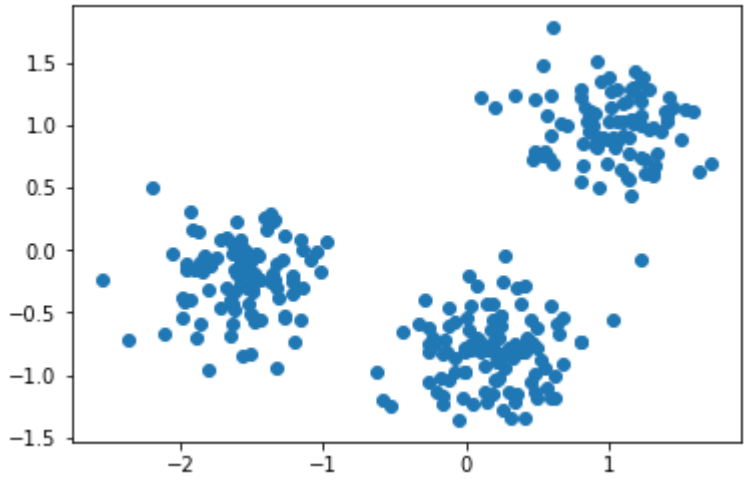
Reference

DataCamp course

Clustering for dataset exploration

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

points = np.loadtxt('points.csv',delimiter = ',')
xs = points[:,0] #For DataFrame, then should be xs = points.loc[:,0]
ys = points[:,1]
plt.scatter(xs,ys)
plt.show()
```

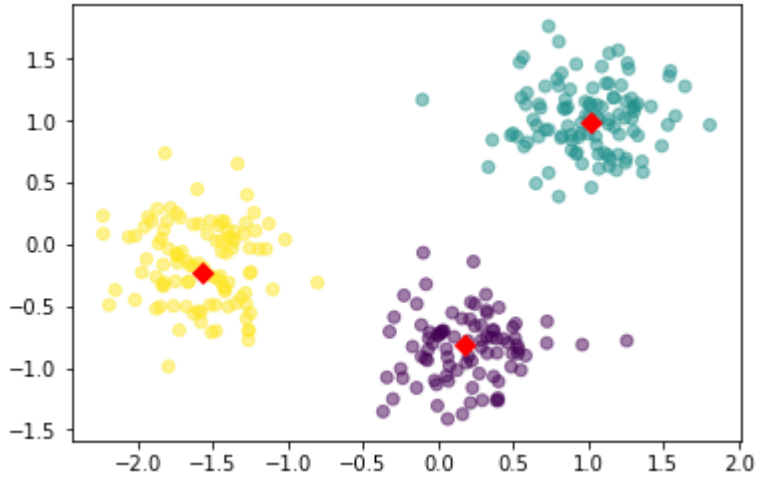


In [3]:
```
from sklearn.cluster import KMeans
new_points = np.loadtxt('new_points.csv', delimiter=",")

model = KMeans(n_clusters = 3)
model.fit(points)
labels = model.predict(new_points)
```

Inspect your clustering

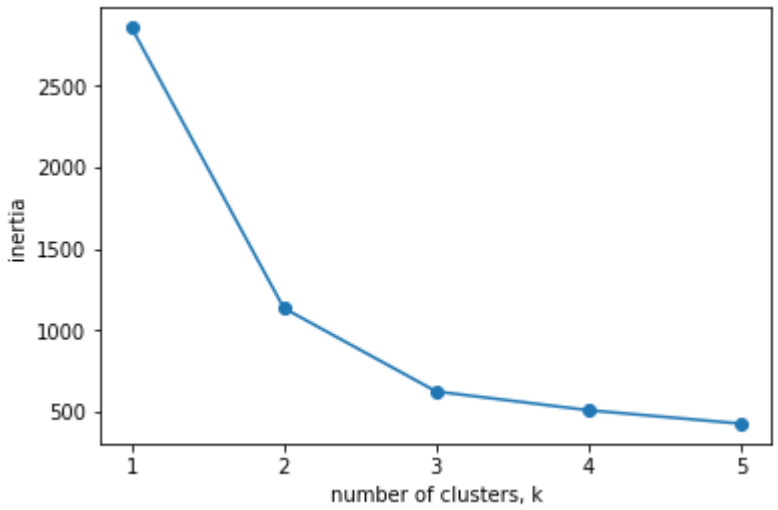
Now inspect the clustering performed in the previous exercise. A solution to the previous exercise has already run, so new_points is an array of points and labels is the array of their cluster labels.

In [3]:
```
import matplotlib.pyplot as plt
xs = new_points[:,0]
ys = new_points[:,1]

plt.scatter(xs,ys,c=labels,alpha = 0.5) # c=labels is to color the points, by what default?
centroids = model.cluster_centers_
centroids_x = centroids[:,0]
centroids_y = centroids[:,1]

plt.scatter(centroids_x,centroids_y, marker = 'D', s = 50, c = 'red')
# s= 50 sets the marker size.
plt.show()
```



How many clusters of grain?

In [4]:
```
import pandas as pd
samples = pd.read_csv('seeds.csv').values

ks = range(1, 6)
inertias = []

for k in ks:
    model = KMeans(n_clusters = k)
    model.fit(samples)
    inertias.append(model.inertia_)
plt.plot(ks, inertias, '-o')
plt.xlabel('number of clusters, k')
plt.ylabel('inertia')
plt.xticks(ks)
plt.show()
```



The inertia decreases very slowly from 3 clusters to 4, so it looks like 3 clusters would be a good choice for this data.

In fact, the grain samples come from a mix of 3 different grain varieties: "Kama", "Rosa" and "Canadian". In this exercise, cluster the grain samples into three clusters, and compare the clusters to the grain varieties using a cross-tabulation.

You have the array samples of grain samples, and a list varieties giving the grain variety for each sample.

In [5]:
```
import pandas as pd
import pickle
with open ('varieties', 'rb') as fp:
    varieties = pickle.load(fp)

model = KMeans(n_clusters = 3)

labels = model.fit_predict(samples)
# Using .fit_predict() is the same as using .fit() followed by .predict()

df = pd.DataFrame({'labels': labels, 'varieties': varieties})
ct = pd.crosstab(df['labels'],df['varieties'])
print(ct)
```

```
varieties  Canadian wheat  Kama wheat  Rosa wheat
labels
0                       0           1          60
1                      70           5           0
2                       0          64          10
```

Scaling fish data for clustering

In [7]:
```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
scaler = StandardScaler()
kmeans = KMeans(n_clusters = 4)

pipeline = make_pipeline(scaler,kmeans)
```

Use the **standardization** and clustering pipeline from the previous exercise to cluster the fish by their measurements, and then create a cross-tabulation to compare the cluster labels with the fish species.

In [8]:
```
import pandas as pd
import numpy as np
import pickle
with open ('species', 'rb') as fp:
    species = pickle.load(fp)

samples = pd.read_csv('fish.csv',delimiter = ',', index_col = 0, header = None).values
pipeline.fit(samples)
labels = pipeline.predict(samples)
df = pd.DataFrame({'labels': labels, 'species': species})
ct = pd.crosstab(df['labels'], df['species'])
print(ct)
```

