# 08B-Clustering-III

October 31, 2016

## 1 Clustering data with k-means

Today we'll do an extended example showing k-means clustering in practice and in the context of the python libraries **scikit-learn.**

### 1.1 Synthetic data

Generally, when learning about or developing a new unsupervised method, it's a good idea to try it out on a dataset in which you already know the "right" answer.

One way to do that is to generate synthetic data that has some known properties.

**scikit-learn** is the main python library for machine learning functions. It is extensive. You will use it a lot :). It is here: http://scikit-learn.org/stable/documentation.html.

Among other things, it contains tools for generating synthetic data for testing.

```
In [63]: X, y = sk_data.make_blobs(n_samples=100, centers=3, n_features=30,
                                    center_box=(-10.0, 10.0),random_state=0)
```

To get a sense of the raw data we can visualize it.

```
In [95]: _ = sns.heatmap(X, xticklabels=False, yticklabels=False, linewidths=0,cbar
```
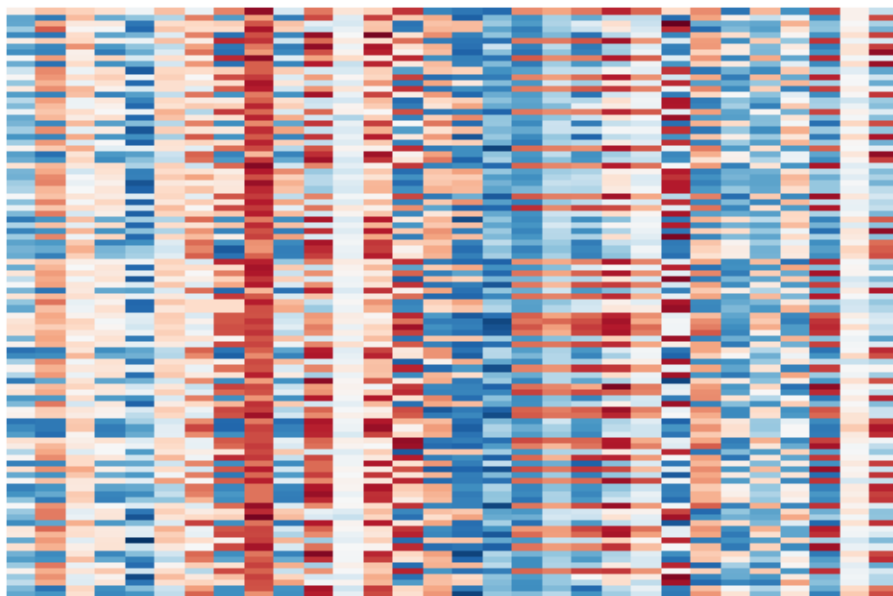
Now let's compute the pairwise distances for visualization purposes.

We can compute pairwise distances using the **sklearn.metrics** functions summarized here: http://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics

```
In [97]: euclidean_dists = metrics.euclidean_distances(X)
         # euclidean_dists
```

That plot shows all the data. These data live in 30 dimensions, so we cannot directly visualize them.

However, there is a method that can turn a set of pairwise distances into an approximate 2-D representation **in some cases.**

### 1.1.1 Visualizing with MultiDimensional Scaling

The idea behind Multidimensional Scaling (MDS) is: given a dissimilarity or distance matrix, find a set of coordinates for the points that approximates those distances as well as possible.

This is done by a gradient descent algorithm that starts with random positions and moves points in a way that reduces the disparity between true distance and euclidean distance.

Note that there are many ways that this can fail!

- Perhaps the dissimilarities are not well modeled as euclidean distances
- It may be necessary to use more than 2 dimensions to capture any clustering via euclidean distances

```
In [66]: import sklearn.manifold
         mds = sklearn.manifold.MDS(n_components=2, max_iter=3000, eps=1e-9, randon
                           dissimilarity="precomputed", n_jobs=1)
         fit = mds.fit(euclidean_dists)
         pos = fit.embedding_
         _ = plt.scatter(pos[:, 0], pos[:, 1], s=8)
```

We can also visualize the data using the heatmap of pairwise distances.

```
In [67]: _ = sns.heatmap(euclidean_dists, xticklabels=False, yticklabels=False, lir
                         square=True, cbar=False)
```
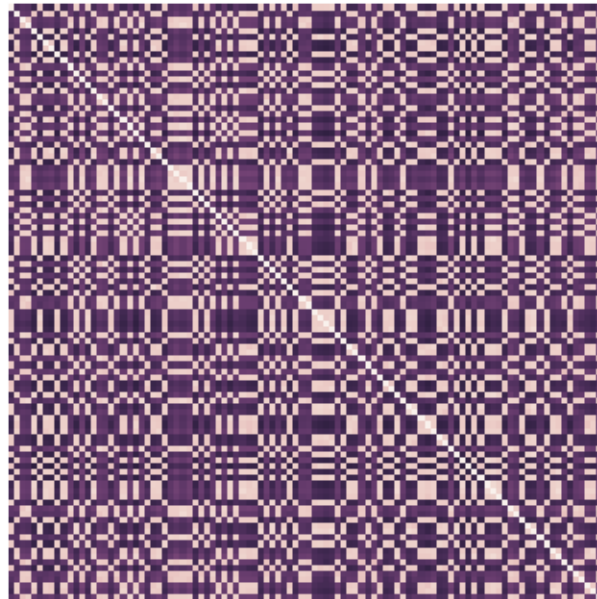
## 1.2  Applying k-means

**scikit-learn** has a huge set of tools for unsupervised learning generally, and clustering specifically.
These are in sklearn.cluster. http://scikit-learn.org/stable/modules/clustering.html
There are 3 functions in all the clustering classes,

- **fit()**,
- **predict()**, and
- **fit_predict()**.

