In [1]:
```
%matplotlib inline
import matplotlib
import seaborn as sns
sns.set()
matplotlib.rcParams['figure.dpi'] = 144
```

Unsupervised Learning

Unsupervised learning is a type of learning that occurs when you only have features $X = \{X_{ji}\}$ but not labels y_j. In this notebook, we will discuss two types of unsupervised learning: **dimensionality reduction** and **clustering**.

When we have a really high-dimensional data set, we often want to lower the dimensionality of the problem. For example, we might have a corpus of unlabeled documents. Rather than using each word as a potential feature, we want to develop a (shorter) list of topics and specify how related each document is to each topic. Similarly, even a collection of 100 x 100 image would (naively) be a problem with 10,000 features. These features will be highly redundant, and we would like to learn with a reduced set of independent features.

Relatedly, we may want to group observations into several clusters. For example, we might

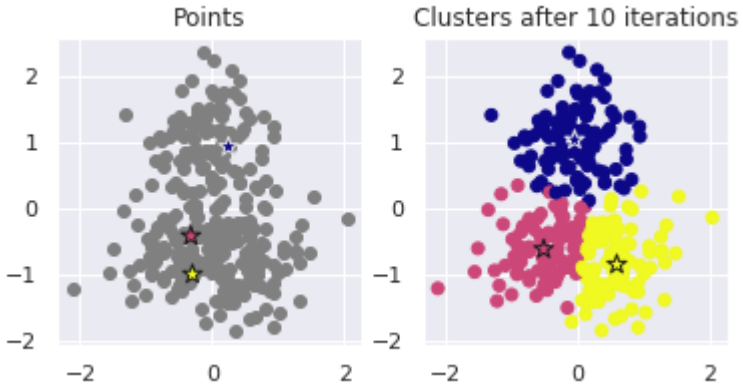
- Group genes and proteins that have similar functionality
- Group stocks with similar price fluctuations

Clustering is *not* classification. There is no "right" answer. Instead, there are a number of algorithms to use and metrics to judge them by. Below, we generate a cloud of points and use an algorithm called k-means to cluster them into three groups. (Note that this is an input. How do we choose the right number?)

In [2]:
```
# generate a blob of points
import numpy as np
from matplotlib import pylab as plt

np.random.seed(42)
X1 = np.vstack((np.random.normal(0, 0.5, (100, 2)) + np.array([0, 1]),
                np.random.normal(0, 0.5, (100, 2)) + np.array([-0.5, -0.7]),
                np.random.normal(0, 0.5, (100, 2)) + np.array([0.5, -0.7])))
```

In [3]:
```
from sklearn.cluster import KMeans

def plot_kmeans(X, n, start):
    def func(step=0):
        iters = step // 2
        if iters:
            km = KMeans(n_clusters=n, max_iter=iters, n_init=1, init=start)
            km.fit(X)
            centers = km.cluster_centers_.T
        else:
            centers = start.T
        if step % 2:
            km = KMeans(n_clusters=n, max_iter=iters+1, n_init=1, init=start)
            km.fit(X)
        plt.scatter(*X.T, c=(km.labels_ if step else '0.5'), cmap=plt.cm.plasm
a)
        plt.scatter(*centers, c=range(n), cmap=plt.cm.plasma, marker='*', s=10
0,
                    linewidth=1, edgecolors=['k' if i else '0.9' for i in rang
e(n)])
        plt.title(['Set centroids', 'Assign clusters'][step % 2])
    return func
```

In [4]:
```
plt.subplot(121)
plot_kmeans(X1, 3, X1[::100,:])(0)
plt.axis('image')
plt.title('Points')
plt.subplot(122)
plot_kmeans(X1, 3, X1[::100,:])(20)
plt.axis('image')
plt.title('Clusters after 10 iterations');
```



Metrics for clustering

For unsupervised learning, we will assume a set of feature vectors x_j. These correspond to the rows of the features matrix X_{ji}. These algorithms revolve around clustering, i.e. assigning each row $X_{j\cdot}$ to a cluster C_k such that the rows that share a cluster are more *similar* than ones from different clusters. There are several ways to do this, some which require ground truth labeling and some which do not.

Below we present mathematical details for **3 common metrics used in clustering**:

- Inertia, which can be thought of as a "within-cluster" sum of squares
- Silhouette Coefficient, which measures how well-defined your clusters are (dense and well-separated clusters = good)
- Mutual Information, which measures agreement between the ground truth values and your predicted values

Here are the details. The simplest metric for clustering is **Inertia** which is based on the second moment of each cluster (and if you are a physicist, this might be more appropriately called potential energy). Let μ_k be the mean of all rows in cluster k, i.e. for all elements $X_{j\cdot} \in C_k$,

$$\mu_k = \frac{1}{|C_k|} \sum_{X_{j\cdot} \in C_k} X_{j\cdot}.$$

Then the inertia is defined as the sum of the inertia (or second moment) of each cluster:

$$\sum_k \frac{1}{|C_k|} \sum_{X_{j\cdot} \in C_k} \|X_{j\cdot} - \mu_k\|_2^2 \ .$$

Another metric is called the **Silhouette Coefficient**. If a_j is the mean distance between a point $X_{j\cdot}$ and the other points in the same cluster C_k , and b_j is the mean distance between $X_{j\cdot}$ and the other points in the next nearest cluster C_k', then the coefficient is given by

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

Assuming you have ground truth values $\tilde{C}_k$ and predicted classification labels C_k, it is easy to calculate their **Mutual Information**. Mutual information has deep ties to information theory (see the Wikipedia article (http://en.wikipedia.org/wiki/Mutual_Information)). If N is the total number of samples (number of rows of X), we can define the probabilities

$$P_k = \frac{|C_k|}{N} \qquad \tilde{P}_k = \frac{|\tilde{C}_k|}{N} \qquad P_{k,l} = \frac{|C_k \cap \tilde{C}_l|}{N}$$

and then the mutual information is defined as

$$\sum_{k,l} P_{k,l} \log\left(\frac{P_{k,l}}{P_k \tilde{P}_l}\right) .$$

There are a plethora of metrics. For more information, including advantages and disadvantages of each metric, see this Scikit Learn page on Clustering Metrics (http://scikit-learn.org/stable/modules/clustering.html#clustering-evaluation).

