In this example, we illustrate various unsupervised learning techniques (Clustering, PCA, SVD) using an example term-document matrix as the data.

In [1]:
```
import numpy as np
import pylab as pl
import pandas as pd
from sklearn.cluster import KMeans
```

In [2]:
```
cd D:\Documents\Class\CSC478\Data
```

```
D:\Documents\Class\CSC478\Data
```

In [4]:
```
Data = pd.read_csv('term-doc-mat.csv', header=None)
TD = Data.ix[:,1:]
TD
```

Out[4]:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	24	32	12	6	43	2	0	3	1	6	4	0	0	0	0
1	9	5	5	2	20	0	1	0	0	0	27	14	3	2	11
2	0	3	0	0	3	7	12	4	27	4	0	1	0	0	0
3	3	0	0	0	0	16	0	2	25	23	7	12	21	3	2
4	1	0	0	0	0	33	2	0	7	12	14	5	12	4	0
5	12	2	0	0	27	0	0	0	0	22	9	4	0	5	3
6	0	0	0	0	0	18	32	22	34	17	0	0	0	0	0
7	1	0	0	0	2	0	0	0	3	9	27	7	5	4	4
8	21	10	16	7	31	0	0	0	0	0	0	0	0	1	0
9	2	0	0	2	0	27	4	2	11	8	33	16	14	7	3

In [5]:
```
terms = Data.ix[:,0]
terms
```

Out[5]:
```
0      database
1         index
2    likelihood
3        linear
4        matrix
5         query
6    regression
7     retrieval
8           sql
9        vector
Name: 0, dtype: object
```

First, we want to do some document clustering. Since the data is in term-document format, we need to obtain the transpose of the TD matrix.

In [6]:
```
DT = TD.T
```

Now we have a document-term matrix:

In [8]:
```
DT
```

Out[8]:

	0	1	2	3	4	5	6	7	8	9
1	24	9	0	3	1	12	0	1	21	2
2	32	5	3	0	0	2	0	0	10	0
3	12	5	0	0	0	0	0	0	16	0
4	6	2	0	0	0	0	0	0	7	2
5	43	20	3	0	0	27	0	2	31	0
6	2	0	7	16	33	0	18	0	0	27
7	0	1	12	0	2	0	32	0	0	4
8	3	0	4	2	0	0	22	0	0	2
9	1	0	27	25	7	0	34	3	0	11
10	6	0	4	23	12	22	17	9	0	8
11	4	27	0	7	14	9	0	27	0	33
12	0	14	1	12	5	4	0	7	0	16
13	0	3	0	21	12	0	0	5	0	14
14	0	2	0	3	4	5	0	4	1	7
15	0	11	0	2	0	3	0	4	0	3

In [12]:
```
numTerms=len(terms)
numTerms
```

Out[12]: 10

Next, we will transform the data to TFxIDF weights:

In [13]:
```
# Find doucment frequencies for each term
DF = np.array([(DT!=0).sum(0)])
print DF
```

```
[[10 11  8 10  9  8  5  9  6 12]]
```

In [15]:
```
NDocs = len(DT[0])
print NDocs
```

```
15
```

In [16]:
```
# Create a matrix with all entries = NDocs
NMatrix=np.ones(np.shape(DT), dtype=float)*NDocs
```

In [17]:
```
# Convert each entry into IDF values
# Note that IDF is only a function of the term, so all rows will be identical.
DivM = np.divide(NMatrix, DF)
IDF = np.log2(DivM)
```

In [18]:
```
np.set_printoptions(precision=2,suppress=True)
print IDF[0:2,]
```

```
[[ 0.58  0.45  0.91  0.58  0.74  0.91  1.58  0.74  1.32  0.32]
 [ 0.58  0.45  0.91  0.58  0.74  0.91  1.58  0.74  1.32  0.32]]
```

In [19]:
```
# Finally compute the TFxIDF values for each document-term entry
DT_tfidf = DT * IDF
```

In [21]:
```
DT_tfidf
```

Out[21]:

	0	1	2	3	4	5	6	7
1	14.039100	4.027131	0.000000	1.754888	0.736966	10.882687	0.000000	0.7369
2	18.718800	2.237295	2.720672	0.000000	0.000000	1.813781	0.000000	0.0000
3	7.019550	2.237295	0.000000	0.000000	0.000000	0.000000	0.000000	0.0000
4	3.509775	0.894918	0.000000	0.000000	0.000000	0.000000	0.000000	0.0000
5	25.153388	8.949180	2.720672	0.000000	0.000000	24.486046	0.000000	1.4739
6	1.169925	0.000000	6.348234	9.359400	24.319865	0.000000	28.529325	0.0000
7	0.000000	0.447459	10.882687	0.000000	1.473931	0.000000	50.718800	0.0000
8	1.754888	0.000000	3.627562	1.169925	0.000000	0.000000	34.869175	0.0000
9	0.584963	0.000000	24.486046	14.624063	5.158759	0.000000	53.888725	2.2108
10	3.509775	0.000000	3.627562	13.454138	8.843587	19.951593	26.944363	6.6326
11	2.339850	12.081392	0.000000	4.094738	10.317518	8.162015	0.000000	19.898
12	0.000000	6.264426	0.906891	7.019550	3.684828	3.627562	0.000000	5.1587
13	0.000000	1.342377	0.000000	12.284213	8.843587	0.000000	0.000000	3.6848
14	0.000000	0.894918	0.000000	1.754888	2.947862	4.534453	0.000000	2.9478
15	0.000000	4.922049	0.000000	1.169925	0.000000	2.720672	0.000000	2.9478

Now we are ready for clustering

In [22]:
```
import kMeans
```

In [23]:
```
reload(kMeans)
```

Out[23]:
```
<module 'kMeans' from 'kMeans.pyc'>
```

In [25]:
```
DT_tfidf = np.array(DT_tfidf)
centroids_tfidf, clusters_tfidf = kMeans.kMeans(DT_tfidf, 3, kMeans.distCosine
, kMeans.randCent)
```

```
Iteration  1
Iteration  2
Iteration  3
```

Let's take a look at the cluster centroids

In [26]:
```
print "\t\tCluster0\tCluster1\tCluster2"
for i in range(len(terms)):
    print "%10s\t%.4f\t\t%.4f\t\t%.4f" %(terms[i],centroids_tfidf[0][i],centro
ids_tfidf[1][i],centroids_tfidf[2][i])
```

```
               Cluster0        Cluster1        Cluster2
  database     0.4680          1.4039          13.6881
     index     5.1010          0.0895          3.6692
likelihood     0.1814          9.7944          1.0883
    linear     5.2647          7.7215          0.3510
    matrix     5.1588          7.9592          0.1474
     query     3.8089          3.9903          7.4365
regression     0.0000          38.9901         0.0000
 retrieval     6.9275          1.7687          0.4422
       sql     0.2644          0.0000          22.4728
    vector     4.7002          3.3481          0.2575
```

Because the centroids are based on TFxIDF weights, they are not as descriptive as raw term frequencies or binary occurrence data. Let's redo the clustering with the original raw term frequencies.

In [27]:
```
DT = np.array(DT)
centroids, clusters = kMeans.kMeans(DT, 3, kMeans.distCosine, kMeans.randCent)
```

```
Iteration  1
Iteration  2
Iteration  3
Iteration  4
Iteration  5
```

In [28]:
```
print "\t\tCluster0\tCluster1\tCluster2"
for i in range(len(terms)):
    print "%10s\t%.4f\t\t%.4f\t\t%.4f" %(terms[i],centroids[0][i],centroids[1][i],centroids[2][i])
```

```
               Cluster0        Cluster1        Cluster2
  database     1.7143          23.4000         1.3333
     index     8.1429          8.2000          0.3333
likelihood     1.7143          1.2000          14.3333
    linear     12.0000         0.6000          9.0000
    matrix     11.4286         0.2000          3.0000
     query     6.1429          8.2000          0.0000
regression     5.0000          0.0000          29.3333
 retrieval     8.0000          0.6000          1.0000
       sql     0.1429          17.0000         0.0000
    vector     15.4286         0.8000          5.6667
```

