The first part of this is adapted directly from VanderPlas 05.09-Principal-Component-Analysis.ipynb, with a number of additions and changes to the code, but most of the text is verbatim from the above.
It came with the disclaimer: [The text is released under the CC-BY-NC-ND license, and code is released under the MIT license]

In Depth: Principal Component Analysis

Up until now, we have been looking in depth at supervised learning estimators: those estimators that predict labels based on labeled training data. Here we begin looking at several unsupervised estimators, which can highlight interesting aspects of the data without reference to any known labels.

In this section, we explore what is perhaps one of the most broadly used of unsupervised algorithms, principal component analysis (PCA). PCA is fundamentally a dimensionality reduction algorithm, but it can also be useful as a tool for visualization, for noise filtering, for feature extraction and engineering, and much more. After a brief conceptual discussion of the PCA algorithm, we will see a couple examples of these further applications.

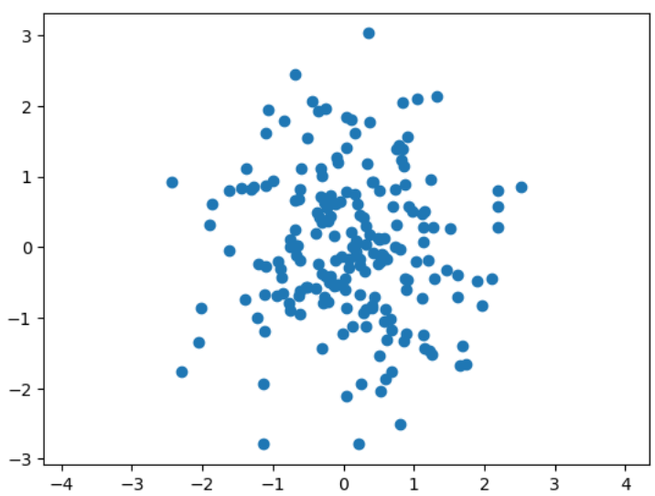
We begin with the standard imports:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
#import seaborn as sns; sns.set()
```

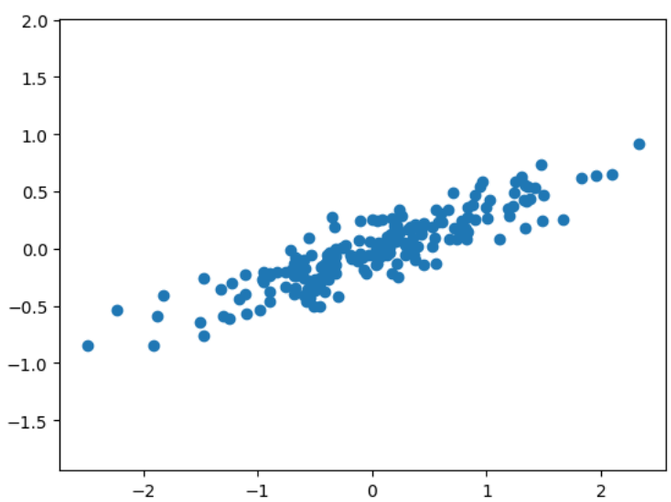
Introducing Principal Component Analysis

Principal component analysis is a fast and flexible unsupervised method for dimensionality reduction in data, which we saw briefly in Introducing Scikit-Learn. Its behavior is easiest to visualize by looking at a two-dimensional dataset. Consider the following 200 points:

```
#take 200 points
rng = np.random.RandomState(1)
plt.scatter(*rng.randn(2, 200))
plt.axis('equal');
```




```
rng = np.random.RandomState(1)
X = np.dot(rng.rand(2, 2), rng.randn(2, 200)).T
plt.scatter(*X.T)
plt.axis('equal');
```



By eye, it is clear that there is a nearly linear relationship between the x and y variables. This is reminiscent of the linear regression data we explored in In Depth: Linear Regression, but the problem setting here is slightly different: rather than attempting to *predict* the y values from the x values, the unsupervised learning problem attempts to learn about the *relationship* between the x and y values.

In principal component analysis, this relationship is quantified by finding a list of the *principal axes* in the data, and using those axes to describe the dataset. Using Scikit-Learn's `PCA` estimator, we can compute this as follows:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(X)
```

```
PCA(n_components=2)
```

The fit learns some quantities from the data, most importantly the "components" and "explained variance":

```
print(pca.components_) # necessarily orthogonal unit vectors
```

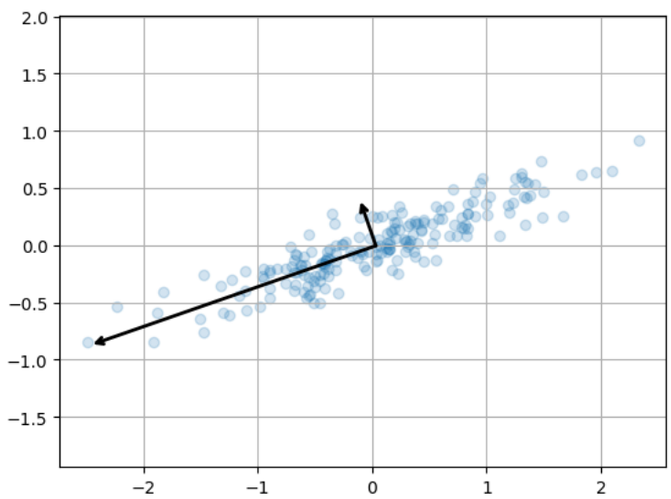
```
[[-0.94446029 -0.32862557]
 [-0.32862557  0.94446029]]
```

```
print('diagonal variances:', pca.explained_variance_)
print('divided by sum to get percentages of total:', pca.explained_variance_ratio_)
```

```
diagonal variances: [0.7625315 0.0184779]
divided by sum to get percentages of total: [0.97634101 0.02365899]
```