Questions:

1. How do the values of each of the measures scale with the scaling of the feature space (e.g. $X \mapsto 2X$)?
2. For a fixed number of points, how do each of the measures scale with the number of clusters?

3. From the the cloud of points in this example (http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html#example-cluster-plot-

In [5]:
```
from sklearn.cluster import KMeans
from sklearn import metrics
from sklearn import datasets

# Fit KMeans onto the Iris dataset
dataset = datasets.load_iris()
X = dataset.data
k_means = KMeans().fit(X)
y = k_means.predict(X)

# Compute Silhoutte score on clusters:
print("Silhouette Score", metrics.silhouette_score(X, y, metric='euclidean'))

# Entropy
labels_true = [1, 1, 0, 1, 2, 2, 1, 2, 0]
labels_pred = [1, 0, 1, 1, 2, 1, 1, 2, 2]

print("Adjusted Mutual Information ", metrics.adjusted_mutual_info_score(label
s_true, labels_pred))
print("Mutual Information ", metrics.mutual_info_score(labels_true, labels_pre
d))
```

```
Silhouette Score 0.3458227822664392
Adjusted Mutual Information  0.03115451461295648
Mutual Information  0.32075748037462026
```

K-Means clustering

We saw K-Means in action above. Here are the mathematical details.

The specification of K-means is simple: assign a collection of clusters C_k that minimize

$$\text{argmin}_C \sum_{k=1}^{K} \sum_{X_j \in C_k} \|X_{j\cdot} - \mu_k\|_2^2$$

where μ_k is the center of the points in C_k. The algorithm to implement this is simple:

Initialize μ_k (with possibly random values). Then iterate between

1. Assign $X_{j\cdot}$ to the cluster C_k that minimizes $\|X_{j\cdot} - \mu_k\|_2^2$.
2. Recompute μ_k by averaging over all the points $X_{j\cdot}$ in the cluster C_k.

Notice that both iterative steps lower the objective (the algorithm is greedy) and there are only a finite number of possible partitions of the points $X_{j\cdot}$ so the algorithm is guaranteed to converge. The converged solution may not be globally optimal.

In [6]:
```
from ipywidgets import interact

interact(plot_kmeans(X1, 3, X1[::100,:]), step=(0, 21, 1));
```

Other notes on K-Means:

- It is sensitive to outliers.
- It has difficulties with non-spherical clusters and clusters of different sizes.

Gaussian mixture models

We can generalize the notion of K-Means in two ways:

1. Instead of requiring that each $X_{j\cdot}$ strictly belong to a single C_k, we say it belongs to C_k with a probability described by the multivariate Gaussian distribution (https://en.wikipedia.org/wiki/Multivariate_normal_distribution)
2. Instead of just having μ_k as a degree of freedom, we have the pair (μ_k, Σ_k)

Here, μ_k is a k-dimensional mean vector and Σ_k is the $k \times k$ covariance matrix. Furthermore, k is the number of clusters.

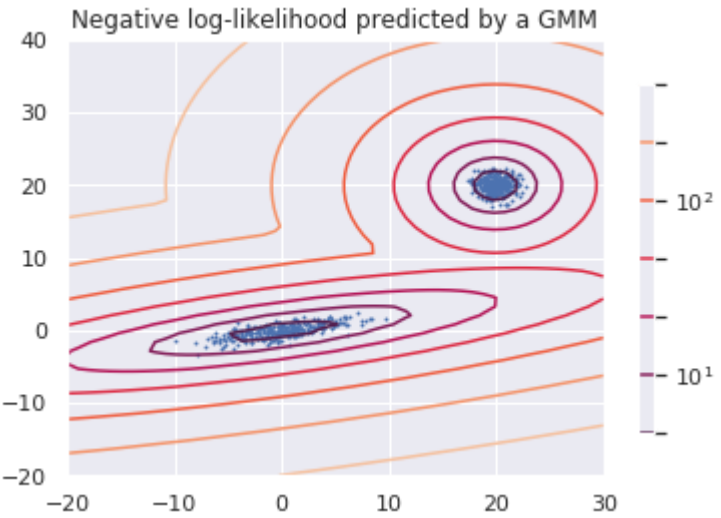
Then each $X_{j\cdot}$ is of type k with probability proportional to

$$p_k \exp\left(\frac{1}{2}(X_{j\cdot} - \mu_k) \cdot \Sigma_k^{-1}(X_{j\cdot} - \mu_k)\right)$$

where $p_k > 0$ is a proportionality constant. While this is not strictly a clustering algorithm, it can be turned into one by choosing the probability cluster with the highest probability.