**fit()** builds the model from the training data (e.g. for kmeans, it finds the centroids),
**predict()** assigns labels to the data after building the model, and
**fit_predict()** does both at the same data (e.g in kmeans, it finds the centroids and assigns
the labels to the dataset).

```
In [98]: from sklearn.cluster import KMeans
         kmeans = KMeans(init='k-means++', n_clusters=3, n_init=100)
         kmeans.fit_predict(X)

Out[98]: array([0, 2, 1, 1, 2, 0, 2, 2, 0, 2, 1, 0, 1, 0, 2, 1, 1, 0, 1, 2, 1, 2, 1
                 0, 2, 2, 0, 1, 1, 1, 1, 0, 1, 0, 1, 2, 1, 2, 1, 2, 2, 2, 0, 1, 0, 1
                 0, 2, 0, 1, 1, 0, 0, 0, 0, 1, 0, 2, 2, 1, 0, 1, 2, 0, 0, 2, 1, 0, 0
                 2, 2, 2, 0, 0, 1, 0, 2, 0, 2, 0, 2, 2, 2, 0, 1, 1, 2, 0, 0, 1, 0, 2
                 2, 1, 0, 1, 1, 2, 2, 1], dtype=int32)
```

All the tools in scikit-learn are implemented as python objects.
Thus, the general sequence for using a tool from scikit-learn is:

- Create the object, probably with some parameter settings or intialization,
- Run the method, generally by using the fit() function, and
- Examine the results, which are generally instance variables of the object.

```
In [99]: centroids = kmeans.cluster_centers_
         labels = kmeans.labels_
         error = kmeans.inertia_

In [100]: print("The total error of the clustering is: {}.".format(error))
          print('\nCluster labels:')
          print(labels)
          print('\nCluster centroids:')
          print(centroids)

The total error of the clustering is: 2733.843081897791.

Cluster labels:
[0 2 1 1 2 0 2 2 0 2 1 0 1 0 2 1 1 0 1 2 1 2 1 0 2 2 0 1 1 1 1 0 1 0 1 2 1
 2 1 2 2 2 0 1 0 1 0 2 0 1 1 0 0 0 0 1 0 2 2 1 0 1 2 0 0 2 1 0 0 2 2 2 0 0
 1 0 2 0 2 0 2 2 2 0 1 1 2 0 0 1 0 2 2 1 0 1 1 2 2 1]
```

4

```
Cluster centroids:
[[ 0.88697885   4.29142902   1.93200132   1.10877989  -1.55994342   2.80616392
  -1.11495818   7.74595341   8.92512875  -2.29656298   6.09588722   0.47062896
   1.36408008   8.63168509  -8.54512921  -8.59161818  -9.64308952   6.92270491
   5.65321496   7.29061444   9.58822315   5.79602014  -0.84970449   5.46127493
  -7.77730238   2.75092191  -7.17026663   9.07475984   0.04245798  -1.98719465]
 [-4.7833887    5.32946939  -0.87141823   1.38900567  -9.59956915   2.35207348
   2.22988468   2.03394692   8.9797878    3.67857655  -2.67618716  -1.17595897
   3.76433199  -8.46317271   3.28114395   3.73803392  -5.73436869  -7.0844462
  -3.75643598  -3.07904369   1.36974653  -0.95918462   9.91135428  -8.17722281
  -5.8656831   -6.76869078   3.12196673  -4.85745245  -0.70449349  -4.94582258]
 [-7.0489904   -7.92501873   2.89710462  -7.17088692  -6.01151677  -2.66405834
   6.43970052  -8.20341647   6.54146052  -7.92978843   9.56983319  -0.86327902
   9.25897119   1.73061823   4.84528928  -9.26418246  -4.54021612  -7.47784575
  -4.15060719  -7.85665458  -3.76688414  -1.6692291   -8.78048843   3.78904162
   1.24247168  -4.73618733   0.27327032  -7.93180624   1.59974866   8.78601576]]
```