```
species  Bream  Pike  Roach  Smelt
labels
0            1     0     19      1
1           33     0      1      0
2            0    17      0      0
3            0     0      0     13
```

Clustering stocks using KMeans

Some stocks are more expensive than others. To account for this, include a Normalizer at the beginning of your pipeline. The Normalizer will separately transform each company's stock price to a relative scale before the clustering begins.

Note that Normalizer() is different to StandardScaler(). While StandardScaler() standardizes features by removing the mean and scaling to unit variance, Normalizer() rescales each sample - here, each company's stock price - independently of the other.

In [14]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import Normalizer

df = pd.read_csv('company-stock-movements-2010-2015-incl.csv',
                         delimiter = ',', header = None, skiprows = 1, index_col = 0)
print(df.shape)
movements = df.values

normalizer = Normalizer()
kmeans = KMeans(n_clusters = 10)

pipeline = make_pipeline(normalizer,kmeans)
pipeline.fit(movements)
```

```
(60, 963)
```

Out[14]:
```
Pipeline(memory=None,
     steps=[('normalizer', Normalizer(copy=True, norm='l2')), ('kmeans', KMeans(algorithm='auto', copy_x=True,
init='k-means++', max_iter=300,
    n_clusters=10, n_init=10, n_jobs=1, precompute_distances='auto',
    random_state=None, tol=0.0001, verbose=0))])
```

Which stocks move together?

In the previous exercise, you clustered companies by their daily stock price movements. So which company have stock prices that tend to change in the same way? You'll now inspect the cluster labels from your clustering to find out.

In [11]:
```
import pandas as pd

with open ('companies', 'rb') as fp:
     companies = pickle.load(fp)
samples = pd.read_csv('fish.csv',delimiter = ',', index_col = 0, header = None).values

labels = pipeline.predict(movements)
df = pd.DataFrame({'labels': labels, 'companies': companies})

#print(df.sort_values('labels'))
```

Visualization with hierarchical clustering and t-SNE

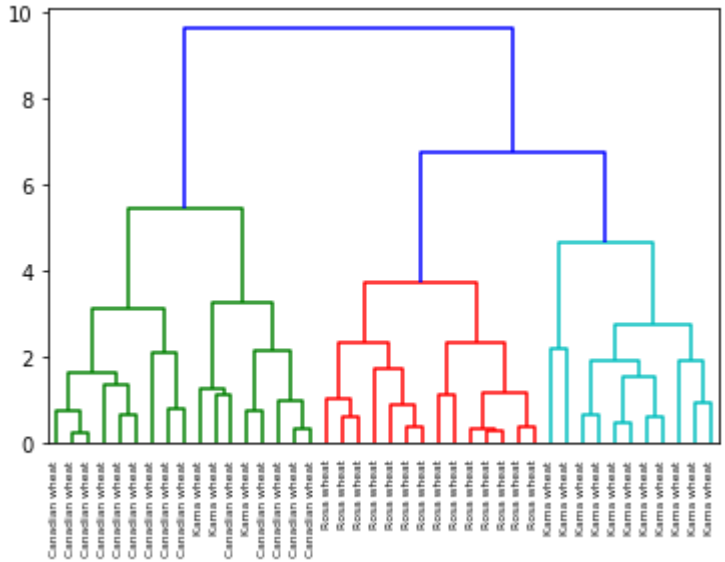
With 5 data samples, there would be 4 merge operations, and with 6 data samples, there would be 5 merges, and so on.

Hierarchical clustering of the grain data

SciPy linkage() function performs hierarchical clustering on an array of samples. Use the linkage() function to obtain a hierarchical clustering of the grain samples, and use dendrogram() to visualize the result. A sample of the grain measurements is provided in the array samples, while the variety of each grain sample is given by the list varieties.

In [16]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import linkage, dendrogram

with open ('anotherSeeds', 'rb') as fp:
    samples = pickle.load(fp)

with open ('another_varieties', 'rb') as fp:
    varieties = pickle.load(fp)

# Calculate the linkage: mergings
mergings = linkage(samples, method = 'complete')

# Plot the dendrogram, using varieties as labels
dendrogram(mergings,
           labels= varieties,
           leaf_rotation=90,
           leaf_font_size=6,
)
plt.show()
print(mergings.shape)
```



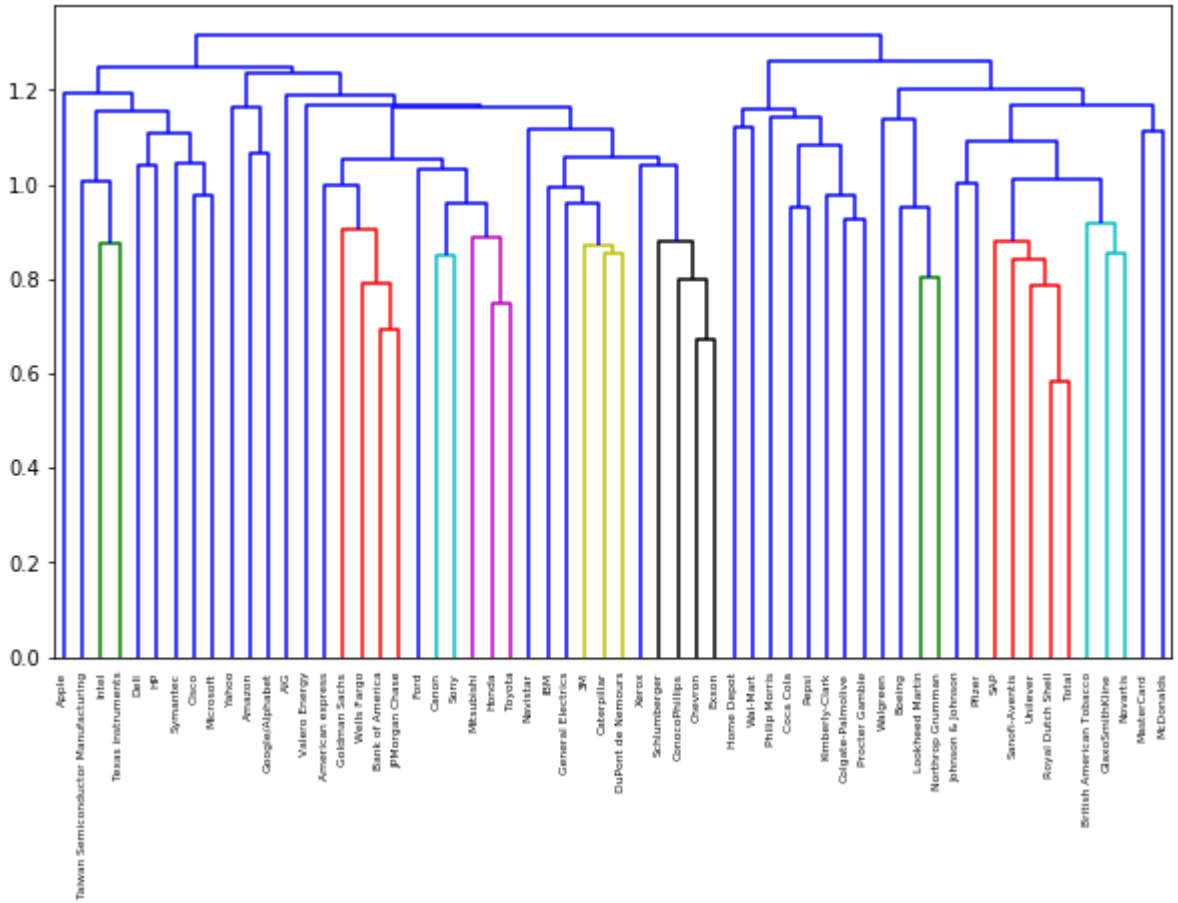

