

Figure 5-54. Three perfect linear discriminative classifiers for our data

These are three *very* different separators that, nevertheless, perfectly discriminate between these samples. Depending on which one you choose, a new data point (e.g., the one marked by the “X” in Figure 5-54) will be assigned a different label! Evidently our simple intuition of “drawing a line between classes” is not enough, and we need to think a bit deeper.

Support Vector Machines: Maximizing the Margin

Support vector machines offer one way to improve on this. The intuition is this: rather than simply drawing a zero-width line between the classes, we can draw around each line a *margin* of some width, up to the nearest point. Here is an example of how this might look (Figure 5-55):

```
In[4]:
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')

for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:
    yfit = m * xfit + b
    plt.plot(xfit, yfit, '-k')
    plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none', color='AAAAAA',
                    alpha=0.4)

plt.xlim(-1, 3.5);
```

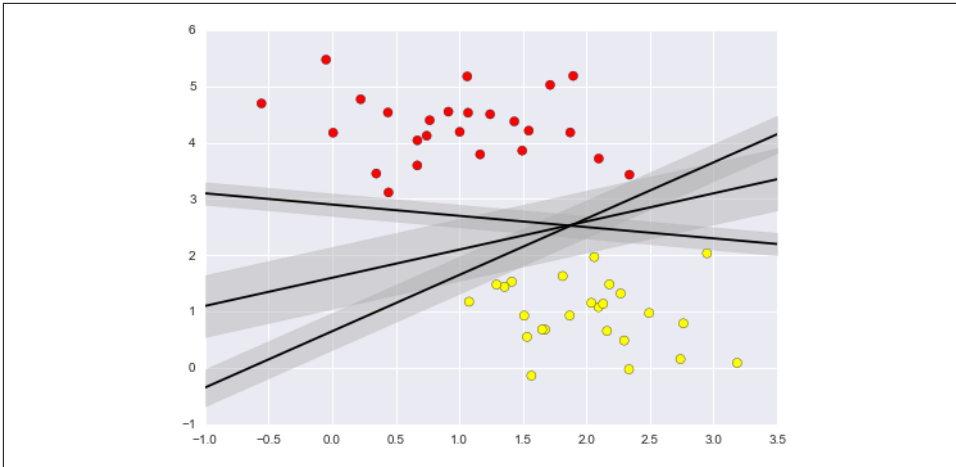


Figure 5-55. Visualization of “margins” within discriminative classifiers

In support vector machines, the line that maximizes this margin is the one we will choose as the optimal model. Support vector machines are an example of such a *maximum margin* estimator.

Fitting a support vector machine

Let’s see the result of an actual fit to this data: we will use Scikit-Learn’s support vector classifier to train an SVM model on this data. For the time being, we will use a linear kernel and set the C parameter to a very large number (we’ll discuss the meaning of these in more depth momentarily):

```
In[5]: from sklearn.svm import SVC # "Support vector classifier"
       model = SVC(kernel='linear', C=1E10)
       model.fit(X, y)

Out[5]: SVC(C=10000000000.0, cache_size=200, class_weight=None, coef0=0.0,
          decision_function_shape=None, degree=3, gamma='auto', kernel='linear',
          max_iter=-1, probability=False, random_state=None, shrinking=True,
          tol=0.001, verbose=False)
```

To better visualize what’s happening here, let’s create a quick convenience function that will plot SVM decision boundaries for us (Figure 5-56):

```
In[6]: def plot_svc_decision_function(model, ax=None, plot_support=True):
       """Plot the decision function for a two-dimensional SVC"""
       if ax is None:
           ax = plt.gca()
       xlim = ax.get_xlim()
       ylim = ax.get_ylim()

       # create grid to evaluate model
       x = np.linspace(xlim[0], xlim[1], 30)
```

```

y = np.linspace(ylim[0], ylim[1], 30)
Y, X = np.meshgrid(y, x)
xy = np.vstack([X.ravel(), Y.ravel()]).T
P = model.decision_function(xy).reshape(X.shape)

# plot decision boundary and margins
ax.contour(X, Y, P, colors='k',
           levels=[-1, 0, 1], alpha=0.5,
           linestyles=['--', '-', '--'])

# plot support vectors
if plot_support:
    ax.scatter(model.support_vectors_[0], 0),
               model.support_vectors_[1], 1,
               s=300, linewidth=1, facecolors='none');
ax.set_xlim(xlim)
ax.set_ylim(ylim)

In[7]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
       plot_svc_decision_function(model);

