11-Dimensionality-Reduction-SVD-II

October 18, 2016

1 Dimensionality Reduction - SVD II

In the last lecture we learned about the SVD as a tool for constructing low-rank matrices.

Today we'll look at it as a way to transform our data objects.

As a reminder, here is what the SVD looks like:

$$\text{objects}\left\{\begin{array}{c} \\ \\ \\ \\ \end{array}\right.\overbrace{\begin{bmatrix} \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ \mathbf{a_1} & \mathbf{a_2} & \dots & \mathbf{a_n} \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \end{bmatrix}}^{\text{features}} = \overbrace{\begin{bmatrix} \vdots & \vdots \\ \vdots & \vdots \\ \sigma_1\mathbf{u}_1 & \sigma_k\mathbf{u}_k \\ \vdots & \vdots \\ \vdots & \vdots \end{bmatrix}}^{k} \times \begin{bmatrix} \dots & \dots & \mathbf{v}_1 & \dots & \dots \\ \dots & \dots & \mathbf{v}_k & \dots & \dots \end{bmatrix}$$

$$A = U\Sigma V^T$$

Notice that U contains a row for each object.

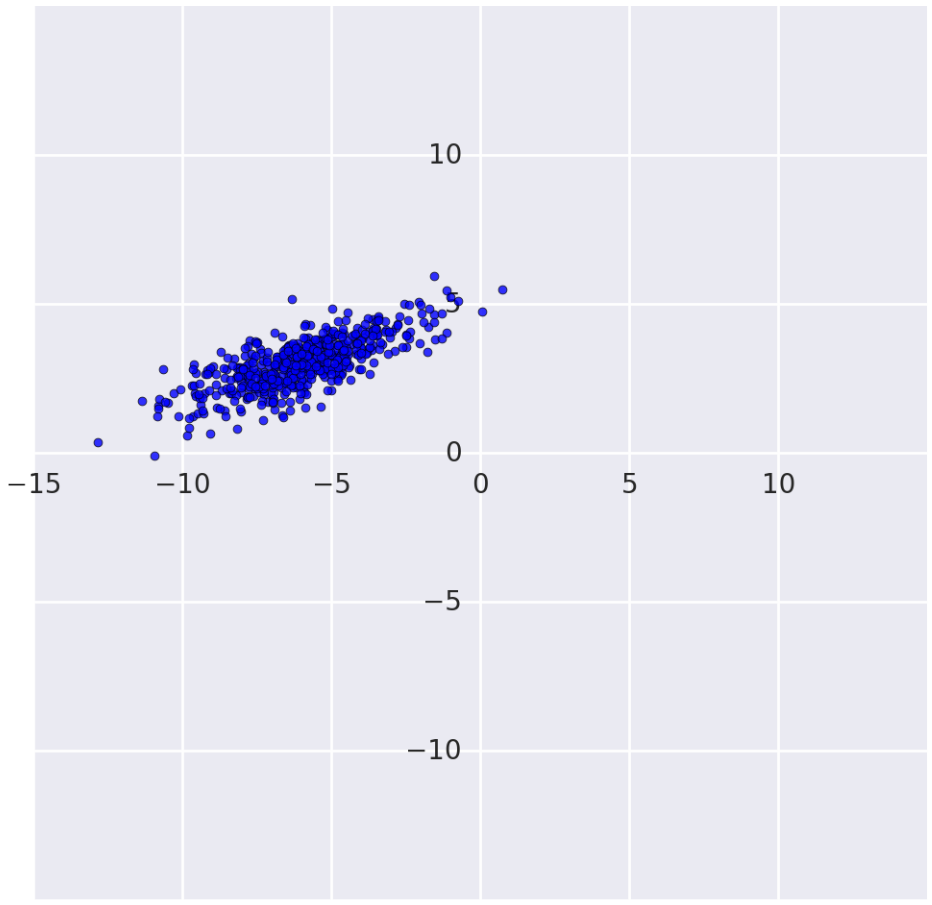
In a sense we have transformed objects from an n dimensional space to a k dimensional space, where k is (probably much) smaller than n.

This suggests an idea: is there an **optimal** transformation of the data into k dimensions?

What would that mean?

One criterion: a transformation that captures the maximum **variance** in the data.

```
In [88]: n_samples = 500
         C = np.array([[0.1, 0.6], [2., .6]])
         X = np.random.randn(n_samples, 2) @ C + np.array([-6, 3])
         ax = ut.plotSetup(-10,10,-10,10,(6,6))
         ut.centerAxes(ax)
         plt.axis('equal')
         _ = plt.scatter(X[:, 0], X[:, 1], s=10, alpha=0.8)
```



What would happen if we used SVD, and kept only rank-1 approximation to the data?

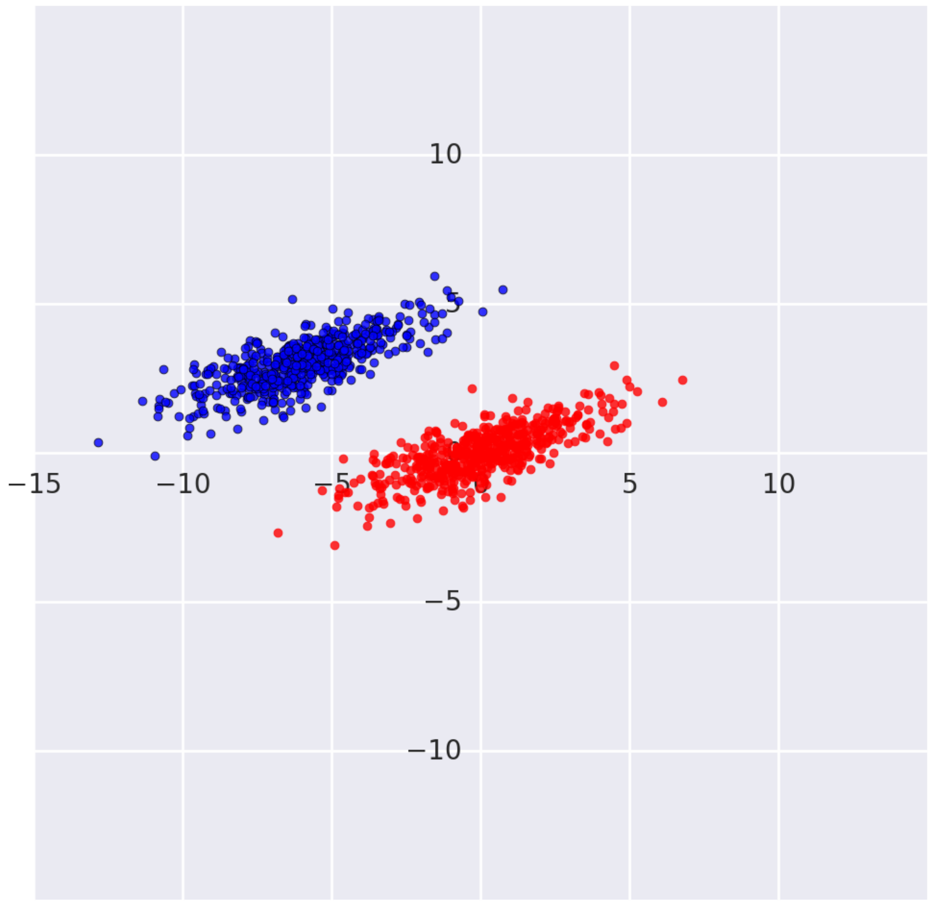
This would be the 1-D **subspace** that approximates the data best.

However the variance in the data is defined with respect to the data mean, so we need to mean-center the data first, before using SVD.

That is, SVD in this case finds the best 1-D subspace, not the best line though the data (which might not pass through the origin).

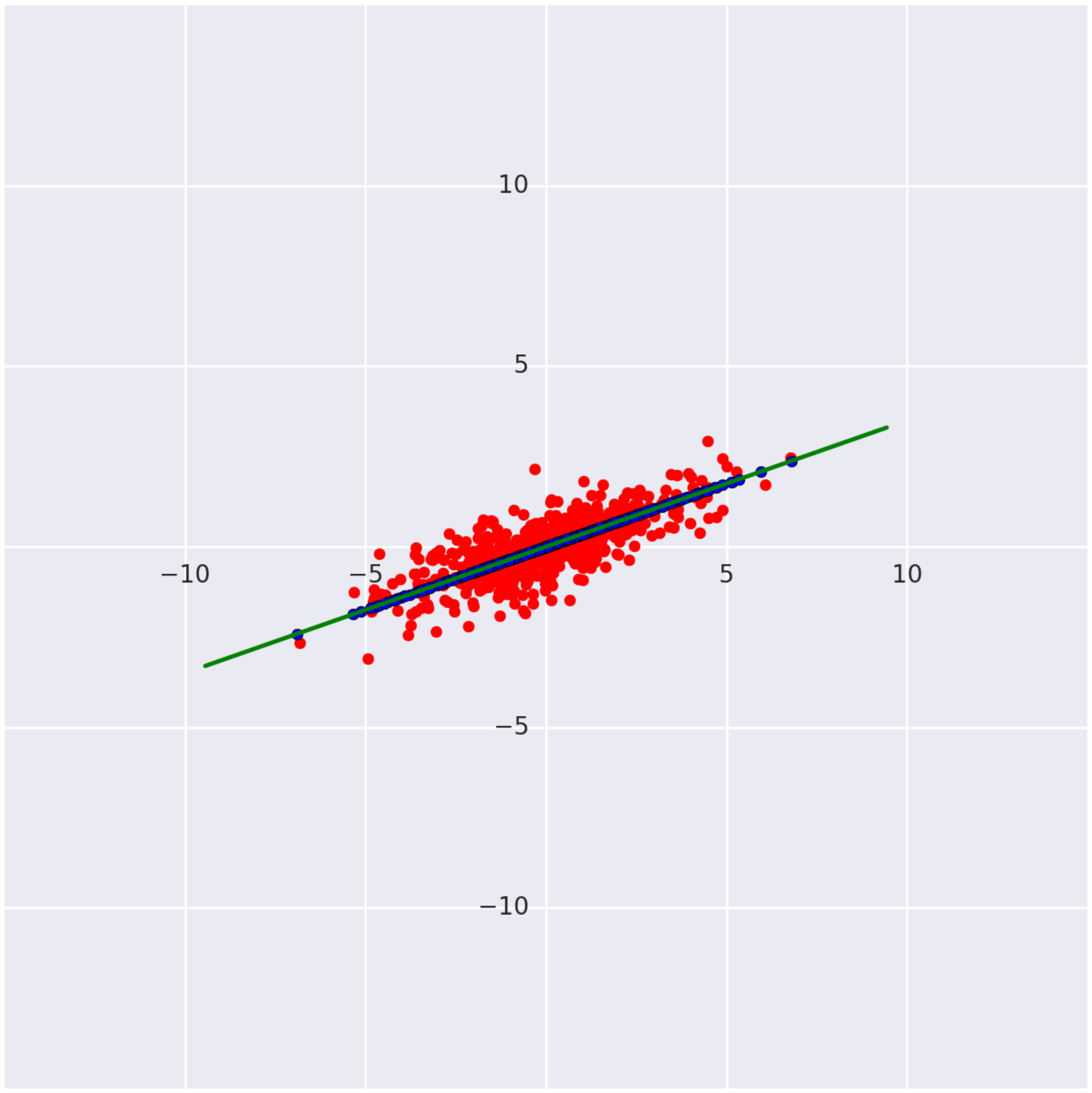
So to capture the best line through the data, we first move the data points to the origin:

```
In [89]: Xc = X - np.mean(X,axis=0)
         ax = ut.plotSetup(-10,10,-10,10,(6,6))
         ut.centerAxes(ax)
         plt.axis('equal')
         plt.scatter(X[:, 0], X[:, 1], s=10, alpha=0.8)
         _ = plt.scatter(Xc[:, 0], Xc[:, 1], s=10, alpha=0.8, color='r')
```