To see what these numbers mean, let's visualize them as vectors over the input data, using the "components" to define the direction of the vector, and the "explained variance" to define the squared-length of the vector:

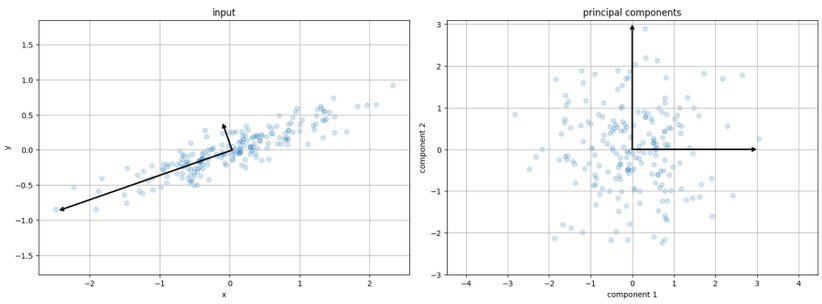
```
def draw_vector(v0, v1, ax=None):
    ax = ax or plt.gca()
    arrowprops=dict(arrowstyle='->', linewidth=2, shrinkA=0, shrinkB=0)
    ax.annotate('', v1, v0, arrowprops=arrowprops)

# plot data
plt.scatter(*X.T, alpha=0.2)
for length, vector in zip(pca.explained_variance_, pca.components_):
    v = vector * 3 * np.sqrt(length)
    draw_vector(pca.mean_, pca.mean_ + v)
plt.grid(True)
plt.axis('equal');
```



These vectors represent the *principal axes* of the data, and the length of the vector is an indication of how "important" that axis is in describing the distribution of the data—more precisely, it is a measure of the variance of the data when projected onto that axis. The projection of each data point onto the principal axes are the "principal components" of the data.

If we plot these principal components beside the original data, we see the plots shown here:

```
#(can skip in class)

pca = PCA(n_components=2, whiten=True) #whiten to get unit variance
pca.fit(X)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

# plot data
ax[0].scatter(*X.T, alpha=0.2)
for length, vector in zip(pca.explained_variance_, pca.components_):
    v = vector * 3 * np.sqrt(length)
    draw_vector(pca.mean_, pca.mean_ + v, ax=ax[0])
ax[0].grid(True)
ax[0].axis('equal');
ax[0].set(xlabel='x', ylabel='y', title='input')

# plot principal components
X_pca = pca.transform(X)
ax[1].scatter(*X_pca.T, alpha=0.2)
draw_vector([0, 0], [0, 3], ax=ax[1])
draw_vector([0, 0], [3, 0], ax=ax[1])
ax[1].grid(True)
ax[1].axis('equal')
ax[1].set(xlabel='component 1', ylabel='component 2',
          title='principal components',
          xlim=(-5, 5), ylim=(-3, 3.1));
```



This transformation from data axes to principal axes is an *affine transformation*, which basically means it is composed of a translation, rotation, and uniform scaling.

While this algorithm to find principal components may seem like just a mathematical curiosity, it turns out to have very far-reaching applications in the world of machine learning and data exploration.

PCA as dimensionality reduction

Using PCA for dimensionality reduction involves zeroing out one or more of the smallest principal components, resulting in a lower-dimensional projection of the data that preserves the maximal data variance.

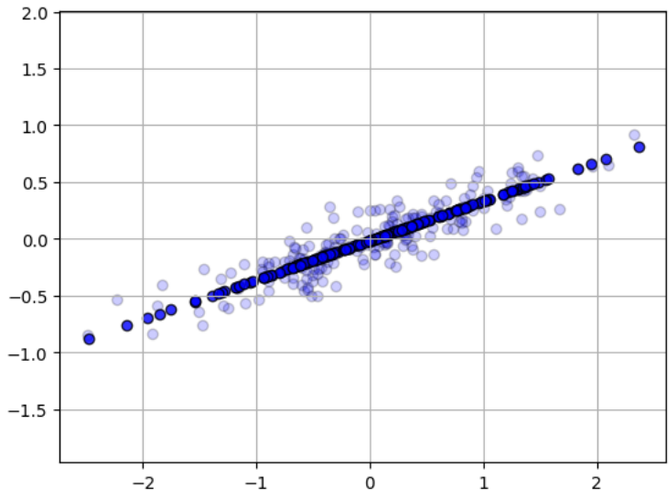
Here is an example of using PCA as a dimensionality reduction transform:

```
pca = PCA(n_components=1)
pca.fit(X)
X_pca = pca.transform(X)
print("original shape:   ", X.shape)
print("transformed shape:", X_pca.shape)
```

```
original shape:    (200, 2)
transformed shape: (200, 1)
```