To see how this works, see Scikit Learn's example code `plot_gmm_pdf.py` (https://github.com/scikit-learn/scikit-learn/blob/master/examples/mixture/plot_gmm_pdf.py), copied in the cell below:

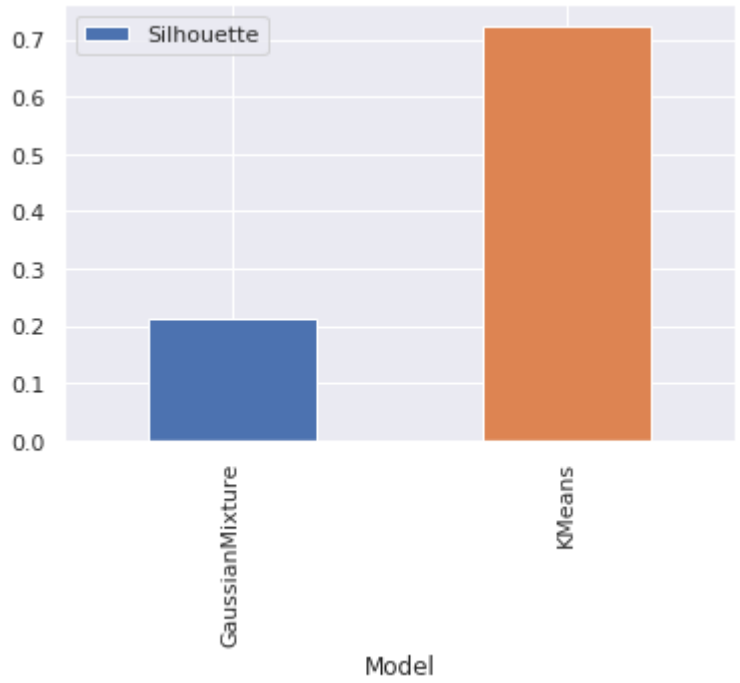
In [7]:
```
# From https://github.com/scikit-learn/scikit-learn/blob/master/examples/mixture/plot_gmm_pdf.py
from matplotlib.colors import LogNorm
from sklearn import mixture

n_samples = 300

# generate random sample, two components
np.random.seed(0)

# generate spherical data centered on (20, 20)
shifted_gaussian = np.random.randn(n_samples, 2) + np.array([20, 20])

# generate zero centered stretched Gaussian data
C = np.array([[0., -0.7], [3.5, .7]])
stretched_gaussian = np.dot(np.random.randn(n_samples, 2), C)

# concatenate the two datasets into the final training set
X_train = np.vstack([shifted_gaussian, stretched_gaussian])

# fit a Gaussian Mixture Model with two components
clf = mixture.GaussianMixture(n_components=2, covariance_type='full')
clf.fit(X_train)

# display predicted scores by the model as a contour plot
x = np.linspace(-20., 30.)
y = np.linspace(-20., 40.)
X, Y = np.meshgrid(x, y)
XX = np.array([X.ravel(), Y.ravel()]).T
Z = -clf.score_samples(XX)
Z = Z.reshape(X.shape)

CS = plt.contour(X, Y, Z, norm=LogNorm(vmin=1.0, vmax=1000.0),
                 levels=np.logspace(0, 3, 10))
CB = plt.colorbar(CS, shrink=0.8, extend='both')
plt.scatter(X_train[:, 0], X_train[:, 1], .8)

plt.title('Negative log-likelihood predicted by a GMM')
plt.axis('tight')
plt.show()
```



Questions:

1. For K-Means, what happens if the features have very different scales?
2. Does this problem exist for Gaussian Mixture Models?

In [8]:
```
from sklearn import cluster, mixture, datasets
import pandas as pd

# Load Boston dataset
boston = datasets.load_boston()

columns = ["CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS","RAD","TAX","PTRA
TIO","B","LSAT"]
X = pd.DataFrame(boston.data, columns=columns)
y = pd.Series(boston.target)

gm_clf = mixture.GaussianMixture(n_components=3, covariance_type='diag', rando
m_state=42).fit(X)
kmeans_clf = cluster.KMeans(n_clusters=3, random_state=42).fit(X)

# Compute the Silhouette
pd.DataFrame([
    ("GaussianMixture",  metrics.silhouette_score(X, gm_clf.predict(X), metric
='euclidean')) ,
    ("KMeans",  metrics.silhouette_score(X, kmeans_clf.labels_, metric='euclid
ean'))
], columns=["Model", "Silhouette"]).plot(x="Model", y="Silhouette", kind="bar"
);
```



Exercise:

1. Normalize KMeans to improve performance (use the scale function)
2. Test the inertia criterion you added earlier and graph it in the above plots

Dimensionality Reduction

Motivation

Certain applications, such as reducing file sizes for faster data transfer, use dimensionality reduction. The idea is to replace a large number of variables with a smaller number, which maintain a good representation of the data. In the map above, we can reduce remove the landscape detail but still preserve the general geography.

Dimensionality reduction is also useful in visualization (reduce features to 3-D or 2-D which you can graph), and as a pre-processing step in your pipeline, when your subsequent methods don't deal well with a large number of dimensions.

Random projections

One really simple method of reducing the data is to do it at random. That is, instead of using X, we use $X' = XP$ where P is a $p \times m$ matrix of randomly generated values. This is effectively projecting the data onto the rows of P. When $p \gg m$, the rows of P are highly likely to be orthogonal. While they don't give a very faithful representation of the original data (PCA or NMF would do better), for classification and regression purposes, they often give enough signal for the job. They are fast to compute and are good for prototyping. They can often be a good standby as part of a learning pipeline which can be upgraded to a more principled technique as necessary.

You can read more about Random Projections in the Scikit Documentation (http://scikit-learn.org/stable/modules/random_projection.html). We give a simple example that implements them below.

Question: Random projections project X onto a random subspace of dimension m, while PCA or NMF project X onto an "optimal" subspace of dimension m. When and why would you use random projections?

In [9]:
```
import numpy as np
from sklearn import random_projection

X = np.random.rand(100, 10000)
transformer = random_projection.GaussianRandomProjection(n_components=30)
X_new = transformer.fit_transform(X)
X_new.shape
```

Out[9]: (100, 30)

Matrix factorization

Recall that for X an $n \times p$ matrix, we have the Singular Value Decomposition (https://en.wikipedia.org/wiki/Singular_value_decomposition) with

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

where U is a unitary $n \times n$ matrix, Σ is a diagonal $n \times p$ matrix, and V is a unitary $p \times p$ matrix. We can think of U as rotating us to a basis wherein Σ will scale along the axes, and V^T rotating back. The unit sphere under this transformation will be transformed into an ellipsoid whose semi-major axes are given by the singular values on the diagonal of Σ, and whose rotation is specified by U and V.

