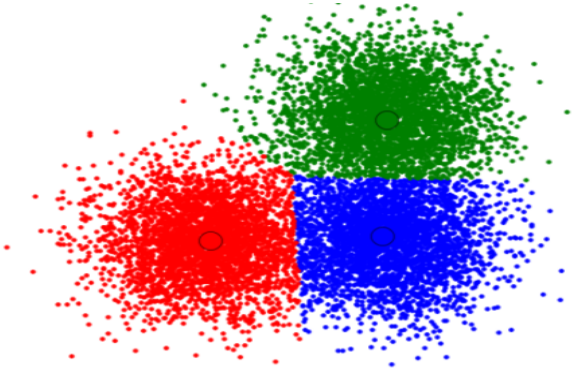
Unsupervised Clustering with Python

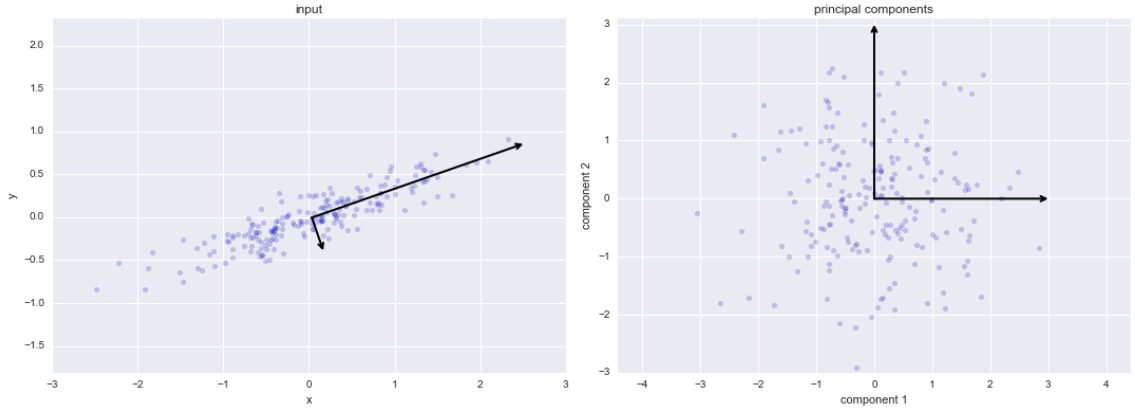


Clustering ...

Luís Garmendia

Index

Principal Component Analysis



Luís Garmendia

Unsupervised estimators

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**.

Principal Component Analysis

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.

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

Principal Component Analysis

Principal component analysis is a fast and flexible unsupervised method for dimensionality reduction in data.

Its behavior is easiest to visualize by looking at a two-dimensional dataset. Consider the following 200 points:

```
rng = np.random.RandomState(1)
X = np.dot(rng.rand(2, 2), rng.randn(2, 200)).T
plt.scatter(X[:, 0], X[:, 1])
plt.axis('equal');
```

Principal Component Analysis

By eye, it is clear that there is a nearly linear relationship between the x and y variables.

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.

Principal Component Analysis

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.

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

Principal Component Analysis

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

```
print(pca.components_)
```

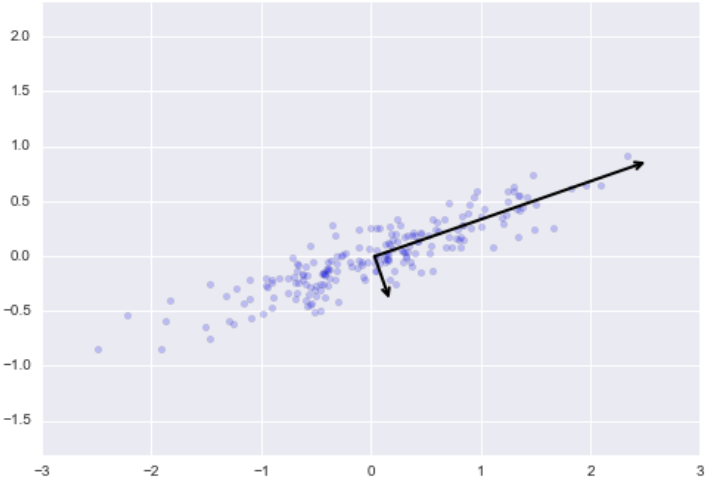
```
print(pca.explained_variance_)
```

```
In [417]: from sklearn.decomposition import PCA
          pca = PCA(n_components=2)
          pca.fit(X)

Out[417]: PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
              svd_solver='auto', tol=0.0, whiten=False)

In [418]: print(pca.components_)

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

In [419]: print(pca.explained_variance_)

          [0.7625315 0.0184779]
```

Principal Component Analysis visualization

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**:

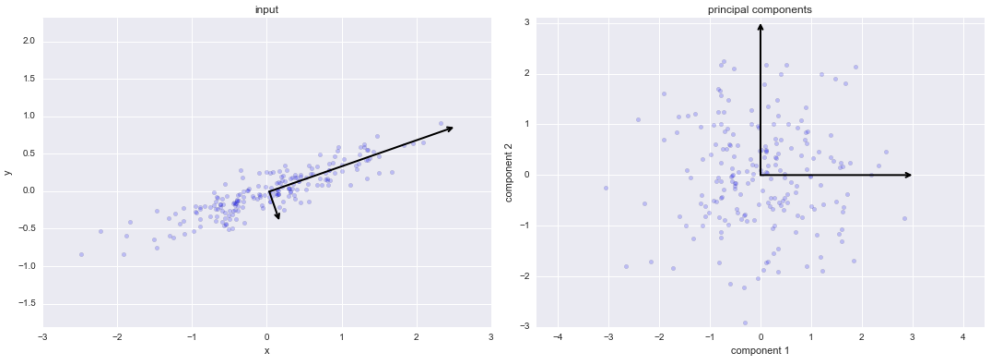
```
def draw_vector(v0, v1, ax=None):
    ax = ax or plt.gca()
    arrowprops=dict(arrowstyle='->',
                    linewidth=2,
                    shrinkA=0, shrinkB=0)
    ax.annotate('', v1, v0, arrowprops=arrowprops)
# plot data
plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
for length, vector in zip(pca.explained_variance_, pca.components_):
    v = vector * 3 * np.sqrt(length)
    draw_vector(pca.mean_, pca.mean_ + v)
plt.axis('equal');
```



Principal Component Analysis affine trasformation

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.

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



Principal Component Analysis PCA as dimensionaly 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)
```

```
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)
```

Principal Component Analysis
PCA as dimensionaly reduction

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[:, 0], X[:, 1], alpha=0.2)
plt.scatter(X_new[:, 0], X_new[:, 1], alpha=0.8)
plt.axis('equal');
```

Principal Component Analysis PCA as dimensionaly reduction

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 to the digits data

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.

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

```
In [426]: from sklearn.datasets import load_digits
          digits = load_digits()
          digits.data.shape

Out[426]: (1797, 64)
```

Recall that the data consists of 8×8 pixel images, meaning that they are 64-dimensional.

PCA to the digits data

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)
```

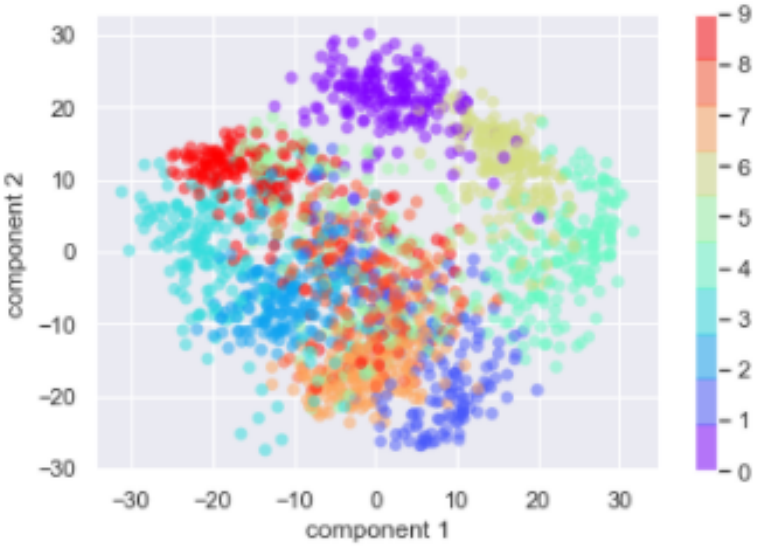
```
pca = PCA(2) # project from 64 to 2 dimensions
projected = pca.fit_transform(digits.data)
print(digits.data.shape)
print(projected.shape)
```

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

PCA to the digits data

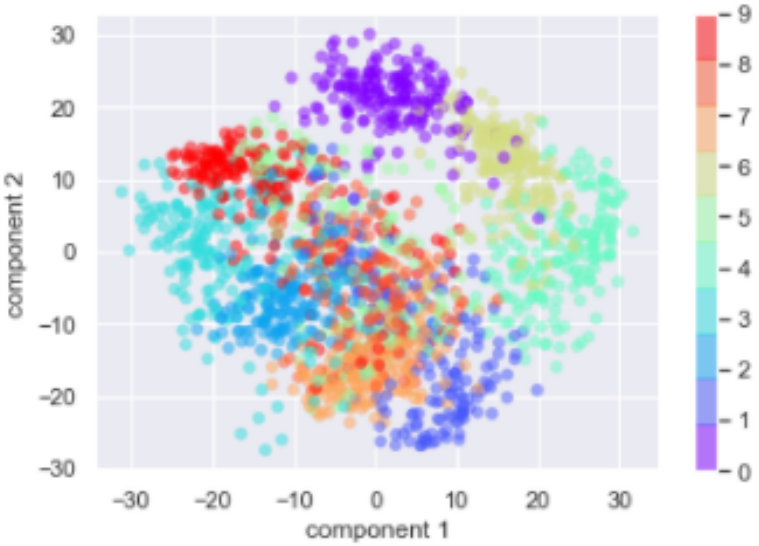
We can now plot the first two principal components of each point to learn about the data:

```
plt.scatter(projected[:, 0], projected[:, 1], c=digits.target,
edgecolor='none', alpha=0.5, cmap=plt.cm.get_cmap('rainbow', 10))
plt.xlabel('component 1')
plt.ylabel('component 2')
plt.colorbar();
```



PCA to the digits data

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**.



PCA components

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:

$x=[x_1,x2,x3\cdots x64]$

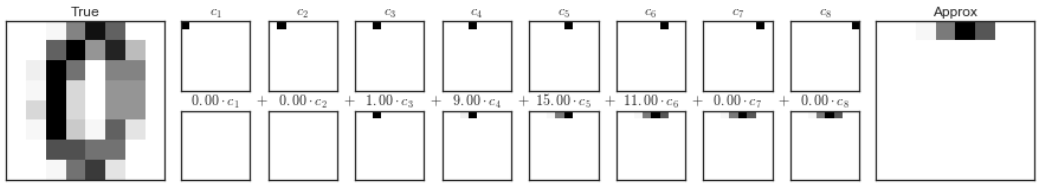
One way we can think about this is in terms of a pixel basis. That is, to construct the image, we multiply each element of the vector by the pixel it describes, and then add the results together to build the image:

$\text{image}(x)=x1\cdot(\text{pixel } 1)+x2\cdot(\text{pixel } 2)+x3\cdot(\text{pixel } 3)\cdots x_{64}\cdot(\text{pixel } 64)$

PCA components

We've thrown out nearly 90% of the pixels!

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!



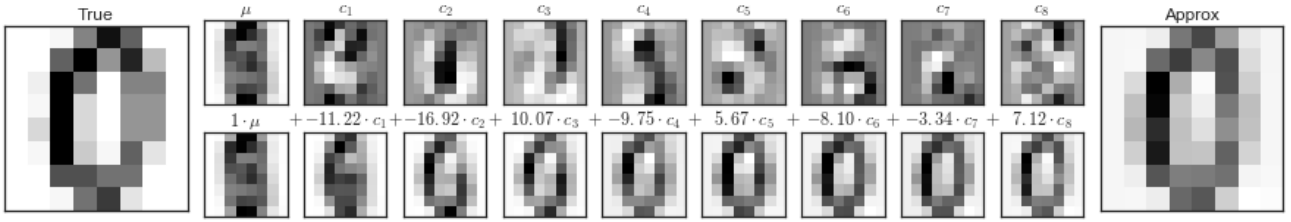
The upper row of panels shows the individual pixels, and the lower row shows the cumulative contribution of these pixels to the construction of the image. Using only eight of the pixel-basis components, we can only construct a small portion of the 64-pixel image. Were we to continue this sequence and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

PCA components

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 components

Unlike e 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.

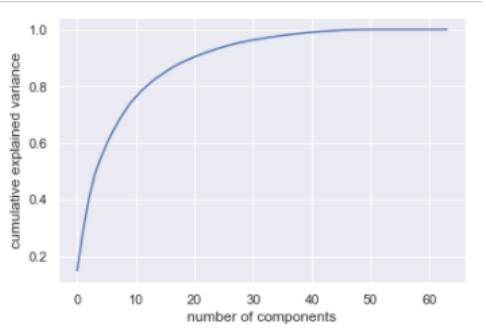
This is the sense in which PCA **provides a low-dimensional representation of the data**: it discovers a set of basis functions that are more efficient than the native pixel-basis of the input data.

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(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```

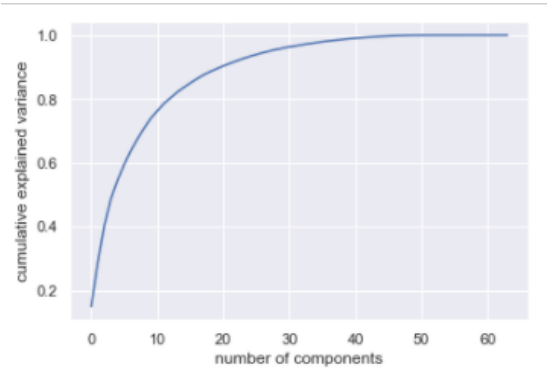


Choosing the number of components

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.



PCA as Noise Filtering

PCA can also be used as a filtering approach for noisy data.

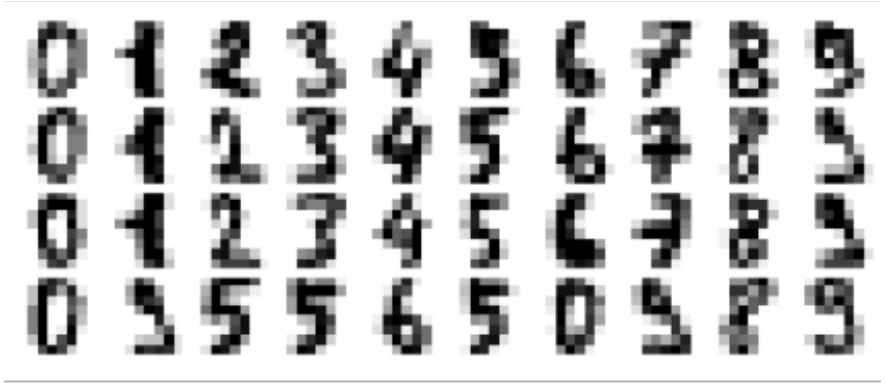
The idea is this: any components with **variance much larger than the effect of the noise should be relatively unaffected by the noise.**

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**.

PCA as Noise Filtering

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))
   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)
```



PCA as Noise Filtering

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

```
np.random.seed(42)
noisy = np.random.normal(digits.data, 4)
plot_digits(noisy)
```

PCA as Noise Filtering

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)
pca.n_components_
```

```
In [440]: pca = PCA(0.50).fit(noisy)
          pca.n_components_

Out[440]: 12
```

PCA as Noise Filtering

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)

```
In [441]: components = pca.transform(noisy)
          filtered = pca.inverse_transform(components)
          plot_digits(filtered)
```

PCA as Noise Filtering

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.

Eigenfaces
PCA as feature selector

We are 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)
```

```
In [443]: 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)
```

Eigenfaces
PCA as feature selector

Let's plot a few of these faces to see what we're working with:

```
fig, ax = plt.subplots(3, 5)
for i, axi in enumerate(ax.flat):
    axi.imshow(faces.images[i], cmap='bone')
    axi.set(xticks=[], yticks=[], xlabel=faces.target_names[faces.target[i]])
```

Eigenfaces
PCA as feature selector

Each image contains [62 × 47] or nearly 3,000 pixels.

We could proceed by simply using each pixel value as a feature, but often it is more effective to use some sort of preprocessor to extract more meaningful features;

Extract 150 fundamental components to feed into our support vector machine classifier.

We can do this most straightforwardly by packaging the preprocessor and the classifier into a single pipeline:

```
from sklearn.svm import SVC
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline
pca = PCA(n_components=150, whiten=True, random_state=42)
svc = SVC(kernel='rbf', class_weight='balanced')
model = make_pipeline(pca, svc)
```

PCA as feature selector

train_test_split

For the sake of testing our classify
er output, we will split the data into a training and testing set:

```
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = (faces.data, faces.target, random_state=42)
```

PCA as feature selector
determine the best model

Finally, we can use a grid search cross-validation to explore combinations of parameters. Here we will adjust C (which controls the margin hardness) and gamma (which controls the size of the radial basis function kernel), and determine the best model:

```
from sklearn.model_selection import GridSearchCV
param_grid = {'svc__C': [1, 5, 10, 50],
              'svc__gamma': [0.0001, 0.0005, 0.001, 0.005]}
grid = GridSearchCV(model, param_grid)
%time grid.fit(Xtrain, ytrain)
print(grid.best_params_)
```

```
from sklearn.model_selection import GridSearchCV
param_grid = {'svc__C': [1, 5, 10, 50],
              'svc__gamma': [0.0001, 0.0005, 0.001, 0.005]}
grid = GridSearchCV(model, param_grid)

%time grid.fit(Xtrain, ytrain)
print(grid.best_params_)
```

```
Wall time: 34.7 s
{'svc__C': 10, 'svc__gamma': 0.001}
```

PCA as feature selector test data

The optimal values fall toward the middle of our grid; if they fell at the edges, we would want to expand the grid to make sure we have found the true optimum.

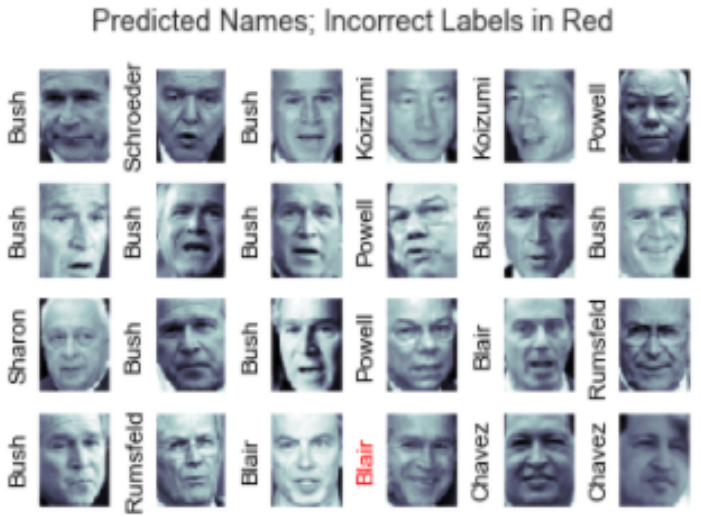
Now with this cross-validated model, we can predict the labels for the test data, which the model has not yet seen:

```
model = grid.best_estimator_
yfit = model.predict(Xtest)
```

PCA as feature selector predicted faces

Let's take a look at a few of the test images along with their predicted values:

```
fig, ax = plt.subplots(4, 6)
for i, axi in enumerate(ax.flat):
   axi.imshow(Xtest[i].reshape(62, 47), cmap='bone')
   axi.set(xticks=[], yticks=[])
   axi.set_ylabel(faces.target_names[yfit[i]].split()[-1],
                  color='black' if yfit[i] == ytest[i] else 'red')
fig.suptitle('Predicted Names; Incorrect Labels in Red', size=14);
```



PCA as feature selector performance

Estimate performance using the classification report, which lists recovery statistics label by label:

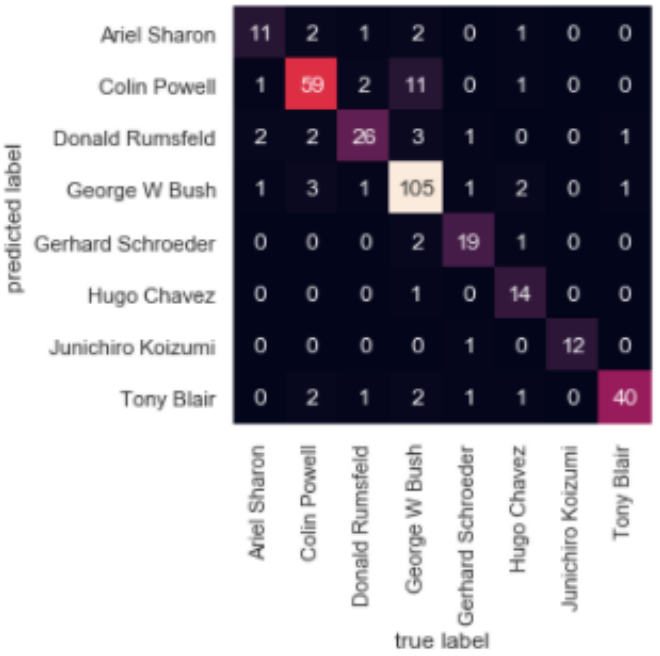
```
from sklearn.metrics import classification_report
print(classification_report(ytest, yfit, target_names=faces.target_names))
```

```
                   precision    recall  f1-score   support

     Ariel Sharon       0.65      0.73      0.69        15
     Colin Powell       0.80      0.87      0.83        68
  Donald Rumsfeld       0.74      0.84      0.79        31
    George W Bush       0.92      0.83      0.88       126
Gerhard Schroeder       0.86      0.83      0.84        23
      Hugo Chavez       0.93      0.70      0.80        20
Junichiro Koizumi       0.92      1.00      0.96        12
       Tony Blair       0.85      0.95      0.90        42

         accuracy                           0.85       337
        macro avg       0.83      0.84      0.84       337
     weighted avg       0.86      0.85      0.85       337
```

PCA as feature selector confusion matrix

We might also display the confusion matrix between these classes:

```
from sklearn.metrics import confusion_matrix
mat = confusion_matrix(ytest, yfit)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
        xticklabels=faces.target_names,
        yticklabels=faces.target_names)
plt.xlabel('true label')
plt.ylabel('predicted label');
```



Eigenfaces

Let's take a look at the principal axes that span this dataset.
We will take a look at the first 150 components:

from sklearn.decomposition import PCA
pca = PCA(150)
pca.fit(faces.data)

```
In [460]: from sklearn.decomposition import PCA
          pca = PCA(150)
          pca.fit(faces.data)

Out[460]: PCA(copy=True, iterated_power='auto', n_components=150, random_state=None,
              svd_solver='auto', tol=0.0, whiten=False)
```

Eigenfaces

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))
for i, ax in enumerate(axes.flat):
    ax.imshow(pca.components_[i].reshape(62, 47), cmap='bone')
```

```
In [461]: fig, axes = plt.subplots(3, 8, figsize=(9, 4),
                                   subplot_kw={'xticks':[], 'yticks':[]},
                                   gridspec_kw=dict(hspace=0.1, wspace=0.1))
          for i, ax in enumerate(axes.flat):
              ax.imshow(pca.components_[i].reshape(62, 47), cmap='bone')
```

Eigenfaces

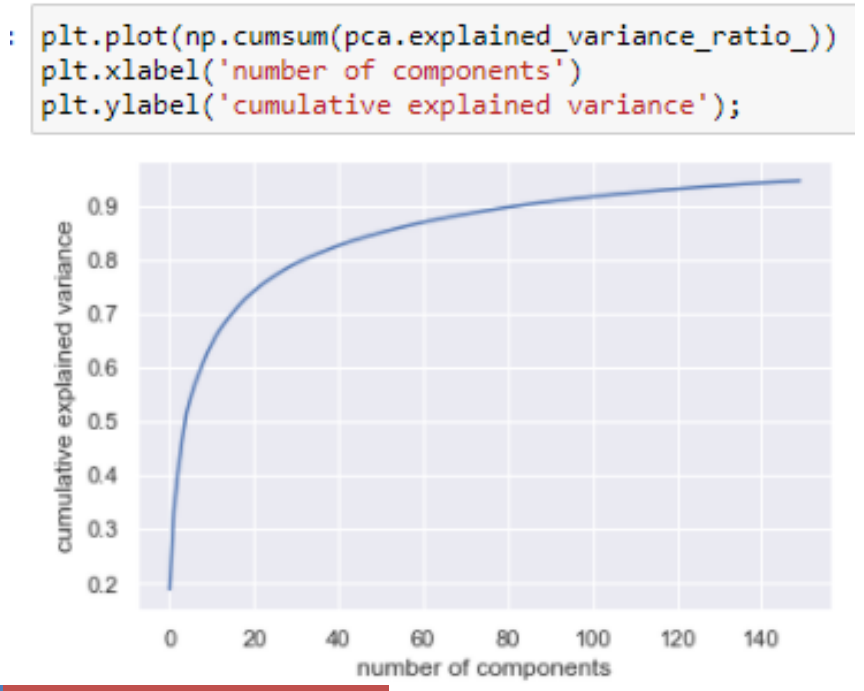
The results are very interesting, and give us insight into how the images vary: for example, the first few eigenfaces (from the top left) seem to be associated with the angle of **lighting** on the face, and later principal vectors seem to be picking out certain features, such as **eyes**, **noses**, and **lips**.

In [461]:
```
fig, axes = plt.subplots(3, 8, figsize=(9, 4),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))
for i, ax in enumerate(axes.flat):
    ax.imshow(pca.components_[i].reshape(62, 47), cmap='bone')
```

Eigenfaces

Let's take a look at the cumulative variance of these components to see how much of the data information the projection is preserving:

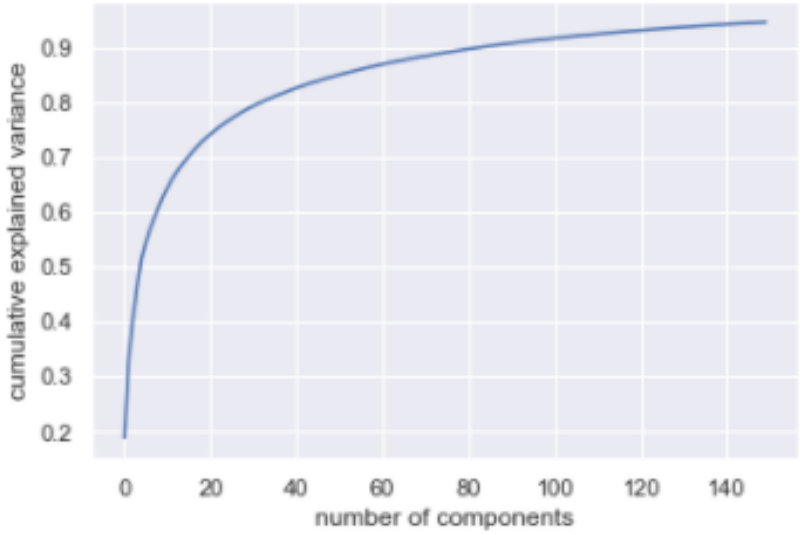
```
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```



Eigenfaces

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.

```
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```



Eigenfaces

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)
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');
```


```
# 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');
```

Eigenfaces

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 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.

Principal Component Analysis Summary

In this section we have discussed the use of principal component analysis for **dimensionality reduction**, for **visualization of high-dimensional data**, for **noise filtering**, and for **feature selection** within high-dimensional data. Because of the versatility and interpretability of PCA, it has been shown to be effective in a wide variety of contexts and disciplines.

Given any **high-dimensional** dataset, start with **PCA** in order to visualize the relationship between points (as we did with the digits), to understand the main variance in the data (as we did with the eigenfaces), and to understand the intrinsic dimensionality (by plotting the explained variance ratio). Certainly PCA is not useful for every high-dimensional dataset, but it offers a straightforward and efficient path to gaining insight into high-dimensional data.

Principal Component Analysis Summary

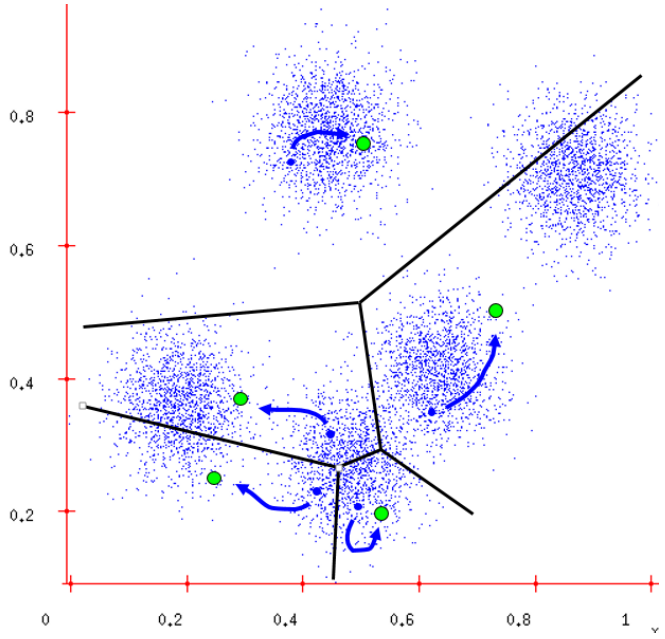
PCA's main **weakness** is that it tends to be highly affected by outliers (**valores atípicos**) in the data.

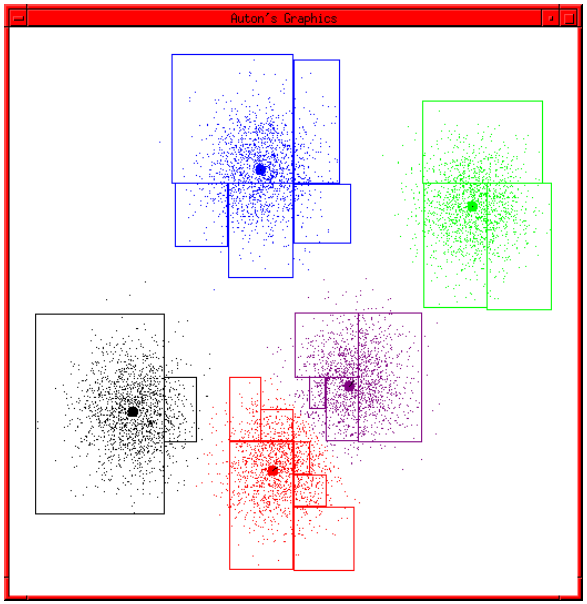
For this reason, many robust variants of PCA have been developed, many of which act to iteratively discard data points that are poorly described by the initial components.