Now let's construct the best 1-D approximation of the mean-centered data:

```
In [123]: u, s, vt = np.linalg.svd(Xc,full_matrices=False)
          scopy = s.copy()
          scopy[1] = 0.
          reducedX = u @ np.diag(scopy) @ vt
          ax = ut.plotSetup(-10,10,-10,10,(8,8))
          ut.centerAxes(ax)
          plt.axis('equal')
          plt.scatter(Xc[:,0],Xc[:,1], color='r')
          plt.scatter(reducedX[:,0], reducedX[:,1])
          endpoints = np.array([[-10],[10]]) @ vt[[0],:]
          _ = plt.plot(endpoints[:,0], endpoints[:,1], 'g-')
```



This method is called **Principal Component Analysis.**

In summary, PCA consists of:

1. Mean center the data, and
2. Reduce the dimension of the mean-centered data via SVD.

This is equivalent to projecting the data onto the subspace that captures the maximum variance in the data.

It winds up constructing the **best low dimensional approximation of the data.**

What are “principal components”?

These are nothing more than the columns of U (or the rows of V^T). Because they capture the direction of maximum variation, they are called “principal” components.

1.1 Uses of PCA/SVD

There are many uses of PCA (and SVD).

We'll cover three of the main uses:

1. Visualization
2. Denoising
3. Anomaly Detection

As already mentioned, SVD is also useful for data compression – we won't discuss it in detail, but it is the principle behind audio and video compression (MP3s, HDTV, etc).

1.2 Visualization and Denoising – Extended Example.

We will study both visualization and denoising in the context of text processing.

As we have seen, a common way to work with documents is using the bag-of-words model (perhaps considering n-grams), which results in a term-document matrix.

Entries in the matrix are generally TF-IDF scores.

Often, terms are correlated – they appear together in combinations that suggest a certain "concept".

That is, term-document matrices often show low effective rank – many columns can be approximated as combinations of other columns.

When PCA is used for dimensionality reduction of documents, it tends to to extract these "concept" vectors.

The application of PCA to term-document matrices is called **Latent Semantic Analysis (LSA).**

Among other benefits, LSA can improve the performance of clustering of documents.

This happens because the important concepts are captured in the most significant principal components.

1.3 Data: 20 Newsgroups

```
In [91]: from sklearn.datasets import fetch_20newsgroups

         categories = ['comp.os.ms-windows.misc', 'sci.space','rec.sport.baseball']
         news_data = fetch_20newsgroups(subset='train', categories=categories)

In [92]: print(news_data.target_names)
         print(news_data.target)

         # import nltk
         # nltk.download()
```

```
['comp.os.ms-windows.misc', 'rec.sport.baseball', 'sci.space']
[2 0 0 ..., 2 1 2]
```

1.3.1 Basic Clustering

To get started, let's compute tf-idf scores.
  Notice that we will let the tokenizer compute n-grams for $n=$1 and 2.
  An n-gram is a set of n consecutive terms.
  We'll compute a document-term matrix `dtm`.

```
In [124]: from sklearn.feature_extraction.text import TfidfVectorizer

          vectorizer = TfidfVectorizer(stop_words='english', min_df=4,max_df=0.8)
          dtm = vectorizer.fit_transform(news_data.data)

In [125]: print(type(dtm), dtm.shape)
          terms = vectorizer.get_feature_names()
          print(terms)
```

```
<class 'scipy.sparse.csr.csr_matrix'> (1781, 9410)
['00', '000', '0005', '0062', '0096b0f0', '00bjgood', '00mbstultz', '01', '0114', '
```

  As a comparison case, let's first cluster the documents using the raw tf-idf scores.
  (This is without any use of PCA, and so includes lots of noisy or meaningless terms.)

```
In [126]: from sklearn.cluster import KMeans
          k = 3
          kmeans = KMeans(n_clusters=k, init='k-means++', max_iter=100, n_init=10,r
          kmeans.fit_predict(dtm)
          centroids = kmeans.cluster_centers_
          labels = kmeans.labels_
          error = kmeans.inertia_
```

  Let's evaluate the clusters. We'll assume that the newgroup the article came from is the 'ground truth.'

```
In [96]: import sklearn.metrics as metrics
         ri = metrics.adjusted_rand_score(labels,news_data.target)
         ss = metrics.silhouette_score(dtm,kmeans.labels_,metric='euclidean')
         print('Rand Index is {}'.format(ri))
         print('Silhouette Score is {}'.format(ss))
```

```
Rand Index is 0.6995017259053622
Silhouette Score is 0.009276829181645699
```

1.3.2 Improvement: Stemming

One source of noise that we can eliminate (before we use LSA) comes from word endings.
  For example: a Google search on 'run' will return web pages on 'running.'
  This is useful, because the difference between 'run' and 'running' in practice is not enough to matter.

The usual solution taken is to simply 'chop off' the part of the word that indicates a variation from the base word.

(For those of you who studied Latin or Greek, this will sound familiar – we are removing the 'inflection.')

The process is called 'stemming.'

A very good stemmer is the "Snowball" stemmer.

You can read more at http://www.nltk.org and http://www.nltk.org/howto/stem.html.

Installation Note: From a cell you need to call `nltk.download()` and select the appropriate packages from the interface that appears. In particular you need to download: `stopwords` from *corpora* and `punkt` and `snowball_data` from *models*.

Let's stem the data using the Snowball stemmer:

```
In [97]: from nltk.stem.snowball import SnowballStemmer
         from nltk.tokenize import word_tokenize, sent_tokenize


         stemmed_data = [" ".join(SnowballStemmer("english", ignore_stopwords=True)
                  for sent in sent_tokenize(message)
                 for word in word_tokenize(sent))
                 for message in news_data.data]

         # stemmed_data = news_data.data
```

```
In [127]: dtm = vectorizer.fit_transform(stemmed_data)
          terms = vectorizer.get_feature_names()
          print(terms)
```

```
['00', '000', '0005', '0062', '0096b0f0', '00bjgood', '00mbstultz', '01', '0114', '
```

And now let's see how well we can cluster on the stemmed data.

```
In [129]: from sklearn.cluster import KMeans
          k = 3
          kmeans = KMeans(n_clusters=k, init='k-means++', max_iter=100, n_init=10,r
          kmeans.fit_predict(dtm)
          centroids = kmeans.cluster_centers_
          labels = kmeans.labels_
          error = kmeans.inertia_
```

```
In [130]: import sklearn.metrics as metrics
          ri = metrics.adjusted_rand_score(labels,news_data.target)
          ss = metrics.silhouette_score(dtm,kmeans.labels_,metric='euclidean')
          print('Rand Index is {}'.format(ri))
          print('Silhouette Score is {}'.format(ss))
```

```
Rand Index is 0.8464232080160293
Silhouette Score is 0.010811400170620348
```

So the Rand Index went from 0.70 to 0.84 as a result of stemming.

1.4 Demonstrating PCA

OK. Now, let's apply PCA.

Our data matrix is in sparse form.

First, we mean center the data. Note that `vectors` is a sparse matrix, but once it is mean centered it is not sparse any longer.

```
In [131]: dtm_dense = dtm.todense()
          centered_dtm = dtm_dense - np.mean(dtm_dense, axis=0)
          np.sum(centered_dtm,axis=0)[:,:10]
```

```
Out[131]: matrix([[ -2.22651758e-15,  -1.82579646e-15,   1.43851944e-15,
                     3.68368530e-15,   5.22748077e-15,  -1.69146381e-15,
                    -2.70269918e-15,   1.06026299e-14,   6.43365569e-15,
                     4.45254727e-15]])
```

```
In [132]: u, s, vt = np.linalg.svd(centered_dtm)
```

Note that if you have sparse data, you may want to use `scipy.sparse.linalg.svds()` and for large data it may be advantageous to use `sklearn.decomposition.TruncatedSVD()`.

$$\text{objects}\left\{\vphantom{\begin{bmatrix}\vdots\\\vdots\\\mathbf{a_1}\\\vdots\\\vdots\end{bmatrix}}\right. \overbrace{\begin{bmatrix} \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ \mathbf{a_1} & \mathbf{a_2} & \dots & \mathbf{a_n} \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \end{bmatrix}}^{\text{features}} = \overbrace{\begin{bmatrix} \vdots & \vdots \\ \vdots & \vdots \\ \sigma_1\mathbf{u}_1 & \sigma_k\mathbf{u}_k \\ \vdots & \vdots \\ \vdots & \vdots \end{bmatrix}}^{k} \times \begin{bmatrix} \dots & \dots & \mathbf{v}_1 & \dots & \dots \\ \dots & \dots & \mathbf{v}_k & \dots & \dots \end{bmatrix}$$

$$A = U\Sigma V^T$$

The principal components (rows of V^T) encode the extracted concepts.