Exercise: What happens if X is a square and symmetric? (The result is called spectral theorem, which roughly says $U = V^T$.) In particular, how do the SVD of $X^T X$ or XX^T relate to that of X? (You can easily work out the algebra.)

Principal Components Analysis (PCA)

We think of the SVD as decomposing X into orthogonal (i.e. independent) components with strengths given by the diagonal of Σ,

$$\Sigma = \begin{bmatrix} \sigma_1 & & \\ & \sigma_2 & \\ & & \ddots \end{bmatrix}$$

By convention, we assume $|\sigma_1| \geq |\sigma_2| \geq \cdots$. One interpretation of this is that σ_1 corresponds to the largest component of variation, σ_2 corresponds to the second largest, etc. We often think of the smaller components as being just noise and the larger components as being being signal. Therefore, it makes sense to truncate Σ to its largest m components and just keep those. In Scikit, this algorithm `sklearn.decomposition.PCA` returns $U\Sigma P_m$ where P_m is the projection operator onto the first m dimensions (in reality, it performs a stochastic SVD, which should be faster than the exact SVD).

Geometrically, you can think about this as fitting an m-dimensional ellipsoid to the (originally p-dimensional) data. We expect that the majority of the variance in the data will be explained by the approximation, and any variation along the truncated axes is negligible. **It's important to note that we should choose m according to how much of the variance we want to preserve vs. how much compression we want.**

We can phrase an m-dimensional PCA as

$$\min_{U,\Sigma_m,V} \|X - U\Sigma_m V^T\|_2$$

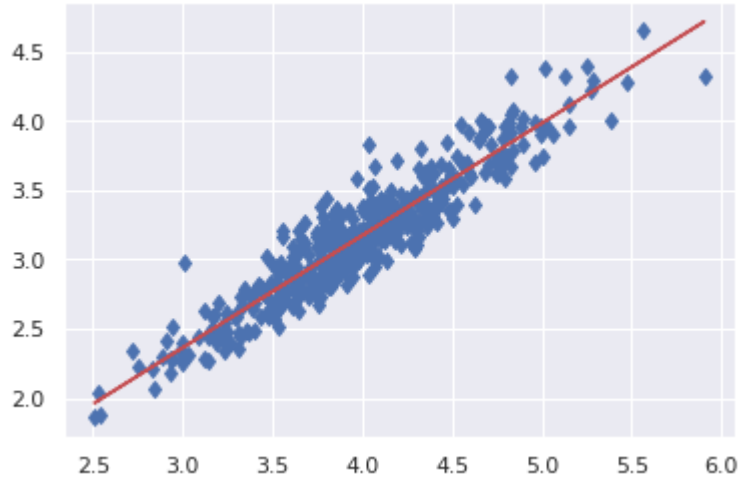
where U is a unitary $n \times n$ matrix, Σ_m is a diagonal $n \times p$ matrix with rank m, and V is a unitary $p \times p$ matrix.

A super simplistic 2-D example of PCA

Here is some data that is clearly correlated.

In [10]:
```
np.random.seed(42)

# Generate some normally distributed random numbers
a = np.random.normal(1,0.3, 400)
b = np.random.normal(3,0.5, 400)

# Look at their sum and L2 norms as our variables, I just wanted things that
# were highly correlated.  Still correlated but much less so if one of the var
iables
# is centered to produce mostly negative numbers (try it!)
pairs = map(lambda x, y: (x + y, np.sqrt(x**2 + y**2)), a, b)

# Trick to unpack the tuples
(a,b) = [list(z) for z in zip(*pairs)]

fitlinePars = np.polyfit(a,b,1)
print("Fit pars = {}".format(fitlinePars))
fitline = np.poly1d(fitlinePars)

plt.plot(a,b,'d',a,fitline(a),'r-');
print("Variance in a = {}, in b = {}".format(np.var(a),np.var(b)))
```

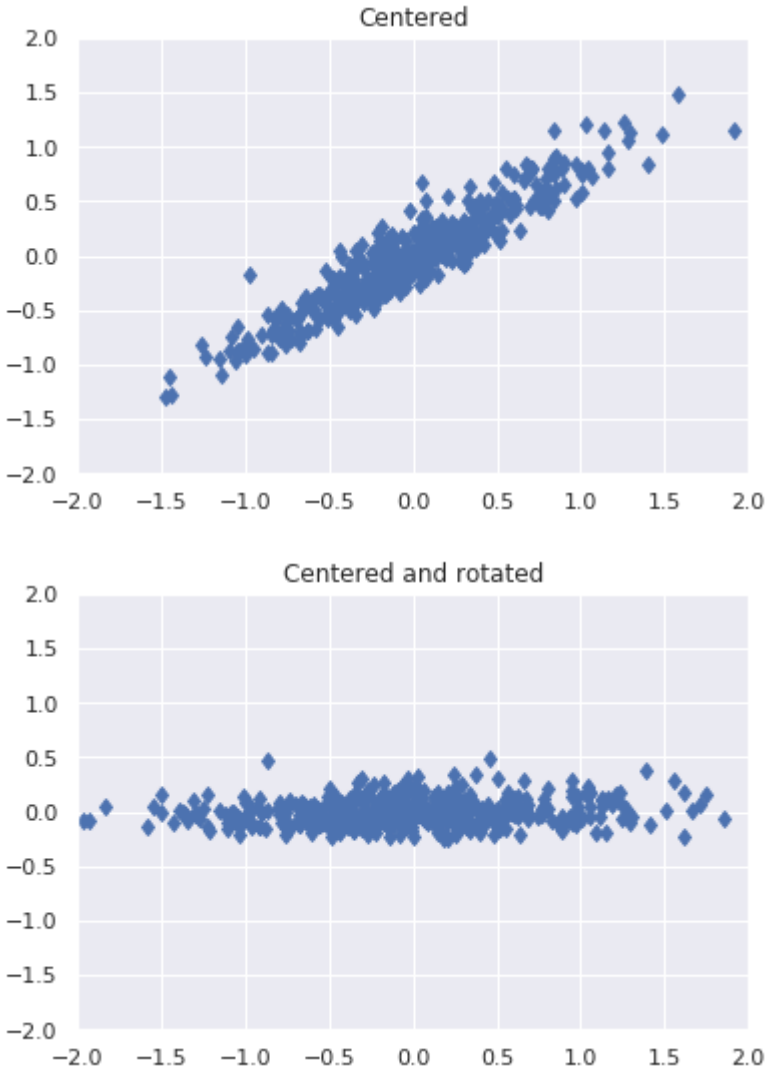
```
Fit pars = [ 0.80851017 -0.06015059]
Variance in a = 0.3026716588217154, in b = 0.2244472302325583
```