In [29]:
```
print clusters
```

```
[[ 1.    0.  ]
 [ 1.    0.01]
 [ 1.    0.01]
 [ 1.    0.01]
 [ 1.    0.  ]
 [ 0.    0.03]
 [ 2.    0.  ]
 [ 2.    0.  ]
 [ 2.    0.  ]
 [ 0.    0.05]
 [ 0.    0.02]
 [ 0.    0.01]
 [ 0.    0.01]
 [ 0.    0.01]
 [ 0.    0.14]]
```

In [30]:
```
print centroids
```

```
[[  1.71   8.14   1.71  12.    11.43   6.14   5.     8.     0.14  15.43]
 [ 23.4    8.2    1.2    0.6    0.2    8.2    0.     0.6   17.     0.8 ]
 [  1.33   0.33  14.33   9.     3.     0.    29.33   1.     0.     5.67]]
```

Next, let's use principal component analysis to reduce the dimensionality of the data:

In [31]:
```
from sklearn import decomposition
```

We'll perform PCA to obtain the top 5 components and then transform the DT matrix into the lower dimensional space of 5 components:

In [32]:
```
pca = decomposition.PCA(n_components=5)
DTtrans = pca.fit(DT).transform(DT)
```

In [33]:
```
np.set_printoptions(precision=2,suppress=True)
print DTtrans
```

```
[[-25.45  -1.8    4.02  -3.12  -0.24]
 [-23.78  -7.29  -0.53  -4.77   6.32]
 [-15.03  -5.44 -10.09  -3.46   2.87]
 [ -6.75  -4.39 -14.84  -3.08   0.44]
 [-46.7    0.01  21.78   3.11  -0.29]
 [ 27.41   7.93  12.27 -15.16  13.47]
 [ 15.67 -21.74  -1.33  11.4    5.81]
 [  8.28 -16.54  -6.94   5.39   1.47]
 [ 29.51 -18.64  16.68   4.82  -1.69]
 [ 11.41   0.44  15.03  -3.03 -17.19]
 [  6.98  37.69   4.5   14.83   5.5 ]
 [  5.97  13.91  -6.64   2.12  -3.32]
 [ 12.36  10.4   -4.93 -11.5   -5.51]
 [  1.67   2.69 -13.18  -2.04  -3.72]
 [ -1.53   2.79 -15.8    4.49  -3.94]]
```

In [34]:
```
print(pca.explained_variance_ratio_)
```

```
[ 0.45  0.23  0.15  0.07  0.05]
```

Looking at the above, it can be obsereved that the first 5 components capture (explain) 95% of the variance in the data.

Now, we can redo the clustering, but this time in the lower dimensional space:

In [35]:
```
centroids_pca, clusters_pca = kMeans.kMeans(DTtrans, 3, kMeans.distCosine, kMe
ans.randCent)
```

```
Iteration  1
Iteration  2
Iteration  3
Iteration  4
Iteration  5
Iteration  6
```

In [36]:
```
print clusters_pca
```

```
[[ 0.    0.  ]
 [ 0.    0.  ]
 [ 0.    0.06]
 [ 2.    0.12]
 [ 0.    0.  ]
 [ 1.    0.03]
 [ 2.    0.12]
 [ 2.    0.02]
 [ 1.    0.21]
 [ 1.    0.14]
 [ 1.    0.2 ]
 [ 1.    0.21]
 [ 1.    0.1 ]
 [ 2.    0.16]
 [ 2.    0.12]]
```

Next, let's actually derive the principal components manaually using linear algebra rather than relying on the PCA package from sklearn:

First step is to obtain the covariance matrix:

In [38]:
```
meanVals = np.mean(DT, axis=0)
meanRemoved = DT - meanVals #remove mean
covMat = np.cov(meanRemoved, rowvar=0)

np.set_printoptions(precision=2,suppress=True,linewidth=100)
print covMat
```

```
[[ 179.7    38.44  -17.06  -50.7   -40.93   66.87  -60.9   -19.62  116.32  -5
9.49]
 [  38.44   67.26  -21.54  -19.81   -6.5    31.83  -55.7    38.13   27.67   2
7.04]
 [ -17.06  -21.54   51.78   31.1     9.36  -11.61   77.41   -8.72  -16.2
4.67]
 [ -50.7   -19.81   31.1    85.97   51.43    2.54   45.59   15.13  -41.97   4
7.47]
 [ -40.93   -6.5     9.36   51.43   80.57   -4.43   25.86   17.64  -35.07   7
4.14]
 [  66.87   31.83  -11.61    2.54   -4.43   72.97  -22.49   15.7    45.17   -
8.39]
 [ -60.9   -55.7    77.41   45.59   25.86  -22.49  162.03  -18.1   -50.37
7.87]
 [ -19.62   38.13   -8.72   15.13   17.64   15.7   -18.1    48.12  -19.18   4
9.06]
 [ 116.32   27.67  -16.2   -41.97  -35.07   45.17  -50.37  -19.18   93.92  -4
8.33]
 [ -59.49   27.04    4.67   47.47   74.14   -8.39    7.87   49.06  -48.33  10
2.26]]
```

In [48]:
```
import numpy.linalg as la
eigVals,eigVects = la.eig(np.mat(covMat))
```

In [49]:
```
print eigVals
```

```
[ 426.77  214.24  144.96   61.87   47.51   26.97   13.24    1.06    3.25
4.71]
```

In [50]:
```
print eigVects
```

```
[[-0.57 -0.09  0.46 -0.14 -0.26 -0.15  0.58  0.05 -0.1  -0.02]
 [-0.18  0.39  0.1   0.51 -0.12 -0.19 -0.22 -0.3  -0.35 -0.49]
 [ 0.18 -0.26  0.3   0.19 -0.11 -0.48 -0.11 -0.06  0.69 -0.2 ]
 [ 0.29  0.1   0.36 -0.35  0.53 -0.49 -0.02 -0.03 -0.36  0.03]
 [ 0.26  0.27  0.31 -0.47 -0.33  0.34 -0.05 -0.53  0.14 -0.11]
 [-0.21  0.13  0.43  0.13  0.56  0.51 -0.09  0.2   0.27 -0.21]
 [ 0.4  -0.51  0.42  0.37 -0.17  0.28  0.01 -0.01 -0.38  0.12]
 [ 0.06  0.38  0.08  0.41  0.14 -0.05  0.28 -0.33  0.17  0.66]
 [-0.42 -0.08  0.23 -0.11 -0.14 -0.07 -0.72 -0.01 -0.05  0.46]
 [ 0.28  0.51  0.21  0.03 -0.37 -0.04 -0.04  0.69  0.01  0.07]]
```

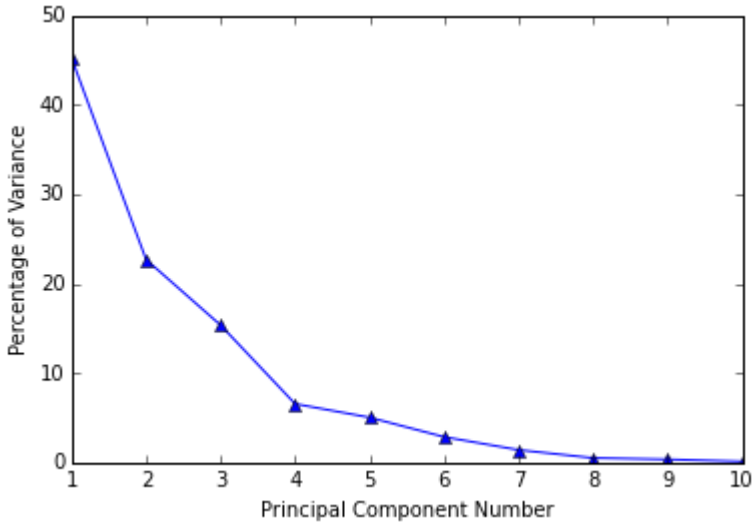
In [51]:
```
eigValInd = np.argsort(eigVals)  #sort, sort goes smallest to largest
eigValInd = eigValInd[::-1]   #reverse
sortedEigVals = eigVals[eigValInd]
print sortedEigVals
total = sum(sortedEigVals)
varPercentage = sortedEigVals/total*100
print varPercentage
```

```
[ 426.77  214.24  144.96   61.87   47.51   26.97   13.24    4.71    3.25
1.06]
[ 45.18  22.68  15.35   6.55   5.03   2.86   1.4    0.5    0.34   0.11]
```