The transformed data has been reduced to a single dimension. To understand the effect of this dimensionality reduction, we can perform the inverse transform of this reduced data and plot it along with the original data:

```
X_new = pca.inverse_transform(X_pca)
plt.scatter(*X.T, alpha=0.2, c='b', ec='k')
plt.scatter(*X_new.T, alpha=0.8, c='b', ec='k')
plt.grid(True)
plt.axis('equal');
```



The light points are the original data, while the dark points are the projected version. This makes clear what a PCA dimensionality reduction means: the information along the least important principal axis or axes is removed, leaving only the component(s) of the data with the highest variance. The fraction of variance that is cut out (proportional to the spread of points about the line formed in this figure) is roughly a measure of how much "information" is discarded in this reduction of dimensionality.

This reduced-dimension dataset is in some senses "good enough" to encode the most important relationships between the points: despite reducing the dimension of the data by 50%, the overall relationship between the data points are mostly preserved.

PCA for visualization: Hand-written digits

The usefulness of the dimensionality reduction may not be entirely apparent in only two dimensions, but becomes much more clear when looking at high-dimensional data. To see this, let's take a quick look at the application of PCA to the digits data we saw in In-Depth: Decision Trees and Random Forests.

We start by loading the data:

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape
```

```
(1797, 64)
```

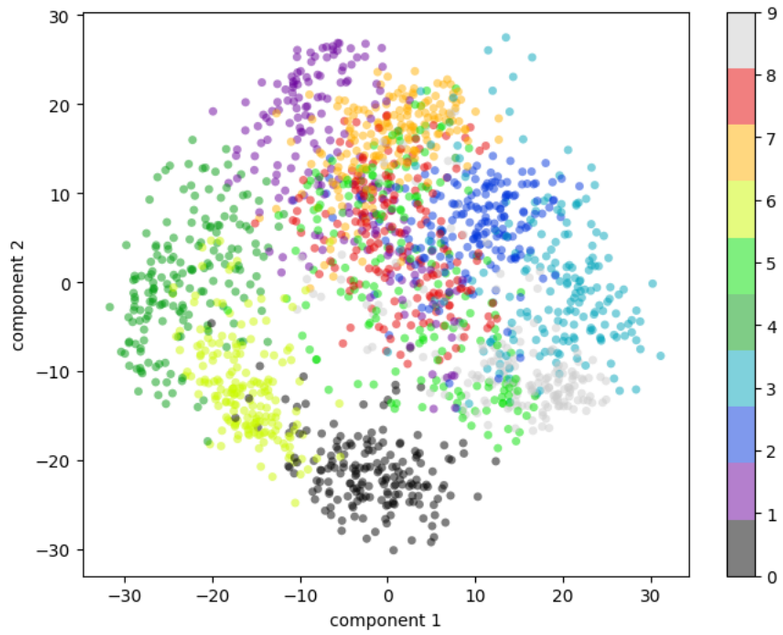
Recall that the data consists of 8×8 pixel images, meaning that they are 64-dimensional. To gain some intuition into the relationships between these points, we can use PCA to project them to a more manageable number of dimensions, say two:

```
pca = PCA(2)  # project from 64 to 2 dimensions
projected = pca.fit_transform(digits.data)
print(digits.data.shape)
print(projected.shape)
```

```
(1797, 64)
(1797, 2)
```

We can now plot the first two principal components of each point to learn about the data:

```
plt.figure(figsize=(8,6))
plt.scatter(*-projected.T, s=25,
            c=digits.target, edgecolor='none', alpha=0.5,
            cmap=plt.get_cmap('nipy_spectral',10))
plt.xlabel('component 1')
plt.ylabel('component 2')
plt.colorbar();
```



Recall what these components mean: the full data is a 64-dimensional point cloud, and these points are the projection of each data point along the directions with the largest variance. Essentially, we have found the optimal stretch and rotation in 64-dimensional space that allows us to see the layout of the digits in two dimensions, and have done this in an unsupervised manner—that is, without reference to the labels.

What do the components mean?

We can go a bit further here, and begin to ask what the reduced dimensions *mean*. This meaning can be understood in terms of combinations of basis vectors. For example, each image in the training set is defined by a collection of 64 pixel values, which we will call the vector x:

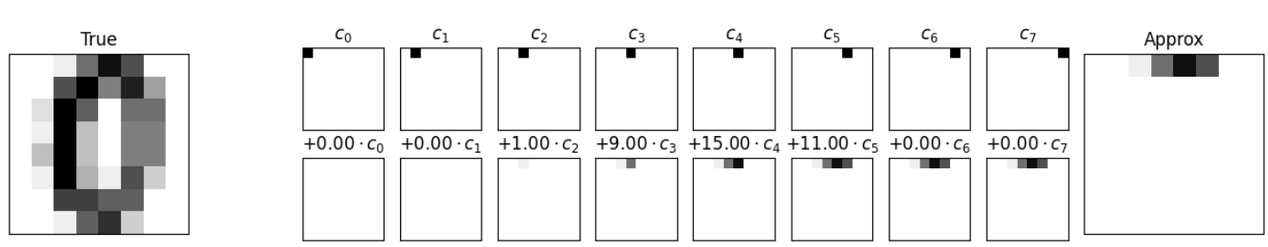
$$x = [x_1, x_2, x_3 \cdots x_{64}]$$

One way we can think about this is in terms of a pixel basis. That is, to construct the image, we multiply each element of the vector by the pixel it describes, and then add the results together to build the image:

$$\text{image}(x) = x_1 \cdot (\text{pixel 1}) + x_2 \cdot (\text{pixel 2}) + x_3 \cdot (\text{pixel 3}) \cdots x_{64} \cdot (\text{pixel 64})$$

One way we might imagine reducing the dimension of this data is to zero out all but a few of these basis vectors. For example, if we use only the first eight pixels, we get an eight-dimensional projection of the data, but it is not very reflective of the whole image: we've thrown out nearly 90% of the pixels!