### 1.2.1 Visualizing the results of clustering

Let's visualize the results. We'll do that by reording the data items according to their cluster.

```
In [71]: idx = np.argsort(labels)
         rX = X[idx,:]
         _ = sns.heatmap( rX,xticklabels=False, yticklabels=False, linewidths=0,cba
```
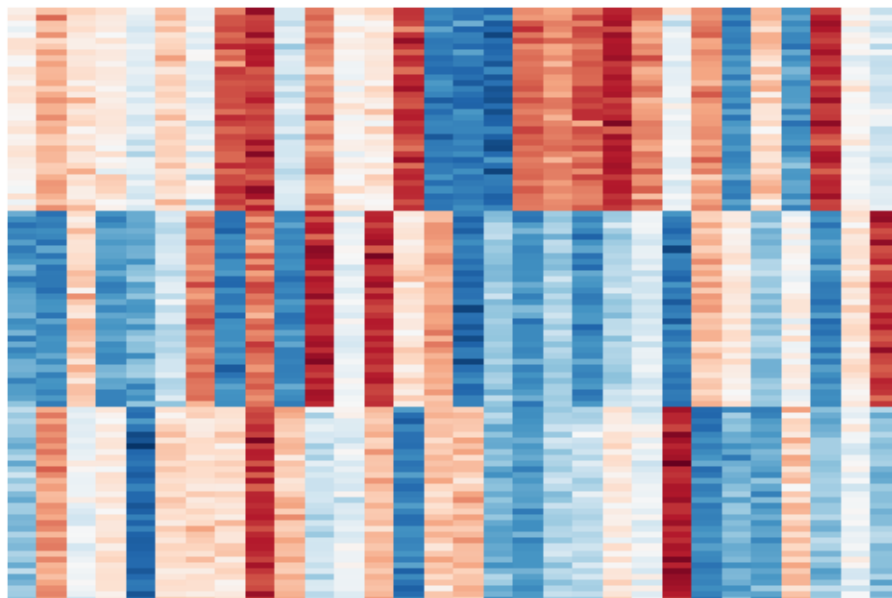


```
In [72]: rearranged_dists = euclidean_dists[idx,:][:,idx]
         sns.heatmap(rearranged_dists, xticklabels=False, yticklabels=False, linewi
```

```
Out[72]: <matplotlib.axes._subplots.AxesSubplot at 0x11d235080>
```
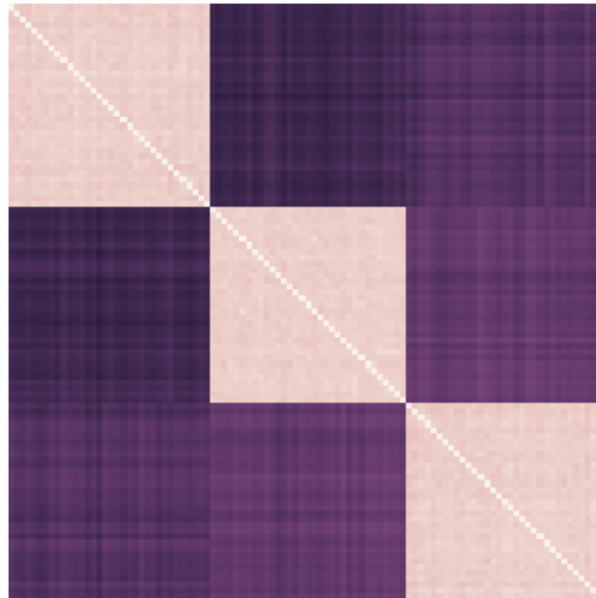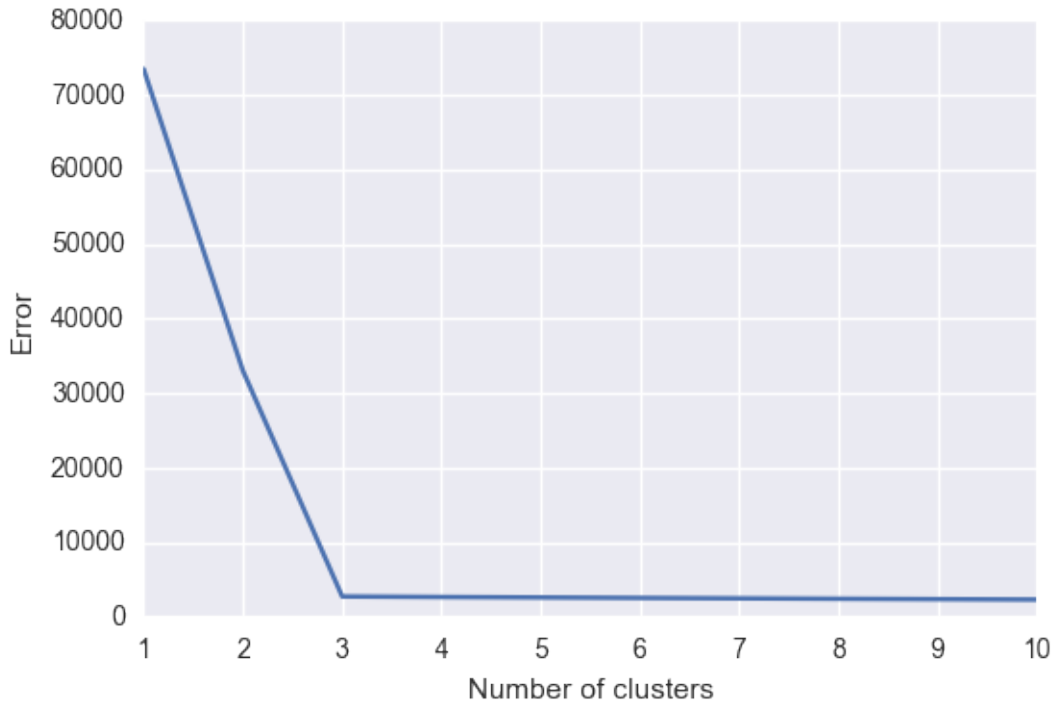


## 1.3  Deciding on the number of clusters

### 1.3.1  Using the error function

```
In [73]: error = np.zeros(11)
         for k in range(1,11):
             kmeans = KMeans(init='k-means++', n_clusters=k, n_init=10)
             kmeans.fit_predict(X)
             error[k] = kmeans.inertia_

In [74]: plt.plot(range(1,len(error)),error[1:])
         plt.xlabel('Number of clusters')
         dummy = plt.ylabel('Error')
```

Let's create a function for later use.

```
In [75]: def evaluate_clusters(X,max_clusters):
             error = np.zeros(max_clusters+1)
             error[0] = 0;
             for k in range(1,max_clusters+1):
                 kmeans = KMeans(init='k-means++', n_clusters=k, n_init=10)
                 kmeans.fit_predict(X)
                 error[k] = kmeans.inertia_

             plt.plot(range(1,len(error)),error[1:])
             plt.xlabel('Number of clusters')
             plt.ylabel('Error')
```

### 1.3.2 Adjusted Rand Index

The Rand Index is a simple similarity measure for clusterings. We can use it to compare two clusterings for similarity.