We can see at a glance that we'd do a pretty good job approximating this data by simply fitting a line to it, and that as it stands the two variables account for approximately equal shares of the variance. Then we'll only need to keep one of the variables, or the distance along the line, and the line equation. We'll reduce our feature space by half! Of course, at the cost of accuracy — but the less variation there is in the direction perpendicular to the line, the less accuracy we lose. We can quantify this by something like Singular Value Decomposition (analogous to eigenvectors and eigenvalues).

First, let's subtract the mean for each variable, so we know those intercepts are all zero (and to allow us to write this simply in matrix language). Then we'll transform our data with a unitary transformation — one that preserves distances, so we know we're not changing it in a meaningful way. In 2D, this is just a rotation.

In [11]:
```
a_center = [z - np.mean(a) for z in a]
b_center = [z - np.mean(b) for z in b]

plt.plot(a_center,b_center,'d');
plt.title('Centered')
plt.axis([-2,2,-2,2])
plt.show()

def rotate(x,y,theta):
    # Note that these are *active* rotations
    x_new = np.cos(theta) * x - np.sin(theta) * y
    y_new = np.sin(theta) * x + np.cos(theta) * y
    return (x_new, y_new)

# Quickly compute the angle corresponding to the slope from our line fit
angle = -np.arctan(fitlinePars[0])

pairs = [rotate(x,y,angle) for (x,y) in zip(a_center,b_center)]

(a_rot,b_rot) = [list(z) for z in zip(*pairs)]
plt.plot(a_rot,b_rot,'d')
plt.title('Centered and rotated')
plt.axis([-2,2,-2,2])
plt.show()
```



In [12]:
```
print("Variance in a_rot = {}, in b_rot = {}".format(np.var(a_rot),np.var(b_rot)))
```

```
Variance in a_rot = 0.5110371506500379, in b_rot = 0.01608173840423586
```

There's a really great brief article on PCA on Stats Stack Exchange (https://stats.stackexchange.com/questions/10251/how-to-find-principal-components-without-matrix-algebra).

Questions:

1. What is the answer to $\max_{u:\|u\|=1} u^T X^T X u$ and how does it relate to SVD.
2. **Gotcha**: Why is it important to subtract the mean (mean centering) before performing PCA?

Non-negative Matrix Factorization (NMF)

Recall our minimization objective for PCA

$$\min_{U,\Sigma_m,V} \|X - U\Sigma_m V^T\|_2$$

If we absorb the Σ_m into U and V^T as W and H, we could rewrite this as

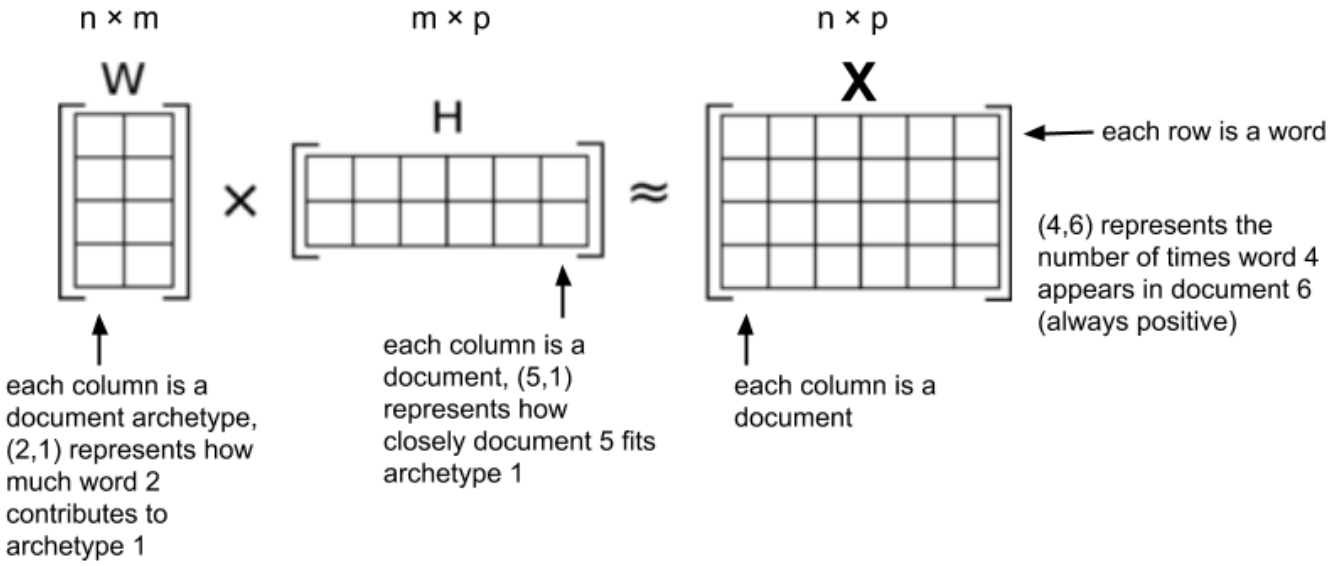
$$\min_{W,H} \|X - WH\|_2$$

where W is a $n \times m$ matrix and H is a $m \times p$ matrix. If X has only non-negative values, we might want W and H to have non-negative values as well. Hence, Non-negative Matrix Factorization is just