```
imshape=(8, 8)
n_components=8
def show_pca_components(X, coeff, show_mean = True):
  fig=plt.figure(figsize=(1.2 * (2 + 1 + n_components + 2), 1.2*2))
  g = plt.GridSpec(2, 2 + 1 + n_components + 2, hspace=0.35)  #make a 2 x (8+2*2+1) grid

  def digshow(i, j, X, title, v={'vmin':0, 'vmax':16}):
    ax = fig.add_subplot(g[i,j], xticks=[], yticks=[])
    ax.imshow(X.reshape(imshape), interpolation='none', cmap='binary', **v)
    ax.set_title(title, fontsize=12)

  digshow(slice(2), slice(2), X, "True")  #actual digit, doublesize at left
  approx = pca.mean_.copy() if show_mean else np.zeros_like(X)
  if show_mean:
      digshow(0, 2, pca.mean_, r'$\mu$')    #mean
      digshow(1, 2, approx, r'$1\cdot\mu$') #mean again

  for i in range(n_components):
    digshow(0, i+3, components[i], r'$c_{}$'.format(i), {})  #show i'th pca component
    approx += coeff[i] * components[i]  # add i'th component
    digshow(1, i+3, approx, r"${:+.2f}\cdot c_{}$".format(coeff[i], i)) #show approx

  digshow(slice(2), slice(-2, None), approx, "Approx") #approx digit, doublesize at right


k = 10 #10th sample, a zero
X = digits.data[k]
coeff = X
components = np.eye(len(X))  # use naive basis

show_pca_components(X, coeff, show_mean=False)
```



The upper row of panels shows the individual pixels, and the lower row shows the cumulative contribution of these pixels to the construction of the image. Using only eight of the pixel-basis components, we can only construct a small portion of the 64-pixel image. Were we to continue

this sequence and use all 64 pixels, we would recover the original image.

But the pixel-wise representation is not the only choice of basis. We can also use other basis functions, which each contain some pre-defined contribution from each pixel, and write something like

$$image(x) = \text{mean} + x_1 \cdot (\text{basis } 1) + x_2 \cdot (\text{basis } 2) + x_3 \cdot (\text{basis } 3) \cdots$$

PCA can be thought of as a process of choosing optimal basis functions, such that adding together just the first few of them is enough to suitably reconstruct the bulk of the elements in the dataset. The principal components, which act as the low-dimensional representation of our data, are simply the coefficients that multiply each of the elements in this series. This figure shows a similar depiction of reconstructing this digit using the mean plus the first eight PCA basis functions:

```
pca = PCA(n_components=8)
Xproj = pca.fit_transform(digits.data)
digits.data.shape, Xproj.shape
```

```
((1797, 64), (1797, 8))
```

The 8 coordinates returned by the PCA are with respect to the mean of the data, so they're centered (recall that each row of data is the 8 projected coordinates of an image, so summing down the column averages the values of each of those coordinates):

```
Xproj.mean(0)
```

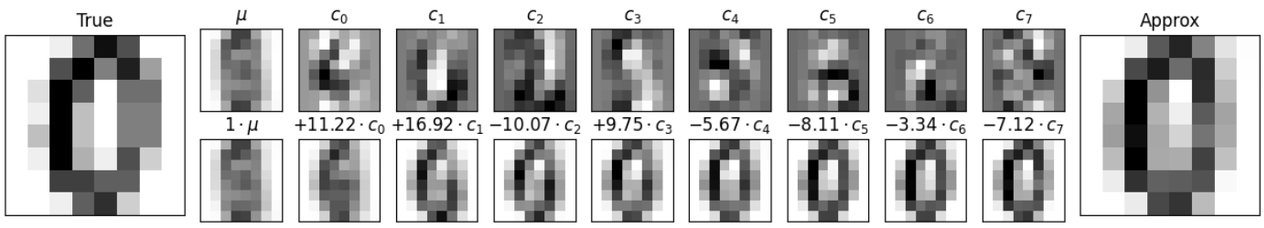
```
array([-1.70456612e-16,  7.81419077e-16, -7.71039697e-17, -2.49105133e-16,
       -2.57013232e-17, -1.25491653e-15, -1.03793805e-15,  4.45077722e-16])
```

Since their mean is zero, averaging the squares of the coordinates computes their variance, which in turn are the variances returned by the PCA (i.e., the variances in the rotated coordinate system that diagonalizes the covariance matrix):