Each LSA **concept** is a linear combination of words.

```
In [133]: pd.DataFrame(vt,columns=vectorizer.get_feature_names())
```

```
Out[133]:           00        000       0005       0062   0096b0f0   00bjgood   00mbstu
          0   0.007885   0.012340   0.000591   0.005538   0.001028   0.002067      0.002
          1  -0.005887   0.009505   0.002096  -0.010693  -0.001650  -0.003485     -0.002
          2  -0.012564  -0.012013  -0.002483   0.001447   0.000431   0.000051      0.000
          3   0.013521   0.017799   0.003623   0.000993   0.003338  -0.000427      0.000
          4  -0.002048  -0.007633  -0.005712   0.001726   0.000315   0.000814      0.000
          5  -0.002852  -0.017763  -0.001141   0.021928   0.003218   0.000670      0.000
          6   0.005453   0.009738  -0.002027  -0.044424  -0.003053  -0.000250     -0.002
          7  -0.017610  -0.001251   0.002503   0.041706   0.001032   0.000374      0.002
          8  -0.005073   0.009931  -0.001484  -0.011088   0.002226  -0.000203     -0.000
          9  -0.012345   0.004288  -0.002194   0.057862  -0.008259  -0.004871      0.002
```

```
10    0.019831  0.009635  0.005410 -0.002062 -0.001075 -0.000955   -0.000
11   -0.011970 -0.011849  0.004851 -0.000717 -0.000383  0.001779   -0.001
12    0.045029  0.003507 -0.001448  0.009667  0.000066  0.007111   -0.005
13    0.001567  0.018366  0.010059 -0.006225 -0.000116 -0.001878   -0.001
14    0.020302 -0.004063  0.001184 -0.001073  0.002073 -0.000296    0.000
15   -0.014156 -0.029345  0.001914 -0.004363  0.002384  0.000032   -0.000
16    0.018073  0.021215  0.007850 -0.003000 -0.001866  0.001739    0.000
17   -0.037704 -0.007862 -0.005199  0.009553 -0.001858 -0.001760   -0.001
18    0.062237  0.015365  0.003215 -0.001070  0.002633  0.004465   -0.001
19    0.009783  0.013588  0.003869  0.010904 -0.007642 -0.006068   -0.004
20   -0.002378 -0.027415  0.000371 -0.005334  0.002974  0.001871   -0.000
21    0.060240 -0.018972 -0.011127 -0.010881  0.002168  0.001426    0.003
22   -0.015973  0.007927 -0.007186 -0.000617  0.005855  0.001812   -0.001
23   -0.043373  0.021941  0.017768 -0.009477  0.004274  0.005240    0.007
24   -0.044484 -0.014512 -0.015176 -0.017307 -0.002763  0.002575    0.009
25    0.016280 -0.016198 -0.015999 -0.004302 -0.001753  0.001907    0.001
26    0.029900  0.033223  0.018585 -0.005890 -0.001656  0.001461    0.004
27   -0.046834  0.036536  0.016103 -0.000384 -0.010913 -0.010460   -0.004
28    0.031240  0.029015  0.006819 -0.011662 -0.002021 -0.007529    0.007
29   -0.055877 -0.010663 -0.001332  0.005908  0.010656 -0.004766   -0.005
...        ...       ...       ...       ...       ...       ...
8029 -0.000263  0.000209  0.013095 -0.000215  0.009371 -0.000752    0.005
8030 -0.001002  0.000254 -0.020639 -0.006260  0.013547 -0.009611   -0.005
8031 -0.000147  0.000114 -0.000661  0.000133 -0.001181 -0.000275   -0.000
8032  0.000167 -0.001747 -0.000373 -0.000336 -0.000963  0.000043    0.000
8033  0.000121 -0.000070 -0.000633 -0.000037 -0.001104 -0.000058   -0.000
8034 -0.000092 -0.001500 -0.000376 -0.000294 -0.001443 -0.000127    0.000
8035 -0.000357 -0.000269 -0.000119  0.000046 -0.000993 -0.000017    0.000
8036 -0.000077 -0.001065 -0.000245 -0.000174 -0.000741 -0.000111    0.000
8037 -0.000492  0.000130  0.000023  0.000134 -0.000247 -0.000067    0.000
8038 -0.000013 -0.000237 -0.000121 -0.000058 -0.000560 -0.000031   -0.000
8039 -0.000195 -0.000011 -0.000084  0.000044 -0.000574  0.000015   -0.000
8040 -0.000007 -0.000260 -0.000014 -0.000144 -0.000471 -0.000010    0.000
8041 -0.000189  0.000567 -0.000693  0.000183 -0.001059 -0.000426    0.000
8042  0.000234 -0.000326 -0.004527 -0.009616 -0.005967  0.003442   -0.000
8043 -0.004218  0.002266 -0.007490  0.003980  0.018893  0.009800    0.002
8044  0.005343  0.000723 -0.000916 -0.001711  0.009222 -0.010413   -0.006
8045 -0.000114  0.000632  0.000053 -0.000074 -0.000981 -0.000284   -0.000
8046  0.000268  0.000128  0.000030 -0.000650 -0.001435  0.000023    0.000
8047  0.000009 -0.000829 -0.000195 -0.000305 -0.000918 -0.000081    0.000
8048 -0.000061 -0.000189 -0.000031 -0.000036 -0.000904 -0.000079   -0.000
8049  0.000088 -0.000320 -0.000039 -0.000114 -0.000596 -0.000038   -0.000
8050  0.000054 -0.000297 -0.000031 -0.000089 -0.000571 -0.000039   -0.000
8051  0.000050 -0.000287 -0.000171  0.000020 -0.000759 -0.000241   -0.000
8052 -0.000030 -0.000136  0.000137 -0.000189 -0.000315 -0.000089    0.000
8053  0.000248 -0.000303 -0.000460 -0.000385 -0.000864  0.000111    0.000
8054 -0.000006 -0.000261  0.000193 -0.000475 -0.004121 -0.000505   -0.000
8055  0.000111  0.000156 -0.000093 -0.000113 -0.000206 -0.000042    0.000
```

```
8056 -0.000038  0.000810 -0.000190  0.000016 -0.000705 -0.000188    -0.000
8057  0.000076  0.000452 -0.000190 -0.000142 -0.000811 -0.000052    -0.000
8058 -0.000362 -0.001279 -0.001475 -0.000446 -0.004056  0.005803     0.000

            01      0114      01wb    ...          zri          zrlk  \
0     0.005580  0.001241  0.000815    ...    -0.000028 -2.473297e-05
1     0.002123 -0.003395  0.002451    ...    -0.000015 -1.318560e-05
2    -0.006843  0.000692 -0.001129    ...    -0.000095 -8.662241e-05
3     0.011659  0.002225  0.001975    ...     0.000205  1.864419e-04
4    -0.002988 -0.000075  0.001228    ...    -0.000245 -2.241907e-04
5    -0.002937  0.002556 -0.001191    ...     0.000376  3.443428e-04
6     0.002229 -0.002426  0.001412    ...     0.000020  2.122207e-05
7    -0.003667 -0.001127 -0.000390    ...    -0.000021 -2.035374e-05
8     0.001156  0.004408  0.003383    ...     0.000161  1.455370e-04
9    -0.007999 -0.007097  0.002261    ...    -0.000257 -2.306489e-04
10    0.003255  0.000781 -0.000850    ...     0.000312  2.835621e-04
11   -0.011424 -0.002380 -0.003327    ...    -0.000001 -3.590004e-06
12    0.011960 -0.012471  0.001854    ...    -0.000015 -1.258691e-05
13    0.001753  0.003740  0.000567    ...     0.000488  4.386558e-04
14    0.004839  0.002087  0.000371    ...     0.000088  8.130063e-05
15   -0.011320  0.003468 -0.002251    ...     0.000125  1.121906e-04
16   -0.001614 -0.008173 -0.000172    ...     0.000104  9.537512e-05
17    0.004476  0.000327  0.000162    ...     0.000022  2.202291e-05
18    0.022621  0.002308  0.000698    ...     0.000490  4.438655e-04
19    0.009096  0.000170  0.001385    ...     0.000212  1.942552e-04
20    0.006061 -0.000789  0.000441    ...     0.000003  1.604733e-06
21    0.009050  0.002282 -0.001160    ...    -0.000072 -6.429727e-05
22    0.002429 -0.002412 -0.001009    ...     0.000140  1.280728e-04
23   -0.002094  0.002038  0.000220    ...    -0.000014 -1.561540e-05
24   -0.009789  0.000585 -0.001681    ...     0.000123  1.158487e-04
25   -0.004250 -0.008427  0.000203    ...    -0.000113 -9.804686e-05
26    0.021854 -0.004961  0.001028    ...     0.000587  5.315453e-04
27   -0.012652  0.000642  0.000813    ...    -0.000413 -3.790918e-04
28    0.018149 -0.003451  0.000777    ...     0.000471  4.295199e-04
29   -0.015745  0.000011 -0.001474    ...     0.000071  6.544670e-05
...        ...       ...       ...    ...          ...           ...
8029 -0.000137  0.000472 -0.000137    ...     0.000046  3.900438e-05
8030  0.001819 -0.003077  0.001185    ...     0.000086  1.220394e-04
8031  0.000619 -0.000188  0.000121    ...     0.000108  8.551956e-05
8032  0.000784  0.000135 -0.000665    ...    -0.000088 -1.193983e-04
8033 -0.000283 -0.000139 -0.000089    ...     0.000070  6.382655e-05
8034  0.001571 -0.000101 -0.000429    ...    -0.000098 -1.530224e-04
8035  0.000744 -0.000064 -0.000155    ...    -0.000060 -9.516217e-05
8036  0.001391 -0.000024 -0.000360    ...    -0.000106 -1.920106e-04
8037  0.000650  0.000078 -0.000049    ...     0.000043  3.757474e-05
8038  0.000162 -0.000064 -0.000167    ...    -0.000082 -5.174702e-05
8039  0.000259 -0.000080 -0.000046    ...    -0.000013  6.343726e-06
8040  0.000044 -0.000059 -0.000150    ...    -0.000002  6.290872e-06
```

```
8041  0.001358 -0.000491 -0.000196    ...     0.000170  1.699829e-04
8042  0.004449 -0.001765 -0.001652    ...    -0.000045 -2.494216e-05
8043  0.007650 -0.001968 -0.001382    ...    -0.000045 -7.882045e-07
8044 -0.014399 -0.004216 -0.007087    ...    -0.000003 -3.167073e-05
8045 -0.000186 -0.000211  0.000386    ...     0.000223  1.898216e-04
8046 -0.001995 -0.000033 -0.000357    ...     0.000019  8.796865e-05
8047  0.000611 -0.000048 -0.000369    ...    -0.000017 -4.050967e-05
8048  0.000323 -0.000066 -0.000057    ...    -0.000175 -3.411776e-04
8049  0.000352 -0.000014 -0.000104    ...     0.999358 -5.001234e-04
8050  0.000348 -0.000013 -0.000098    ...    -0.000499  9.995653e-01
8051 -0.000460  0.000113  0.000138    ...     0.000112  1.098208e-04
8052  0.000079 -0.000049 -0.000052    ...     0.000056  3.593363e-05
8053 -0.000461 -0.000012 -0.000503    ...     0.000146  9.406358e-05
8054  0.000725 -0.000354  0.000124    ...    -0.001385 -1.950026e-03
8055 -0.000599 -0.000001 -0.000012    ...     0.000111  1.006481e-04
8056 -0.000302 -0.000118  0.000284    ...     0.000249  2.114462e-04
8057 -0.000779 -0.000134  0.000032    ...     0.000195  1.789853e-04
8058  0.001322  0.001395 -0.000556    ...    -0.000024 -1.116160e-06

            zs        zt        zu        zv        zw        zx        z
0    -0.000194 -0.000025 -0.000129 -0.000208 -0.000087 -0.000151 -0.00011
1    -0.000048 -0.000013 -0.000042 -0.000100 -0.000025 -0.000063 -0.00004
2    -0.000270 -0.000087 -0.000253 -0.000578 -0.000134 -0.000294 -0.00020
3     0.000452  0.000172  0.000465  0.001143  0.000221  0.000510  0.00035
4    -0.000611 -0.000208 -0.000581 -0.001386 -0.000283 -0.000620 -0.00044
5     0.000675  0.000256  0.000682  0.001909  0.000282  0.000634  0.00047
6     0.000356  0.000084  0.000270  0.000174  0.000124  0.000240  0.00021
7     0.000019 -0.000020 -0.000028 -0.000118 -0.000001 -0.000052 -0.00001
8     0.000410  0.000124  0.000396  0.000966  0.000216  0.000492  0.00031
9    -0.000378 -0.000144 -0.000399 -0.001358 -0.000203 -0.000440 -0.00029
10    0.000649  0.000275  0.000737  0.001835  0.000365  0.000857  0.00056
11   -0.000013 -0.000025 -0.000043 -0.000032 -0.000008 -0.000068 -0.00002
12   -0.000187 -0.000056 -0.000162 -0.000194 -0.000100 -0.000234 -0.00015
13    0.001237  0.000355  0.001104  0.002834  0.000614  0.001285  0.00088
14    0.000304  0.000108  0.000395  0.000619  0.000186  0.000466  0.00030
15    0.000116  0.000034  0.000141  0.000593  0.000059  0.000151  0.00008
16    0.000281  0.000077  0.000248  0.000616  0.000134  0.000304  0.00019
17    0.000313  0.000061  0.000234  0.000242  0.000139  0.000278  0.00019
18    0.000996  0.000369  0.001029  0.002814  0.000527  0.001213  0.00080
19    0.000703  0.000212  0.000634  0.001418  0.000360  0.000782  0.00053
20    0.000411  0.000089  0.000307  0.000247  0.000204  0.000389  0.00028
21   -0.000389 -0.000097 -0.000347 -0.000578 -0.000208 -0.000461 -0.00031
22    0.000152  0.000073  0.000234  0.000733  0.000091  0.000264  0.00015
23   -0.000220 -0.000096 -0.000268 -0.000232 -0.000128 -0.000335 -0.00020
24    0.000491  0.000141  0.000490  0.000882  0.000257  0.000559  0.00038
25   -0.000081 -0.000015 -0.000081 -0.000508 -0.000041 -0.000098 -0.00006
26    0.000992  0.000386  0.001053  0.003252  0.000526  0.001284  0.00082
27   -0.000952 -0.000323 -0.000909 -0.002429 -0.000473 -0.001051 -0.00070
```

```
28    0.001276  0.000399  0.001190  0.002877  0.000645  0.001415  0.00095
29   -0.000046  0.000017 -0.000013  0.000297 -0.000028 -0.000003 -0.00002
...        ...       ...       ...       ...       ...       ...       ..
8029  0.000258  0.000172 -0.000166  0.000790  0.000146 -0.000212  0.00000
8030 -0.000231  0.000435  0.000381  0.000956  0.000076 -0.000605 -0.00010
8031 -0.001859  0.000723 -0.000422  0.000533 -0.000405 -0.002031 -0.00047
8032 -0.000922 -0.000523 -0.003412  0.000497 -0.000315  0.000376 -0.00002
8033 -0.001567  0.000866 -0.001768  0.000733 -0.000740 -0.000525 -0.00069
8034 -0.001050 -0.001155 -0.001834 -0.001009 -0.000016  0.000207  0.00055
8035  0.000164  0.000046  0.000203 -0.000194  0.000190  0.000191  0.00012
8036 -0.000303 -0.000607 -0.001053 -0.001417  0.000164 -0.000156  0.00045
8037  0.000319  0.000117  0.000277  0.000655  0.000074  0.000063  0.00010
8038  0.000152  0.000175  0.000564  0.000719 -0.000006  0.000189 -0.00012
8039  0.000126  0.000127  0.000364  0.000678  0.000089  0.000073 -0.00004
8040  0.000077 -0.000544 -0.000129 -0.000173 -0.000128  0.000263 -0.00026
8041 -0.001423  0.000140 -0.000543  0.000673 -0.000549 -0.001243 -0.00068
8042  0.000044 -0.000126 -0.000216  0.000025 -0.000028  0.000728  0.00010
8043 -0.000251 -0.000302 -0.000196  0.000618 -0.000365  0.000351 -0.00023
8044  0.000376  0.000122 -0.000113  0.000120  0.000342  0.000501  0.00043
8045 -0.000969 -0.001248  0.000661 -0.001325 -0.000316 -0.002343 -0.00087
8046 -0.001056 -0.001461 -0.003491 -0.001071 -0.001789  0.000672 -0.00216
8047 -0.000516 -0.000990 -0.001571 -0.000427 -0.000303 -0.000025 -0.00018
8048  0.000045  0.000113  0.000370 -0.004201  0.000105  0.000171  0.00014
8049  0.000085  0.000053  0.000149 -0.001389  0.000120  0.000268  0.00021
8050  0.000075  0.000030  0.000106 -0.001957  0.000109  0.000225  0.00019
8051  0.997615 -0.000059 -0.000708  0.000005 -0.000667 -0.001047 -0.00070
8052 -0.000052  0.998834 -0.000081 -0.001132 -0.000147 -0.000170 -0.00027
8053 -0.000715 -0.000100  0.995374  0.000626 -0.001079 -0.000732 -0.00118
8054 -0.000299 -0.001182  0.000910  0.976015 -0.000068  0.000086 -0.00010
8055 -0.000638 -0.000149 -0.001069 -0.000145  0.999355 -0.000477 -0.00075
8056 -0.001017 -0.000187 -0.000844  0.000150 -0.000498  0.996905 -0.00110
8057 -0.000671 -0.000274 -0.001188 -0.000173 -0.000748 -0.001117  0.99873
8058 -0.000476 -0.001084 -0.001943  0.000430 -0.000762  0.000999 -0.00068

            zz
0     0.000531
1    -0.001043
2    -0.000015
3     0.000210
4    -0.001012
5    -0.000272
6     0.001156
7    -0.000748
8     0.000612
9     0.000476
10    0.000907
11   -0.000421
12   -0.006928
```

```
13    0.003026
14    0.000319
15    0.002374
16   -0.002199
17    0.000201
18    0.001760
19    0.003546
20    0.000103
21   -0.001745
22   -0.002240
23   -0.002470
24   -0.000134
25    0.001814
26    0.003326
27   -0.004762
28   -0.001678
29   -0.000848
...        ...
8029 -0.000039
8030  0.002681
8031  0.001457
8032 -0.001814
8033 -0.000218
8034 -0.001406
8035 -0.000682
8036 -0.000204
8037 -0.000398
8038 -0.000548
8039 -0.000317
8040 -0.001003
8041  0.000126
8042  0.000589
8043 -0.001254
8044 -0.000235
8045  0.000079
8046 -0.005546
8047 -0.001716
8048  0.000023
8049 -0.000053
8050 -0.000016
8051 -0.000545
8052 -0.001058
8053 -0.002131
8054  0.000535
8055 -0.000853
8056  0.000802
8057 -0.000782
8058  0.978465
```

```
[8059 rows x 8059 columns]
```

