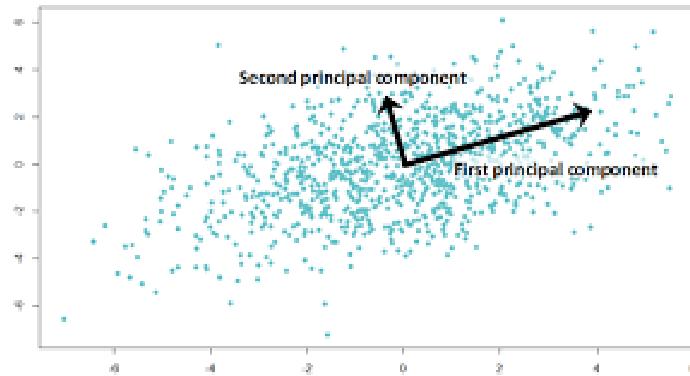
Principal Components

MATH1900: Machine Learning

Location: http://people.sc.fsu.edu/~jburkardt/classes/ml_2019/principal_components/principal_components.pdf



What are the top two trends in my data?

Principal Component Analysis

Find a small set of orthogonal basis vectors that approximate a large data set.

If we ask 1,000 people to fill out a survey of 50 questions, it's likely that every survey will be different. However, it may also be the case that distinct patterns can be observed, corresponding to differences in men and women, young and old, conservative and liberal. Identifying these patterns will help us to understand the data, to replace the raw data by a simpler model that explains much of the results.

We can use principal component analysis (PCA) to search for these patterns. We try to boil down our data to reveal the most information in the fewest number of components.

If there was only one question on the survey, it would make sense to compute the average answer, and then the variance to report how much answers can deviate from it. Now, however, our task is more complicated.

We will think of our data as a big, rectangular, numerical matrix, and we will see that the singular value decomposition (SVD) can give us useful answers.

1 A = U * S * V

The SVD factorization of an $m \times n$ matrix A (as computed in Python) has the form $A = USV$ where the matrix S has the shape of A, but is nonzero only on the diagonal, while the $m \times m$ matrix U and the $n \times n$ matrix V are orthogonal.

In Python, if A is an *np.array()*, then we can request the SVD factorization by:

```
U, s, V = np.linalg.svd ( A );
```

Instead of returning the matrix S, we get back a vector s, of length $mn = \min(m, n)$, containing the diagonal entries of S. We can build S by

```
S = np.zeros ( A.shape )
mn = min ( m, n )
for i in range ( 0, mn ):
  S[i,i]= s[i]
```

Listing 1: Create matrix S from vector s

To verify that the factorization is correct, we can do the following test:

```
  SV = np.matmul ( S, V )
  USV = np.matmul ( U, SV )
  diff = np.linalg.norm ( A - USV )
```

Listing 2: Compare A and U*S*V

Exercise: Compute `A = np.random.randn ( 5, 6 )`, compute the SVD, and verify that the factors, when multiplied, are almost exactly equal to `A`.

Most texts and algorithms return the transpose of the matrix `V`. Python is the only system that I know of where this is not so!

You can look at the code *svd_product.py*, which checks that $A = U * S * V$ for several small matrices.

2 Theory of approximating A with a partial SVD

You know that if column vectors u and v both have length n that you can compute the vector dot product, or vector inner product,

$$u \text{ dot } v = u'\, v = \sum_{i=1}^{n} u_i v_i$$

You may be less familiar with the *outer product* of a column m-vector u and a row n-vector v:

$$u \otimes v = u\, v$$

which results in an $m \times n$ matrix A whose (i, j) entry is:

$$A_{i,j} = u_i\, v_j$$

Thus, if we have

$$u = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, v = \begin{pmatrix} 10 & -2 \end{pmatrix},$$

Then

$$u \otimes v = u\, v = \begin{pmatrix} 10 & -2 \\ 20 & -4 \\ 30 & -6 \end{pmatrix}$$

So, although normally a 1000×50 matrix requires 50,000 values to describe it, we can easily create a matrix that large just by an outer product that requires a total of 1000 + 50 values. The question is, what good is such an "inflated" matrix?

The answer goes the other way. If our 50,000 values can actually be described by 1050 values, then we have compressed the data down to the actual information. Even if we have to create 3 or 4 such outer products and add them together and we only get an approximation to our original big matrix, such an approximation

could be good enough for our purposes, and might also reveal some of the underlying structure of the raw data.

So now let's see how the SVD can be used in this way.

Let's suppose that A is an $m \times n$ matrix. Our survey data, for instance, might be stored as a 1000×50 table of numbers. Our computational puzzle is: can we approximate the large amount of data in A by a small set of numbers, which capture most of the information?