We can plot the principal components based on the percentage of variance they capture:

In [53]:
```
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(range(1, 11), varPercentage[:10], marker='^')
plt.xlabel('Principal Component Number')
plt.ylabel('Percentage of Variance')
plt.show()
```



In [55]:
```
topNfeat = 5
topEigValInd = eigValInd[:topNfeat]  #cut off unwanted dimensions
reducedEigVects = eigVects[:,topEigValInd]   #reorganize eig vects largest to
 smallest
reducedDT = np.dot(meanRemoved, reducedEigVects)    #transform data into new d
imensions
print reducedDT
```

```
[[-25.45  -1.8    4.02  -3.12   0.24]
 [-23.78  -7.29  -0.53  -4.77  -6.32]
 [-15.03  -5.44 -10.09  -3.46  -2.87]
 [ -6.75  -4.39 -14.84  -3.08  -0.44]
 [-46.7    0.01  21.78   3.11   0.29]
 [ 27.41   7.93  12.27 -15.16 -13.47]
 [ 15.67 -21.74  -1.33  11.4   -5.81]
 [  8.28 -16.54  -6.94   5.39  -1.47]
 [ 29.51 -18.64  16.68   4.82   1.69]
 [ 11.41   0.44  15.03  -3.03  17.19]
 [  6.98  37.69   4.5   14.83  -5.5 ]
 [  5.97  13.91  -6.64   2.12   3.32]
 [ 12.36  10.4   -4.93 -11.5    5.51]
 [  1.67   2.69 -13.18  -2.04   3.72]
 [ -1.53   2.79 -15.8    4.49   3.94]]
```

Next, let's look at an application of Singular Value Decomposition. This time, we'll foucs on the term-document matrix in order to find themes based on combinations of terms.

In [56]:
```
u, s, vt = la.svd(TD, full_matrices=False)
```

In [57]:
```
print u
```

```
[[ 0.39 -0.6   0.22  0.17  0.22 -0.1  -0.59  0.07 -0.07  0.04]
 [ 0.3  -0.2  -0.33 -0.5   0.06 -0.17  0.21 -0.08 -0.57 -0.31]
 [ 0.2   0.16  0.33 -0.16  0.04 -0.4   0.   -0.76  0.23 -0.02]
 [ 0.37  0.27  0.    0.42 -0.49 -0.53  0.04  0.27 -0.16 -0.05]
 [ 0.32  0.23 -0.19  0.49  0.31  0.38  0.03 -0.22  0.   -0.52]
 [ 0.29 -0.23 -0.02  0.   -0.66  0.53  0.07 -0.31 -0.03  0.22]
 [ 0.36  0.38  0.62 -0.35  0.08  0.3   0.02  0.34 -0.1  -0.03]
 [ 0.21  0.05 -0.34 -0.38 -0.2  -0.02 -0.28  0.21  0.65 -0.33]
 [ 0.23 -0.44  0.15  0.11  0.15 -0.09  0.72  0.18  0.38 -0.02]
 [ 0.42  0.25 -0.42 -0.03  0.33  0.    0.02  0.01  0.07  0.68]]
```

In [58]:
```
print s
```

```
[ 93.97  77.25  54.14  29.74  26.27  19.76  13.75   9.41   7.88   3.88]
```

In [60]:
```
print np.diag(s)
```

```
[[ 93.97   0.     0.     0.     0.     0.     0.     0.     0.     0.  ]
 [  0.    77.25   0.     0.     0.     0.     0.     0.     0.     0.  ]
 [  0.     0.    54.14   0.     0.     0.     0.     0.     0.     0.  ]
 [  0.     0.     0.    29.74   0.     0.     0.     0.     0.     0.  ]
 [  0.     0.     0.     0.    26.27   0.     0.     0.     0.     0.  ]
 [  0.     0.     0.     0.     0.    19.76   0.     0.     0.     0.  ]
 [  0.     0.     0.     0.     0.     0.    13.75   0.     0.     0.  ]
 [  0.     0.     0.     0.     0.     0.     0.     9.41   0.     0.  ]
 [  0.     0.     0.     0.     0.     0.     0.     0.     7.88   0.  ]
 [  0.     0.     0.     0.     0.     0.     0.     0.     0.     3.88]]
```