```

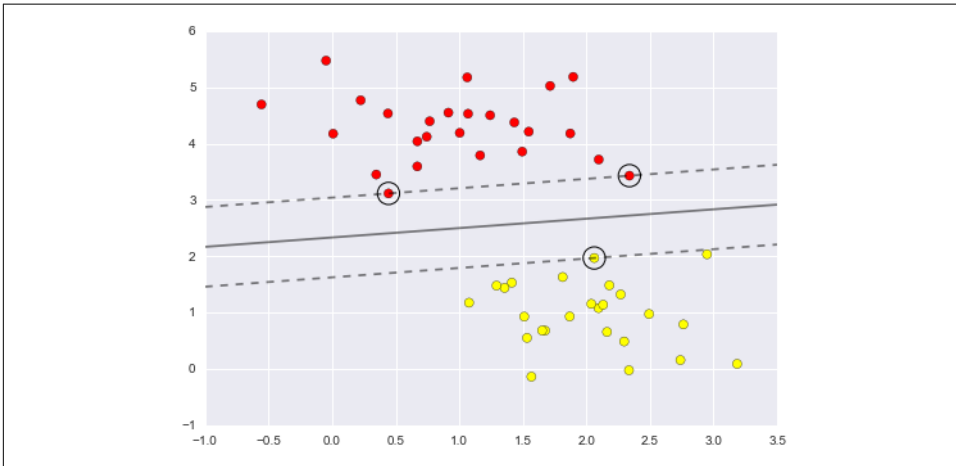


Figure 5-56. A support vector machine classifier fit to the data, with margins (dashed lines) and support vectors (circles) shown

This is the dividing line that maximizes the margin between the two sets of points. Notice that a few of the training points just touch the margin; they are indicated by the black circles in [Figure 5-56](#). These points are the pivotal elements of this fit, and are known as the *support vectors*, and give the algorithm its name. In Scikit-Learn, the identity of these points is stored in the `support_vectors_` attribute of the classifier:

```

In[8]: model.support_vectors_
Out[8]: array([[ 0.44359863,  3.11530945],
               [ 2.33812285,  3.43116792],
               [ 2.06156753,  1.96918596]])

```

A key to this classifier's success is that for the fit, only the position of the support vectors matters; any points further from the margin that are on the correct side do not modify the fit! Technically, this is because these points do not contribute to the loss function used to fit the model, so their position and number do not matter so long as they do not cross the margin.

We can see this, for example, if we plot the model learned from the first 60 points and first 120 points of this dataset (Figure 5-57):

```
In[9]: def plot_svm(N=10, ax=None):
        X, y = make_blobs(n_samples=200, centers=2,
                           random_state=0, cluster_std=0.60)

        X = X[:N]
        y = y[:N]
        model = SVC(kernel='linear', C=1E10)
        model.fit(X, y)

        ax = ax or plt.gca()
        ax.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
        ax.set_xlim(-1, 4)
        ax.set_ylim(-1, 6)
        plot_svc_decision_function(model, ax)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
for axi, N in zip(ax, [60, 120]):
    plot_svm(N, axi)
    axi.set_title('N = {0}'.format(N))
```

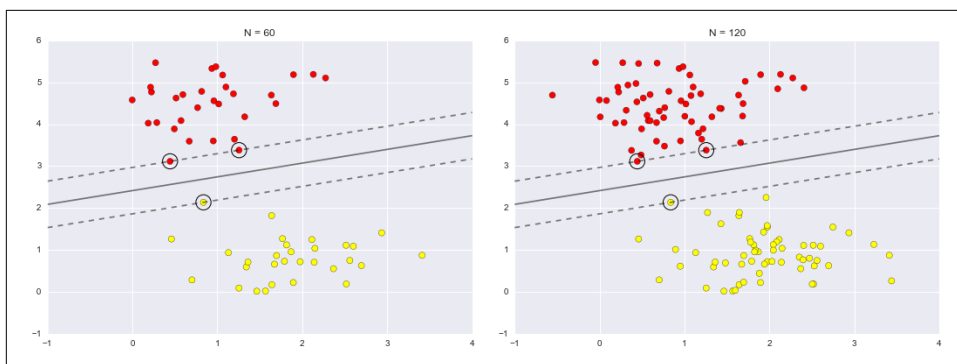


Figure 5-57. The influence of new training points on the SVM model

In the left panel, we see the model and the support vectors for 60 training points. In the right panel, we have doubled the number of training points, but the model has not changed: the three support vectors from the left panel are still the support vectors from the right panel. This insensitivity to the exact behavior of distant points is one of the strengths of the SVM model.