```
print((Xproj**2).mean(0))
print(pca.explained_variance_)
```

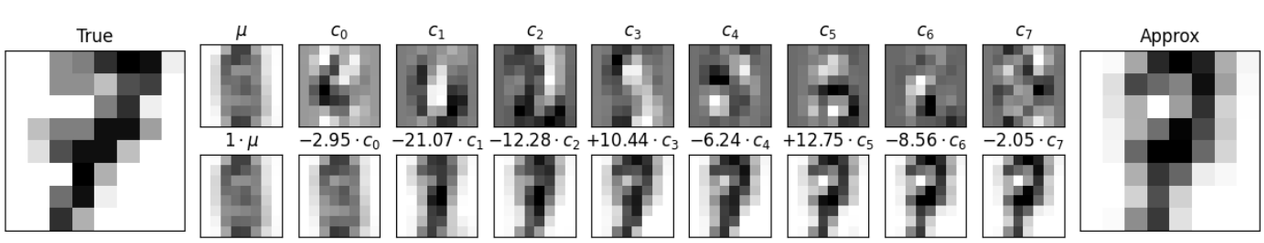
```
[178.90731577 163.62664072 141.7095362  101.04411403  69.47447921
  59.07562603  51.85562879  43.99037912]
[179.00693009 163.71774687 141.78843906 101.10037468  69.5131621
  59.10851891  51.88450163  44.01487265]
```

```
k = 10
X = digits.data[k]
coeff = Xproj[k]
components = pca.components_

show_pca_components(X, coeff)
```

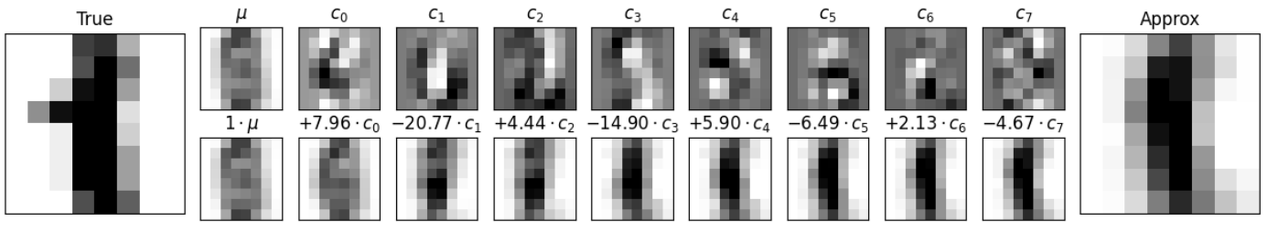


Unlike the pixel basis, the PCA basis allows us to recover the salient features of the input image with just a mean plus eight components! The amount of each pixel in each component is the corollary of the orientation of the vector in our two-dimensional example. This is the sense in which PCA provides a low-dimensional representation of the data: it discovers a set of basis functions that are more efficient than the native pixel-basis of the input data.

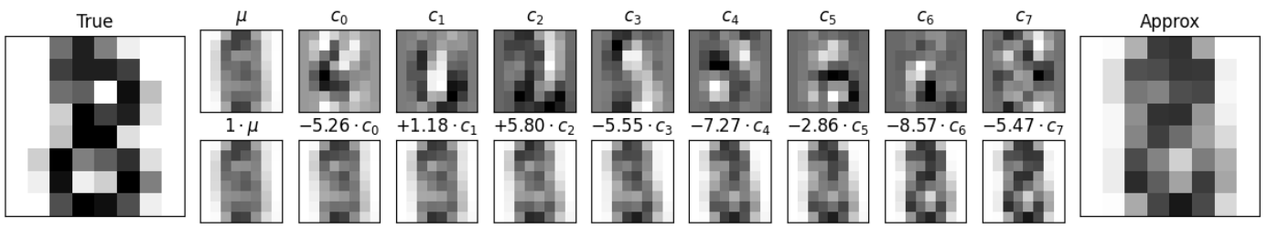
```
k = 7
X = digits.data[k]
coeff = Xproj[k]
show_pca_components(X, coeff)
```




```
k = 1
X = digits.data[k]
coeff = Xproj[k]
show_pca_components(X, coeff)
```



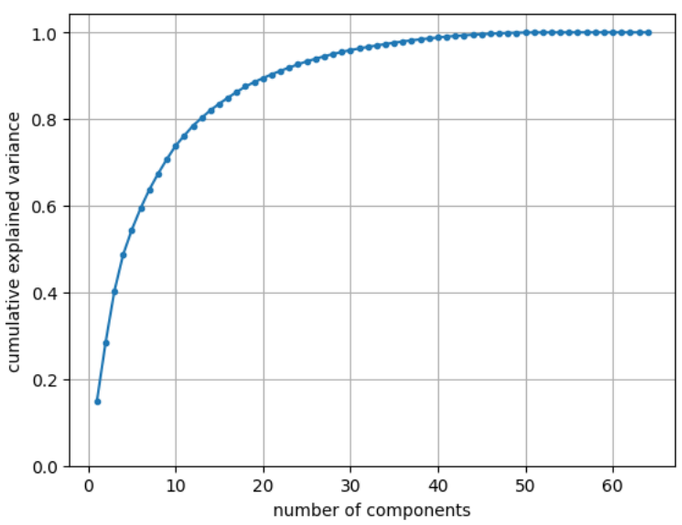