In [64]:
```
originalTD = np.dot(u, np.dot(np.diag(s), vt))
print originalTD
```

```
[[ 24.  32.  12.   6.  43.   2.  -0.   3.   1.   6.   4.   0.   0.   0.   0.]
 [  9.   5.   5.   2.  20.  -0.   1.  -0.  -0.   0.  27.  14.   3.   2.  11.]
 [ -0.   3.  -0.  -0.   3.   7.  12.   4.  27.   4.  -0.   1.  -0.   0.  -0.]
 [  3.  -0.   0.   0.  -0.  16.   0.   2.  25.  23.   7.  12.  21.   3.   2.]
 [  1.  -0.  -0.  -0.  -0.  33.   2.  -0.   7.  12.  14.   5.  12.   4.   0.]
 [ 12.   2.   0.   0.  27.   0.   0.   0.   0.  22.   9.   4.  -0.   5.   3.]
 [ -0.  -0.   0.   0.  -0.  18.  32.  22.  34.  17.  -0.  -0.   0.  -0.  -0.]
 [  1.   0.  -0.   0.   2.   0.  -0.   0.   3.   9.  27.   7.   5.   4.   4.]
 [ 21.  10.  16.   7.  31.  -0.   0.   0.   0.   0.  -0.   0.   0.   1.   0.]
 [  2.  -0.   0.   2.  -0.  27.   4.   2.  11.   8.  33.  16.  14.   7.
3.]]
```

In [66]:
```
numDimensions = 3
u_ld = u[:, :numDimensions]
sigma = np.diag(s)[:numDimensions, :numDimensions]
vt_ld = vt[:numDimensions, :]
lowRankTD = np.dot(u_ld, np.dot(sigma, vt_ld))
```

In [67]:
```
np.set_printoptions(precision=2,suppress=True,linewidth=120)
print lowRankTD
```

```
[[ 25.33  22.89  13.5    6.13  45.48  -1.98   1.99   2.79   1.9    7.87   4.6
1.34  -1.82   1.01   2.14]
 [ 10.71   7.37   4.79   2.6   18.7    6.82  -5.26  -3.14  -3.04   6.47  21.6
9.52   6.96   4.02   4.41]
 [  1.46   2.05   0.6    0.29   2.07  10.24  12.93   8.54  19.25   9.57  -2.7
4  -0.     2.18   0.08  -0.94]
 [  1.25  -0.19  -0.45   0.22   1.03  20.3   10.26   6.46  20.37  14.86  15.4
9   8.63  10.26   3.55   2.17]
 [  0.3   -1.69  -1.06   0.02  -0.57  18.19   4.81   2.78  13.17  12.03  19.7
1  10.15  10.88   4.12   3.05]
 [ 12.74  10.59   6.33   3.07  22.51   4.64   0.76   1.09   3.03   7.25  10.8
1   4.81   3.12   2.17   2.49]
 [  0.58   1.9   -0.08   0.01  -0.06  20.55  25.06  16.42  37.57  18.2   -5.2
3   0.13   4.68   0.16  -1.94]
 [  2.26  -0.13   0.23   0.54   3.39   9.93  -3.09  -2.23   0.62   5.95  19.6
4   9.17   8.18   3.73   3.5 ]
 [ 17.34  15.78   9.34   4.21  31.25  -3.21   0.35   1.28  -0.65   4.05   1.8
8   0.19  -2.16   0.39   1.3 ]
 [  1.34  -2.11  -1.07   0.26   0.94  23.19   1.78   0.66  11.78  14.65  31.6
1  15.6   15.57   6.33   5.21]]
```

The VT matrix can be viewed as the new representation of documents in the lower dimensional space.

In [68]:
```
print vt_ld
```

```
[[ 0.24  0.18  0.1   0.06  0.41  0.39  0.18  0.12  0.37  0.35  0.41  0.21  0.
21  0.09  0.07]
 [-0.34 -0.32 -0.2  -0.08 -0.63  0.33  0.2   0.11  0.36  0.13  0.06  0.07  0.
15  0.02 -0.02]
 [ 0.07  0.14  0.06  0.02  0.13 -0.06  0.4   0.27  0.43  0.08 -0.62 -0.26 -0.
2  -0.1  -0.12]]
```

In information retrieval, a query is compared to documents using vector-space similarity between the query vector and document vectors. In the lower dim. space, this can be achieved by first mapping the query to lower dim. space, and then comparing it to docs in the lower dim. space.