We know that computing the average answer for each question gives us a lot of information to start with, so we can start our analysis by creating a matrix A_0 so that each row contains the same 50 average values. We can compute this as an outer product, $A_0 = u_0 \otimes v_0$ where the m vector u_0 and the n-vector v_0 are:

$$u_0 = \begin{pmatrix} 1 \\ 1 \\ ... \\ 1 \end{pmatrix}, \quad v_0 = \begin{pmatrix} \mu_1 & \mu_2 & ... & \mu_n \end{pmatrix}$$

Now let's compute the SVD of $A - A_0$, so that we have exactly:

$$A = A_0 + U\,S\,V = u_0 \otimes v_0 + U\,S\,V$$

The SVD suggests that, if we wanted to use just one more matrix outer product, we should use the first column of U, the first row of V, and multiply it by the first entry in S. This approximation would have the form:

$$A \approx A_0 + A_1 = u_0 \otimes v_0 + s_1\, u_1 \otimes v_1$$

Using one more column gives us a better approximation:

$$A \approx A_0 + A_1 + A_2 = u_0 \otimes v_0 + s_1\, u_1 \otimes v_1 + s_2\, u_2 \otimes v_2$$

We can tell how good our approximation is. If A_k is formed from the first k columns of the SVD, then:

$$||A - A_0 - A_1 - ... - A_k||_2^2 = \frac{\sum_{i=k+1}^{mn} s_i^2}{\sum_{i=1}^{mn} s_i^2}$$

3 Exercise: approximating A with a partial SVD

Let's test out the theory on a simple 5×5 matrix A. Once we subtract the mean, we expect to have just 4 nonzero singular values. Let's add up the first few outer products and see how we do:

```
1  import numpy as np
2  m = 5
3  n = 5
4  A = np.array ( [
5    [17, 24,  1,  8, 15],
6    [23,  5,  7, 14, 16],
7    [ 4,  6, 13, 20, 22],
8    [10, 12, 19, 21,  3],
9    [11, 18, 25,  2,  9] ] )
```

Listing 3: Define a matrix A

```
1  u0 = np.ones ( m )
2  v0 = np.mean ( A, axis = 0 )
3  A0 = np.outer ( u0, v0 )
```

Listing 4: Compute mean and create $A0$

```
1  U, s, V = np.linalg.svd ( A - A0 )
2  diff = np.linalg.norm ( A - A0 )
```

Listing 5: Request SVD of $A - A_0$

```
1  A1 = s[0] * np.outer ( U[:,0], V[0,:] )
2  diff = np.linalg.norm ( A - A0 - A1 )
```

Listing 6: Create outer product A_1

```
1  A2 = s[1] * np.outer ( U[:,1], V[1,:] )
2  diff = np.linalg.norm ( A - A0 - A1 - A2 )
```

Listing 7: Create outer product A_2

At this halfway point, we can compare A and our approximation A0+A1+A2:

```
   -----A-----            -----A0+A1+A2-----
[[17 24  1  8 15]     [[20.7 20.4  2.9  5.7 15.2]
 [23  5  7 14 16]      [16.0 10.2  4.6 15.8 18.1]
 [ 4  6 13 20 22]      [ 9.6  3.9 13.0 22.0 16.3]
 [10 12 19 21  3]      [ 7.8 10.1 21.3 15.7  9.9]
 [11 18 25  2  9]]     [10.7 20.2 23.0  5.5  5.2]]
```

The SVD promises that our sequence of approximations is the best that can be computed, and always gives us a report of how far off we are by looking at the singular values.

Once we have computed A_0, we can jump to any approximation in one step. This requires setting up the full matrix S, and doing a nested multiplication of $S * V$, then of $U * (S * V)$: For instance, to compute the third approximation A123 = A1 + A2 + A3:

```
1  A123 = np.matmul ( U[:,0:3], np.matmul ( S[0:3,0:3], V[0:3,:] ) )
```

Listing 8: Create an outer product A_{123} in one step

You can look at some of these computations in *svd_approx.py*.

4 PCA by Singular Value Decomposition of A

As a small, but more realistic example, we are ready to consider a dataset containing 214 records, describing 9 measurements of chemical properties of samples of glass.

We will read a data file *glass_data.txt* into a $m \times n$ array A. We compute the average values of the 9 columns and write $A_0 = u_0 \otimes v_0$, where each row of A_0 stores the average values. We compute the SVD of $A - A_0$. The singular values s tell us the importance of each column of U and row of V. If we approximate A with a part of the SVD, then the corresponding singular values tell us how close our approximation is.

```
1  import numpy as np
2  A = np.loadtxt ( 'glass_data.txt' )
3  m, n = A.shape
```

Listing 9: Read the data into A

```
1  u0 = np.ones ( m )
2  v0 = np.mean ( A, axis = 0 )
3  A0 = np.outer ( u0, v0 )
4  print ( '||A-A0|| = ', np.linalg.norm ( A - A0 ) )
```

Listing 10: Create column average matrix A_0

```
U, s, V = np.linalg.svd ( A - A0 )
```

Listing 11: Get SVD of $A - A_0$

```
A1 = s[0] * np.outer ( U[:,0], V[0,:] ) )
print ( '||A-A0-A1|| = ', np.linalg.norm ( A - A0 - A1 ) )
```