$$\min_{W \geq 0, H \geq 0} \|X - WH\|_2$$

when $X \geq 0$ (here, we use $X \geq 0$ to mean that each element of X has non-negative values). While PCA gives you a more accurate low-dimensional representation, NMF can give a more interpretable results since the values are non-negative.

This is often true of text and image data, where words and pixel value features are strictly "positive" and we'd like our lower-dimensional representations to also be positive. For example, it can be useful to think of text data this way:



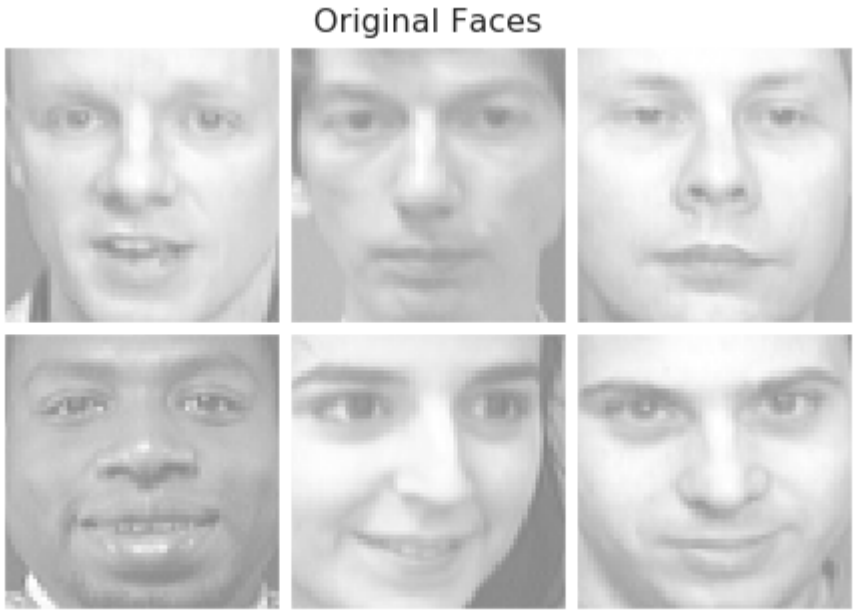
Just like in PCA, we have to choose the number of "archetypes" $m \ll n, p$ in order to compress the data, but the larger m, the more variance is retained.

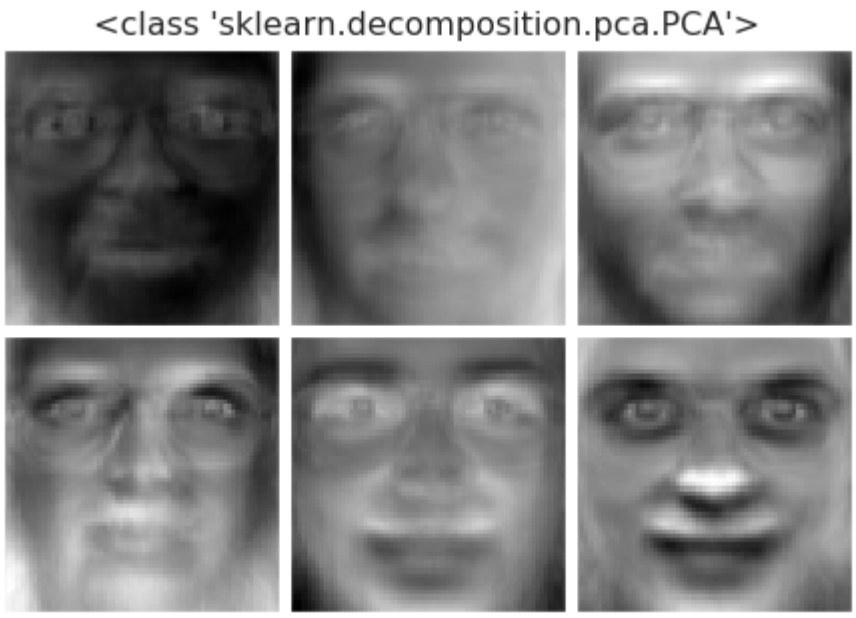
In Scikit, the matrix H is called the `components_` while W is the value returned from `.transform`.

Below, we are analyzing images of faces. Each row is a grayscale 64 x 64 dimensional image so the feature space is 4096 dimensional!

In [13]:
```
# Eigenfaces using NMF and PCA

from sklearn.datasets import fetch_olivetti_faces
from sklearn import decomposition
from matplotlib import pyplot as plt

# Fetch data of 400 faces.  Each face is a greyscale 64 x 64 dimensional imag
e,
# making this a 4096 dimensional feature space
dataset = fetch_olivetti_faces(shuffle=True, random_state=42)
X = dataset.data

n_col = 3
n_row = 2
n_components = n_col * n_row
image_shape = (64, 64)

def plot_gallery(title, images, n_col, n_row):
    plt.figure(figsize=(2. * n_col, 2.26 * n_row))
    plt.suptitle(title, size=16)
    for i, comp in enumerate(images):
        plt.subplot(n_row, n_col, i + 1)
        vmax = max(comp.max(), -comp.min())
        plt.imshow(comp.reshape(image_shape), cmap=plt.cm.gray,
                   interpolation='nearest',
                   vmin=-vmax, vmax=vmax)
        plt.xticks(())
        plt.yticks(())
    plt.subplots_adjust(0.01, 0.05, 0.99, 0.93, 0.04, 0.)
    plt.show()

transformers = [
    decomposition.PCA(n_components=n_components, whiten=True),
    decomposition.NMF(n_components=n_components, init='nndsvda', tol=5e-3),
]

plot_gallery("Original Faces", X[:6,:], n_row=n_row, n_col=n_col)
for transformer in transformers:
    transformer.fit(X)
    plot_gallery(str(transformer.__class__), transformer.components_, n_row=n_
row, n_col=n_col)
```

```
downloading Olivetti faces from https://ndownloader.figshare.com/files/597602
7 to /home/jovyan/scikit_learn_data
```