```
(41, 4)
```

Hierarchies of stocks

- Earlier k-means has been used to cluster companies according to their stock price movements.
- Now perform hierarchical clustering of the companies.
- A NumPy array `movements` of price movements is given, where the rows correspond to companies.
- A list of the company names `companies` is given.
- SciPy hierarchical clustering doesn't fit into a sklearn pipeline so you'll need to use the normalize() function from sklearn.preprocessing instead of Normalizer. **This is because sklearn is based on scipy, and thus earlier developed scipy will not fit into the pipeline of a later developed sklearn, right?**

In [11]:

```
from pylab import rcParams
rcParams['figure.figsize'] = 10, 6

from sklearn.preprocessing import normalize

normalized_movements = normalize(movements)

# Calculate the linkage: mergings
mergings = linkage(normalized_movements, method='complete')

dendrogram(
    mergings,
    labels=companies,
    leaf_rotation=90.,
    leaf_font_size=6
)
plt.show()
```



Which clusters are closest?

In the video, you learned that the linkage method defines how the distance between clusters is measured. In `complete` linkage, the distance between clusters is the distance between the furthest points of the clusters. In `single` linkage, the distance between clusters is the distance between the closest points of the clusters.

Consider the three clusters in the diagram. Which of the following statements are true?

A. In single linkage, cluster 3 is the closest to cluster 2.

B. In complete linkage, cluster 1 is the closest to cluster 2.

Answer: Both A and B

Different linkage, different hierarchical clustering!

In the video, you saw a hierarchical clustering of the voting countries at the Eurovision song contest using 'complete' linkage. Now, perform a hierarchical clustering of the voting countries with 'single' linkage, and compare the resulting dendrogram with the one in the video. Different linkage, different hierarchical clustering!

You are given an array samples. Each row corresponds to a voting country, and each column corresponds to a performance that was voted for. The list country_names gives the name of each voting country. This dataset was obtained from Eurovision.

Nothing new here, we just need set method = 'single' as below

mergings = linkage(samples, method = 'single')

Extracting the cluster labels

In the previous exercise, you saw that the intermediate clustering of the grain samples **at height 6 has 3 clusters**. Now, use the fcluster() function to extract the cluster labels for this intermediate clustering, and compare the labels with the grain varieties using a cross-tabulation.

The hierarchical clustering has already been performed and mergings is the result of the linkage() function. The list varieties gives the variety of each grain sample.

Note the mergings, lables, varieties below are different from datacamp.

In [ ]:
```
import pandas as pd
from scipy.cluster.hierarchy import fcluster

# Use fcluster to extract labels: labels
labels = fcluster(mergings,6,criterion='distance')

df = pd.DataFrame({'labels': labels, 'varieties': varieties})

ct = pd.crosstab(df['labels'], df['varieties'])

print(ct)
```

The output

```
varieties  Canadian wheat  Kama wheat  Rosa wheat
labels
1                      14           3           0
2                       0           0          14
3                       0          11           0
```

t-SNE visualization of grain dataset

In this exercise, you'll apply t-SNE to the grain samples data and inspect the resulting t-SNE features using a scatter plot. You are given an array samples of grain samples and a list variety_numbers giving the variety number of each grain sample.

t-SNE is a powerful tool for **visualizing high dimensional data.**

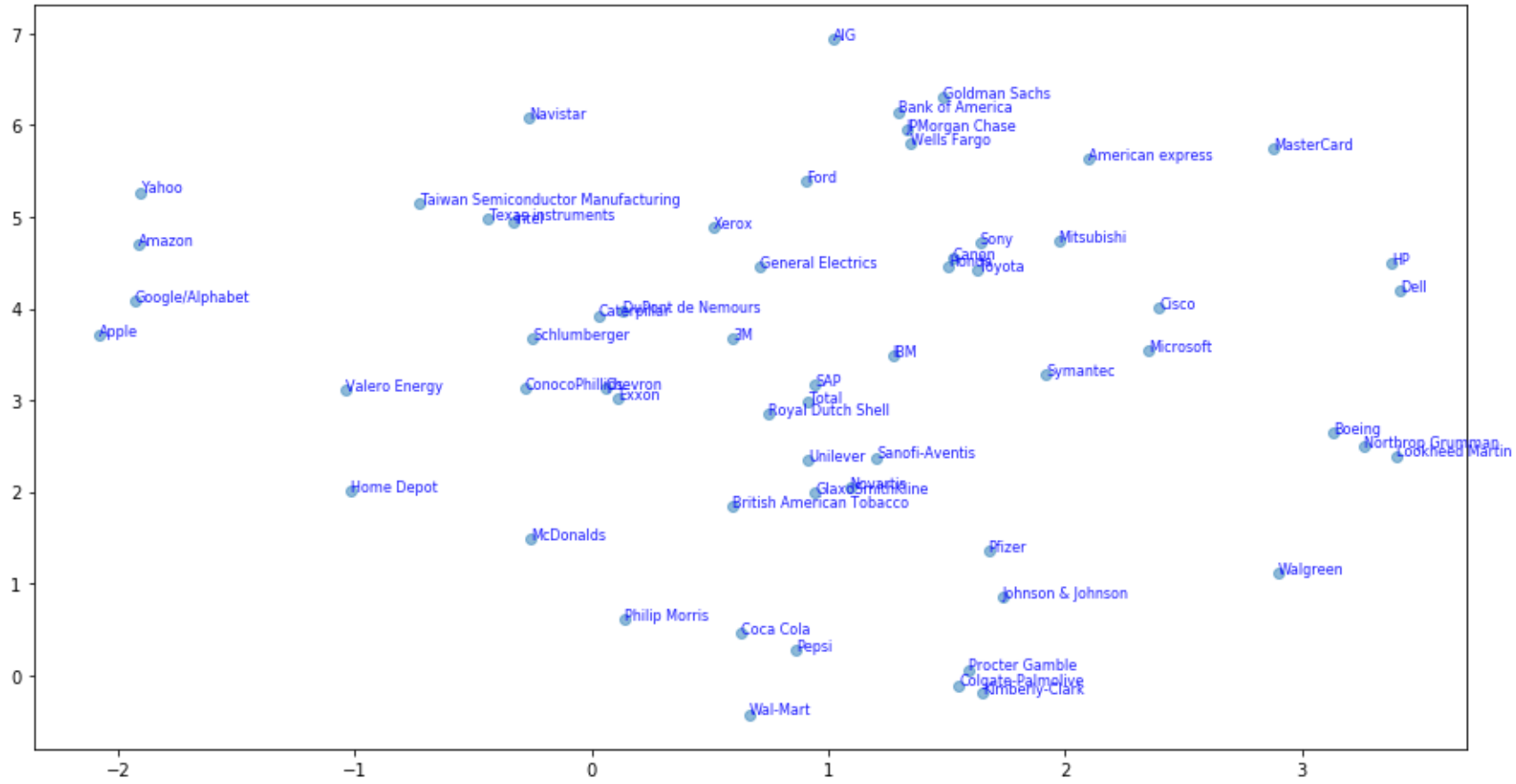
In [ ]:
```
from sklearn.manifold import TSNE