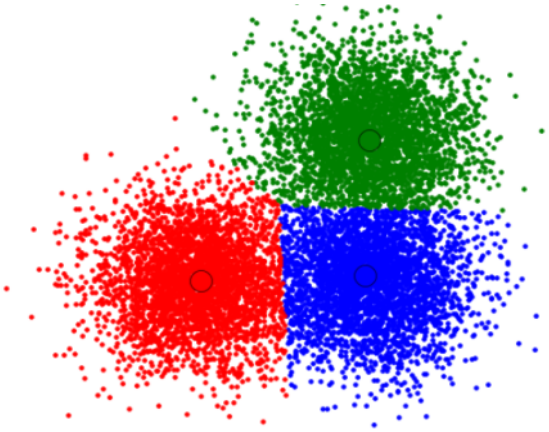
Scikit-Learn contains a couple interesting variants on PCA, including RandomizedPCA and SparsePCA, both also in the sklearn.decomposition submodule.

RandomizedPCA, uses a non-deterministic method to quickly approximate the first few principal components in very high-dimensional data, while SparsePCA introduces a regularization term that serves to enforce sparsity of the components.

K-means clustering





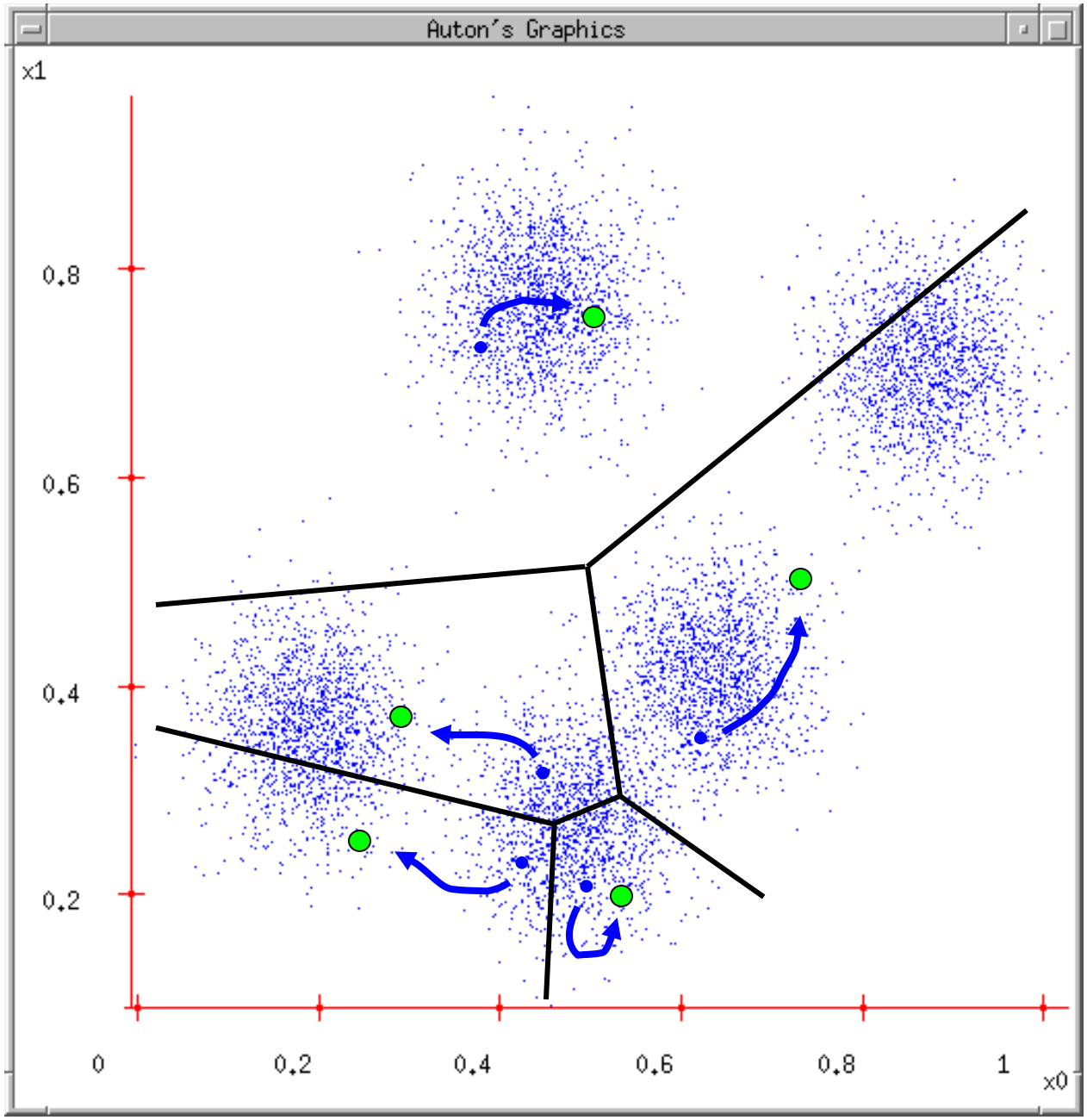


Clustering ...

Luís Garmendia

K-means

1. Preguntar número de clusters *(e.g. k=5)*
2. Escoger al azar k cluster Center locations
3. Cada punto escoge el centro más cercano
4. Cada centro escoge el centroide de sus puntos
5. Repetir….



K-means clustering

Clustering algorithms seek to learn, from the properties of the data, an **optimal division** or **discrete labeling** of groups of points.

Many clustering algorithms are available in Scikit-Learn and elsewhere, but perhaps the simplest to understand is an algorithm known as ***k-means clustering***, which is implemented in sklearn.cluster.KMeans.

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

K-means clustering

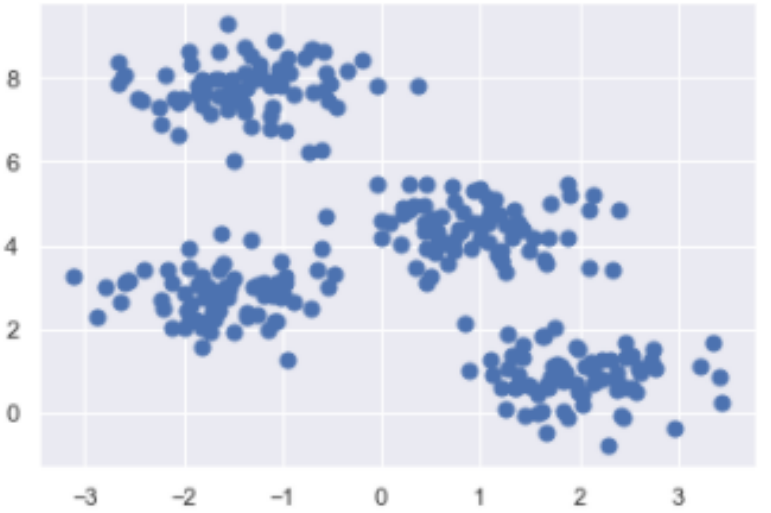
The *k*-means algorithm **searches** for a pre-determined number of **clusters** within an unlabeled multidimensional dataset. It accomplishes this using a simple conception of what the optimal clustering looks like:

- The "**cluster center**" is the arithmetic mean of all the points belonging to the cluster.
- Each point is closer to its own cluster center than to other cluster centers.

K-means clustering

First, let's generate a two-dimensional dataset containing four distinct blobs. To emphasize that this is an unsupervised algorithm, we will leave the labels out of the visualization

```
from sklearn.datasets.samples_generator import make_blobs
X, y_true = make_blobs(n_samples=300, centers=4, cluster_std=0.60,
random_state=0)
plt.scatter(X[:, 0], X[:, 1], s=50);
```

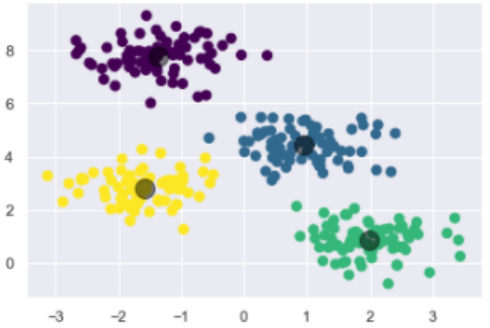


K-means clustering
pick up the clusters

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=4)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)
```

Let's visualize the results by plotting the data colored by these labels. We will also plot the cluster centers as determined by the *k*-means estimator

```
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5);
```

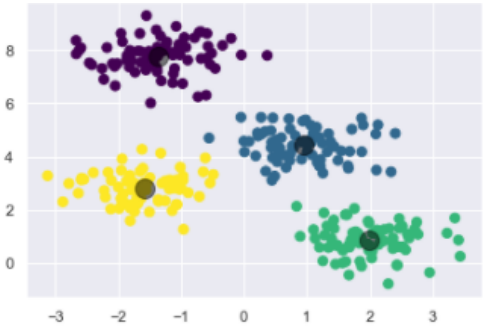


K-means clustering
pick up the clusters

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=4)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)
```

Let's visualize the results by plotting the data colored by these labels. We will also plot the cluster centers as determined by the *k*-means estimator

```
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5);
```



K-Means Algorithm: Expectation–Maximization

- Expectation–maximization (E–M) is a powerful algorithm that comes up in a variety of contexts within data science. *k*-means is a particularly simple and easy-to-understand application of the algorithm, and we will walk through it briefly here. In short, the expectation–maximization approach here consists of the following procedure:
- Guess some cluster centers
- Repeat until converged
  - *E-Step*: assign points to the nearest cluster center
  - *M-Step*: set the cluster centers to the mean