If you are running this notebook live, you can use IPython's interactive widgets to view this feature of the SVM model interactively (Figure 5-58):

```
In[10]: from ipywidgets import interact, fixed
        interact(plot_svm, N=[10, 200], ax=fixed(None));
```

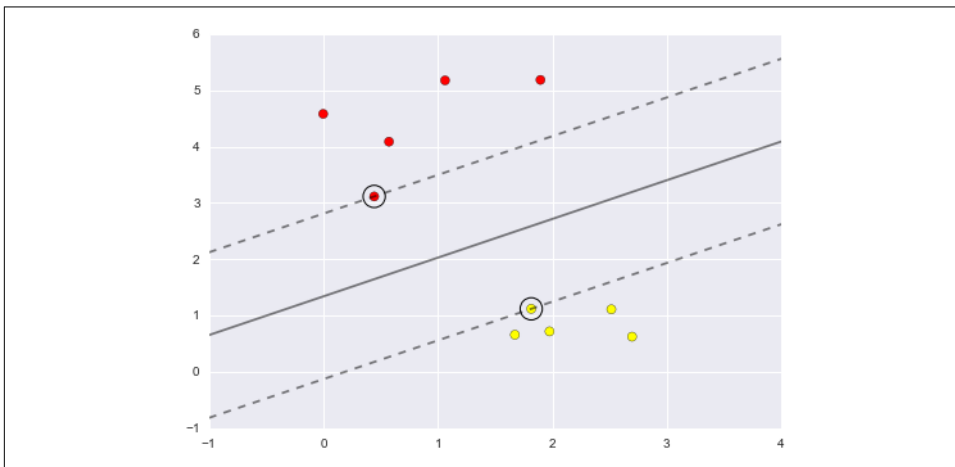


Figure 5-58. The first frame of the interactive SVM visualization (see the [online appendix](#) for the full version)

Beyond linear boundaries: Kernel SVM

Where SVM becomes extremely powerful is when it is combined with *kernels*. We have seen a version of kernels before, in the basis function regressions of “[In Depth: Linear Regression](#)” on page 390. There we projected our data into higher-dimensional space defined by polynomials and Gaussian basis functions, and thereby were able to fit for nonlinear relationships with a linear classifier.

In SVM models, we can use a version of the same idea. To motivate the need for kernels, let's look at some data that is not linearly separable (Figure 5-59):

```
In[11]: from sklearn.datasets.samples_generator import make_circles
        X, y = make_circles(100, factor=.1, noise=.1)

        clf = SVC(kernel='linear').fit(X, y)

        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
        plot_svc_decision_function(clf, plot_support=False);
```