model = TSNE(learning_rate=200)

tsne_features = model.fit_transform(samples)

xs = tsne_features[:,0]

ys = tsne_features[:,1]

plt.scatter(xs, ys, c=variety_numbers)
plt.show()


plt.scatter(xs,ys,c=variety_numbers)
plt.show()
```

A t-SNE map of the stock market

t-SNE provides great visualizations when the individual samples can be labeled. In this exercise, you'll apply t-SNE to the company stock price data. A scatter plot of the resulting t-SNE features, labeled by the company names, gives you a map of the stock market! The stock price movements for each company are available as the array normalized_movements (these have already been normalized for you). The list companies gives the name of each company. PyPlot (plt) has been imported for you.

In [13]:

```
from pylab import rcParams
rcParams['figure.figsize'] = 15, 8

from sklearn.manifold import TSNE

model = TSNE(learning_rate = 50)

tsne_features = model.fit_transform(normalized_movements)

xs = tsne_features[:,0]

ys = tsne_features[:,1]

plt.scatter(xs,ys,alpha = 0.5)


for x, y, company in zip(xs, ys, companies):
    plt.annotate(company, (x, y), fontsize=8, alpha=0.9, color = 'blue')
plt.show()
```



It's visualizations such as this that make t-SNE such a powerful tool for extracting quick insights from high dimensional data.

In [14]:
```
normalized_movements
```

Out[14]:
```
array([[ 0.00302051, -0.00114574, -0.01775851, ..., -0.02791349,
         0.00437463, -0.10202026],
       [-0.02599391, -0.02639998, -0.00852927, ..., -0.00162466,
        -0.01624623,  0.02680614],
       [-0.02208986,  0.01184398, -0.02208986, ...,  0.04502568,
        -0.01654394,  0.03515588],
       ...,
       [ 0.01981027,  0.01059598,  0.02626006, ..., -0.01197837,
         0.01842816,  0.02211388],
       [ 0.0200991 ,  0.00223323, -0.01786587, ..., -0.0066997 ,
         0.00446647, -0.0066997 ],
       [ 0.01796837,  0.00112314,  0.        , ..., -0.00673829,
         0.02919855,  0.01123007]])
```

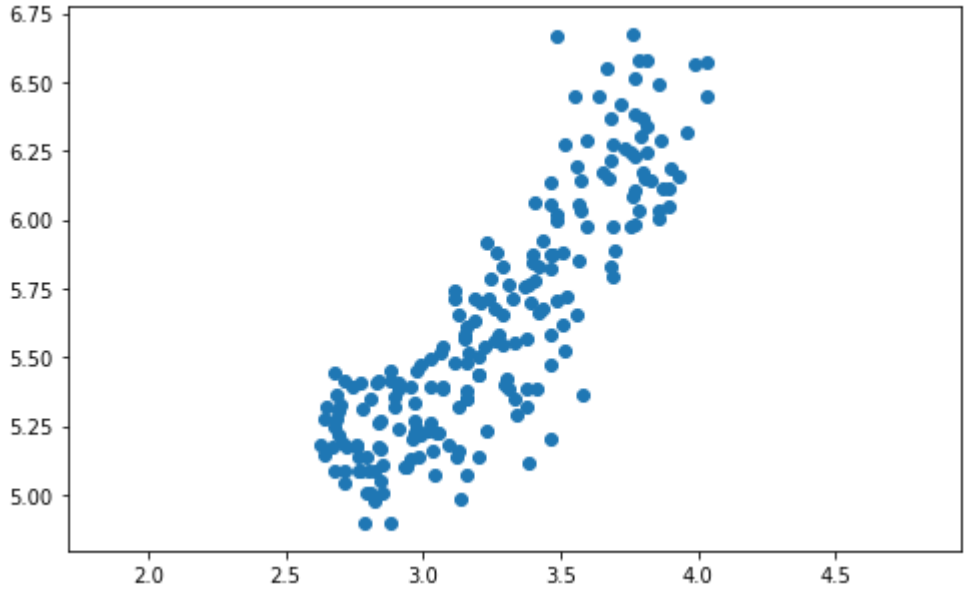
Decorrelating your data and dimension reduction

PCA is often used before supervised learning to improve model performance and generalization. It can also be useful for unsupervised learning. Here a variant of PCA will be used to cluster Wikipedia articles by their content.

Correlated data in nature

You are given an array grains giving the width and length of samples of grain. You suspect that width and length will be correlated. To confirm this, make a scatter plot of width vs length and measure their Pearson correlation.

In [4]:
```
import matplotlib.pyplot as plt
from scipy.stats import pearsonr
from pylab import rcParams
import pickle
rcParams['figure.figsize'] = 8, 5

import numpy as np
with open ('grains_1', 'rb') as fp:
    grains = pickle.load(fp)

#There are better ways to plot multiply such figures.
width = grains[:,0]
length = grains[:,1]
plt.scatter(width,length)
plt.axis('equal')
plt.show()

correlation, pvalue = pearsonr(width, length)
print(correlation,pvalue)
#What is the pvalue for here?
```




```
0.8604149377143467 8.121332906193427e-63
```

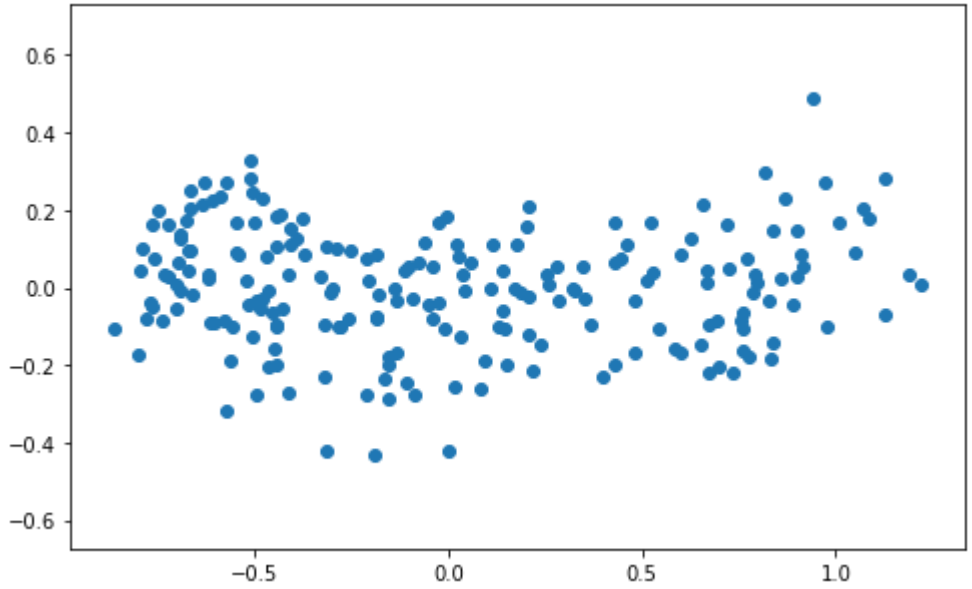
Decorrelating the grain measurements with PCA

You observed in the previous exercise that the width and length measurements of the grain are correlated. Now, you'll use PCA to decorrelate these measurements, then plot the decorrelated points and measure their Pearson correlation.

In [8]:

```
from sklearn.decomposition import PCA

model = PCA()
#PCA(n_components=None, copy=True, whiten=False, svd_solver='auto', tol=0.0, iterated_power='auto', random_state=
#We can choose different implementations of SVD. It can also use the scipy.sparse.linalg ARPACK implementation of

pca_features = model.fit_transform(grains)
#Return X_new : array-like, shape (n_samples, n_components)

xs = pca_features[:,0]
ys = pca_features[:,1]

plt.scatter(xs, ys)
plt.axis('equal')
plt.show()
correlation, pvalue = pearsonr(xs, ys)
print(correlation, pvalue)
# The p-value roughly indicates the probability of an uncorrelated system
# producing datasets that have a Pearson correlation at least as extreme
# as the one computed from these datasets. The p-values are not entirely
# reliable but are probably reasonable for datasets larger than 500 or so.
```

```
(210, 2)
```




```
7.47465689945304e-17 1.0
```

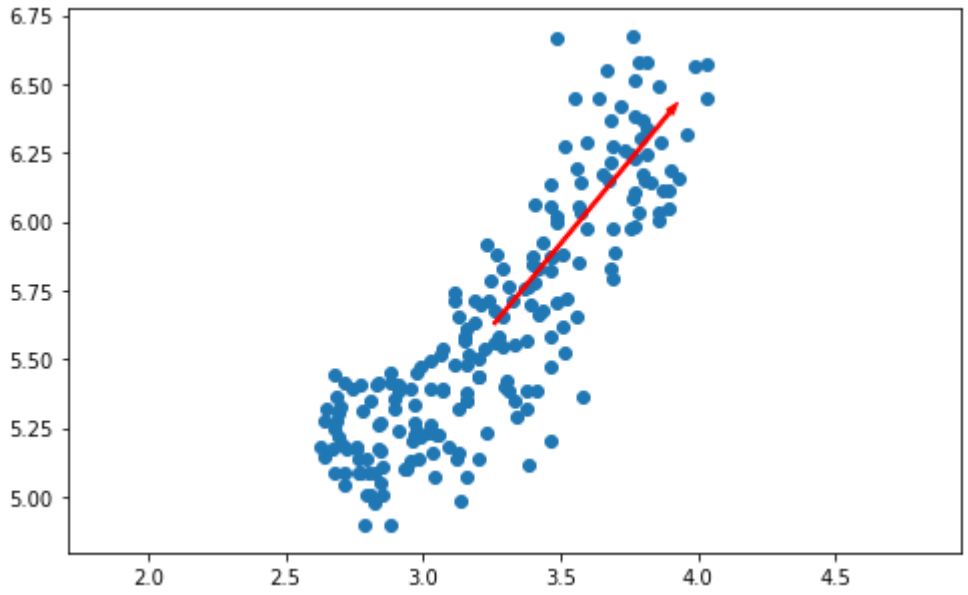
The first principal component

The first principal component of the data is the direction in which the data varies the most. In this exercise, your job is to use PCA to find the first principal component of the length and width measurements of the grain samples, and represent it as an arrow on the scatter plot.

The array grains gives the length and width of the grain samples.

In [16]:
```
# Make a scatter plot of the untransformed points
plt.scatter(grains[:,0], grains[:,1])
model = PCA()
model.fit(grains)
mean = model.mean_
first_pc = model.components_[0,:]
# components_ : array, shape (n_components, n_features)
# Principal axes in feature space, representing the directions of maximum variance in the data.
# So this is the eigenvector V but not XV, as in other naming convention.
# The components are sorted by explained_variance_.
# explained_variance_ : array, shape (n_components,)
# The amount of variance explained by each of the selected components.
# Equal to n_components largest eigenvalues of the covariance matrix of X.

print(type(first_pc))
print(first_pc.shape)
print(first_pc)

plt.arrow(mean[0], mean[1], first_pc[0], first_pc[1], color='red', width=0.01)
#Note each component here is a unit vector. Check it out!

# Keep axes on same scale
plt.axis('equal')
plt.show()
print(grains.shape)
```

```
<class 'numpy.ndarray'>
(2,)
[0.63910027 0.76912343]
```



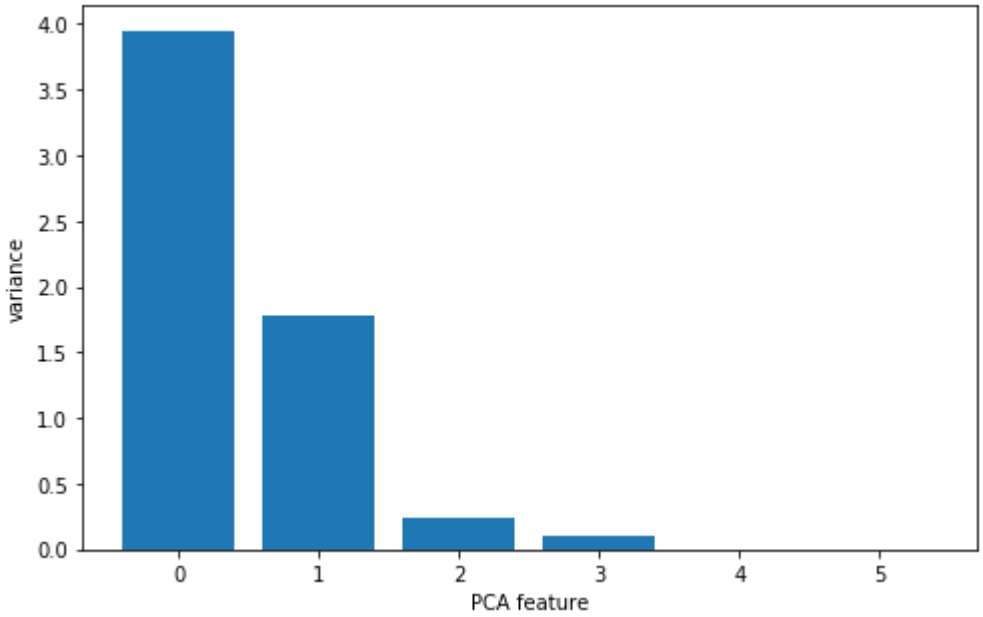