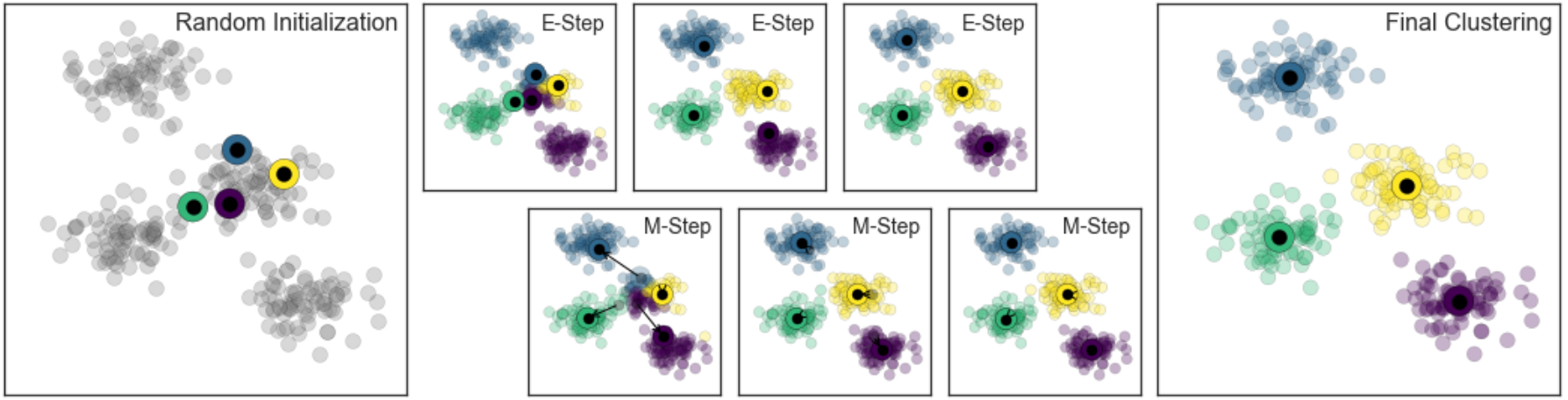
K-Means Algorithm: Expectation–Maximization

Here the "E-step" or "Expectation step" is so-named because it involves updating our expectation of which cluster each point belongs to. The "M-step" or "Maximization step" is so-named because it involves maximizing some fitness function that defines the location of the cluster centers—in this case, that maximization is accomplished by taking a simple mean of the data in each cluster.

Under typical circumstances, each repetition of the E-step and M-step will always result in a better estimate of the cluster characteristics.

K-Means Algorithm: Expectation–Maximization

For the particular initialization shown here, the clusters converge in just three iterations.

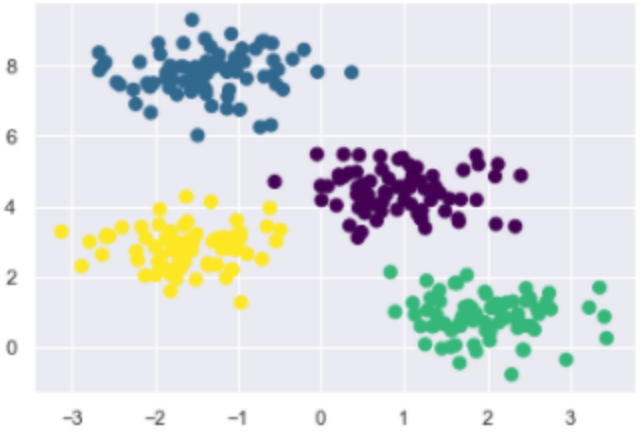


K-Means Algorithm: Expectation–Maximization

The *k*-Means algorithm is simple enough that we can write it in a few lines of code.

from sklearn.metrics import pairwise_distances_argmin

```
def find_clusters(X, n_clusters, rseed=2):
    # 1. Randomly choose clusters
    rng = np.random.RandomState(rseed)
    i = rng.permutation(X.shape[0])[:n_clusters]
    centers = X[i]
    while True:
        # 2a. Assign labels based on closest center
        labels = pairwise_distances_argmin(X, centers)
        # 2b. Find new centers from means of points
        new_centers = np.array([X[labels == i].mean(0)
                                for i in range(n_clusters)])
        # 2c. Check for convergence
        if np.all(centers == new_centers):
            break
        centers = new_centers
    return centers, labels
centers, labels = find_clusters(X, 4)
plt.scatter(X[:, 0], X[:, 1], c=labels,
            s=50, cmap='viridis');
```

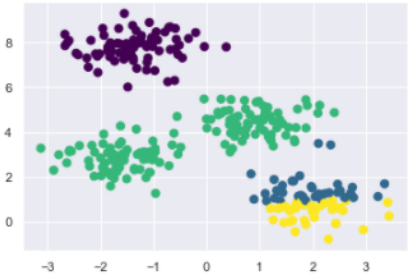


Expectation–Maximization issues

- The globally optimal result may not be achieved

First, although the E–M procedure is guaranteed to improve the result in each step, there is no assurance that it will lead to the *global* best solution. For example, if we use a different random seed in our simple procedure, the particular starting guesses lead to poor results:

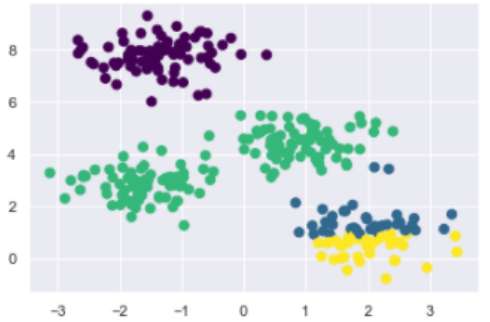
```
centers, labels = find_clusters(X, 4, rseed=0)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=50, cmap='viridis');
```



Expectation–Maximization issues

- The globally optimal result may not be achieved

```
centers, labels = find_clusters(X, 4, rseed=0)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=50, cmap='viridis');
```



Here the E–M approach has converged, but has not converged to a globally optimal configuration. For this reason, it is common for the algorithm to be run for multiple starting guesses, as indeed Scikit-Learn does by default (set by the n_init parameter, which defaults to 10).

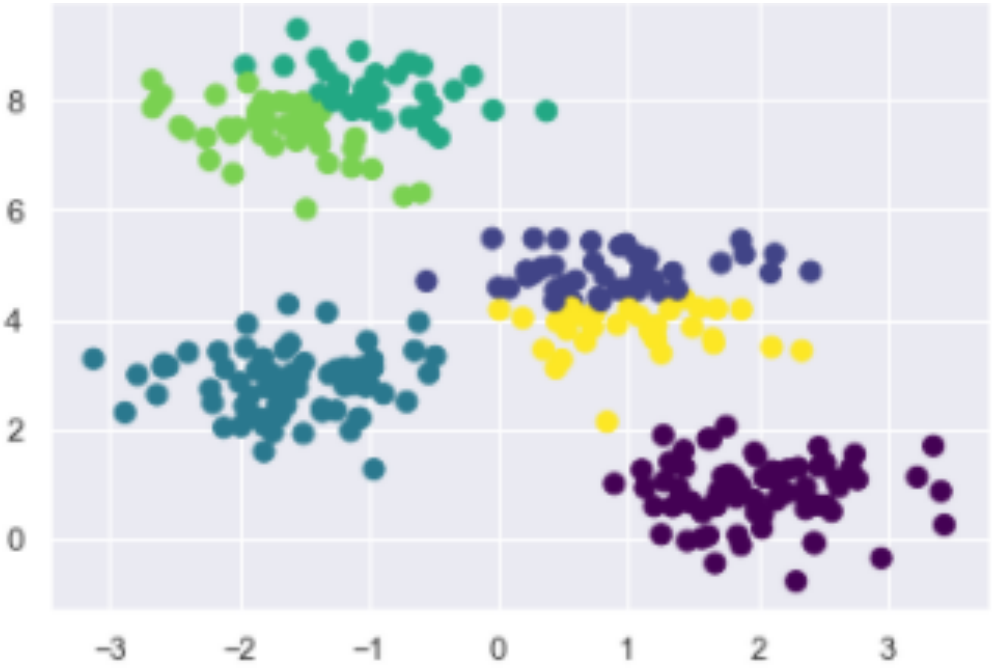
Expectation–Maximization issues

- The number of clusters must be selected beforehand

Another common challenge with k-means is that you must tell it how many clusters you expect: it cannot learn the number of clusters from the data. For example, if we ask the algorithm to identify six clusters, it will happily proceed and find the best six clusters

Expectation–Maximization issues

```
labels = KMeans(6, random_state=0).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=50, cmap='viridis');
```



Expectation–Maximization issues

- k-means is limited to linear cluster boundaries

The fundamental model assumptions of *k*-means (points will be closer to their own cluster center than to others) means that the algorithm will often be ineffective if the clusters have complicated geometries.

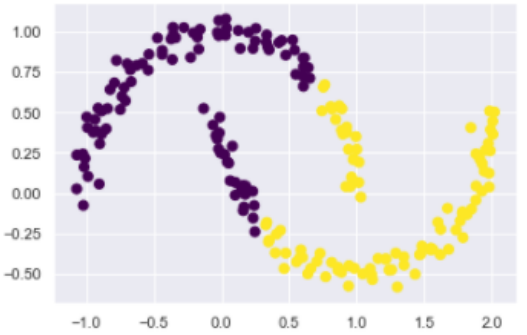
In particular, the boundaries between *k*-means clusters will always be linear, which means that it will fail for more complicated boundaries. Consider the following data, along with the cluster labels found by the typical *k*-means approach:

Expectation–Maximization issues

- k-means is limited to linear cluster boundaries

Consider the following data, along with the cluster labels found by the typical *k*-means approach:

```
from sklearn.datasets import make_moons
X, y = make_moons(200, noise=.05, random_state=0)

labels = KMeans(2, random_state=0).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=50, cmap='viridis');
```



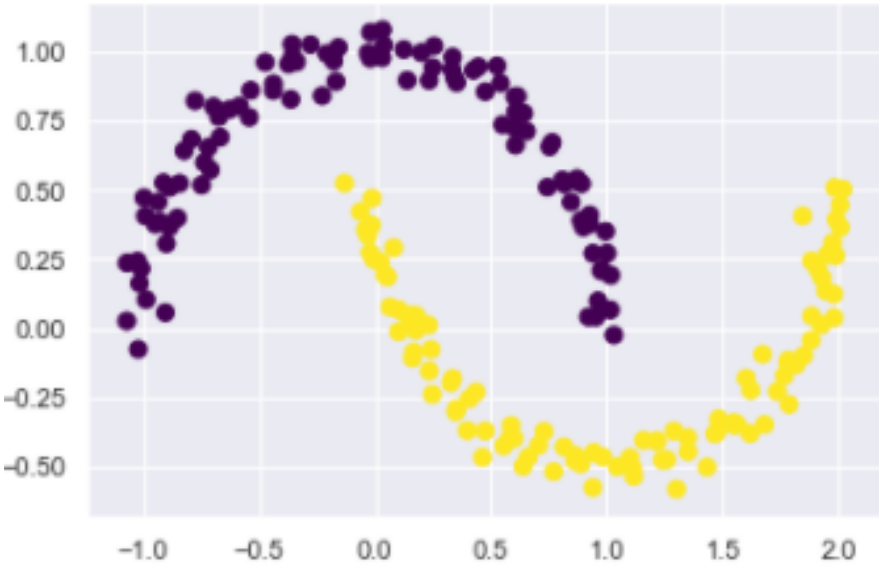
Expectation–Maximization issues
Spectral Clustering

In Support Vector Machines, where we used a **kernel transformation** to project the data into a higher dimension where a linear separation is possible. We might imagine using the same trick to allow *k*-means to discover non-linear boundaries.