The rows of U correpond to documents, which are linear combinations of **concepts**.

1.5 Denoising

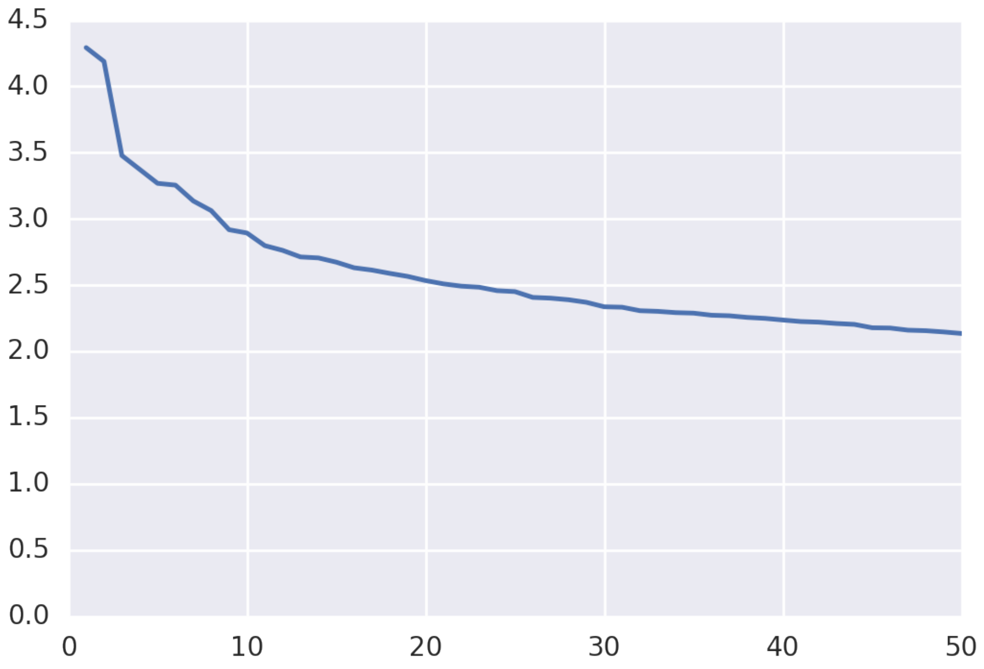
In order to improve our clustering accuracy, we will **exclude** the less significant concepts from the documents' feature vectors.

That is, we will choose the leftmost k columns of U and the topmost k rows of V^T.