```
(210, 2)
```

Check another way of put different scaled-data in the same axis.

Variance of the PCA features

The fish dataset is 6-dimensional. But what is its intrinsic dimension? Standardize the features first and then use PCA to find it. **Check what happens if no standardization.**

In [18]:
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
import matplotlib.pyplot as plt

samples = pd.read_csv('fish.csv',delimiter = ',', index_col = 0, header = None).values
scaler = StandardScaler()
pca = PCA()
pipeline = make_pipeline(scaler,pca)
pipeline.fit(samples)
features = range(pca.n_components_)
plt.bar(features, pca.explained_variance_)
plt.xlabel('PCA feature')
plt.ylabel('variance')
plt.xticks(features)
plt.show()
```



Intrinsic dimension of the fish data

In the previous exercise, you plotted the variance of the PCA features of the fish measurements. Looking again at your plot, what do you think would be a reasonable choice for the "intrinsic dimension" of the the fish measurements? Recall that the **intrinsic dimension is the number of PCA features with significant variance.**

Dimension reduction of the fish measurements

In a previous exercise, you saw that 2 was a reasonable choice for the "intrinsic dimension" of the fish measurements. Now use PCA for dimensionality reduction of the fish measurements, retaining only the 2 most important components.

The fish measurements have already been scaled, and are available as scaled_samples.

In [25]:
```
from sklearn.decomposition import PCA

with open ('scaled_sample', 'rb') as fp:
    scaled_samples = pickle.load(fp)

print(scaled_samples.shape)
print(scaled_samples[0:4,0:6])

pca = PCA(n_components=2)
pca.fit(scaled_samples)
pca_features = pca.transform(scaled_samples)
print(pca_features.shape)
print(pca_features[0:4,0:6])
```

```
(85, 6)
[[-0.50109735 -0.36878558 -0.34323399 -0.23781518  1.0032125   0.25373964]
 [-0.37434344 -0.29750241 -0.26893461 -0.14634781  1.15869615  0.44376493]
 [-0.24230812 -0.30641281 -0.25242364 -0.15397009  1.13926069  1.0613471 ]
 [-0.18157187 -0.09256329 -0.04603648  0.02896467  0.96434159  0.20623332]]
(85, 2)
[[-0.57640502 -0.94649159]
 [-0.36852393 -1.17103598]
 [-0.28028168 -1.59709225]
 [-0.00955427 -0.81967711]]
```

A tf-idf word-frequency array

tf-idf stands for Term frequency-inverse document frequency. The tf-idf weight is a weight often used in information retrieval and text mining. Variations of the tf-idf weighting scheme are often used by search engines in scoring and ranking a document's relevance given a query.

In this exercise, you'll create a tf-idf word frequency array for a toy collection of documents. For this, use the TfidfVectorizer from sklearn. It transforms a list of documents into a word frequency array, which it outputs as a csr_matrix. It has fit() and transform() methods like other sklearn objects.

You are given a list documents of toy documents about pets.

In [24]:
```
from sklearn.feature_extraction.text import TfidfVectorizer

documents = ['cats say meow', 'dogs say woof', 'dogs chase cats']
tfidf = TfidfVectorizer()

csr_mat = tfidf.fit_transform(documents)

print(csr_mat.toarray())

words = tfidf.get_feature_names()

print(words)
print(csr_mat)
```

```
[[0.51785612 0.         0.         0.68091856 0.51785612 0.        ]
 [0.         0.         0.51785612 0.         0.51785612 0.68091856]
 [0.51785612 0.68091856 0.51785612 0.         0.         0.        ]]
['cats', 'chase', 'dogs', 'meow', 'say', 'woof']
  (0, 0)        0.5178561161676974
  (0, 4)        0.5178561161676974
  (0, 3)        0.680918560398684
  (1, 4)        0.5178561161676974
  (1, 2)        0.5178561161676974
  (1, 5)        0.680918560398684
  (2, 0)        0.5178561161676974
  (2, 2)        0.5178561161676974
  (2, 1)        0.680918560398684
```

Clustering Wikipedia part I

- TruncatedSVD is able to perform PCA on sparse arrays in csr_matrix format, such as word-frequency arrays. Combine your knowledge of TruncatedSVD and k-means to cluster some popular pages from Wikipedia. In this exercise, build the pipeline. In the next exercise, you'll apply it to the word-frequency array of some Wikipedia articles.
- Create a Pipeline object consisting of a TruncatedSVD followed by KMeans. (This time, we've precomputed the word-frequency matrix for you, so there's no need for a TfidfVectorizer).

Comments

Sometimes we need compressed-sparse-row format. However, we cannot use PCA to do dimension deduciton as PCA does not accept csr matrix. Therefore, we use TruncatedSVD (still from sklearn) instead. It does the same thing as PCA but accept csr matrix. Also, the following has not only do the PCA-like job by TruncatedSVD (i.e. reduce dimension) but also using the results (the reduced dimension 50?) to do cluster with KMeans?.

In [23]:
```
from sklearn.decomposition import TruncatedSVD
from sklearn.cluster import KMeans
from sklearn.pipeline import make_pipeline
from sklearn.pipeline import Pipeline
#Note always from lowercase pipepline import uppercase Pipeline.


# Create a TruncatedSVD instance: svd
svd = TruncatedSVD(n_components = 50)

#svd.fit(articles)
#need figure out what are n_components here.
#Note TruncatedSVD accept csr_matrix argument, which is not the numpy array as PCA accept.
#So the n_components here should be different from that of PCA case earlier.

kmeans = KMeans(n_clusters = 6)

pipeline = make_pipeline(svd,kmeans)
```

Clustering Wikipedia part II

- An array articles of tf-idf word-frequencies of some popular Wikipedia articles, and a list titles of their titles are given. * Use the pipeline to cluster the Wikipedia articles.

In [30]:

```
import pandas as pd
from scipy import sparse

np_array = pd.read_csv('wikipedia-vectors.csv', index_col = 0).values.transpose()

# I read in this csv file and transform it into Compressed Sparse Row (csr) format. However it is different from
# the 'articles' in DataCamp. Later I transpose it and then it becomes same.
# That might be from the following reasons: The original EXCEL is with shape (13125,60). However, 13125 should be
# 60 are number of articles to be clustered. But because showing 13125 as column is no convenient in EXCEL, so th
# it, right? That is reason I have to transpose again because conventionally, features are with column names.
articles = sparse.csr_matrix(np_array)
#This method sparse.csr_matrix for transforming np_array to csr format is what I think about for np array. It sho
#within the data frame context because here we have values. There must be method for doing the opposite way: tran
#sparse matrix to regualar metrix.

print(articles.toarray().shape)
#print(titles)

# import pickle
# with open ('C:/Users/ljyan/Desktop/courseNotes/UnsupervisedLearningPython/titles', 'wb') as fp:
#      pickle.dump(titles,fp)
with open('titles','rb') as fp:
    titles = pickle.load(fp)


import pandas as pd
pipeline.fit(articles)

# Calculate the cluster labels: labels
labels = pipeline.predict(articles)
print(labels)
#Note that within the pipeline we have TruncatedSVD which is like PCA. So running it in different time will give
#different results. The labels will be different after each running. This is same either here or in DateCamp.
#Thus result below is also different each time of running.

df = pd.DataFrame({'label': labels, 'article': titles})

#Display df sorted by cluster label
#print(df.sort_values('label'))

#Note the labels for each cluster is exactly same, 10 each. How does this happens?
print(articles.toarray())
```

```
(60, 13125)
[4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 3 3 3 3 3 3 3 3 3 1 1 1 1 1 1 1
 1 1 1 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0]
[[0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.02960744 ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.01159441 0.         0.        ]
 ...
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.00610985 0.         ... 0.         0.00547551 0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]]
```

Discovering interpretable features

- A dimension reduction technique called "Non-negative matrix factorization" ("NMF") that expresses samples as combinations of interpretable parts. For example, it expresses documents as combinations of topics, and images in terms of commonly occurring visual patterns.
- Use NMF to build recommender systems that can find similar articles to read, or musical artists that match a person's listening history!