One version of this kernelized *k*-means is implemented in Scikit-Learn within the **SpectralClustering** estimator. It uses the graph of nearest neighbors to compute a higher-dimensional representation of the data, and then assigns labels using a *k*-means algorithm:

Expectation–Maximization issues
Spectral Clustering

```
from sklearn.cluster import SpectralClustering
model = SpectralClustering(n_clusters=2,
affinity='nearest_neighbors', assign_labels='kmeans')
labels = model.fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=50, cmap='viridis');
```



Expectation–Maximization issues big samples

- k-means can be slow for large numbers of samples

Because each iteration of *k*-means must access every point in the dataset, the algorithm can be relatively slow as the number of samples grows.

You might wonder if this requirement to use all data at each iteration can be relaxed; for example, you might just use a subset of the data to update the cluster centers at each step.

This is the idea behind batch-based *k*-means algorithms, one form of which is implemented in sklearn.cluster.MiniBatchKMeans. The interface for this is the same as for standard KMeans;

k-means on digits

Here we will attempt to use *k*-means to try to identify similar digits *without using the original label information*; this might be similar to a first step in extracting meaning from a new dataset about which you don't have any *a priori* label information.

Recall that the digits consist of 1,797 samples with 64 features, where each of the 64 features is the brightness of one pixel in an 8×8 image:

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

```
(1797, 64)
```

k-means on digits

The clustering can be performed

```
kmeans = KMeans(n_clusters=10, random_state=0)
clusters = kmeans.fit_predict(digits.data)
kmeans.cluster_centers_.shape
```

```
kmeans = KMeans(n_clusters=10, random_state=0)
clusters = kmeans.fit_predict(digits.data)
kmeans.cluster_centers_.shape
```

```
(10, 64)
```

The result is 10 clusters in 64 dimensions. Notice that the cluster centers themselves are 64-dimensional points, and can themselves be interpreted as the "typical" digit within the cluster. Let's see what these cluster centers look like:

k-means on digits

Notice that the cluster centers themselves are 64-dimensional points, and can themselves be interpreted as the "typical" digit within the cluster. Let's see what these cluster centers look like:

```
fig, ax = plt.subplots(2, 5, figsize=(8, 3))
centers = kmeans.cluster_centers_.reshape(10, 8, 8)
for axi, center in zip(ax.flat, centers):
    axi.set(xticks=[], yticks=[])
    axi.imshow(center, interpolation='nearest', cmap=plt.cm.binary)
```

k-means on digits accuracy

Because *k*-means knows nothing about the identity of the cluster, the 0–9 labels may be permuted. We can fix this by matching each learned cluster label with the true labels found in them

```
from scipy.stats import mode
labels = np.zeros_like(clusters)
for i in range(10):
    mask = (clusters == i)
    labels[mask] = mode(digits.target[mask])[0]
```

```
clusters.shape
```

```
(1797,)
```

```
np.zeros_like(clusters).shape
```

```
(1797,)
```

```
np.zeros_like(clusters)
```

```
array([0, 0, 0, ..., 0, 0, 0])
```

```
from scipy.stats import mode
labels = np.zeros_like(clusters)
for i in range(10):
    mask = (clusters == i)
    labels[mask] = mode(digits.target[mask])[0]
```

```
labels
```

```
array([0, 8, 8, ..., 8, 9, 9])
```

```
mask
```

```
array([False,  True,  True, ...,  True, False, False])
```

```
labels.shape
```

```
(1797,)
```

k-means on digits accuracy

Now we can check how accurate our unsupervised clustering was in finding similar digits within the data:

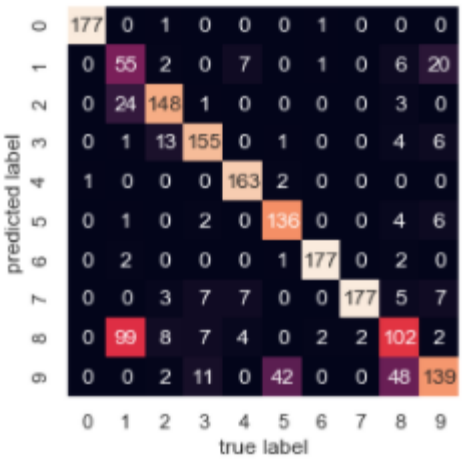
```
from sklearn.metrics import accuracy_score
accuracy_score(digits.target, labels)
```

```
In [501]: from sklearn.metrics import accuracy_score
          accuracy_score(digits.target, labels)

Out[501]: 0.7952142459654981
```

k-means on digits accuracy

With just a simple *k*-means algorithm, we discovered the correct grouping for 80% of the input digits! Let's check the confusion matrix for this:

```
from sklearn.metrics import confusion_matrix
mat = confusion_matrix(digits.target, labels)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
xticklabels=digits.target_names, yticklabels=digits.target_names)
plt.xlabel('true label')
plt.ylabel('predicted label');
```



k-means for color compression

One interesting application of clustering is in color compression within images.

For example, imagine you have an image with millions of colors. In most images, a large number of the colors will be unused, and many of the pixels in the image will have similar or even identical colors.

k-means for color compression

For example, consider the image shown in the following figure, which is from the Scikit-Learn datasets module (for this to work, you'll have to have the pillow Python package installed).

```
# Note: this requires the ``pillow`` package to be installed
from sklearn.datasets import load_sample_image
china = load_sample_image("china.jpg")
ax = plt.axes(xticks=[], yticks=[])
ax.imshow(china);
```

k-means for color compression

The image itself is stored in a three-dimensional array of size (height, width, RGB), containing red/blue/green contributions as integers from 0 to 255:

```
china.shape
```

```
(427, 640, 3)
```

k-means for color compression reshape and rescale

One way we can view this set of pixels is as a cloud of points in a three-dimensional color space. We will reshape the data to [n_samples x n_features], and rescale the colors so that they lie between 0 and 1:

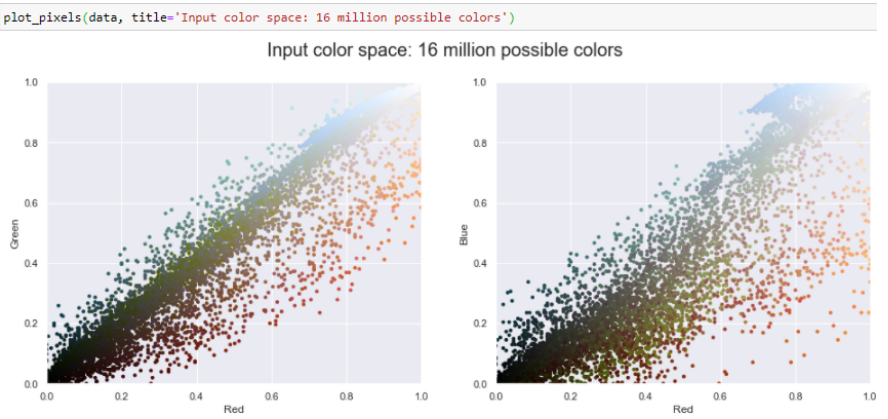
```
data = china / 255.0 # use 0...1 scale
data = data.reshape(427 * 640, 3)
data.shape
```

```
In [505]: data = china / 255.0 # use 0...1 scale
          data = data.reshape(427 * 640, 3)
          data.shape

Out[505]: (273280, 3)

In [506]: data

Out[506]: array([[0.68235294, 0.78823529, 0.90588235],
                 [0.68235294, 0.78823529, 0.90588235],
                 [0.68235294, 0.78823529, 0.90588235],
                 ...,
                 [0.16862745, 0.19215686, 0.15294118],
                 [0.05098039, 0.08235294, 0.02352941],
                 [0.05882353, 0.09411765, 0.02745098]])
```

k-means for color compression visualize

We can visualize these pixels in this color space, using a subset of 10,000 pixels for efficiency:

```
def plot_pixels(data, title, colors=None, N=10000):
    if colors is None:
        colors = data
    # choose a random subset
    rng = np.random.RandomState(0)
    i = rng.permutation(data.shape[0])[:N]
    colors = colors[i]
    R, G, B = data[i].T
    fig, ax = plt.subplots(1, 2, figsize=(16, 6))
    ax[0].scatter(R, G, color=colors, marker='.')
    ax[0].set(xlabel='Red', ylabel='Green', xlim=(0, 1), ylim=(0, 1))
    ax[1].scatter(R, B, color=colors, marker='.')
    ax[1].set(xlabel='Red', ylabel='Blue', xlim=(0, 1), ylim=(0, 1))
    fig.suptitle(title, size=20);

plot_pixels(data, title='Input color space: 16 million possible colors')
```

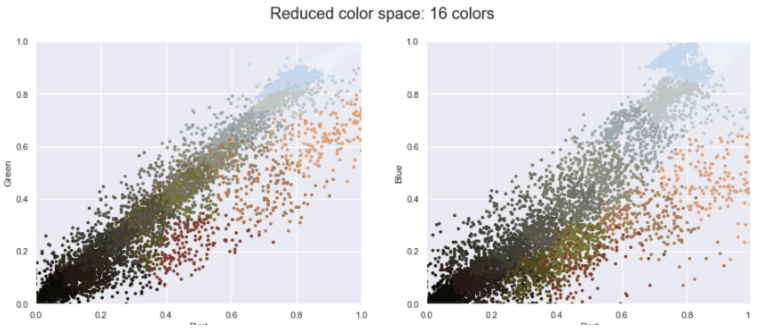


k-means for color compression reduce 16M colors to 16 colors

colors to just 16 colors, using a *k*-means clustering across the pixel space.