The reduced set of columns of U are our new document encodings, and it is those that we will cluster.

```
In [134]: plt.xlim([0,50])
          plt.plot(range(1,len(s)+1),s)

Out[134]: [<matplotlib.lines.Line2D at 0x15f645940>]
```



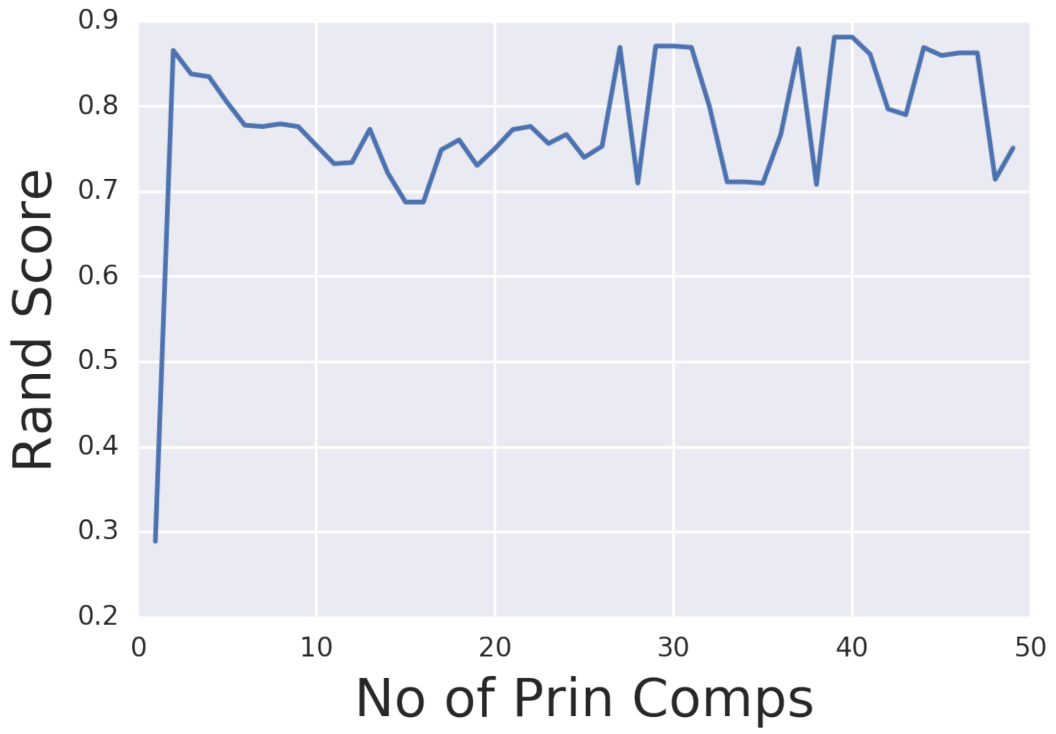
It looks like 2 is a reasonable number of principal components.

```
In [135]: ri = []
          ss = []
          max = len(u)
          for k in range(1,50):
              vectorsk = u[:,:k] @ np.diag(s[:k])
              kmeans = KMeans(n_clusters=3, init='k-means++', max_iter=100, n_init=
              kmeans.fit_predict(vectorsk)
```

```
            labelsk = kmeans.labels_
            ri.append(metrics.adjusted_rand_score(labelsk,news_data.target))
            ss.append(metrics.silhouette_score(vectorsk,kmeans.labels_,metric='eu
```

```
In [136]: plt.plot(range(1,50),ri)
          plt.ylabel('Rand Score',size=20)
          plt.xlabel('No of Prin Comps',size=20)

Out[136]: <matplotlib.text.Text at 0x158bbf320>
```

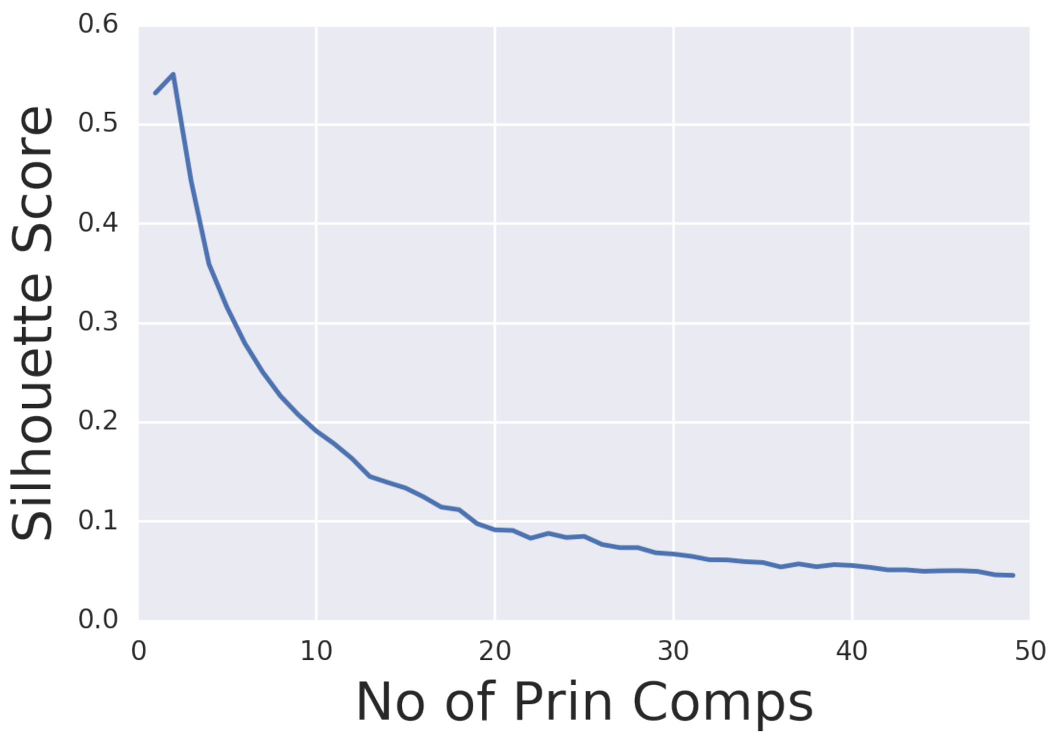


```
In [137]: news_data.target_names

Out[137]: ['comp.os.ms-windows.misc', 'rec.sport.baseball', 'sci.space']

In [138]: plt.plot(range(1,50),ss)
          plt.ylabel('Silhouette Score',size=20)
          plt.xlabel('No of Prin Comps',size=20)

Out[138]: <matplotlib.text.Text at 0x15ba5cd30>
```

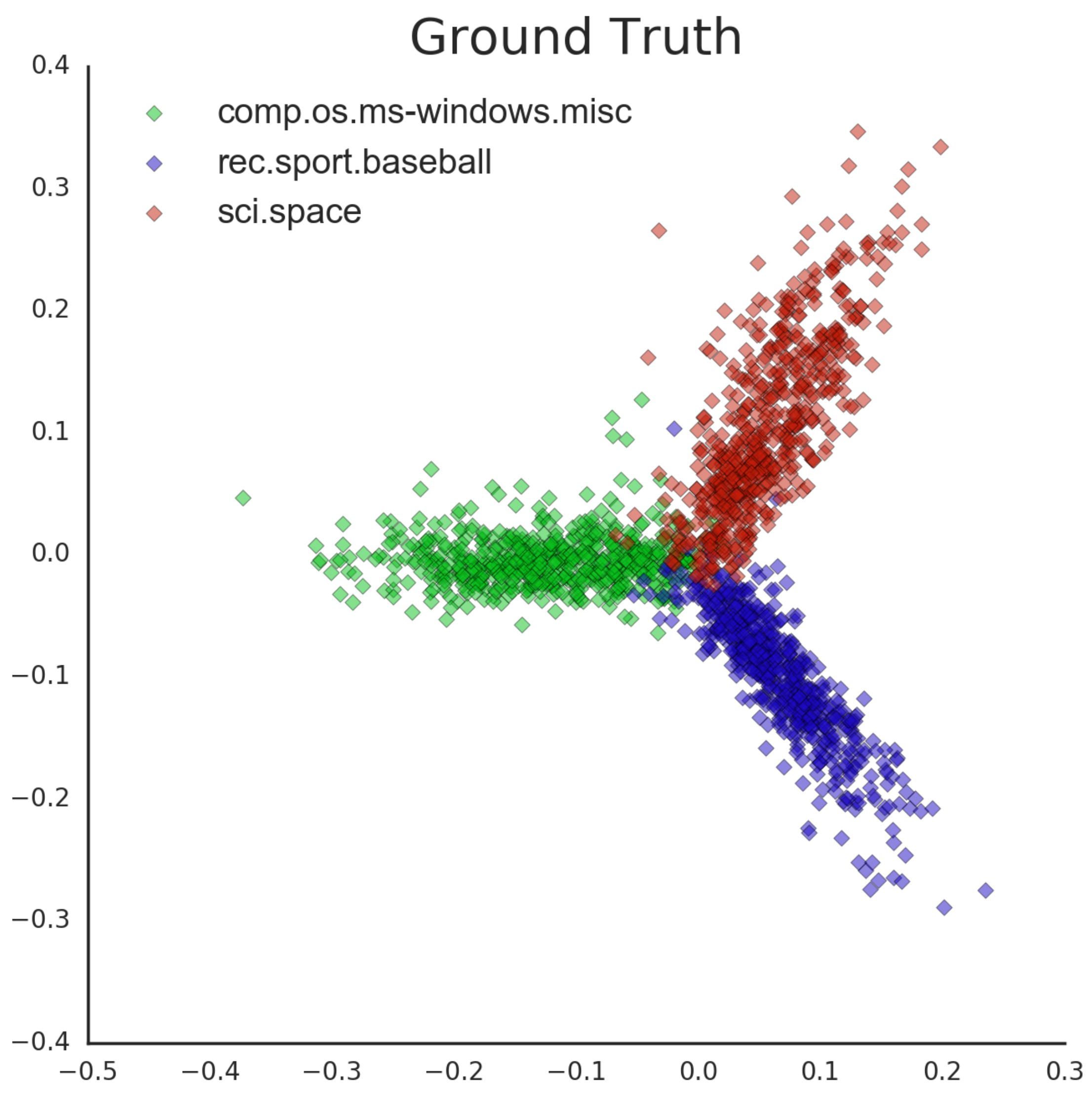


Note that we can get good accuracy with just **two** principal components.

1.6 Visualization

That's a good thing, because it means that we can **visualize** the data well with the help of PCA.
Recall that the challenge of visualization is that the data live in a high dimensional space.
We can only look at 2 (or maybe 3) dimensions at a time, so it's not clear **which** dimensions to look at.
The idea behind using PCA for visualization is that since low-numbered principal components capture most of the **variance** in the data, these are the "directions" from which it is most useful to inspect the data.
We saw that the first two principal components were particularly large – let's start by using them for visualization.

```
In [139]: with sns.axes_style("white"):
              fig, ax = plt.subplots(1,1,figsize=(7,7))
              cmap = sns.hls_palette(n_colors=3, h=0.35, l=0.4, s=0.9)
              for i, label in enumerate(set(news_data.target)):
                  point_indices = np.where(news_data.target == label)[0]
                  point_indices = point_indices.tolist()
                  plt.scatter(Xk[point_indices,0], Xk[point_indices,1], s=20, alpha
          label=news_data.target_names[i])
                  plt.legend(prop={'size':14}, loc=2)
              sns.despine()
          _ = plt.title('Ground Truth', size=20)
```



Points in this plot have been labelled with their "true" (aka "ground truth") cluster labels.