NMF applied to Wikipedia articles

- Apply NMF using the tf-idf word-frequency array of Wikipedia articles (given as a csr matrix articles)
- Fit the model and transform the articles.

Comments: Unlike PCA cannot work with csr_matrix and have to resort to Truncated..., NMF work both with numpy array and csr_matrix.

In [29]:
```
from sklearn.decomposition import NMF

model = NMF(n_components = 6)

model.fit(articles)

# Transform the articles: nmf_features
nmf_features = model.transform(articles)

# So the components of samples are 60?
# However, in the model = NMF(n_components = 6) We set 6, so only 6 components are considered. Let print.
# print(model.components_)
# print(len(model.components_[:,0]) ) #output 6
# print(len(model.components_[0]) ) #ouput 13125

# In NMF case, dimension of components is equal to the dimension of samples. Note it is the dimenstion of the NMF
# equals to the dimension of sampels.
# Print the NMF features
#print(nmf_features)
print(nmf_features.shape)
#ouput (60,6), can be (60,60) in n_components = 60 (maximum, as there are 60 rows)
print(model.components_.shape)
#output (6, 13125), can be (60, 13125) if n_components = 60.

#The features and components_ of the NMF model can be used to approximately to reconstruct the data set samples.
```

```
(60, 6)
(6, 13125)
```

NMF features of the Wikipedia articles

- Explore the NMF features created in the previous exercise. A solution to the previous exercise has been pre-loaded, so the array nmf_features is available. **Also available is a list titles giving the title of each Wikipedia article.**

Comments:

- The header in the wiki excel file such as HTT4 etc. are the title of each wikipedia article. However, those titles should not be the 'features' of the problem. It might be because the feature columns are too large (13125) that the EXCEL are displayed that way. When creating csr_matrix from EXCEL array, I need transpose the array to let the column names as column names, and rows are for article names.

- When investigating the features, notice that for both actors (my comments: the wiki article title is just the actor name), the NMF feature 3 has by far the highest value. This means that both articles are reconstructed using mainly the 3rd NMF component. In the next video, you'll see why: NMF components represent topics (for instance, acting!).

In [124]:
```
import pandas as pd

df = pd.DataFrame(nmf_features, index=titles)

print(df.loc['Anne Hathaway'])

# Print the row for 'Denzel Washington'
print(df.loc['Denzel Washington'])
#print(titles)
#df
```

```
0    0.003846
1    0.000000
2    0.000000
3    0.575686
4    0.000000
5    0.000000
Name: Anne Hathaway, dtype: float64
0    0.000000
1    0.005601
2    0.000000
3    0.422362
4    0.000000
5    0.000000
Name: Denzel Washington, dtype: float64
```

NMF reconstructs samples

- Study how NMF reconstructs samples from its components using the NMF feature values. On the right are the components of an NMF model(below).

  [[1. 0.5 0.] [0.2 0.1 2.1]]

If the NMF feature values of a sample are [2, 1], then which of the following is most likely to represent the original sample? [0.1203 0.1764 0.3195 0.141].

Answer is [2.2, 1.0, 2.0] 2 *1* + *1* 0.2 = 2.2 2 *0.5* + *1* 0.1 = 1.1 2 *0* + *1* 2.1 = 2.1

My Comments Note the above process can be represented by matrix multiplication. This is the origin of the name "non-negative matrix factorization".

NMF learns topics of documents

In the video, you learned when NMF is applied to documents, the components correspond to topics of documents, and the NMF features reconstruct the documents from the topics. Verify this for yourself for the NMF model that you built earlier using the Wikipedia articles. Previously, you saw that the 3rd NMF feature value was high for the articles about actors Anne Hathaway and Denzel Washington. In this exercise, identify the topic of the corresponding NMF component.

The NMF model you built earlier is available as model, while words is a list of the words that label the columns of the word-frequency array.

After you are done, take a moment to recognise the topic that the articles about Anne Hathaway and Denzel Washington have in common!

In [32]:
```
import pickle
# with open ('C:/Users/ljyan/Desktop/courseNotes/UnsupervisedLearningPython/words',  'wb') as fp:
#     #fp.dump(words) This is wrong
#     pickle.dump(words, fp)
with open ('words','rb') as fp:
    words = pickle.load(fp)
#In DataCamp, if I only type words and Enter then it only print part of results and thus I cannot copy them
#here and write to file. To avoid this, use print(words).
#by the way, there is a text file called 'wikipedia-vocabulary-utf8'. It contains all the words here but all stic
#without space. How can I use that file to obtain the word list?

# with open ('C:/Users/ljyan/Desktop/courseNotes/UnsupervisedLearningPython/titles', 'wb') as fp:
#     pickle.dump(titles,fp)
with open('titles','rb') as fp:
    titles = pickle.load(fp)
# Import pandas
import pandas as pd

# Create a DataFrame: components_df
components_df = pd.DataFrame(model.components_, columns = words)

# Print the shape of the DataFrame
print(components_df.shape)

# Select row 3: component
component = components_df.iloc[3]

# Print result of nlargest
print(component.nlargest())
# This gives the five words with the highest values for that component.
```

```
(6, 13125)
film       0.627993
award      0.253179
starred    0.245330
role       0.211490
actress    0.186432
Name: 3, dtype: float64
```

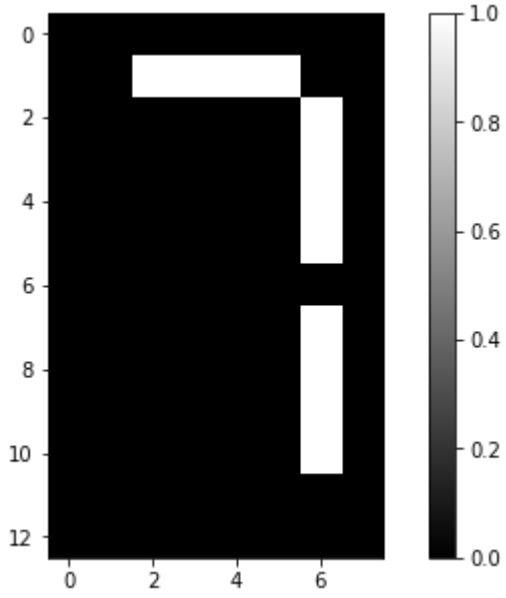
Take a moment to recognize the topics that the articles about Anne Hathaway and Denzel Washington have in common!