```
k = 8
X = digits.data[k]
coeff = Xproj[k]
show_pca_components(X, coeff)
```



Choosing the number of components

A vital part of using PCA in practice is the ability to estimate how many components are needed to describe the data. This can be determined by looking at the cumulative *explained variance ratio* as a function of the number of components:

```
pca = PCA().fit(digits.data)
plt.plot(range(1,65), np.cumsum(pca.explained_variance_ratio_), '.-')
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance')
plt.ylim(0,)
plt.grid(True);
```



This curve quantifies how much of the total, 64-dimensional variance is contained within the first N components. For example, we see that with the digits the first 10 components contain approximately 75% of the variance, while you need around 50 components to describe close to 100% of the variance.

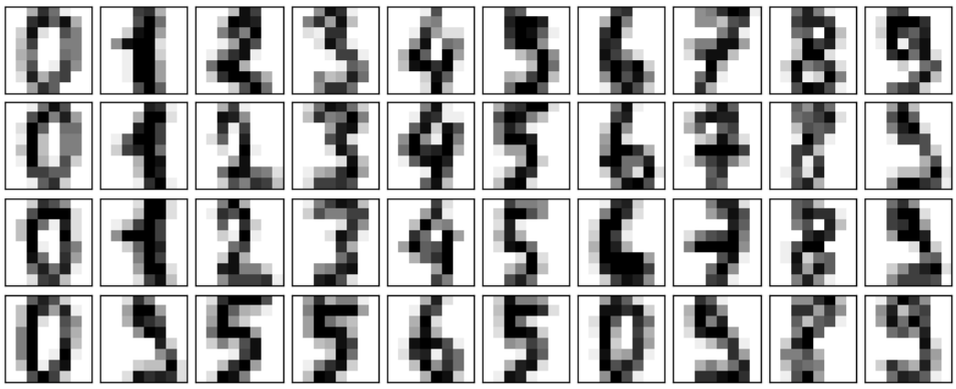
Here we see that our two-dimensional projection loses a lot of information (as measured by the explained variance) and that we'd need about 20 components to retain 90% of the variance. Looking at this plot for a high-dimensional dataset can help you understand the level of redundancy present in multiple observations.

PCA as Noise Filtering

PCA can also be used as a filtering approach for noisy data. The idea is this: any components with variance much larger than the effect of the noise should be relatively unaffected by the noise. So if you reconstruct the data using just the largest subset of principal components, you should be preferentially keeping the signal and throwing out the noise.

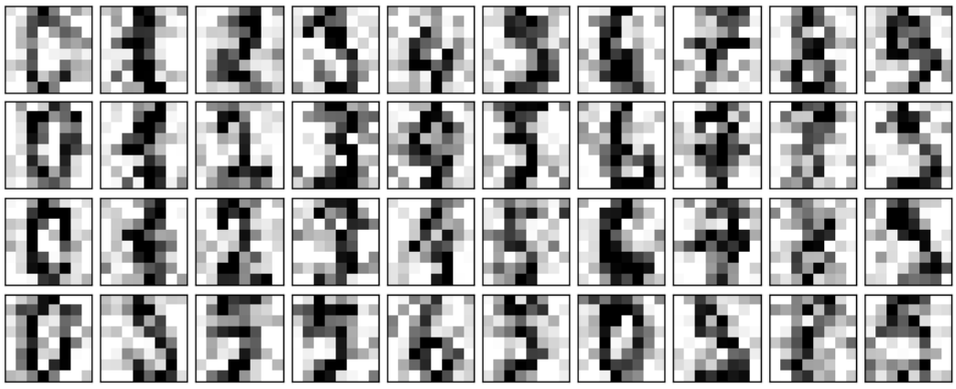
Let's see how this looks with the digits data. First we will plot several of the input noise-free data:

```
def plot_digits(data):
    fig, axes = plt.subplots(4, 10, figsize=(10, 4),
                             subplot_kw={'xticks':[], 'yticks':[]},
                             gridspec_kw=dict(hspace=0.1, wspace=0.1))
    # first 40 images
    for i, ax in enumerate(axes.flat):
        ax.imshow(data[i].reshape(8, 8),
                  cmap='binary', interpolation='nearest',
                  clim=(0, 16))
plot_digits(digits.data)
```



Now lets add some random noise to create a noisy dataset, and re-plot it:

```
np.random.seed(42)
#add normal noise with standard deviation 4
noisy = np.random.normal(digits.data,4)
plot_digits(noisy)
```



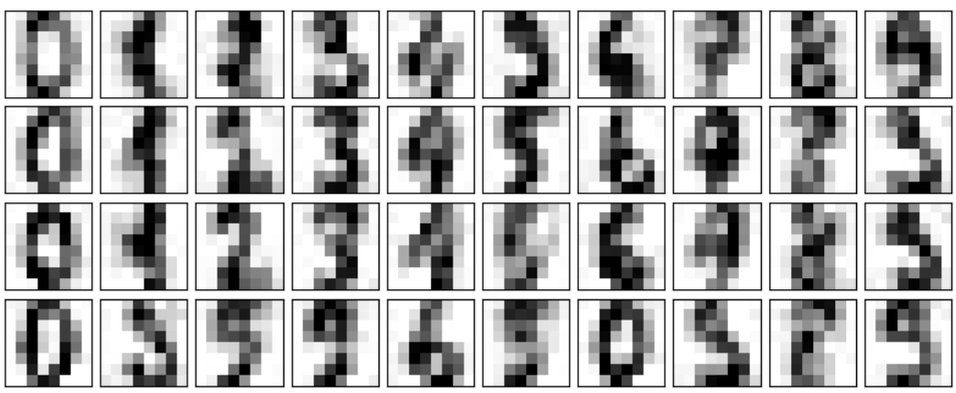
It's clear by eye that the images are noisy, and contain spurious pixels. Let's train a PCA on the noisy data, requesting that the projection preserve 50% of the variance:

```
pca = PCA(0.50).fit(noisy)  #automatically keep n_components that give at least 50% of variance of data
pca.n_components_
```

```
12
```

Here 50% of the variance amounts to 12 principal components. Now we compute these components, and then use the inverse of the transform to reconstruct the filtered digits:

```
components = pca.transform(noisy)
filtered = pca.inverse_transform(components)
plot_digits(filtered)
```



This signal preserving/noise filtering property makes PCA a very useful feature selection routine—for example, rather than training a classifier on very high-dimensional data, you might instead train the classifier on the lower-dimensional representation, which will automatically serve to filter out random noise in the inputs.

Example: Eigenfaces

Earlier we explored an example of using a PCA projection as a feature selector for facial recognition with a support vector machine (see In-Depth: Support Vector Machines). Here we will take a look back and explore a bit more of what went into that. Recall that we were using the Labeled Faces in the Wild dataset made available through Scikit-Learn:

```
from sklearn.datasets import fetch_lfw_people
faces = fetch_lfw_people(min_faces_per_person=60)
print(faces.target_names)
print(faces.images.shape)
```

```
['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush'
 'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']
(1348, 62, 47)
```

```
print (faces.DESCR)
```

```
.. _labeled_faces_in_the_wild_dataset:

The Labeled Faces in the Wild face recognition dataset
------------------------------------------------------

This dataset is a collection of JPEG pictures of famous people collected
over the internet, all details are available on the official website:

    http://vis-www.cs.umass.edu/lfw/

Each picture is centered on a single face. The typical task is called
Face Verification: given a pair of two pictures, a binary classifier
must predict whether the two images are from the same person.

An alternative task, Face Recognition or Face Identification is:
given the picture of the face of an unknown person, identify the name
of the person by referring to a gallery of previously seen pictures of
identified persons.

Both Face Verification and Face Recognition are tasks that are typically
performed on the output of a model trained to perform Face Detection. The
most popular model for Face Detection is called Viola-Jones and is
implemented in the OpenCV library. The LFW faces were extracted by this
face detector from various online websites.

**Data Set Characteristics:**

    =================   =======================
    Classes                                5749
    Samples total                         13233
    Dimensionality                         5828
    Features            real, between 0 and 255
    =================   =======================

Usage
~~~~~

``scikit-learn`` provides two loaders that will automatically download,
cache, parse the metadata files, decode the jpeg and convert the
interesting slices into memmapped numpy arrays. This dataset size is more
than 200 MB. The first load typically takes more than a couple of minutes
to fully decode the relevant part of the JPEG files into numpy arrays. If
the dataset has  been loaded once, the following times the loading times
less than 200ms by using a memmapped version memoized on the disk in the
``~/scikit_learn_data/lfw_home/`` folder using ``joblib``.

The first loader is used for the Face Identification task: a multi-class
classification task (hence supervised learning)::

  >>> from sklearn.datasets import fetch_lfw_people
  >>> lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)

  >>> for name in lfw_people.target_names:
  ...     print(name)
  ...
  Ariel Sharon
  Colin Powell
  Donald Rumsfeld
```

if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient 'randomized' method is enabled.

faces.data.shape = (1348, 2914), 150 < .8 *2914 so will auto use randomized

Let's take a look at the principal axes that span this dataset. Because this is a large dataset, we will use `RandomizedPCA`—it contains a randomized method to approximate the first N principal components much more quickly than the standard `PCA` estimator, and thus is very useful for high-dimensional data (here, a dimensionality of nearly 3,000). We will take a look at the first 150 components:

```
pca = PCA(150)
pca.fit(faces.data)
```