Or, if we are testing an algorithm on data for which we know ground truth, we can use it to assess the algorithm's accuracy.

If $T$ is a ground truth label assignment and $C$ the clustering.

Let $a$ be: the number of pairs of elements that have the same label in $T$ and the same label in $C$.

Also let $b$ be: the number of pairs of elements that have different labels in $T$ and different labels in $C$.

7

Then the Rand Index is:
$$\text{RI}(T, C) = \frac{a + b}{\binom{n}{2}}$$

However the RI score does not guarantee that random label assignments will get a value close to zero

(especially if the number of clusters is in the same order of magnitude as the number of samples).

To counter this effect we can use the expected RI $E[\text{RI}]$ of random labelings to define the adjusted Rand index as follows:

$$\text{ARI} = \frac{\text{RI} - E[\text{RI}]}{\max(\text{RI}) - E[\text{RI}]} \tag{1}$$

The computation of the expected RI is simple combinatorics (and left as an exercise for you :).

```
In [76]: ri = metrics.adjusted_rand_score(labels,y)
         print(ri)

1.0
```

Again, we'll create a function to evaluate clusterings.

```
In [77]: def ri_evaluate_clusters(X,max_clusters,ground_truth):
             ri = np.zeros(max_clusters+1)
             ri[0] = 0;
             for k in range(1,max_clusters+1):
                 kmeans = KMeans(init='k-means++', n_clusters=k, n_init=10)
                 kmeans.fit_predict(X)
                 ri[k] = metrics.adjusted_rand_score(kmeans.labels_,ground_truth)
             plt.plot(range(1,len(ri)),ri[1:])
             plt.xlabel('Number of clusters')
             plt.ylabel('Adjusted Rand Index')

         ri_evaluate_clusters(X,10,y)
```
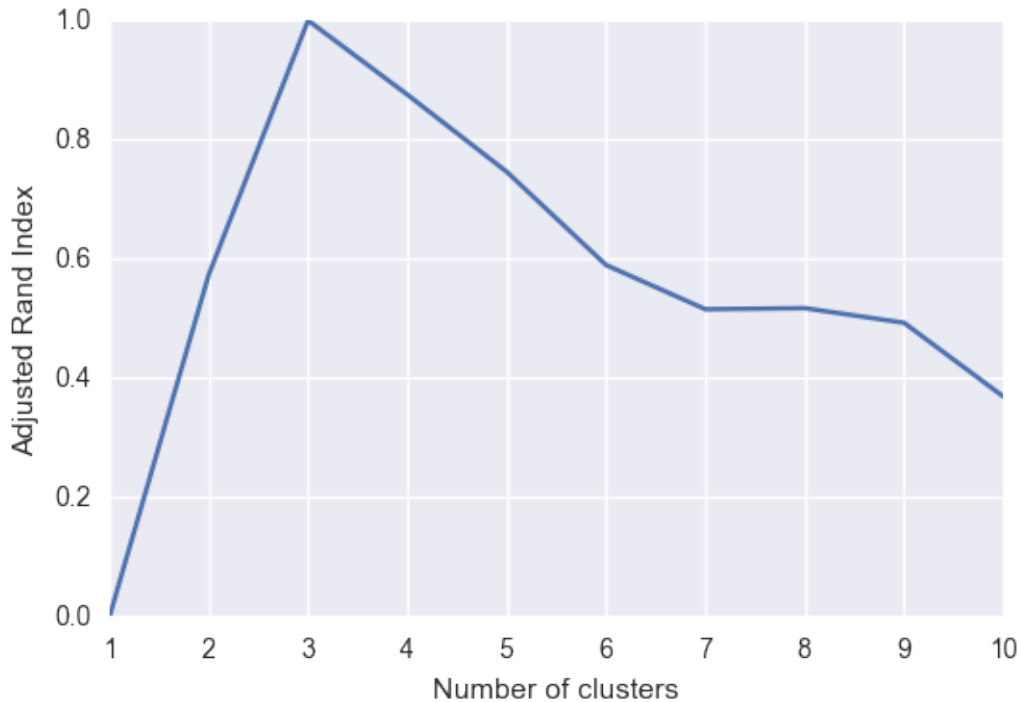
### 1.3.3  Silhouette Coefficient

If the ground truth labels are not known, evaluation must be performed using the model itself.

The Silhouette Coefficient (**sklearn.metrics.silhouette_score**) is an example of such an evaluation, where a higher Silhouette Coefficient score relates to a model with "better defined" clusters.

Let $a$ be the mean distance between a data point and all other points in the same cluster.

Let $b$ be the mean distance between a data point and all other points in the next nearest cluster.

Then the **Silhouette Coefficient** for a clustering is:

$$s = \frac{b - a}{\max(a, b)}$$

```
In [78]: sc = metrics.silhouette_score(X, labels, metric='euclidean')
         print(sc)

0.83193488414


In [101]: def sc_evaluate_clusters(X,max_clusters):
              s = np.zeros(max_clusters+1)
              s[0] = 0;
              s[1] = 0;
              for k in range(2,max_clusters+1):
                  kmeans = KMeans(init='k-means++', n_clusters=k, n_init=10)
                  kmeans.fit_predict(X)
```
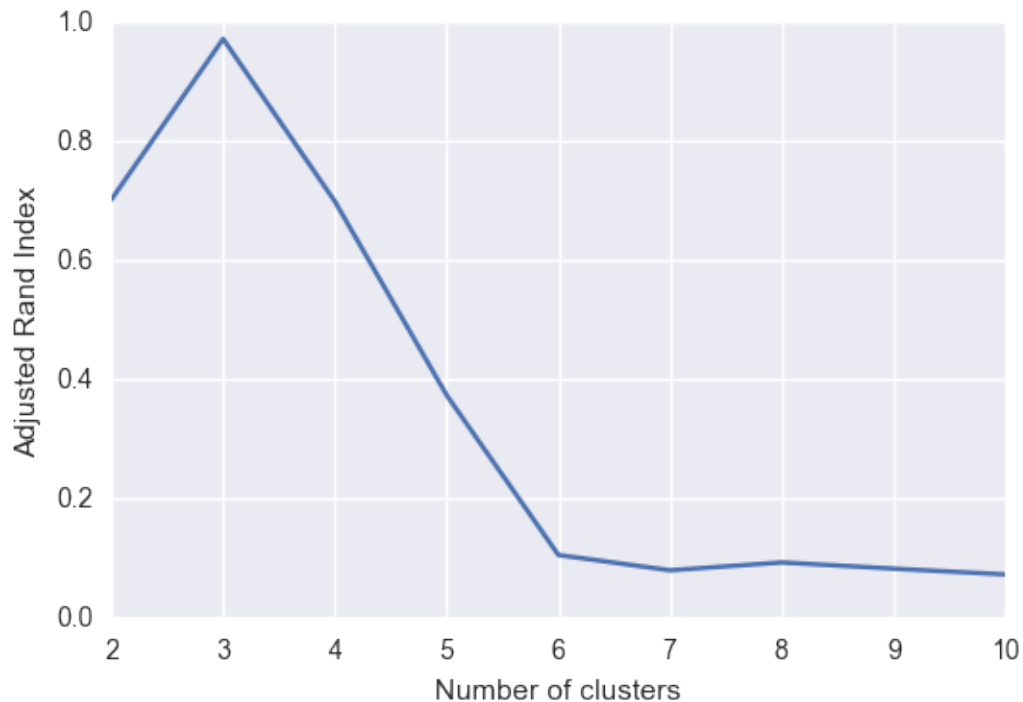
```
        s[k] = metrics.silhouette_score(X,kmeans.labels_,metric='cosine')
    plt.plot(range(2,len(s)),s[2:])
    plt.xlabel('Number of clusters')
    plt.ylabel('Adjusted Rand Index')

sc_evaluate_clusters(X,10)
```



## 1.4 Working with real data

As a "real world" example, we'll use the "20 Newsgroup" data provided as example data in sklearn.

(http://scikit-learn.org/stable/datasets/twenty_newsgroups.html).

```
In [80]: from sklearn.datasets import fetch_20newsgroups

         """
         categories = [
          'comp.windows.x',
          'misc.forsale',
          'rec.autos',
          'rec.motorcycles',
          'rec.sport.baseball',
          'rec.sport.hockey',
          'talk.religion.misc',
```

```
            'comp.graphics',
            'sci.space',
            'rec.autos',
            'rec.sport.baseball'
        ]
        """
        categories = ['comp.os.ms-windows.misc', 'sci.space','rec.sport.baseball']
        news_data = fetch_20newsgroups(subset='train', categories=categories)
        print(news_data.target, len(news_data.target))
        print(news_data.target_names)

[2 0 0 ..., 2 1 2] 1781
['comp.os.ms-windows.misc', 'rec.sport.baseball', 'sci.space']


In [81]: news_data.data[0]

Out[81]: 'From: aws@iti.org (Allen W. Sherzer)\nSubject: Re: DC-X update???\nOrgani
```

### 1.4.1 Feature Extraction

We've discussed a bit the challenges of feature engineering. One of the most basic issues concerns how to encode categorical or text data in a form usable by algorithms that expect numeric input.

As we've discussed already, one can encode a set using a binary vector with one component for each potential set member.

The so-called *bag of words* encoding for a document is to treat the document as a **multi**set of words.

That is, we simply count how many times each word occurs. It is a "bag" because all the order of the words in the document is lost.

Surprisingly, we can still tell a lot about the document even without knowing its word ordering.

Counting the number of times each word occurs in a document yields a vector of **term frequencies.**

However, simply using the "bag of words" directly has a number of drawbacks. First of all, large documents will have more words than small documents. Hence it often makes sense to normalize the frequency vectors.

$\ell_1$ or $\ell_2$ normalization are common.

Next, as noted in **scikit-learn**:

> In a large text corpus, some words will be very present (e.g. "the", "a", "is" in English) hence carrying very little meaningful information about the actual contents of the document.

> If we were to feed the direct count data directly to a classifier those very frequent terms would shadow the frequencies of rarer yet more interesting terms.

> In order to re-weight the count features into floating point values suitable for usage by a classifier it is very common to use the tf–idf transform.

11

**Tf** means **term-frequency** while **tf–idf** means **term-frequency times inverse document-frequency.**

This is a originally a term weighting scheme developed for information retrieval (as a ranking function for search engines results), that has also found good use in document classification and clustering.

The idea is that rare words are more informative than common words. (This has connections to information theory).

Hence, the definition of tf-idf is as follows.

First:

$$\text{tf}(t, d) = \text{Number of times term } t \text{ occurs in document } d$$

Next, if $N$ is the total number of documents in the corpus $D$ then:

$$\text{idf}(t, D) = \frac{N}{|\{d \in D : t \in d\}|}$$

where the denominator is the number of documents in which the term $t$ appears.