Explore the LED digits dataset

Use NMF to decompose grayscale images into their commonly occurring patterns. Firstly, explore the image dataset and see how it is encoded as an array. There are 100 images as a 2D array samples, where each row represents a single 13x8 image.

In [40]:
```
import pandas as pd
samples = pd.read_csv('lcd-digits.csv', header = None).values #.values but not values()
samples.shape

from matplotlib import pyplot as plt

digit = samples[0,:]

print(digit)

bitmap = digit.reshape(13,8)

print(bitmap)

plt.imshow(bitmap, cmap='gray', interpolation='nearest')
plt.colorbar()
plt.show()
```

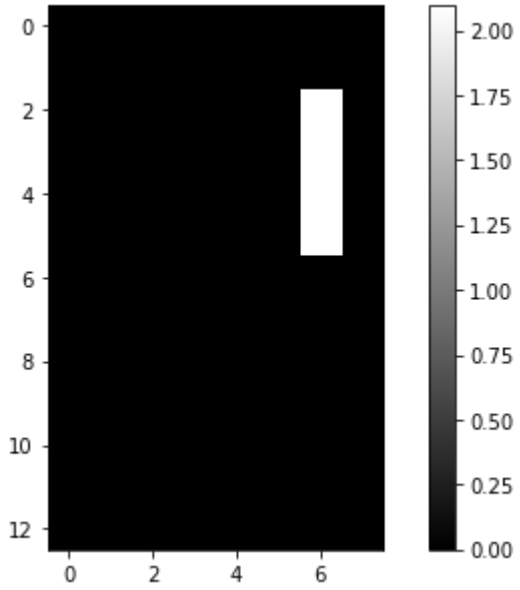
```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.
 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0.
 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0.]
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 1. 1. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```



NMF learns the parts of images

- Now use what you've learned about NMF to decompose the digits dataset. Again the digit images are given as 2D array samples.
- A function show_as_image() that displays the image encoded by any 1D array.
- After decomposition, take a moment to look through the plots and notice how NMF has expressed the digit as a sum of the components!

In [47]:
```
def show_as_image(sample):
    bitmap = sample.reshape((13, 8))
    plt.figure()
    plt.imshow(bitmap, cmap='gray', interpolation='nearest')
    plt.colorbar()
    plt.show()

from sklearn.decomposition import NMF

model = NMF(n_components = 7)
# 7 is the number of cells in a LED display.

features = model.fit_transform(samples)

for component in model.components_:
    show_as_image(component)


digit_features = features[0,:]

# Print digit_features
print(digit_features)
```

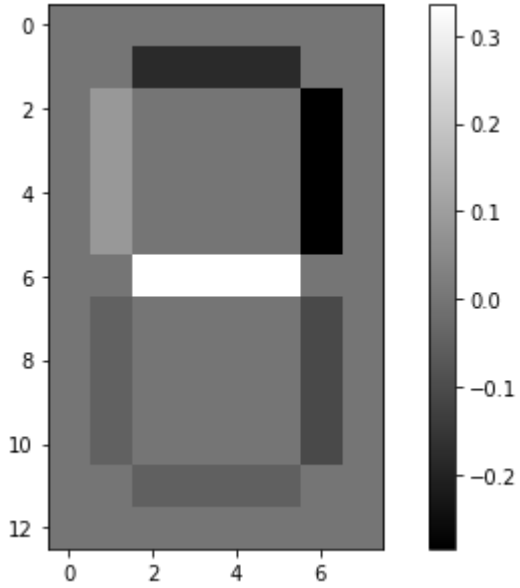


Take a moment to look through the plots and notice how NMF has expressed the digit as a sum of the components!

PCA doesn't learn parts

- Unlike NMF, PCA doesn't learn the parts of things. Its components do not correspond to topics (in the case of documents) or to parts of images, when trained on images. Verify this for yourself by inspecting the components of a PCA model fit to the dataset of LED digit images from the previous exercise. The images are available as a 2D array samples. Also available is a modified version of the show_as_image() function which colors a pixel red if the value is negative.
- Notice that the components of PCA do not represent meaningful parts of images of LED digits.

In [48]:
```
from sklearn.decomposition import PCA

model = PCA(n_components = 7)

features = model.fit_transform(samples)


for component in model.components_:  #IMPORTANT, understand model.components from here
    show_as_image(component)


#In
```



Notice that the components of PCA do not represent meaningful parts of images of LED digits!

Which articles are similar to 'Cristiano Ronaldo'?

Use NMF features and the cosine similarity to find similar articles. Apply this to the NMF model for popular Wikipedia articles, by finding the articles most similar to the article about the footballer Cristiano Ronaldo. The NMF features obtained earlier are available as nmf_features, while titles is a list of the article titles.

In [49]:
```
import pandas as pd
from sklearn.preprocessing import normalize

norm_features = normalize(nmf_features)
#nmf_features are calculated while ago.

df = pd.DataFrame(norm_features, index=titles)

article = df.loc['Cristiano Ronaldo']

# Compute the dot products: similarities
similarities = df.dot(article)

print(similarities.nlargest())
```

```
Cristiano Ronaldo               1.000000
Franck Ribéry                   0.999972
Radamel Falcao                  0.999942
Zlatan Ibrahimović              0.999942
France national football team   0.999923
dtype: float64
```

Recommend musical artists part I

- Use NMF to recommend popular music artists! You are given a sparse array artists whose rows correspond to artists and whose column correspond to users. The entries give the number of times each artist was listened to by each user.
- Build a pipeline and transform the array into normalized NMF features. The first step in the pipeline, MaxAbsScaler, transforms the data so that all users have the same influence on the model, regardless of how many different artists they've listened to. In the next exercise, you'll use the resulting normalized NMF features for recommendation!

In [ ]:
```
#artists is a csr_matrix converted from a (111,500) regular array.

from sklearn.decomposition import NMF
from sklearn.preprocessing import Normalizer, MaxAbsScaler
from sklearn.pipeline import make_pipeline

scaler = MaxAbsScaler()

nmf = NMF(n_components = 20)

normalizer = Normalizer()

pipeline = make_pipeline(scaler,nmf,normalizer)

norm_features = pipeline.fit_transform(artists)
```

Recommend musical artists part II

- Suppose you were a big fan of Bruce Springsteen - which other musicial artists might you like? Use your NMF features from the previous exercise and the cosine similarity to find similar musical artists.
- A solution to the previous exercise has been run, so norm_features is an array containing the normalized NMF features as rows. The names of the musical artists are available as the list artist_names.

In [ ]:
```
import pandas as pd

df = pd.DataFrame(norm_features,index = artist_names)

artist = df.loc['Bruce Springsteen']

similarities = df.dot(artist)

print(similarities.nlargest())
```

output

```
Bruce Springsteen    1.000000
Neil Young           0.956288
Van Morrison         0.872420
Leonard Cohen        0.862712
Bob Dylan            0.860227
dtype: float64
```