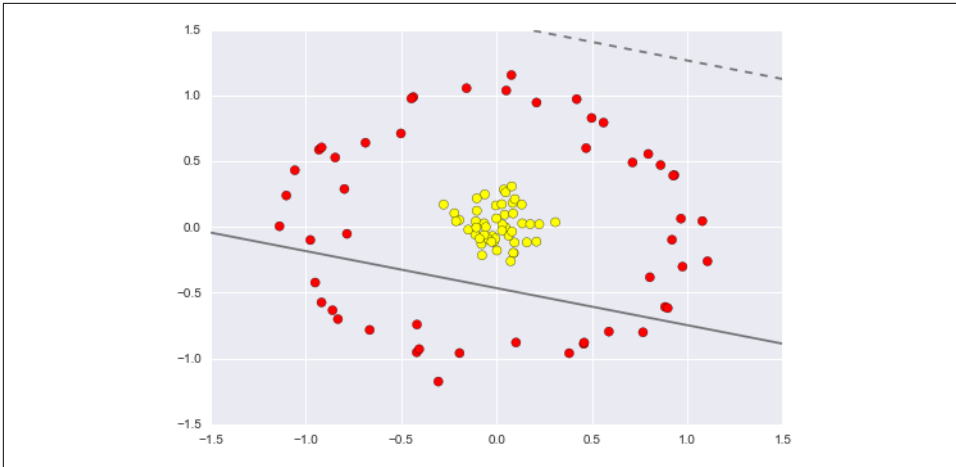


Figure 5-59. A linear classifier performs poorly for nonlinear boundaries

It is clear that no linear discrimination will *ever* be able to separate this data. But we can draw a lesson from the basis function regressions in “[In Depth: Linear Regression](#)” on page 390, and think about how we might project the data into a higher dimension such that a linear separator *would* be sufficient. For example, one simple projection we could use would be to compute a *radial basis function* centered on the middle clump:

```
In[12]: r = np.exp(-(X ** 2).sum(1))
```

We can visualize this extra data dimension using a three-dimensional plot—if you are running this notebook live, you will be able to use the sliders to rotate the plot (Figure 5-60):

```
In[13]: from mpl_toolkits import mplot3d

def plot_3D(elev=30, azim=30, X=X, y=y):
    ax = plt.subplot(projection='3d')
    ax.scatter3D(X[:, 0], X[:, 1], r, c=y, s=50, cmap='autumn')
    ax.view_init(elev=elev, azim=azim)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('r')

interact(plot_3D, elev=[-90, 90], azim=(-180, 180),
        X=fixed(X), y=fixed(y));
```

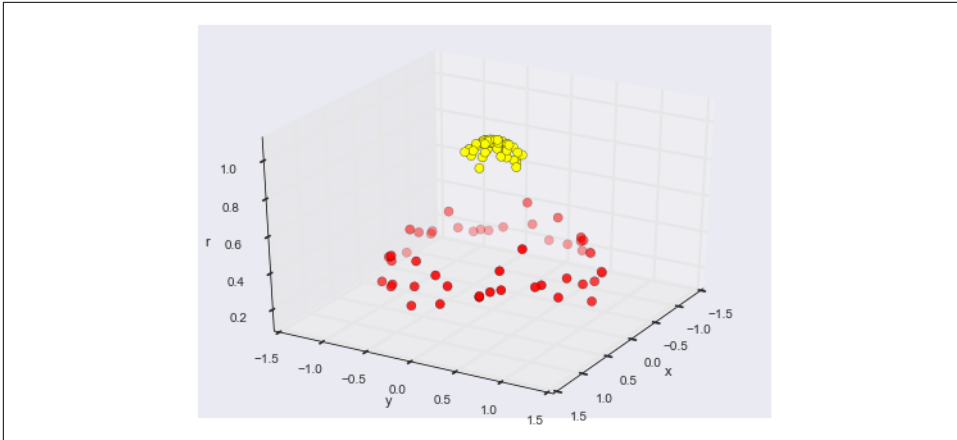


Figure 5-60. A third dimension added to the data allows for linear separation

We can see that with this additional dimension, the data becomes trivially linearly separable, by drawing a separating plane at, say, $r=0.7$.

Here we had to choose and carefully tune our projection; if we had not centered our radial basis function in the right location, we would not have seen such clean, linearly separable results. In general, the need to make such a choice is a problem: we would like to somehow automatically find the best basis functions to use.

One strategy to this end is to compute a basis function centered at *every* point in the dataset, and let the SVM algorithm sift through the results. This type of basis function transformation is known as a *kernel transformation*, as it is based on a similarity relationship (or kernel) between each pair of points.

A potential problem with this strategy—projecting N points into N dimensions—is that it might become very computationally intensive as N grows large. However, because of a neat little procedure known as the *kernel trick*, a fit on kernel-transformed data can be done implicitly—that is, without ever building the full N -dimensional representation of the kernel projection! This kernel trick is built into the SVM, and is one of the reasons the method is so powerful.