And finally:

$$\text{tf-idf}(t, d) = \text{tf}(t, d) \times \text{idf}(t, D)$$

```
In [82]: from sklearn.feature_extraction.text import TfidfVectorizer
         vectorizer = TfidfVectorizer(stop_words='english', min_df=4, max_df=0.8)
         data = vectorizer.fit_transform(news_data.data)
```

### 1.4.2 Getting to know the Data

```
In [83]: print(type(data), data.shape)

<class 'scipy.sparse.csr.csr_matrix'> (1781, 9410)


In [84]: fig, ax1 = plt.subplots(1,1,figsize=(15,10))
         dum = sns.heatmap(data[1:100,1:200].todense(), xticklabels=False, yticklab
                    linewidths=0, cbar=False, ax=ax1)
```
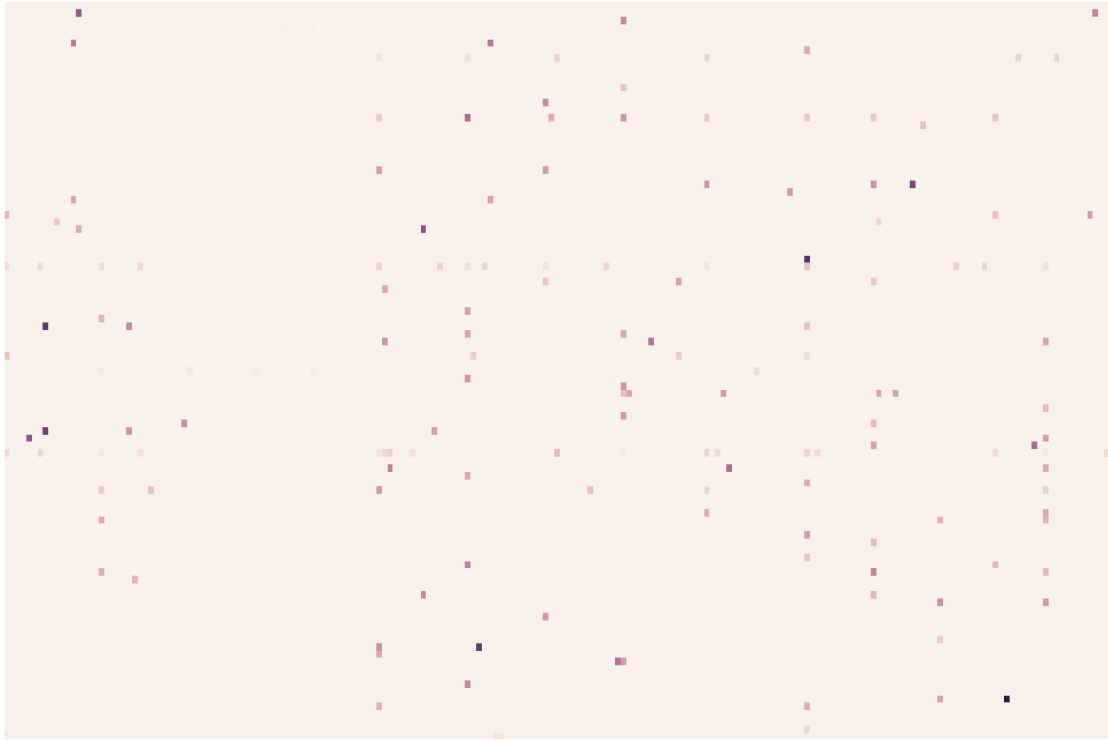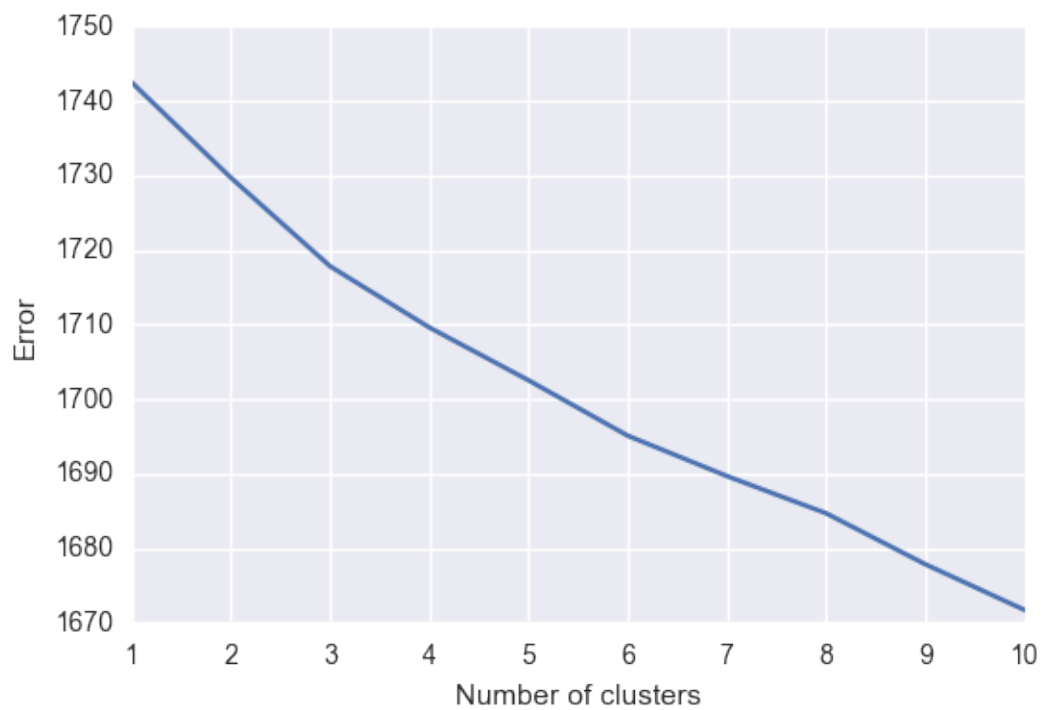
```
In [85]: print(news_data.target)
         print(news_data.target_names)

[2 0 0 ..., 2 1 2]
['comp.os.ms-windows.misc', 'rec.sport.baseball', 'sci.space']
```
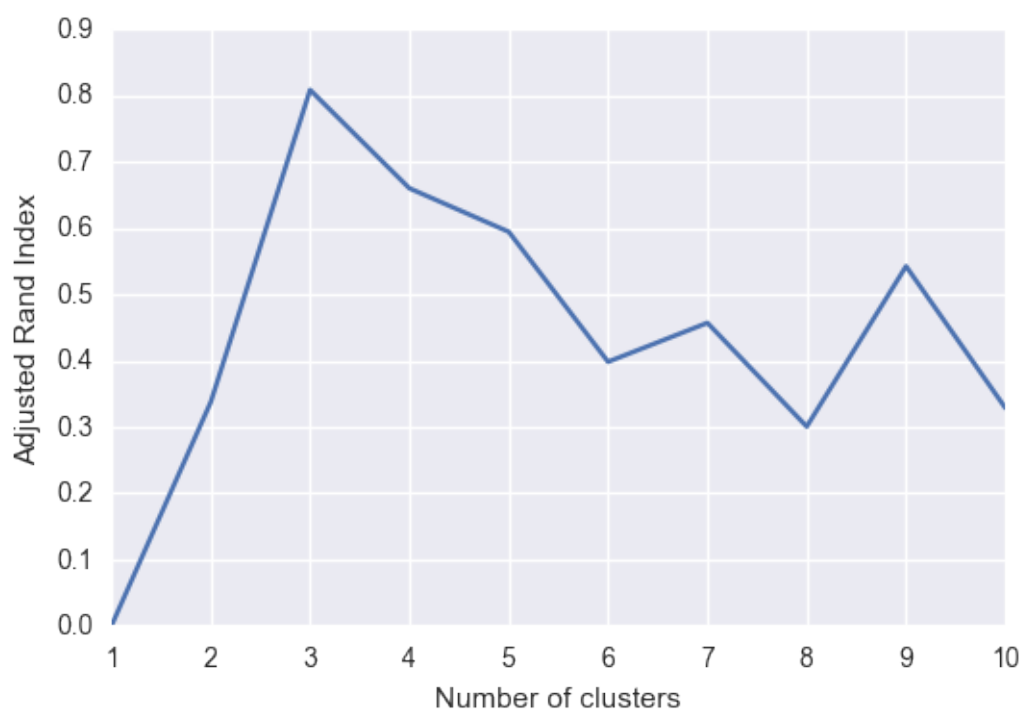
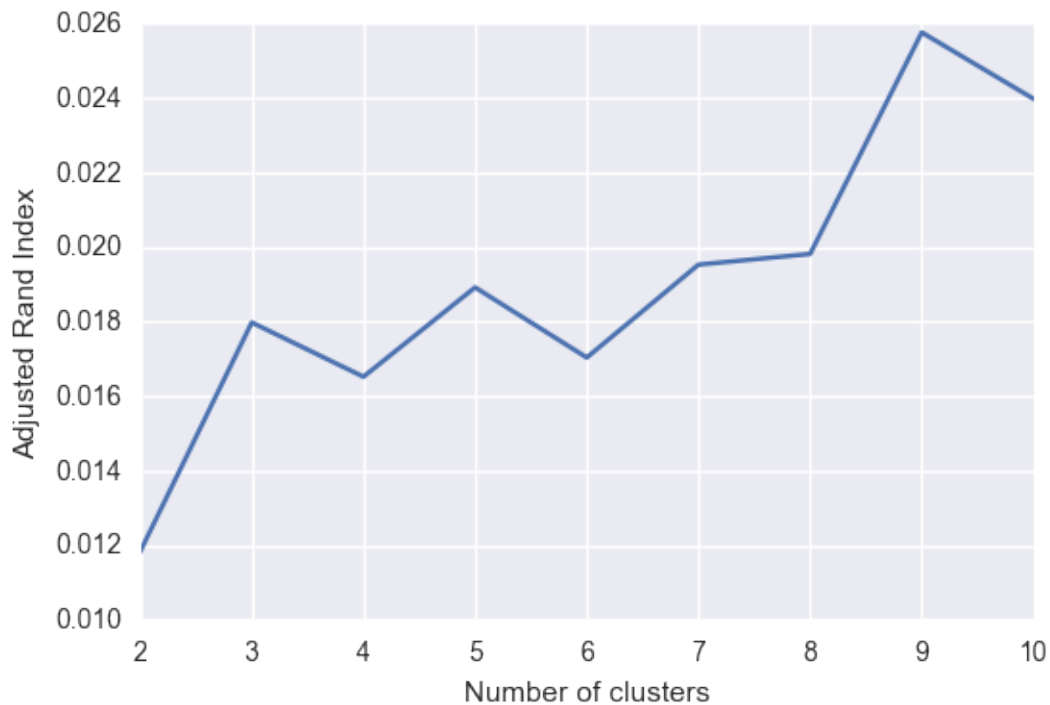### 1.4.3   Selecting the Number of Clusters

```
In [86]: evaluate_clusters(data, 10)
```