<class 'sklearn.decomposition.nmf.NMF'>

Exercise:

1. `np.dot(pca.transform(X), pca.components_)` will project the transformed (low-dimensional) compressed features `pca.transform(X)` back to the original 4096-dimensional space. Use to this visualize the reconstructed "compressed" images for different faces. Do the same for `nmf`. There's a lot of faces, you probably only want to visualize them for a fraction of the faces.
2. Vary `n_components` and see how the `pca` low-dimensional reconstructed varies with different values of `n_components`.

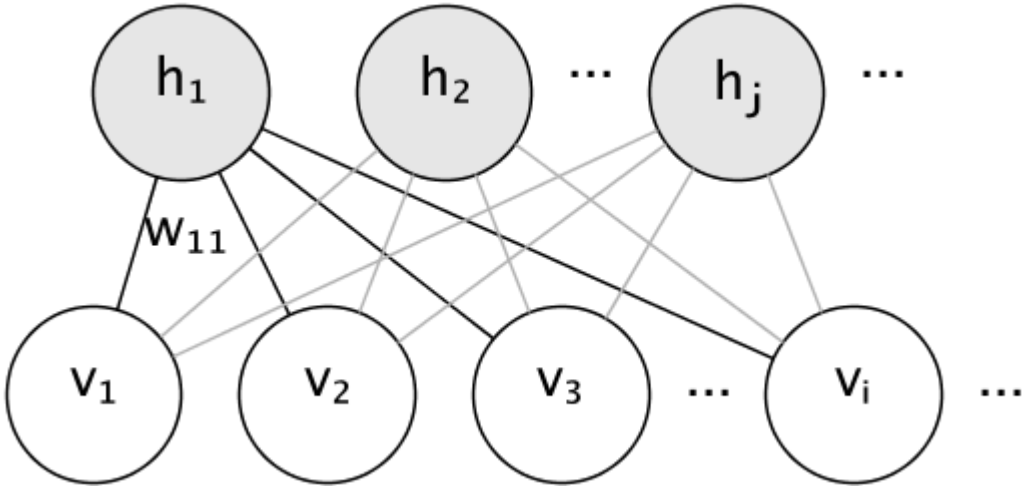
In [14]:
```
# Topic modelling using NMF
# Adapted from http://scikit-learn.org/stable/auto_examples/applications/topics_extraction_with_nmf.html

from time import time

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import NMF
from sklearn.datasets import fetch_20newsgroups

n_samples = 20000
n_features = 1000
n_topics = 10
n_top_words = 20

# Load the 20 newsgroups dataset and vectorize it. We use a few heuristics
# to filter out useless terms early on: the posts are stripped of headers,
# footers and quoted replies, and common English words, words occurring in
# only one document or in at least 95% of the documents are removed.

t0 = time()
print("Loading dataset and extracting TF-IDF features...")
dataset = fetch_20newsgroups(shuffle=True, random_state=1,
                             remove=('headers', 'footers', 'quotes'))

vectorizer = TfidfVectorizer(max_df=0.95, min_df=2, max_features=n_features,
                             stop_words='english')
tfidf = vectorizer.fit_transform(dataset.data[:n_samples])
print("done in %0.3fs." % (time() - t0))

# Fit the NMF model
print("Fitting the NMF model with n_samples=%d and n_features=%d..."
      % (n_samples, n_features))
nmf = NMF(n_components=n_topics, random_state=1).fit(tfidf)
print("done in %0.3fs." % (time() - t0))

feature_names = vectorizer.get_feature_names()

for topic_idx, topic in enumerate(nmf.components_):
    print("Topic #%d:" % topic_idx)
    print(" ".join([feature_names[i]
                    for i in topic.argsort()[:-n_top_words - 1:-1]]))
    print()
```

```
Downloading 20news dataset. This may take a few minutes.
Downloading dataset from https://ndownloader.figshare.com/files/5975967 (14 M
B)
```

```
Loading dataset and extracting TF-IDF features...
done in 19.571s.
Fitting the NMF model with n_samples=20000 and n_features=1000...
done in 20.134s.
Topic #0:
don just people think like know right ve did time say really good way make sa
id going want things thing

Topic #1:
card video monitor cards drivers bus vga driver color memory bit ram board mo
de pc graphics apple 16 modem speed

Topic #2:
god jesus bible believe christ faith christian christians church does life si
n truth lord say man hell christianity love belief

Topic #3:
game team year games season players play hockey win league player teams nhl g
ood runs better best hit think time

Topic #4:
new car 00 10 sale price space condition offer used good 20 bike 50 shipping
15 old power interested 30

Topic #5:
thanks know does mail advance anybody hi looking info help appreciated inform
ation email address post need interested send like appreciate

Topic #6:
windows file use files window dos program using problem running version run a
pplication server screen ms image help software ftp

Topic #7:
edu soon com university cs email article internet send mail ftp david pub mit
address au reply 1993 subject apr

Topic #8:
key chip encryption clipper keys government use public escrow law algorithm s
ecurity phone data nsa bit number secure chips des

Topic #9:
drive scsi hard drives disk ide floppy controller cd mac tape internal power
rom cable computer apple problem mb switch
```

Restricted Boltzmann Machines

A restricted Boltzmann Machine is a type of neural-network that's used for machine learning. It's given in the diagram of a bipartite graph below:



The p nodes labeled v are the *visible* nodes and represent data (a row of X in our notation). The m nodes labeled h represent hidden states and the $m \times p$ edges represent a matrix W that gives the interactions between the visible and hidden nodes (NB: while W is the standard notation for this area, it plays the analogous role to H in the PCA, NMF notation above). Why does it have a funny name?