Notice how clearly the clusters separate and how coherently they present themselves. This is obvious an excellent visualization that is provided by PCA.

Since this visualization is so clear, we can use it to examine the results of our various clustering methods and get some insight into how they differ.

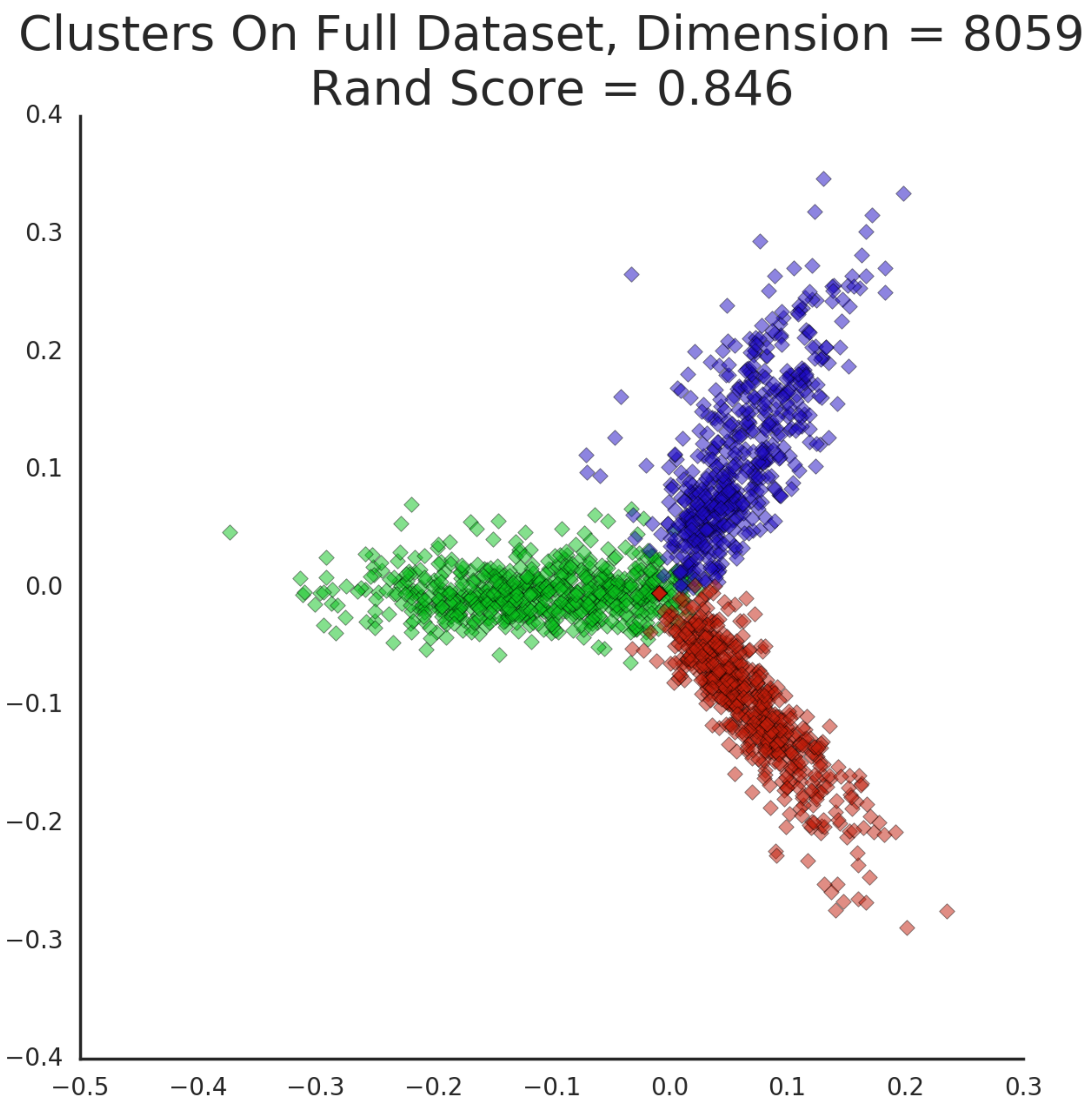
```
In [140]: k = 3
          kmeans = KMeans(n_clusters=k, init='k-means++', max_iter=100, n_init=10,r
          kmeans.fit_predict(dtm)
          centroids = kmeans.cluster_centers_
          labels = kmeans.labels_
          error = kmeans.inertia_

          with sns.axes_style("white"):
```

```
        fig, ax = plt.subplots(1,1,figsize=(7,7))
        cmap = sns.hls_palette(n_colors=3, h=0.35, l=0.4, s=0.9)
        for i in range(k):
            point_indices = np.where(labels == i)[0]
            point_indices = point_indices.tolist()
            plt.scatter(Xk[point_indices,0], Xk[point_indices,1], s=20, alpha
    label=news_data.target_names[i])
        sns.despine()
    plt.title('Clusters On Full Dataset, Dimension = {}\nRand Score = {:0.3f}

              size=20)
```

```
Out[140]: <matplotlib.text.Text at 0x11655cb38>
```

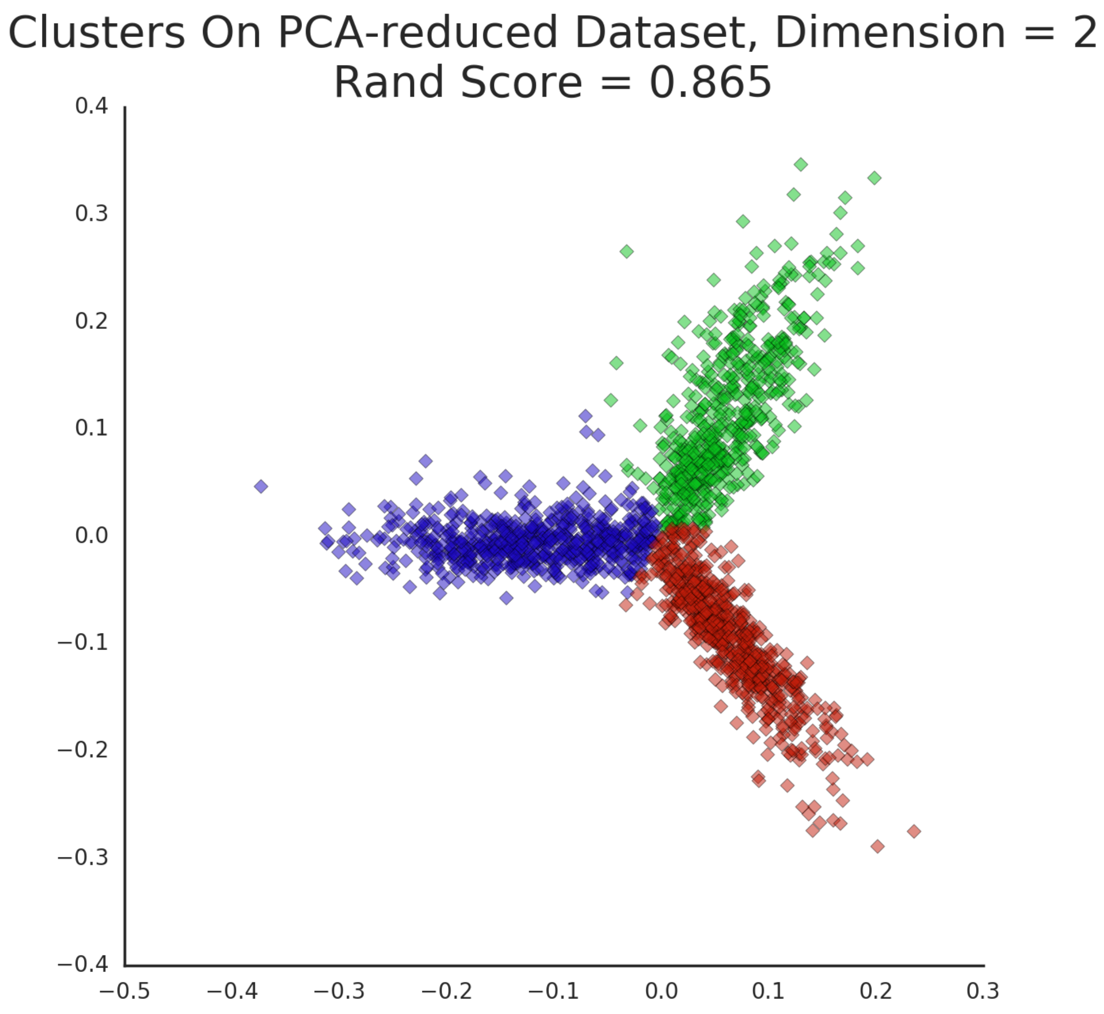


```
In [141]: k = 3
          kmeans = KMeans(n_clusters=k, init='k-means++', max_iter=100, n_init=10,r
          kmeans.fit_predict(Xk[:,:2])
          centroids = kmeans.cluster_centers_
          Xklabels = kmeans.labels_
          error = kmeans.inertia_

          with sns.axes_style("white"):
              fig, ax = plt.subplots(1,1,figsize=(7,7))
              cmap = sns.hls_palette(n_colors=3, h=0.35, l=0.4, s=0.9)
              for i, label in enumerate(set(news_data.target)):
                  point_indices = np.where(Xklabels == label)[0]
                  point_indices = point_indices.tolist()
                  plt.scatter(Xk[point_indices,0], Xk[point_indices,1], s=20, alpha
              sns.despine()
          plt.title('Clusters On PCA-reduced Dataset, Dimension = 2\nRand Score = {

                    size=20)

Out[141]: <matplotlib.text.Text at 0x1612f59b0>
```