In [87]: ri_evaluate_clusters(data,10,news_data.target)

```
In [88]: sc_evaluate_clusters(data,10)
```



### 1.4.4  Looking into the clusters

```
In [89]: k=3
         kmeans = KMeans(n_clusters=k, init='k-means++', max_iter=100, n_init=1)
         kmeans.fit_predict(data)

Out[89]: array([1, 0, 0, ..., 2, 1, 2], dtype=int32)

In [90]: print("Top terms per cluster:")
         asc_order_centroids = kmeans.cluster_centers_.argsort()#[:, ::-1]
         order_centroids = asc_order_centroids[:,::-1]
         terms = vectorizer.get_feature_names()
         for i in range(k):
             print("Cluster {}:".format(i))
             for ind in order_centroids[i, :10]:
                 print(' {}'.format(terms[ind]))
             print('')

Top terms per cluster:
Cluster 0:
 windows
 file
```

```
 edu
 dos
 files
 com
 card
 driver
 drivers
 use

Cluster 1:
 edu
 com
 year
 baseball
 university
 article
 writes
 cs
 team
 game

Cluster 2:
 space
 nasa
 edu
 henry
 access
 gov
 alaska
 com
 digex
 moon
```
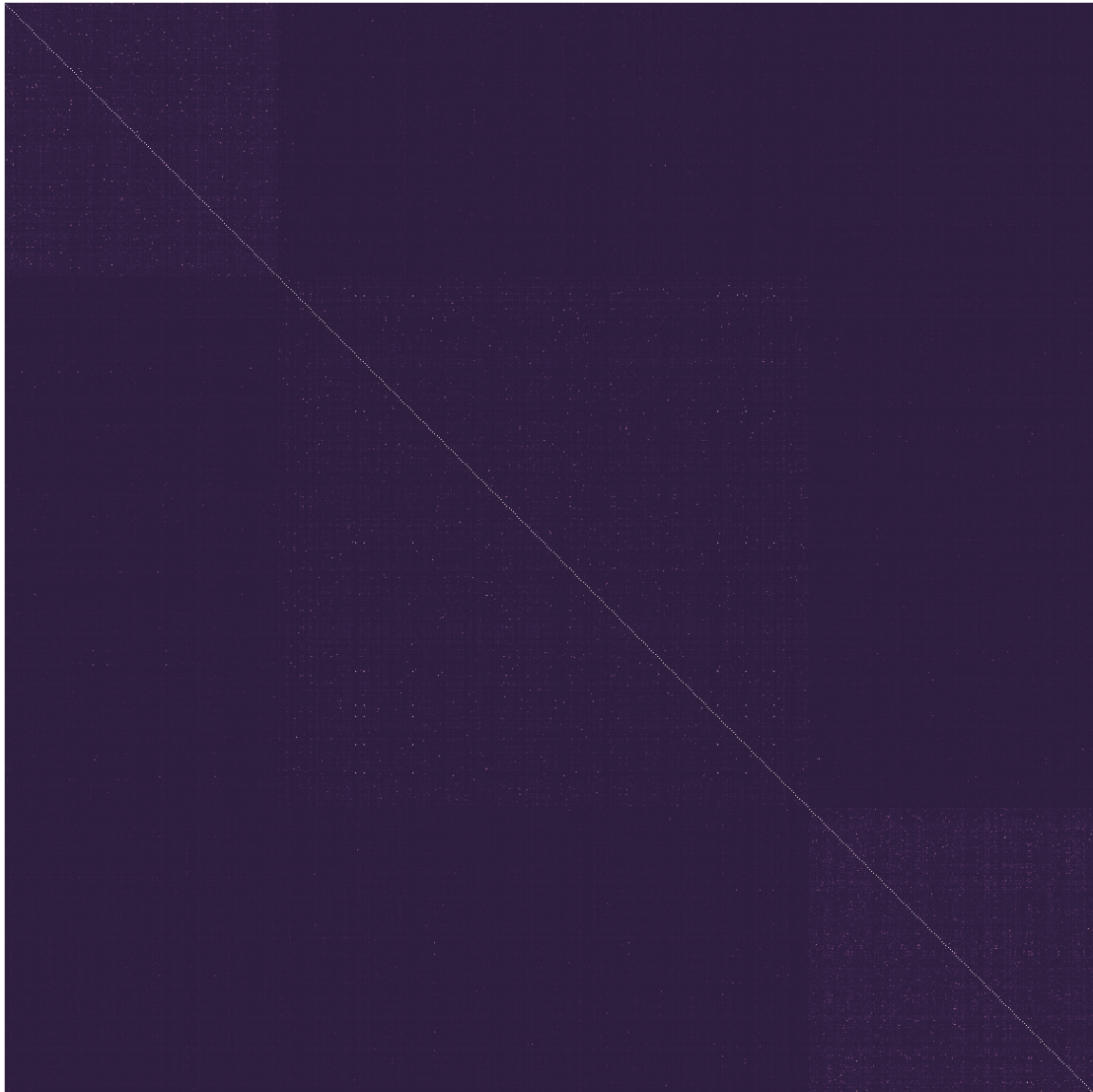
```
In [91]: euclidean_dists = metrics.euclidean_distances(data)
         labels = kmeans.labels_
         idx = np.argsort(labels)
         clustered_dists = euclidean_dists[idx][:,idx]
         fig, ax1 = plt.subplots(1,1,figsize=(15,15))
         dum = sns.heatmap(clustered_dists, xticklabels=False, yticklabels=False, ]
```

Let's visualize with MDS. Note that MDS is a slow algorithm and we can't do all 1700+ data points quickly, so we will take a random sample.

```
In [92]: dim = euclidean_dists.shape[0]
         subset = np.random.random_integers(0,dim-1,500)

         fit = mds.fit(euclidean_dists[subset][:,subset])
         pos = fit.embedding_

In [93]: cols = [['r','g','b'][l] for l in labels[subset]]
         plt.scatter(pos[:, 0], pos[:, 1], s=12, c=cols)

Out[93]: <matplotlib.collections.PathCollection at 0x11cfbddd8>
```