In [72]:
```
queryVector = np.array([0,0,1,5,4,0,6,0,0,2])
lowDimQuery = np.dot(la.inv(sigma), np.dot(u_ld.T, queryVector))
print lowDimQuery
```

```
[ 0.07  0.07  0.05]
```

In [74]:
```
# Compute Cosine sim between the query and docs in the lower dimensional space

qNorm = lowDimQuery / la.norm(lowDimQuery)
```

In [77]:
```
docNorm = np.array([vt_ld[:,i]/la.norm(vt_ld[:,i]) for i in range(len(vt_ld[0
]))])
print docNorm
```

```
[[ 0.57 -0.81  0.16]
 [ 0.47 -0.8   0.37]
 [ 0.45 -0.85  0.27]
 [ 0.55 -0.82  0.15]
 [ 0.54 -0.83  0.17]
 [ 0.75  0.64 -0.12]
 [ 0.37  0.41  0.83]
 [ 0.38  0.33  0.86]
 [ 0.55  0.54  0.64]
 [ 0.92  0.33  0.2 ]
 [ 0.55  0.08 -0.83]
 [ 0.61  0.2  -0.77]
 [ 0.64  0.47 -0.61]
 [ 0.64  0.16 -0.75]
 [ 0.53 -0.13 -0.84]]
```

In [78]:
```
sims = np.dot(qNorm, docNorm.T)
# return indices of the docs in decending order of similarity to the query
simInds = sims.argsort()[::-1]
for i in simInds:
    print "Cosine similarity between Document %d and the query is: %.4f" %(i,s
ims[i])
```

```
Cosine similarity between Document 8 and the query is: 0.9693
Cosine similarity between Document 9 and the query is: 0.8843
Cosine similarity between Document 6 and the query is: 0.8604
Cosine similarity between Document 5 and the query is: 0.8362
Cosine similarity between Document 7 and the query is: 0.8309
Cosine similarity between Document 12 and the query is: 0.4358
Cosine similarity between Document 13 and the query is: 0.1840
Cosine similarity between Document 11 and the query is: 0.1767
Cosine similarity between Document 10 and the query is: 0.0404
Cosine similarity between Document 1 and the query is: -0.0555
Cosine similarity between Document 0 and the query is: -0.0810
Cosine similarity between Document 3 and the query is: -0.1039
Cosine similarity between Document 14 and the query is: -0.1097
Cosine similarity between Document 4 and the query is: -0.1119
Cosine similarity between Document 2 and the query is: -0.1375
```

In [79]:
```
centroids_svd, clusters_svd = kMeans.kMeans(vt_ld.T, 3, kMeans.distCosine, kMe
ans.randCent)
```

```
Iteration  1
Iteration  2
Iteration  3
```

In [80]:
```
print clusters_svd
```

```
[[ 0.    0.  ]
 [ 0.    0.  ]
 [ 0.    0.  ]
 [ 0.    0.  ]
 [ 0.    0.  ]
 [ 2.    0.05]
 [ 2.    0.01]
 [ 2.    0.01]
 [ 2.    0.  ]
 [ 2.    0.01]
 [ 1.    0.  ]
 [ 1.    0.  ]
 [ 1.    0.  ]
 [ 1.    0.  ]
 [ 1.    0.  ]]
```

In [81]:
```
print "\t\tCluster0\tCluster1\tCluster2"
for i in range(numDimensions):
    print "Theme %d\t\t%.4f\t\t%.4f\t\t%.4f" %(i,centroids_svd[0][i],centroids
_svd[1][i],centroids_svd[2][i])
```

```
                Cluster0        Cluster1        Cluster2
Theme 0         0.1994          0.1982          0.2813
Theme 1         -0.3141         0.0570          0.2243
Theme 2         0.0844          -0.2611         0.2239
```

In [ ]:
```

```