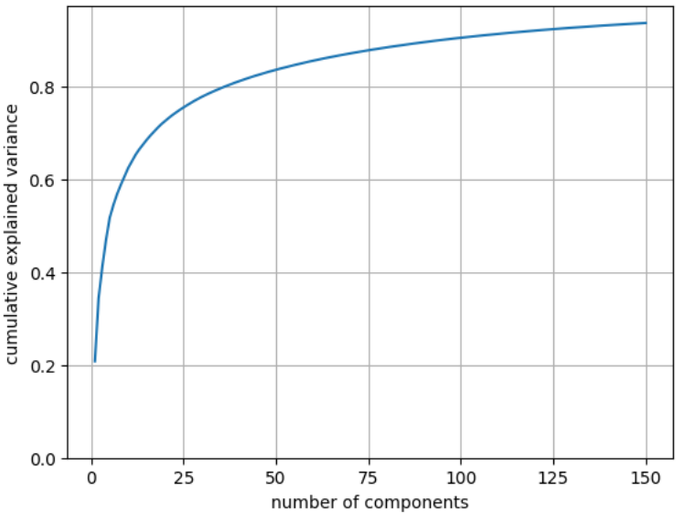
```
▾         PCA
PCA(n_components=150)
```

In this case, it can be interesting to visualize the images associated with the first several principal components (these components are technically known as "eigenvectors," so these types of images are often called "eigenfaces"). As you can see in this figure, they are as creepy as they sound:

```
fig, axes = plt.subplots(3, 8, figsize=(9, 4),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))
# see first 24 components
for i, ax in enumerate(axes.flat):
    ax.imshow(pca.components_[i].reshape(62, 47), cmap='bone', interpolation='gaussian')
```

The results are very interesting, and give us insight into how the images vary: for example, the first few eigenfaces (from the top left) seem to be associated with the angle of lighting on the face, and later principal vectors seem to be picking out certain features, such as eyes, noses, and lips. Let's take a look at the cumulative variance of these components to see how much of the data information the projection is preserving:

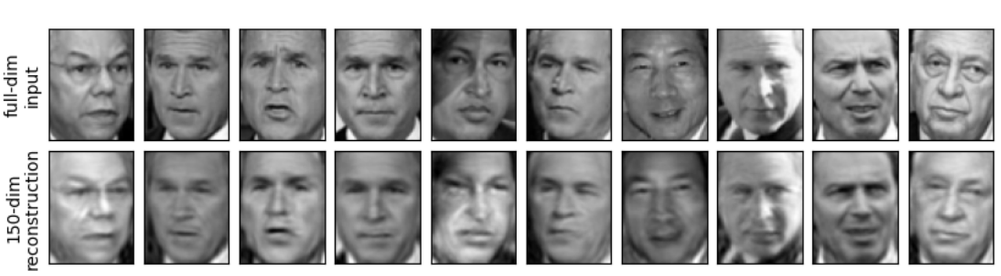
```
plt.plot(range(1,len(pca.explained_variance_ratio_)+1),
          np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance')
plt.xticks(range(0,155,25))
plt.ylim(0,)
plt.grid(True);
```



We see that these 150 components account for just over 90% of the variance. That would lead us to believe that using these 150 components, we would recover most of the essential characteristics of the data. To make this more concrete, we can compare the input images with the images reconstructed from these 150 components:

```
# Compute the components and projected faces
pca = PCA(150).fit(faces.data) #same as three cells above
components = pca.transform(faces.data)
projected = pca.inverse_transform(components)
```

```
# Plot the results
fig, ax = plt.subplots(2, 10, figsize=(10, 2.5),
                       subplot_kw={'xticks':[], 'yticks':[]},
                       gridspec_kw=dict(hspace=0.1, wspace=0.1))
for i in range(10):
    ax[0, i].imshow(faces.data[i].reshape(62, 47), cmap='binary_r')
    ax[1, i].imshow(projected[i].reshape(62, 47), cmap='binary_r')

ax[0, 0].set_ylabel('full-dim\ninput')
ax[1, 0].set_ylabel('150-dim\nreconstruction');
```



The top row here shows the input images, while the bottom row shows the reconstruction of the images from just 150 of the ~3,000 initial features. This visualization makes clear why the PCA feature selection used in In-Depth: Support Vector Machines was so successful: although it reduces the dimensionality of the data by nearly a factor of 20, the projected images contain enough information that we might, by eye, recognize the individuals in the image. What this means is that our classification algorithm needs to be trained on 150-dimensional data rather than 3,000-dimensional data, which depending on the particular algorithm we choose, can lead to a much more efficient classification.

[PG added]: as for the digits, see how an image is built up from the principal components as a series of successive approximations:

```
k=79 #look at 80th image in the set
coeff = components[k]

plt.figure(figsize=(12, 23))
plt.suptitle ('successive approximations to k={}, up to 100th principal component'.format(k), y=.905, fontsize=15)
approx = pca.mean_.copy()
for i in range(100):
    approx += pca.components_[i]*coeff[i]
    plt.subplot(15,10,i+1)
    plt.imshow(approx.reshape(62,47), cmap='binary_r', vmin=0, vmax=1)
    plt.title(r"${:+.2f}\cdot c_{{{}}}$".format(coeff[i], i), fontsize=8)
    plt.axis('off')
```