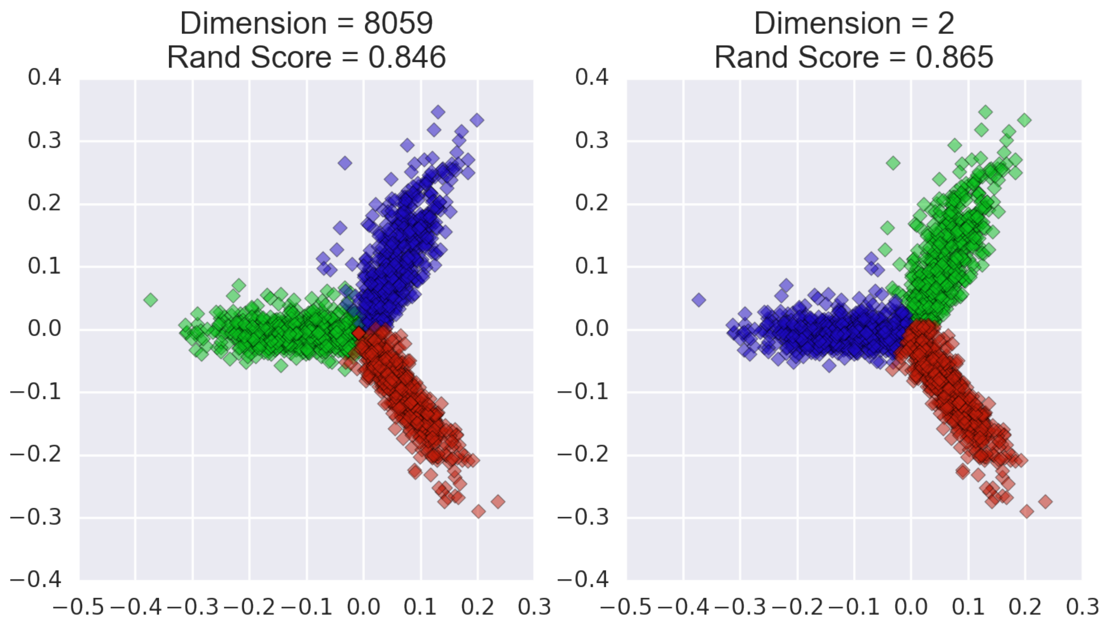
```
In [142]: plt.figure(figsize=(8,4))
          plt.subplot(121)
          cmap = sns.hls_palette(n_colors=3, h=0.35, l=0.4, s=0.9)
          for i in range(k):
                  point_indices = np.where(labels == i)[0]
                  point_indices = point_indices.tolist()
                  plt.scatter(Xk[point_indices,0], Xk[point_indices,1], s=20, alpha
          sns.despine()
          plt.title('Dimension = {}\nRand Score = {:0.3f}'.format(dtm.shape[1],

                    size=14)
          plt.subplot(122)
          cmap = sns.hls_palette(n_colors=3, h=0.35, l=0.4, s=0.9)
          for i in range(k):
                  point_indices = np.where(Xklabels == i)[0]
                  point_indices = point_indices.tolist()
```

```
            plt.scatter(Xk[point_indices,0], Xk[point_indices,1], s=20, alpha
        sns.despine()
        plt.title('Dimension = 2\nRand Score = {:0.3f}'.format(

                  size=14)
```

```
Out[142]: <matplotlib.text.Text at 0x158bbe400>
```



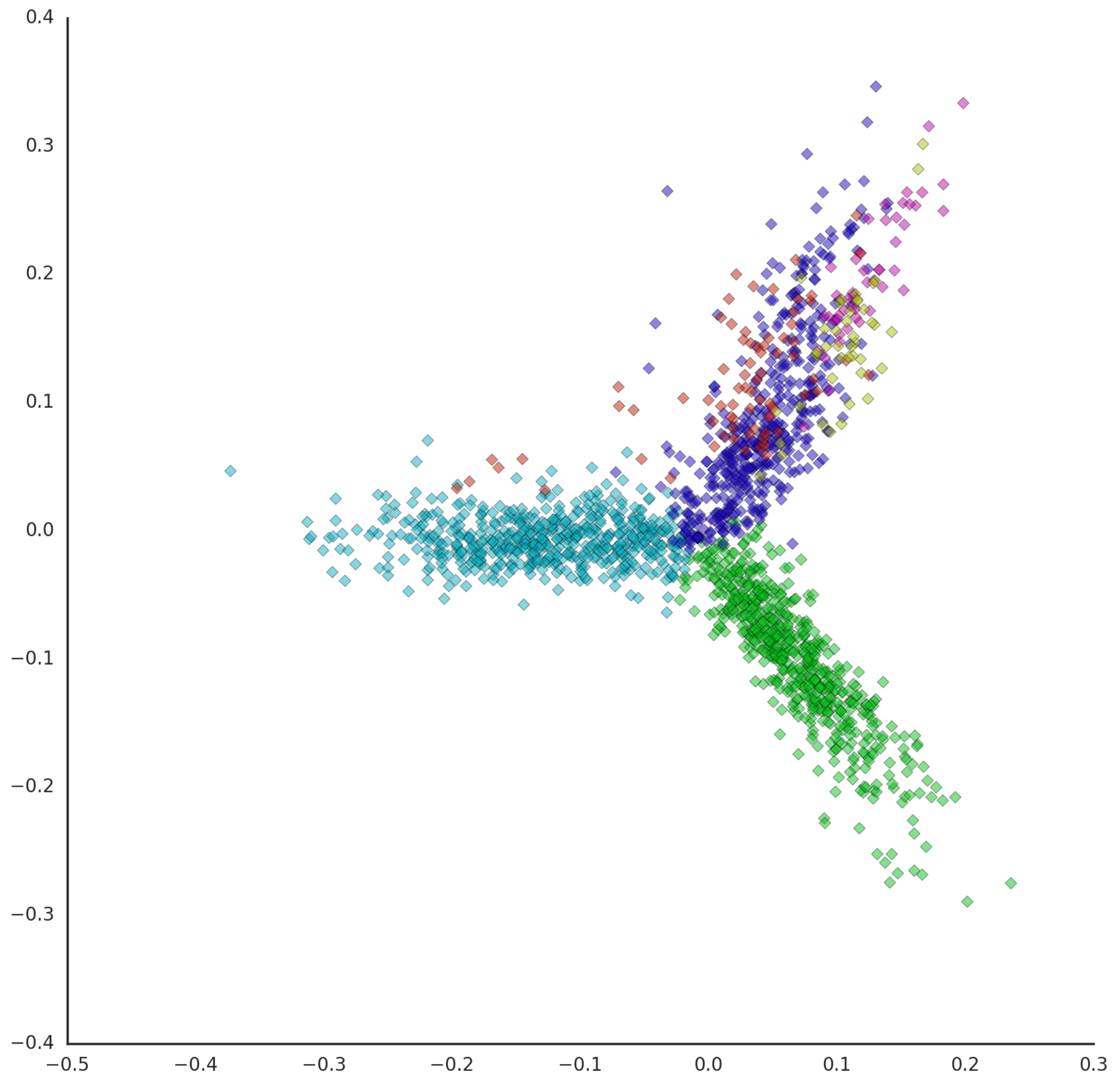
What happens if we misjudge the number of clusters? Let's form 6 clusters.

```
In [143]: k = 6
          kmeans = KMeans(n_clusters=k, init='k-means++', max_iter=100, n_init=10,r
          kmeans.fit_predict(Xk[:,:6])
          centroids = kmeans.cluster_centers_
          labels = kmeans.labels_
          error = kmeans.inertia_

          with sns.axes_style("white"):
              fig, ax = plt.subplots(1,1,figsize=(10,10))
              cmap = sns.hls_palette(n_colors=k, h=0.35, l=0.4, s=0.9)
              for i in range(k):
                  point_indices = np.where(labels == i)[0]
                  point_indices = point_indices.tolist()
                  plt.scatter(Xk[point_indices,0], Xk[point_indices,1], s=20, alpha
              sns.despine()

          print(metrics.adjusted_rand_score(labels,news_data.target))
```

```
0.7589216806677921
```



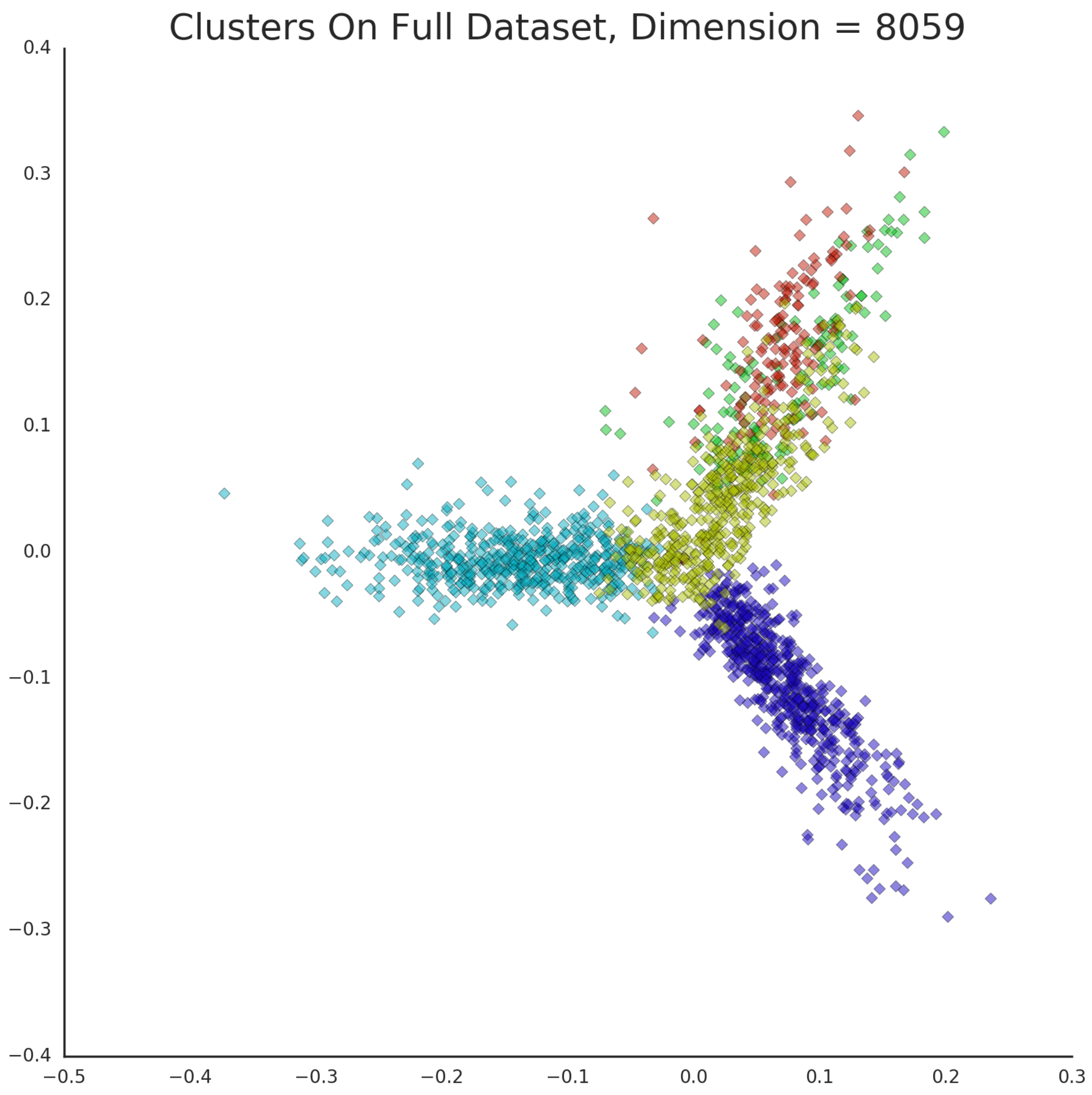
```
In [144]: k = 6
          kmeans = KMeans(n_clusters=k, init='k-means++', max_iter=100, n_init=10,r
          kmeans.fit_predict(dtm)
          centroids = kmeans.cluster_centers_
          labels = kmeans.labels_
          error = kmeans.inertia_

          with sns.axes_style("white"):
              fig, ax = plt.subplots(1,1,figsize=(10,10))
              cmap = sns.hls_palette(n_colors=k, h=0.35, l=0.4, s=0.9)
              for i in range(k):
```

```
            point_indices = np.where(labels == i)[0]
            point_indices = point_indices.tolist()
            plt.scatter(Xk[point_indices,0], Xk[point_indices,1], s=20, alpha
        sns.despine()
    plt.title('Clusters On Full Dataset, Dimension = {}'.format(dtm.shape[1])

    print(metrics.adjusted_rand_score(labels,news_data.target))
```

