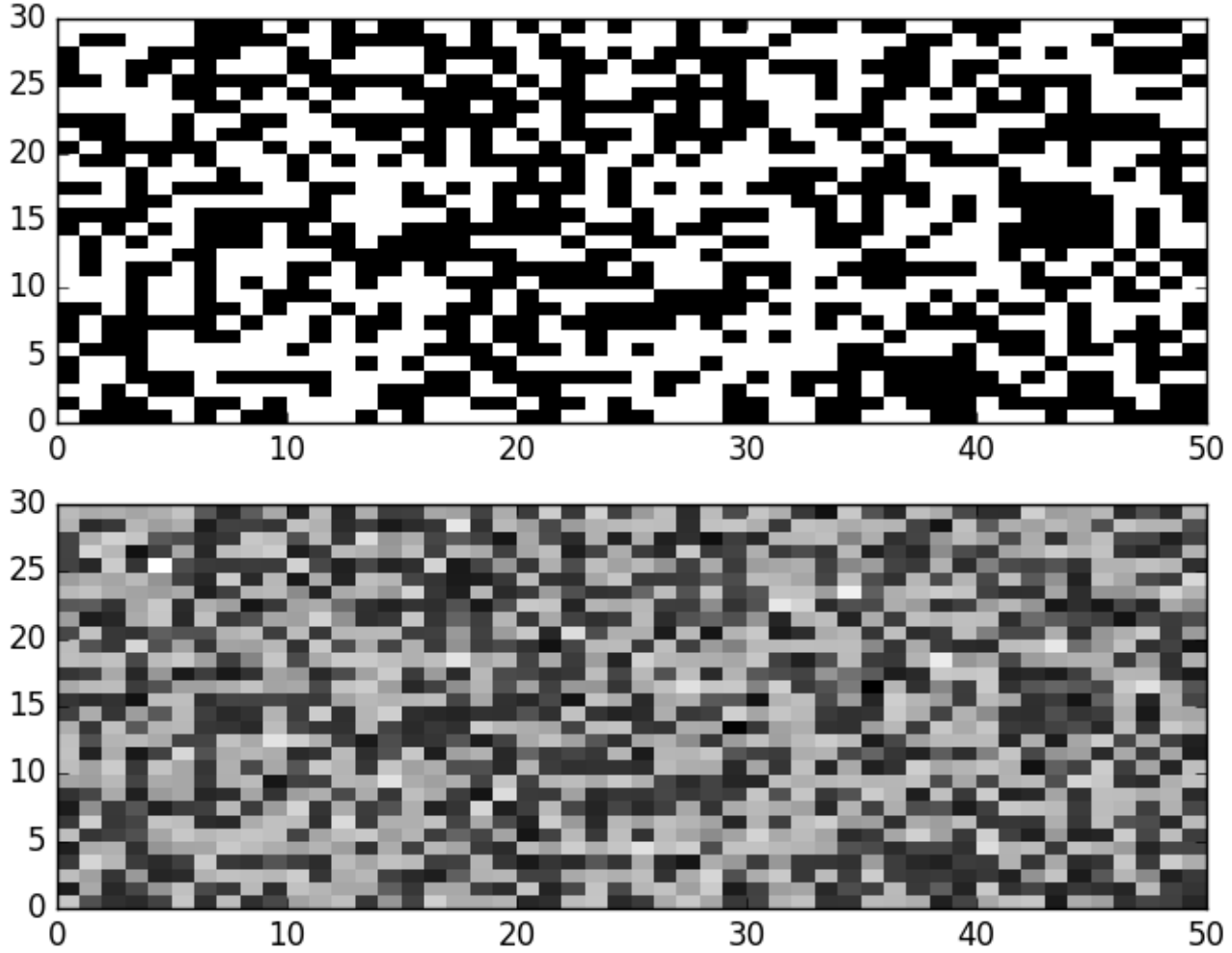
numpy.linalg.svd()

```
[[ 0.  1.  0.  0.]
 [ 1.  0.  0.  0.]
 [ 0.  0.  0. -1.]
 [ 0.  0.  1.  0.]]

[ 3.          2.23606798  2.          0.        ]

[[-0.          0.          1.         -0.          0.        ]
 [ 0.4472136  -0.         -0.         -0.          0.89442719]
 [-0.          1.          0.         -0.          0.        ]
 [ 0.          0.          0.          1.          0.        ]]

[[ 3.          0.          0.          0.        ]
 [ 0.          2.23606798  0.          0.        ]
 [ 0.          0.          2.          0.        ]
 [ 0.          0.          0.          0.        ]]

[[ 1.  0.  0.  0.  2.]
 [ 0.  0.  3.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  2.  0.  0.  0.]]
```

So where's the magic?

- Note that the values in Σ – called the eigenvalues – come in decreasing magnitudes.
- So the trick is to zero out some of the lower-right values, providing us with the "blurring" or "smearing" we want in the reconstructed M matrix.
- Let's try this with a bigger example ….

```python
import numpy as np
import matplotlib.pyplot as plt

# Step 1 Build a (fake, random) co-occurrence matrix
M = np.random.random_integers(0, 1, (30,50))

# Step 2: Add 1 to eliminate zeros
M += 1

# Step 3: Take the logarithm to reduce big values
M = np.log(M)

# Step 4: Run the Singular Value Decomposition
U, sigma, V = np.linalg.svd(M, full_matrices=False)

# Step 5: Eliminate lower-order eigenvalues
sigma[-10:-1] = 0

# Step 6: Reconstruct the matrix
Msvd = np.dot(U, np.dot(np.diag(sigma), V))

# Plot the original and SVD co-occurrence matrices
plt.set_cmap('gray')
plt.subplot(2,1,1)
plt.pcolor(M)
plt.subplot(2,1,2)
plt.pcolor(Msvd)
plt.show()
```


30
25
20
15
10
5
0
0
10
20
30
40
50
30
25
20
15
10
5
0
0
10
20
30
40
50