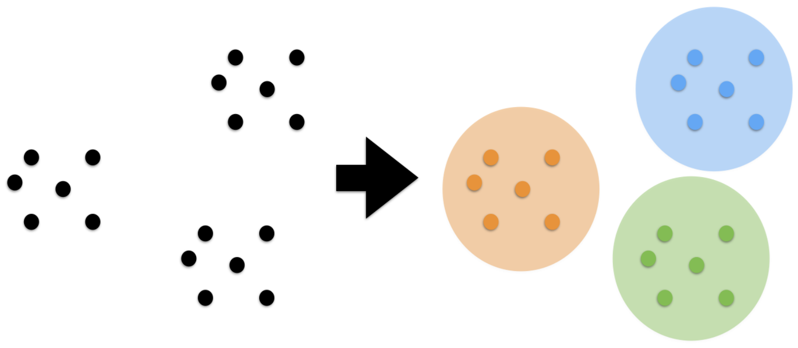
In [1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

Unsupervised Learning Part 2 -- Clustering

Clustering is the task of gathering samples into groups of similar samples according to some predefined similarity or distance (dissimilarity) measure, such as the Euclidean distance.



In this section we will explore a basic clustering task on some synthetic and real-world datasets.

Here are some common applications of clustering algorithms:

- Compression for data reduction
- Summarizing data as a reprocessing step for recommender systems
- Similarly:
  - grouping related web news (e.g. Google News) and web search results
  - grouping related stock quotes for investment portfolio management
  - building customer profiles for market analysis
- Building a code book of prototype samples for unsupervised feature extraction

Let's start by creating a simple, 2-dimensional, synthetic dataset:

In [2]:

```
from sklearn.datasets import make_blobs

X, y = make_blobs(random_state=42)
X.shape
```

Out[2]:

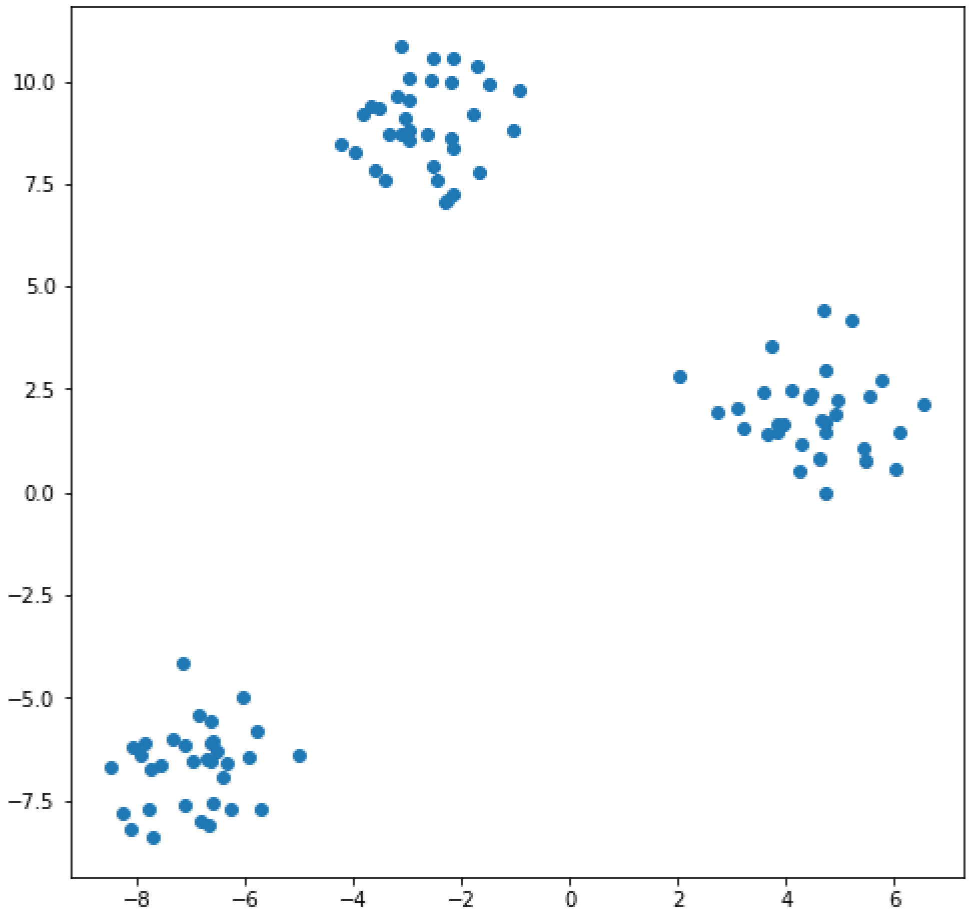
```
(100, 2)
```

In [3]:

```
plt.figure(figsize=(8, 8))
plt.scatter(X[:, 0], X[:, 1])
```

Out[3]:

```
<matplotlib.collections.PathCollection at 0x1a16afc748>
```



In the scatter plot above, we can see three separate groups of data points and we would like to recover them using clustering -- think of "discovering" the class labels that we already take for granted in a classification task.

Even if the groups are obvious in the data, it is hard to find them when the data lives in a high-dimensional space, which we can't visualize in a single histogram or scatterplot.

Now we will use one of the simplest clustering algorithms, K-means. This is an iterative algorithm which searches for three cluster centers such that the distance from each point to its cluster is minimized. The standard implementation of K-means uses the Euclidean distance, which is why we want to make sure that all our variables are measured on the same scale if we are working with real-world datastets. In the previous notebook, we talked about one technique to achieve this, namely, standardization.

Question:

- what would you expect the output to look like?

In [4]:

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=3, random_state=42)
```

We can get the cluster labels either by calling fit and then accessing the `labels_` attribute of the K means estimator, or by calling `fit_predict`. Either way, the result contains the ID of the cluster that each point is assigned to.

In [5]:

```
labels = kmeans.fit_predict(X)
```

In [6]:

```
labels
```

Out[6]:

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

In [7]:

```
np.all(y == labels)
```

Out[7]:

```
False
```

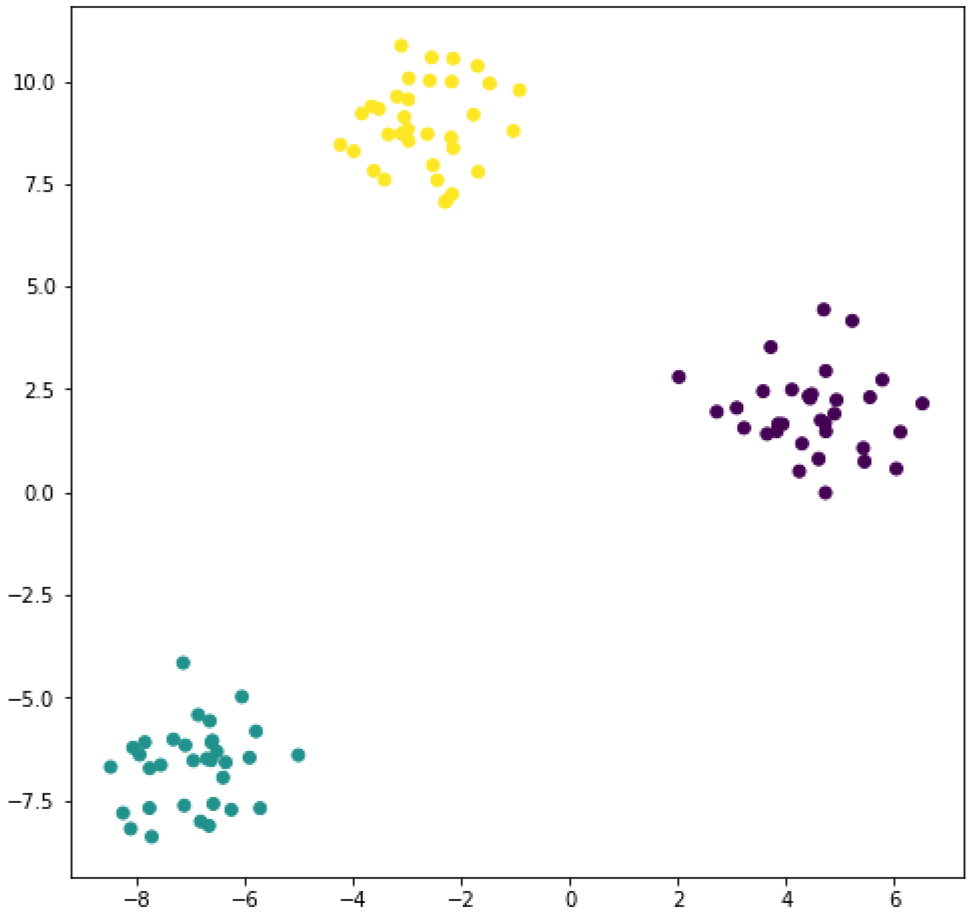
Let's visualize the assignments that have been found

In [8]:

```
plt.figure(figsize=(8, 8))
plt.scatter(X[:, 0], X[:, 1], c=labels)
```

Out[8]:

```
<matplotlib.collections.PathCollection at 0x1a179b8198>
```



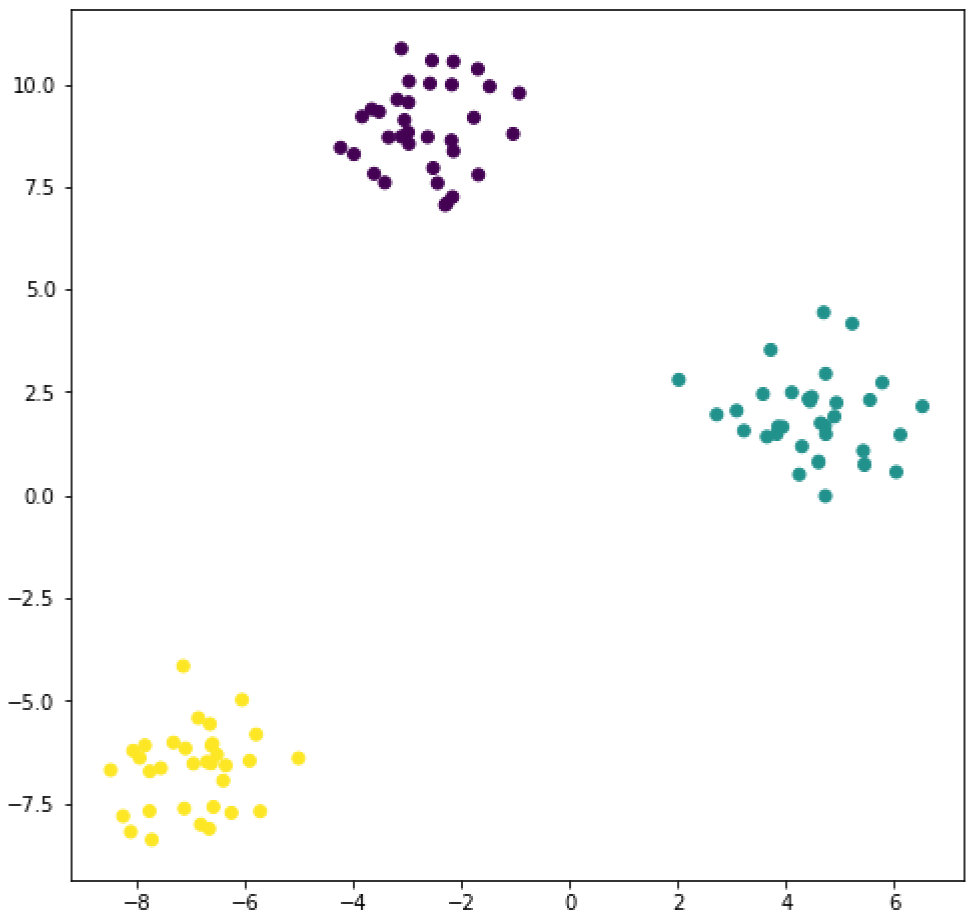
Compared to the true labels:

In [9]:

```
plt.figure(figsize=(8, 8))
plt.scatter(X[:, 0], X[:, 1], c=y)
```

Out[9]:

```
<matplotlib.collections.PathCollection at 0x1a17a299b0>
```



Here, we are probably satisfied with the clustering results. But in general we might want to have a more quantitative evaluation. How about comparing our cluster labels with the ground truth we got when generating the blobs?

In [10]:

```
from sklearn.metrics import confusion_matrix, accuracy_score

print('Accuracy score:', accuracy_score(y, labels))
print(confusion_matrix(y, labels))
```

```
Accuracy score: 0.0
[[ 0  0 34]
 [33  0  0]
 [ 0 33  0]]
```

In [11]:

```
np.mean(y == labels)
```

Out[11]:

```
0.0
```

EXERCISE:

- After looking at the "True" label array y, and the scatterplot and `labels` above, can you figure out why our computed accuracy is 0.0, not 1.0, and can you fix it?

Even though we recovered the partitioning of the data into clusters perfectly, the cluster IDs we assigned were arbitrary, and we can not hope to recover them. Therefore, we must use a different scoring metric, such as `adjusted_rand_score`, which is invariant to permutations of the labels:

In [12]:

```
from sklearn.metrics import adjusted_rand_score

adjusted_rand_score(y, labels)
```

Out[12]:

```
1.0
```

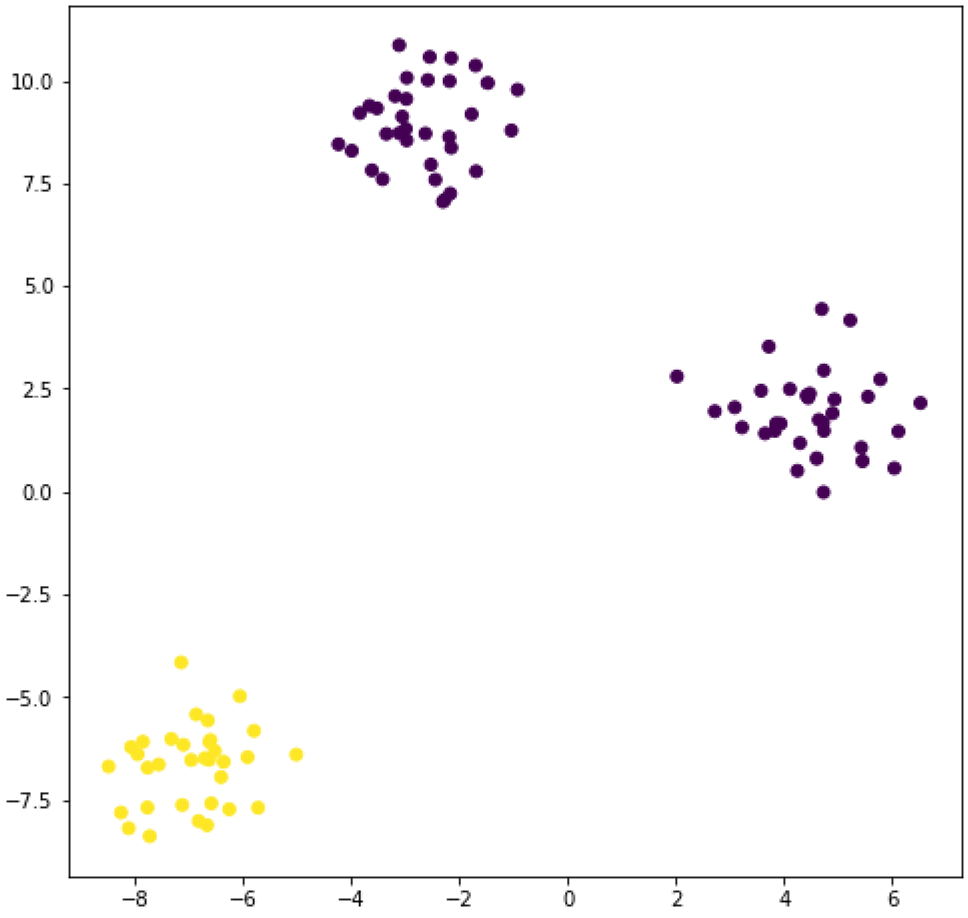
One of the "short-comings" of K-means is that we have to specify the number of clusters, which we often don't know *apriori*. For example, let's have a look what happens if we set the number of clusters to 2 in our synthetic 3-blob dataset:

In [13]:

```
kmeans = KMeans(n_clusters=2, random_state=42)
labels = kmeans.fit_predict(X)
plt.figure(figsize=(8, 8))
plt.scatter(X[:, 0], X[:, 1], c=labels)
```

Out[13]:

```
<matplotlib.collections.PathCollection at 0x1a17ba5400>
```



In [14]:

```
kmeans.cluster_centers_
```

Out[14]:

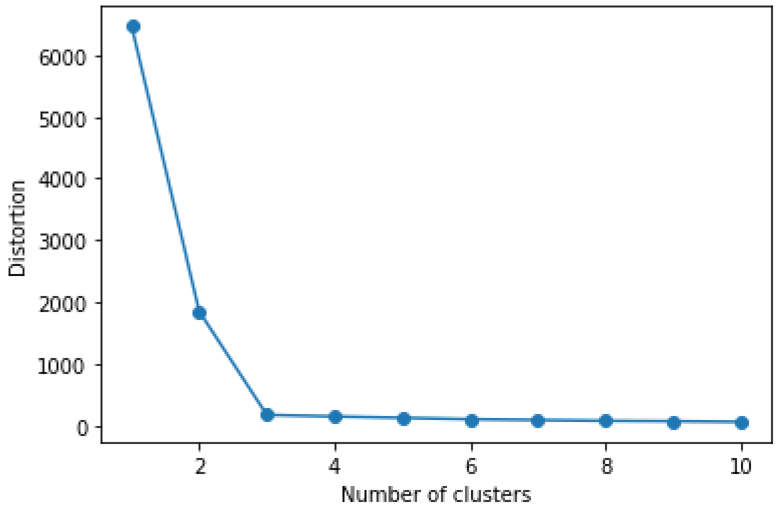
```
array([[ 0.86236563,  5.48955564],
       [-6.95170962, -6.67621669]])
```

The Elbow Method

The Elbow method is a "rule-of-thumb" approach to finding the optimal number of clusters. Here, we look at the cluster dispersion for different values of k:

In [15]:

```
distortions = []
for i in range(1, 11):
    km = KMeans(n_clusters=i,
                random_state=0)
    km.fit(X)
    distortions.append(km.inertia_)

plt.plot(range(1, 11), distortions, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Distortion')
plt.show()
```



Then, we pick the value that resembles the "pit of an elbow." As we can see, this would be k=3 in this case, which makes sense given our visual expection of the dataset previously.

Clustering comes with assumptions: A clustering algorithm finds clusters by making assumptions with samples should be grouped together. Each algorithm makes different assumptions and the quality and interpretability of your results will depend on whether the assumptions are satisfied for your goal. For K-means clustering, the model is that all clusters have equal, spherical variance.

In general, there is no guarantee that structure found by a clustering algorithm has anything to do with what you were interested in.

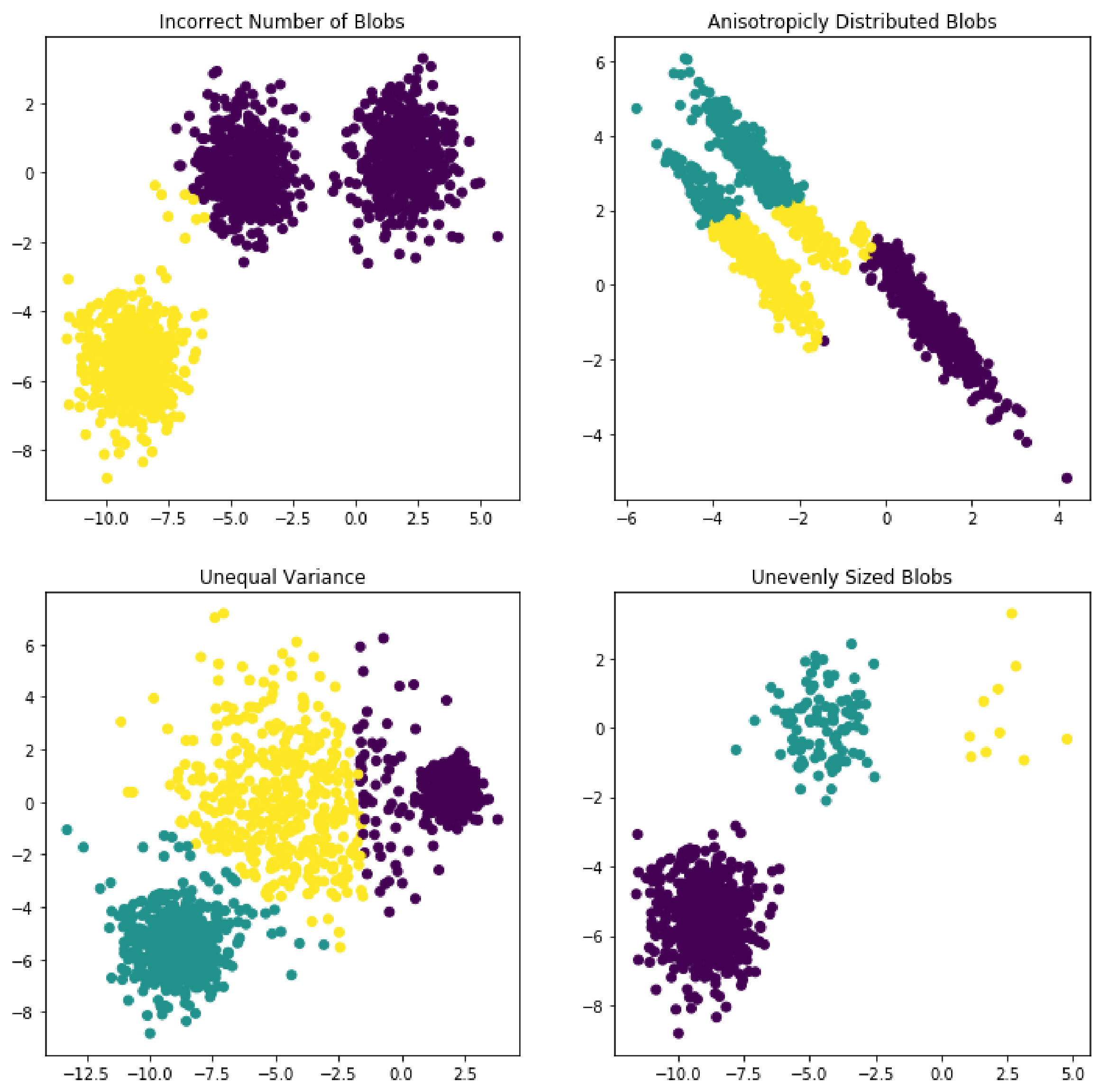
We can easily create a dataset that has non-isotropic clusters, on which kmeans will fail:

In [16]:

```
plt.figure(figsize=(12, 12))

n_samples = 1500
random_state = 170
X, y = make_blobs(n_samples=n_samples, random_state=random_state)

# Incorrect number of clusters
y_pred = KMeans(n_clusters=2, random_state=random_state).fit_predict(X)

plt.subplot(221)
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
plt.title("Incorrect Number of Blobs")

# Anisotropicly distributed data
transformation = [[0.60834549, -0.63667341], [-0.40887718, 0.85253229]]
X_aniso = np.dot(X, transformation)
y_pred = KMeans(n_clusters=3, random_state=random_state).fit_predict(X_aniso)

plt.subplot(222)
plt.scatter(X_aniso[:, 0], X_aniso[:, 1], c=y_pred)
plt.title("Anisotropicly Distributed Blobs")

# Different variance
X_varied, y_varied = make_blobs(n_samples=n_samples,
                                cluster_std=[1.0, 2.5, 0.5],
                                random_state=random_state)
y_pred = KMeans(n_clusters=3, random_state=random_state).fit_predict(X_varied)

plt.subplot(223)
plt.scatter(X_varied[:, 0], X_varied[:, 1], c=y_pred)
plt.title("Unequal Variance")

# Unevenly sized blobs
X_filtered = np.vstack((X[y == 0][:500], X[y == 1][:100], X[y == 2][:10]))
y_pred = KMeans(n_clusters=3,
                random_state=random_state).fit_predict(X_filtered)

plt.subplot(224)
plt.scatter(X_filtered[:, 0], X_filtered[:, 1], c=y_pred)
plt.title("Unevenly Sized Blobs")
```

Out[16]:

```
Text(0.5, 1.0, 'Unevenly Sized Blobs')
```

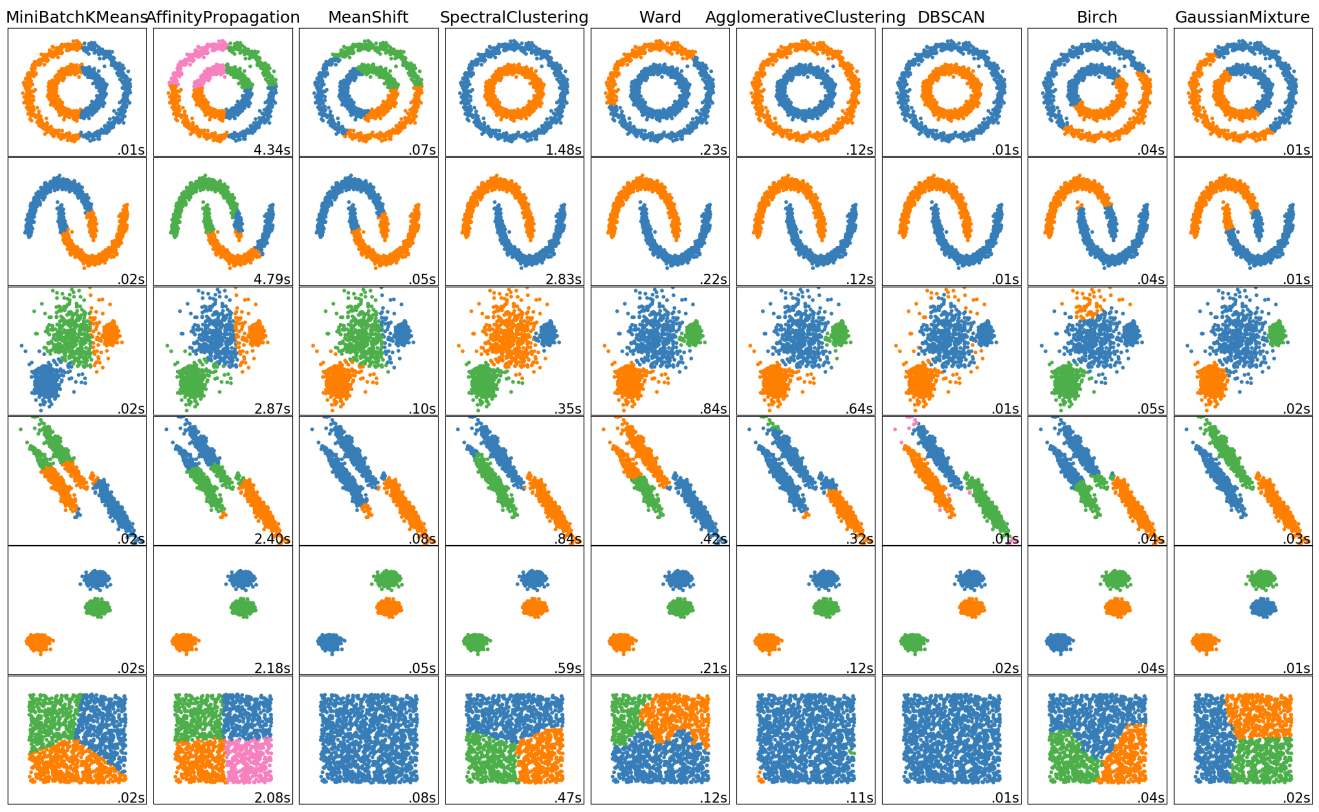


Some Notable Clustering Routines

The following are two well-known clustering algorithms.

- `sklearn.cluster.KMeans` :
  The simplest, yet effective clustering algorithm. Needs to be provided with the number of clusters in advance, and assumes that the data is normalized as input (but use a PCA model as preprocessor).
- `sklearn.cluster.MeanShift` :
  Can find better looking clusters than KMeans but is not scalable to high number of samples.
- `sklearn.cluster.DBSCAN` :
  Can detect irregularly shaped clusters based on density, i.e. sparse regions in the input space are likely to become inter-cluster boundaries. Can also detect outliers (samples that are not part of a cluster).
- `sklearn.cluster.AffinityPropagation` :
  Clustering algorithm based on message passing between data points.
- `sklearn.cluster.SpectralClustering` :
  KMeans applied to a projection of the normalized graph Laplacian: finds normalized graph cuts if the affinity matrix is interpreted as an adjacency matrix of a graph.
- `sklearn.cluster.Ward` :
  Ward implements hierarchical clustering based on the Ward algorithm, a variance-minimizing approach. At each step, it minimizes the sum of squared differences within all clusters (inertia criterion).

Of these, Ward, SpectralClustering, DBSCAN and Affinity propagation can also work with precomputed similarity matrices.



EXERCISE: digits clustering:

- Perform K-means clustering on the digits data, searching for ten clusters. Visualize the cluster centers as images (i.e. reshape each to 8x8 and use ``plt.imshow``) Do the clusters seem to be correlated with particular digits? What is the ``adjusted_rand_score``?
- Visualize the projected digits as in the last notebook, but this time use the cluster labels as the color. What do you notice?

In [17]:

```
from sklearn.datasets import load_digits
digits = load_digits()
# ...
```

In [18]:

```
# %load solutions/08B_digits_clustering.py
```