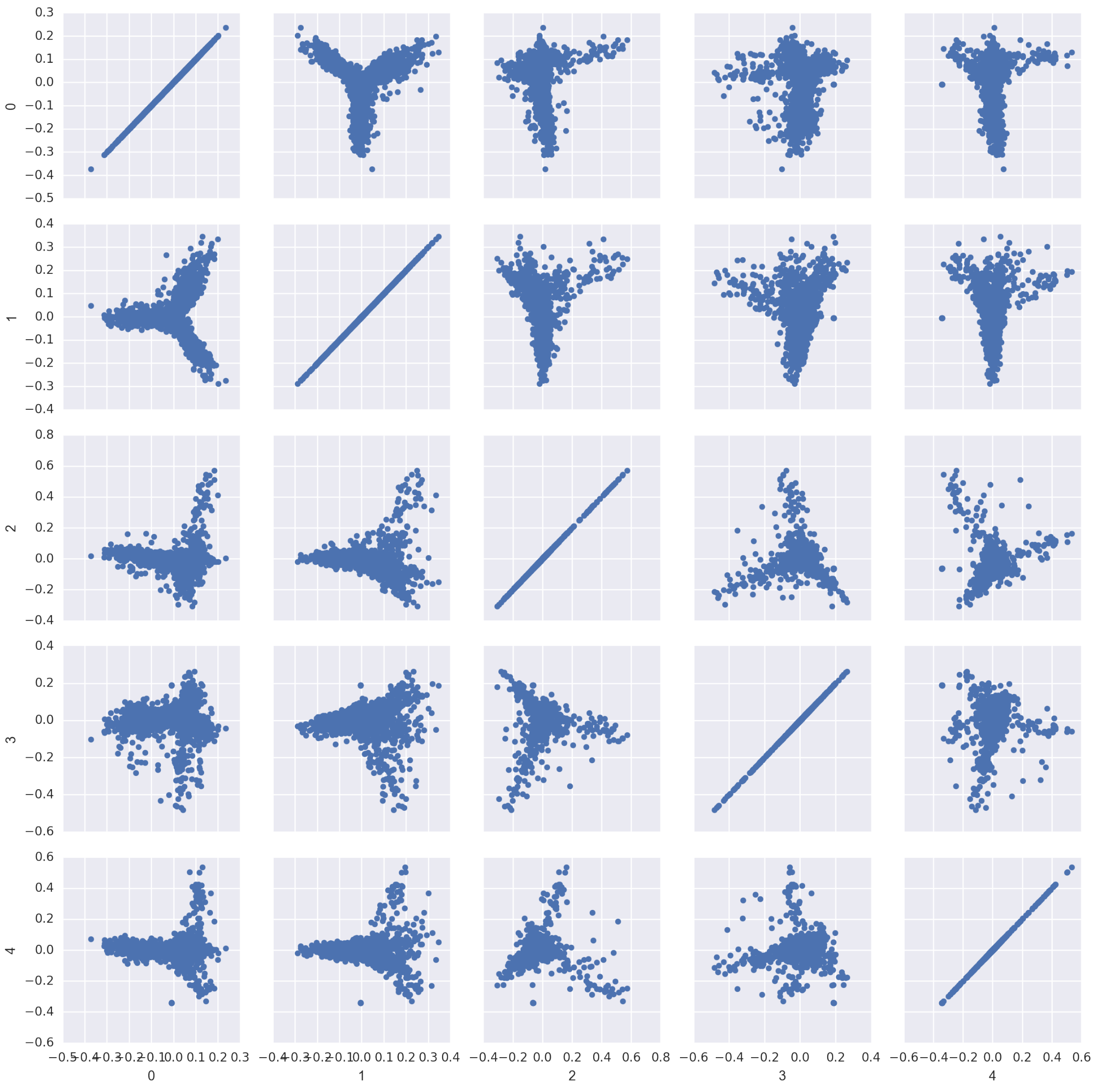
```
0.6599524401707585
```



What about the other principal components? Are they useful for visualization?
A common approach is to look at all pairs of (low-numbered) principal components.

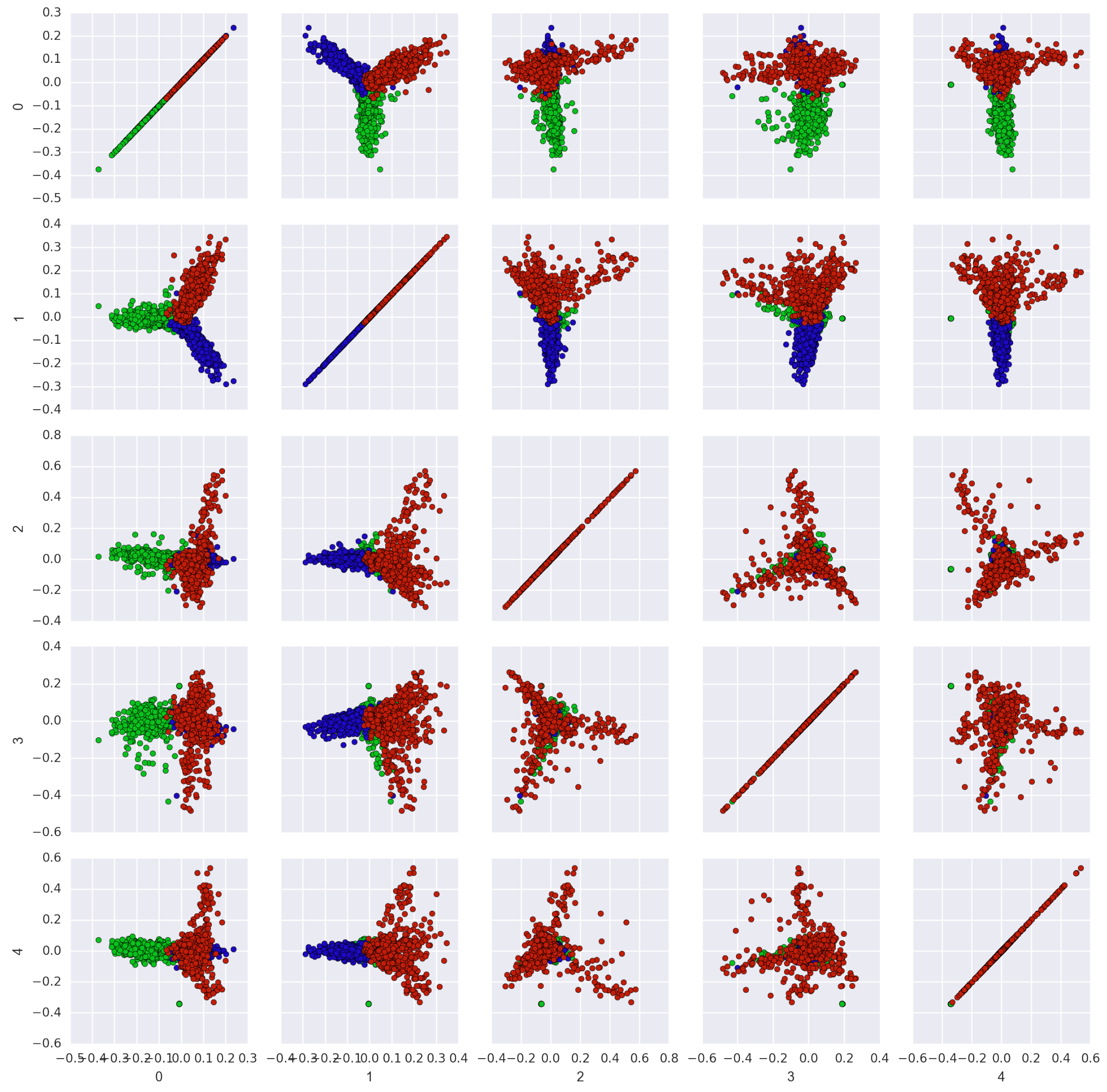
```
In [145]: import seaborn as sns
          k = 5
          Xk = u[:,:k] @ np.diag(s[:k])
          X_df = pd.DataFrame(Xk)
          g = sns.PairGrid(X_df)
          g.map(plt.scatter)

Out[145]: <seaborn.axisgrid.PairGrid at 0x1615cf9b0>
```



```
In [146]: k = 5
          Xk = u[:,:k] @ np.diag(s[:k])
          X_df = pd.DataFrame(Xk)
          g = sns.PairGrid(X_df)
          def pltColor(x,y,label,color):
              cmap = sns.hls_palette(n_colors=3, h=0.35, l=0.4, s=0.9)
              for i in range(3):
                  point_indices = np.where(news_data.target == i)[0]
                  point_indices = point_indices.tolist()
                  plt.scatter(x[point_indices], y[point_indices], c=cmap[i])
              sns.despine()
          g.map(pltColor)
```

```
Out[146]: <seaborn.axisgrid.PairGrid at 0x124eaf2b0>
```



1.7 Looking at the Topics

```
In [147]: for i in range(6):
              top = np.argsort(vt[i])
              topterms = [terms[top[0,f]] for f in range(12)]
              print (i, topterms)
```

```
0 ['window', 'file', 'driver', 'dos', 'use', 'card', 'font', 'ms', 'problem', 'prog
1 ['game', 'team', 'player', 'pitch', 'run', 'basebal', 'win', 'hit', 'edu', 'score
2 ['access', 'nasa', 'digex', 'pat', 'gov', 'jpl', 'baalk', 'com', '___', 'kelvin',
3 ['access', 'digex', 'pat', 'com', 'prb', 'net', 'express', 'onlin', 'communic', '
4 ['ax', 'henri', 'toronto', 'zoo', 'com', 'spencer', 'zoolog', 'access', 'jpl', 'b
5 ['window', 'run', 'nasa', 'year', 'file', 'team', 'game', 'win', 'dos', 'hit', 'h
```