Because we are dealing with a very large dataset, we will use the **mini batch *k*-means**, which operates on subsets of the data to compute the result much more quickly than the standard *k*-means algorithm:

```
import warnings; warnings.simplefilter('ignore')  # Fix NumPy issues.
from sklearn.cluster import MiniBatchKMeans
kmeans = MiniBatchKMeans(16)
kmeans.fit(data)
new_colors = kmeans.cluster_centers_[kmeans.predict(data)]
plot_pixels(data, colors=new_colors,
        title="Reduced color space: 16 colors")
```



k-means for color compression recolored picture

The result is a re-coloring of the original pixels, where each pixel is assigned the color of its closest cluster center. Plotting these new colors in the image space rather than the pixel space shows us the effect of this:

```
china_recolored = new_colors.reshape(china.shape)
fig, ax = plt.subplots(1, 2, figsize=(16, 6),
                       subplot_kw=dict(xticks=[], yticks=[]))
fig.subplots_adjust(wspace=0.05)
ax[0].imshow(china)
ax[0].set_title('Original Image', size=16)
ax[1].imshow(china_recolored)
ax[1].set_title('16-color Image', size=16);
```

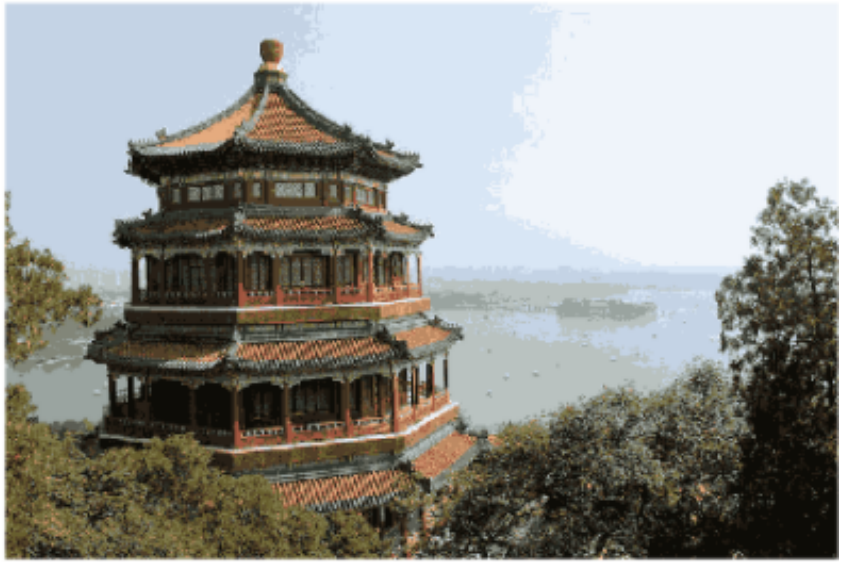
k-means for color compression
1M compression factor

Some detail is certainly lost in the rightmost panel, but the overall image is still easily recognizable. This image on the right achieves a compression factor of around 1 million!

While this is an interesting application of *k*-means, there are certainly better way to compress information in images.

Original Image

16-color Image



References

https://ipython.org/

https://numpy.org/

https://jupyter.org/try

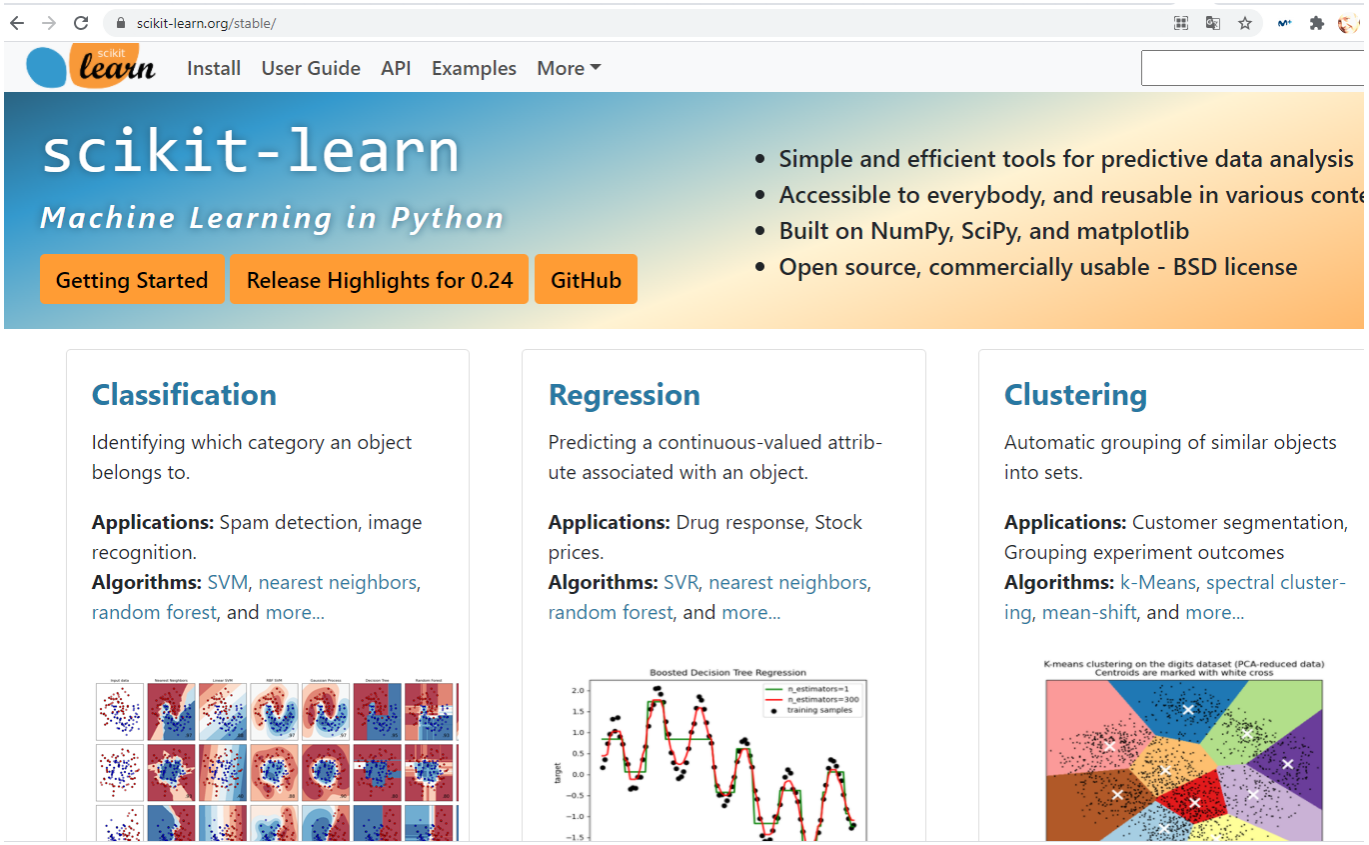
https://pandas.pydata.org/

http://seaborn.pydata.org/

https://scikit-learn.org/stable/

References

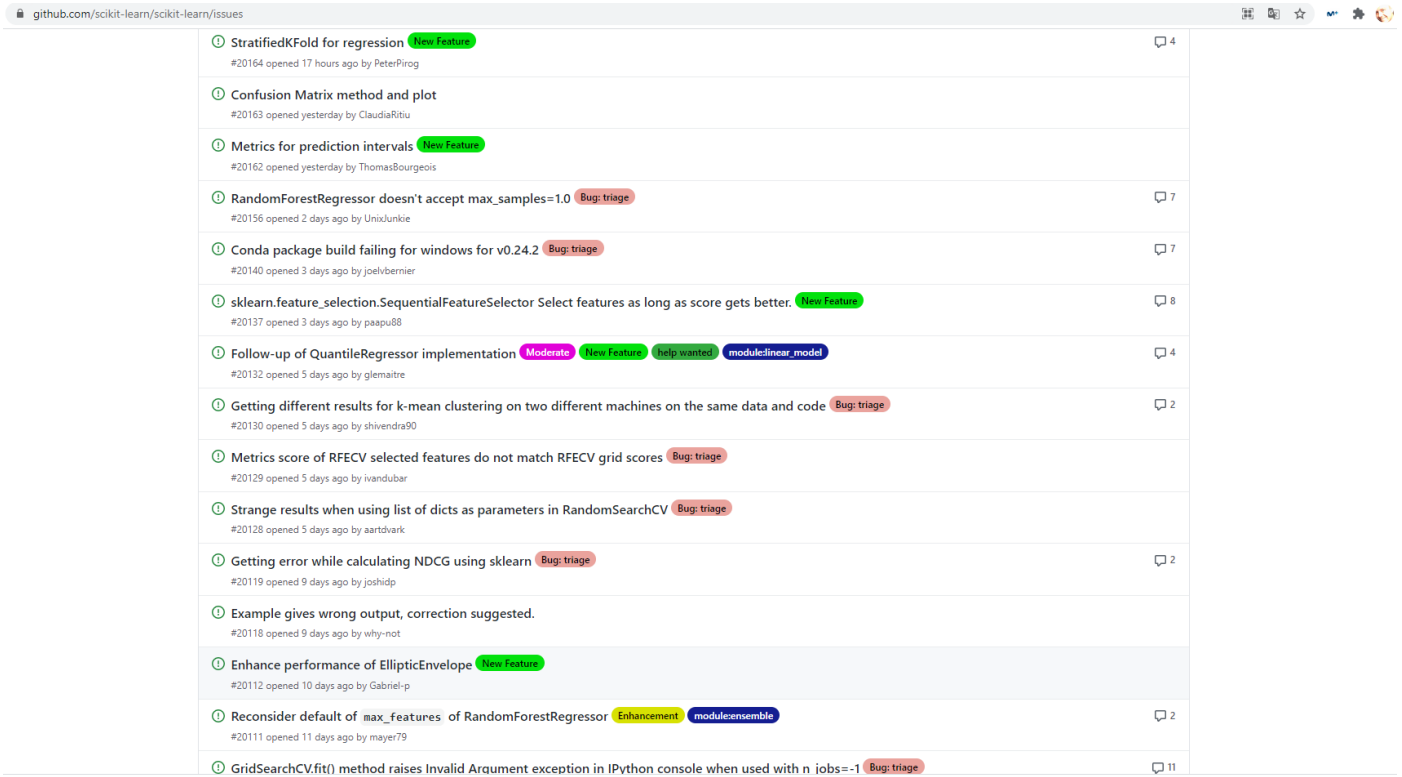
https://scikit-learn.org/stable/



References

Github:

https://github.com/scikit-learn/scikit-learn/issues



References

Github:

https://github.com/joanby/python-ml-course

https://github.com/rasbt/python-machine-learning-book-3rd-edition

https://github.com/tirthajyoti/Machine-Learning-with-Python

https://github.com/PacktPublishing/Python-Machine-Learning-Second-Edition

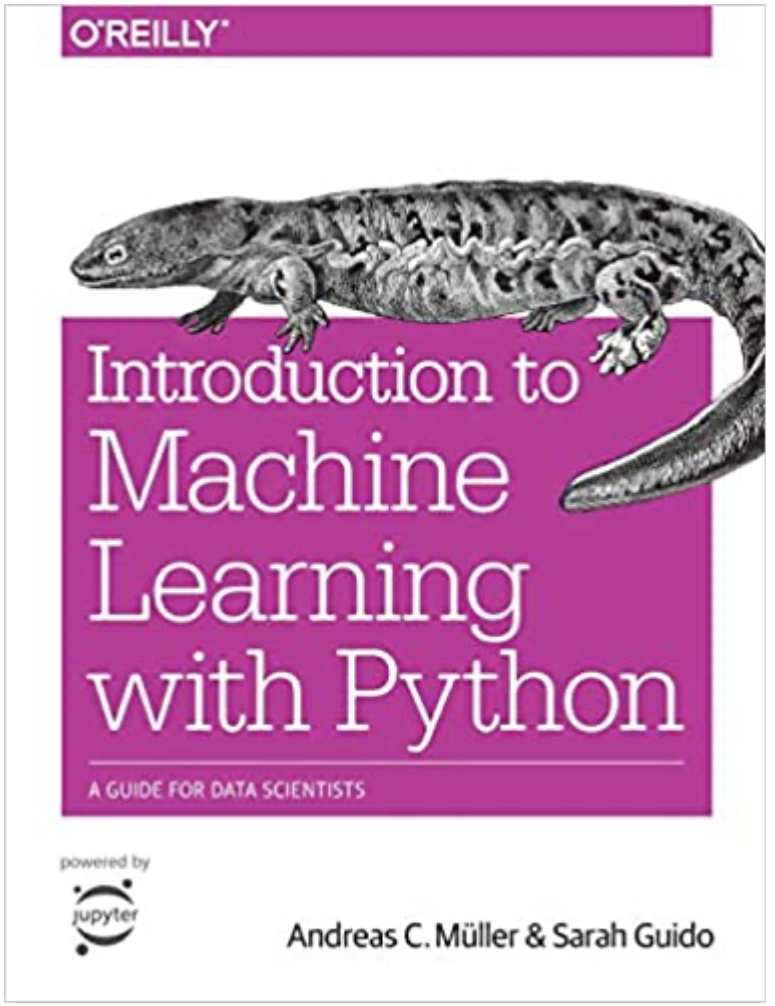
ML with Python References

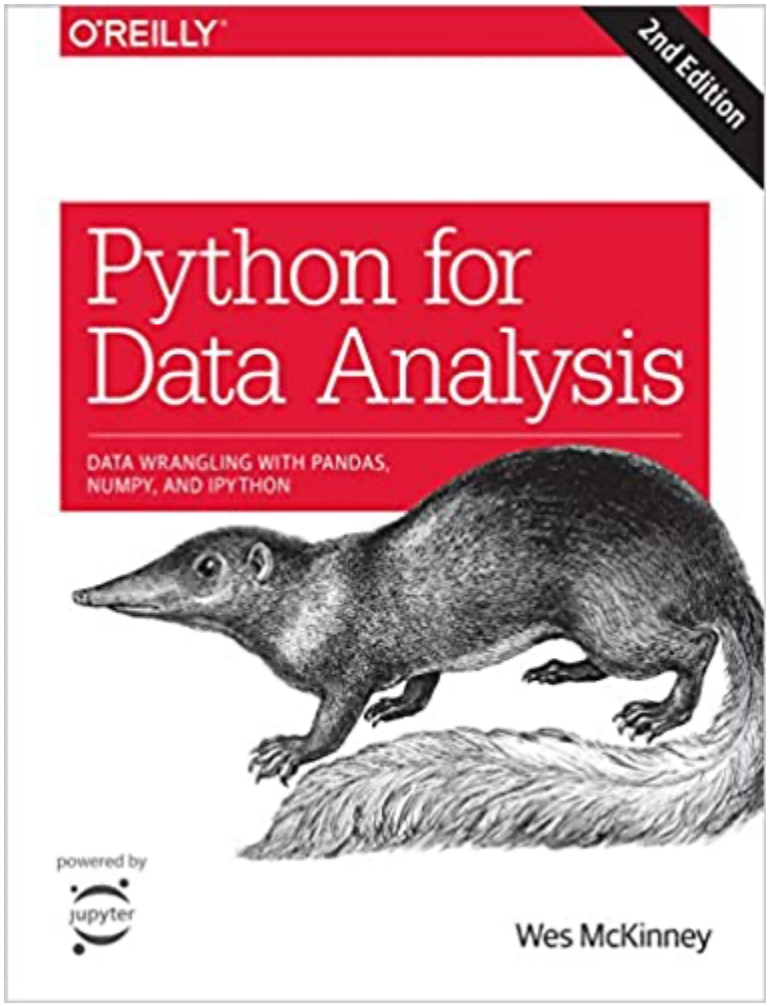
- The Scikit-Learn website: The Scikit-Learn website has an impressive breadth of documentation and examples covering some of the models discussed here, and much, much more. If you want a brief survey of the most important and often-used machine learning algorithms, this website is a good place to start.
- *SciPy, PyCon, and PyData tutorial videos*: Scikit-Learn and other machine learning topics are perennial favorites in the tutorial tracks of many Python-focused conference series, in particular the PyCon, SciPy, and PyData conferences. You can find the most recent ones via a simple web search.
- *Introduction to Machine Learning with Python*: Written by Andreas C. Mueller and Sarah Guido, this book includes a fuller treatment of the topics in this chapter. If you're interested in reviewing the fundamentals of Machine Learning and pushing the Scikit-Learn toolkit to its limits, this is a great resource, written by one of the most prolific developers on the Scikit-Learn team.
- *Python Machine Learning*: Sebastian Raschka's book focuses less on Scikit-learn itself, and more on the breadth of machine learning tools available in Python. In particular, there is some very useful discussion on how to scale Python-based machine learning approaches to large and complex datasets.

ML References

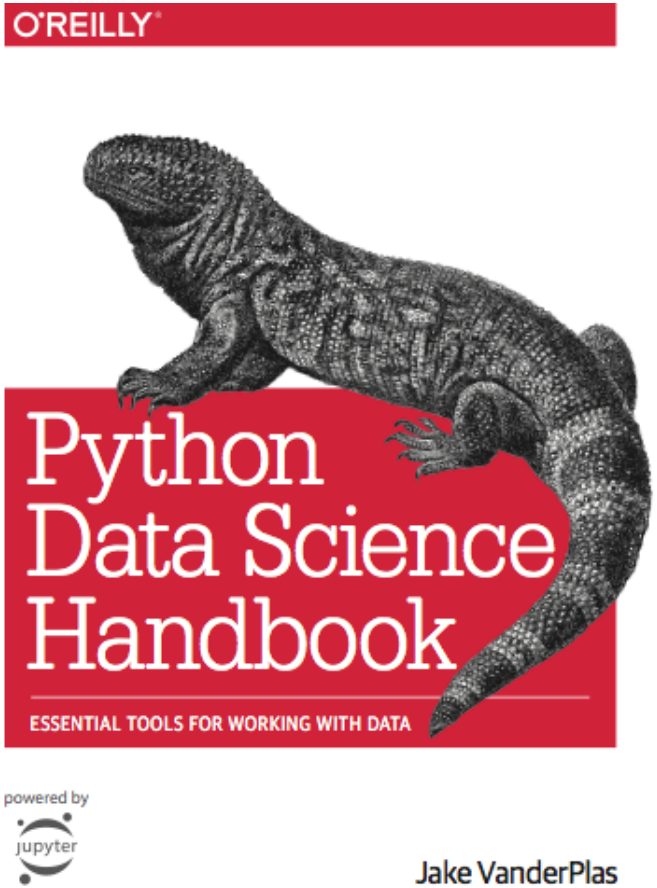
- *Machine Learning*: Taught by Andrew Ng (Coursera), this is a very clearly-taught free online course which covers the basics of machine learning from an algorithmic perspective. It assumes undergraduate-level understanding of mathematics and programming, and steps through detailed considerations of some of the most important machine learning algorithms. Homework assignments, which are algorithmically graded, have you actually implement some of these models yourself.
- *Pattern Recognition and Machine Learning*: Written by Christopher Bishop, this classic technical text covers the concepts of machine learning discussed in this chapter in detail. If you plan to go further in this subject, you should have this book on your shelf.
- *Machine Learning: a Probabilistic Perspective*: Written by Kevin Murphy, this is an excellent graduate-level text that explores nearly all important machine learning algorithms from a ground-up, unified probabilistic perspective.

References


O'REILLY
Introduction to
Machine
Learning
with Python
A GUIDE FOR DATA SCIENTISTS
powered by
jupyter
Andreas C. Müller & Sarah Guido


O'REILLY
2nd Edition
Python for
Data Analysis
DATA WRANGLING WITH PANDAS,
NUMPY, AND IPYTHON
powered by
jupyter
Wes McKinney

References



https://jakevdp.github.io/PythonDataScienceHandbook/