Listing 12: Create A_1 first outer product

```
A2 = s[1] * np.outer ( U[:,1], V[1,:] ) )
print ( '||A-A0-A1-A2|| = ', np.linalg.norm ( A - A0 - A1 -A2 ) )
```

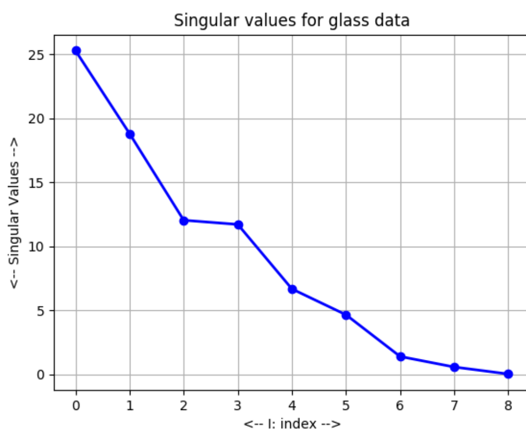
Listing 13: Create A_2 second outer product

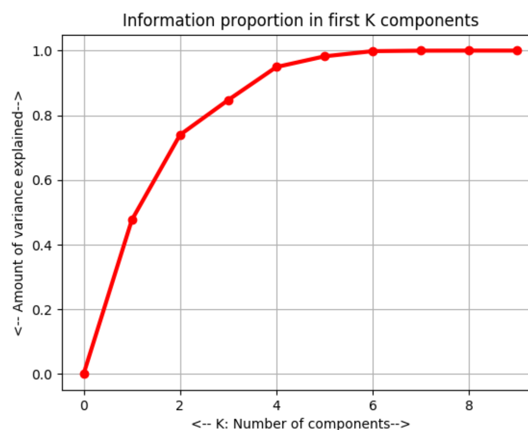
```
import matplotlib.pyplot as plt
mn = min ( m, n )
x = np.arange ( mn )
plt.plot ( x, s )
```

Listing 14: Plot the singular values

```
x = np.arange ( mn + 1 )
y = [ 0.0 ]
y = np.append ( y, np.cumsum ( s**2 ) )
y = y / np.sum ( s**2 )
plt.plot ( x, y )
```

Listing 15: Plot variance captured by first K vectors





Singular values and variance capture

The point of this exercise is to suggest how the SVD can be used to handle datasets much larger than this example. We have only looked at the idea of approximating the data with a small set of outer products, but with the SVD there are many other tools that we can build.

You can look at some of these computations in *svd_glass.py*.

5 PCA for Image Data

Consider an black and white image as an $m \times n$ array A of numbers. Now imagine each column of this array as being a data item, so that we have n items, each of dimension m. (You need the PIL library for this!)

If we compute the SVD of A, we can approximate the image using combinations of a small set of singular vectors, and compute how much information we have captured.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from PIL import image
4  I = Image.open ( 'casablanca.png' )
5  I = I.convert ( 'LA' )
```

Listing 16: Start up and get the image data

```
1  A = np.array ( list ( I.getdata(band=0)),float )
2  A.shape = ( I.size[1], I.size[0] )
3  plt.imshow ( A, cmap = 'gray' )
4  plt.title ( 'Numeric version of image' )
5  plt.show ( )
```

Listing 17: Convert image to numeric format and display

```
1  u0 = np.ones ( m )
2  v0 = np.mean ( A, axis = 0 )
3  A0 = np.outer ( u0, v0 )
```

Listing 18: Compute column averages

```
1  U, s, V = np.linalg.svd ( A - A0 )
2  S = np.zeros ( A.shape )
3  for i in range ( 0, min ( m, n ) ):
4    S[i,i] = s[i]
```

Listing 19: Compute SVD and build S

```
1  r = 5
2  A5 = A0 + np.matmul ( U[:,0:r], np.matmul ( S[0:r,0:r], V[0:r,:] ) )
3  plt.imshow ( A5, cmap = 'gray' )
4  plt.title ( 'Reconstruction using %d components' % ( r ) )
5  plt.show ( )
6  plt.clf ( )
```

Listing 20: Construct average using A0 and 5 outer products

The last block of code can be modified to use 10, 20 or 40 outer products.

You can look at some of these computations in *svd_image.py*.

6 Computing Assignment #3

Write a Python program which

- Sets `A` to the data in *concrete_data.txt* (on web page);
- Sets `A0` to the matrix in which each row contains the column averages;
- Computes `U, s, V` of the SVD of `A-A0`;
- Computes `A1` through `A3`, the first three SVD outer products;
- Prints $||A||, ||A-A0||, ||A-A0-A1||, ||A-A0-A1-A2||, ||A-A0-A1-A2-A3||$.

Email a copy of your program to Dr Schneier **mhs64@pitt.edu** by 11:59pm Friday, 27 September.