1. This machine is *Restricted* because the connections between states are confined to a bipartite graph (there are no connections between h's or v's. This means all its relationships can be represented by a matrix W. This also implies conditional independence amongst the h's and the v's,

$$h_i \perp h_j | v \qquad v_i \perp v_j | h$$

2. This machine is *Boltzmann* because for a given v and h, we assign it an energy

$$E(v, h) = v^T W h + b^T v + c^T h$$

where in addition to W, we have the visible and hidden intercepts, b and c. The probability of a state is given by

$$P(v, h) = \frac{e^{-E(v,h)}}{Z}$$

where Z is called a *partition function* in physics and is just the necessary coefficient to make P a probability distribution. We then train the RBM to maximize the probability

$$\max_{W,b,c} P(v, h) = \sum_h P(v, h)$$

This is often done using contrastive divergence (http://www.cs.toronto.edu/~hinton/absps/guideTR.pdf). It is then easy to compute

$$P(h|v)$$

to obtain the hidden nodes from the visible nodes.

In Scikit, `sklearn.neural_network.BernoulliRBM` maximizes log-likelihood using Stochastic Maximum Likelihood (see the documentation (http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.BernoulliRBM.html)). This is an iterative stochastic algorithm that takes a number of parameters,

Parameter	**Meaning**
`learning_rate`	Learning Rate
`n_iter`	Iterations to run
`n_components`	Number of hidden states

Once the model is trained, the outputs map to the mathematical notation we have expressed above members on the return object

Math	**Scikit**
W	`.components_`
b	`.intercept_visible_`
c	`.intercept_hidden_`

Exit Tickets

1. Explain how performing PCA or NMF affects the variance-bias tradeoff.
2. How do you choose the k in k-means?
3. How do you choose m in dimensionality reductions from $n \rightarrow m$?

Spoilers

Answers

Metrics for Unsupervised Learning

1. As you scale in feature space, inertia scales as the square, Silhouette is invariant, and Mutual Information does not scale (why?).
2. For a fixed number of points, increasing the number of clusters tends to decrease Inertia. The effect on Silhouette is less easily determined but should be a minimum at the optimal number of clusters. Mutual information will tend to decrease as well.
3. You can make a case for either 2 or 4 clusters, whereas 3 or 5 clusters make less sense. The silhouette coefficient reaches a similar maximum for those two values.

Clustering

1. For KMeans, the largest-scaled feature will dominate the clustering. If the features have very different scales you should normalize the data first.
2. This problem does not exist for Gaussian Mixture Models. In the 2D case (two features), the curves of constant probability are ellipses whose major and minor axes are dictated by the eigenvalues and eigenvectors of the covariance matrix, Σ. 2D KMeans clusters, on the other hand, are assumed to be circular.

Random Projections

1. While the subspace chosen by PCA or NMF might be "optimal", these algorithms are much more computationally expensive. You would use random projects for prototyping (where speed matters) or because you have much more "signal" than you need to do your classification / regression job and some random subset of the signal will do.

Principle Component Analysis

1. $\max_{u:\|u\|=1} u^T X^T X u = \sigma_1^2$ and the answer should be obvious at this point. This should give some intuition for why σ_1 corresponds to the largest component of variation.
2. It important to subtract the mean (mean centering) before performing PCA. Otherwise the first component might just be the mean of the data.

In [ ]:
```
# Reconstructed faces at n_components = 10

n_col = 3
n_row = 2
n_samples = n_col * n_row

np.random.seed(42)
ix_sample = np.random.choice(X.shape[0], size=n_samples, replace=False)
X_sample = X[ix_sample]

n_components = 10

pca = decomposition.PCA(n_components=n_components, whiten=True).fit(X)
nmf = decomposition.NMF(n_components=n_components, init='nndsvda', tol=5e-3).f
it(X)

def projector(M):
    U, _, V = np.linalg.svd(M)
    S = np.ones([U.shape[0], V.shape[1]])
    return np.array(np.matrix(U) * np.matrix(S) * np.matrix(V))

proj = projector(nmf.components_)

plot_gallery("Before", X_sample, n_row=n_row, n_col=n_col)
plot_gallery("After NMF", np.dot(nmf.transform(X_sample), nmf.components_), n_
row=n_row, n_col=n_col)
plot_gallery("After PCA", np.dot(pca.transform(X_sample), pca.components_), n_
row=n_row, n_col=n_col)
```

In [ ]:
```
# A reconstructed face compressed to various numbers of components

n_col = 3
n_row = 3
n_show = n_col * n_row
components_delta = 10

n_components = 10

pcas = [decomposition.PCA(n_components=n_components, whiten=True).fit(X)
            for n_components in range(components_delta, n_show * components_de
lta, components_delta)]
plot_data = [np.dot(pca.transform(X[84:85]), pca.components_) for pca in pcas]
+ [X[84]]
plot_gallery("PCA at progressively higher resolution", plot_data, n_col=n_col,
n_row=n_row)
```

Exit Tickets

1. Decreasing dimensionality decreases variance and increases bias.
2. Minimize the silhouette coefficient
3. Treat m as a hyperparameter, and cross validate. Sometimes, it may be limited by computational power.