In Scikit-Learn, we can apply kernelized SVM simply by changing our linear kernel to an RBF (radial basis function) kernel, using the kernel model hyperparameter (Figure 5-61):

```
In[14]: clf = SVC(kernel='rbf', C=1E6)
        clf.fit(X, y)

Out[14]: SVC(C=1000000.0, cache_size=200, class_weight=None, coef0=0.0,
            decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',
            max_iter=-1, probability=False, random_state=None, shrinking=True,
            tol=0.001, verbose=False)
```

```
In[15]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
        plot_svc_decision_function(clf)
        plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
                    s=300, lw=1, facecolors='none');
```

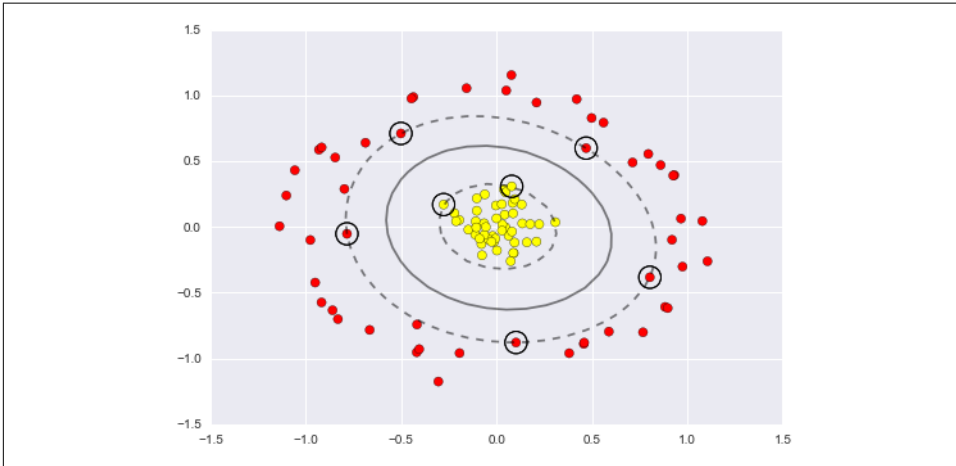


Figure 5-61. Kernel SVM fit to the data

Using this kernelized support vector machine, we learn a suitable nonlinear decision boundary. This kernel transformation strategy is used often in machine learning to turn fast linear methods into fast nonlinear methods, especially for models in which the kernel trick can be used.

Tuning the SVM: Softening margins

Our discussion so far has centered on very clean datasets, in which a perfect decision boundary exists. But what if your data has some amount of overlap? For example, you may have data like this (Figure 5-62):

```
In[16]: X, y = make_blobs(n_samples=100, centers=2,
                          random_state=0, cluster_std=1.2)
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```

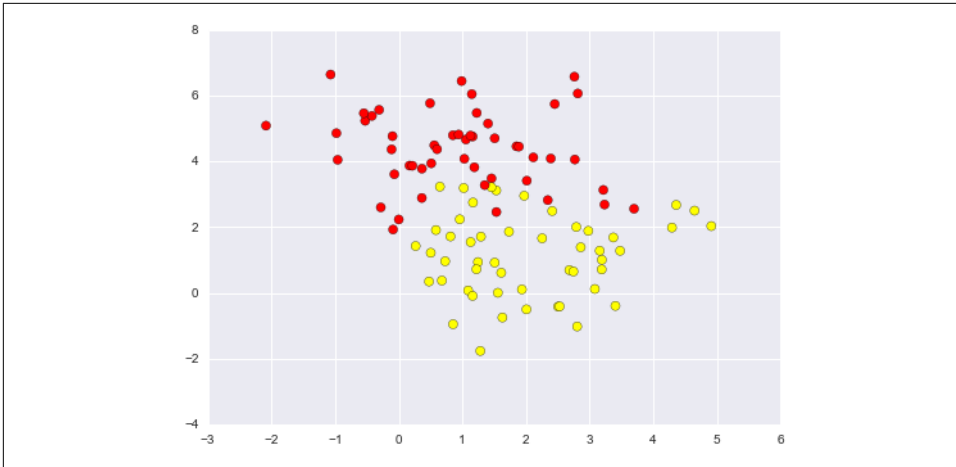



Figure 5-62. Data with some level of overlap

To handle this case, the SVM implementation has a bit of a fudge-factor that “softens” the margin; that is, it allows some of the points to creep into the margin if that allows a better fit. The hardness of the margin is controlled by a tuning parameter, most often known as C . For very large C , the margin is hard, and points cannot lie in it. For smaller C , the margin is softer, and can grow to encompass some points.

The plot shown in Figure 5-63 gives a visual picture of how a changing C parameter affects the final fit, via the softening of the margin:

```
In[17]: X, y = make_blobs(n_samples=100, centers=2,
                           random_state=0, cluster_std=0.8)

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

for axi, C in zip(ax, [10.0, 0.1]):
    model = SVC(kernel='linear', C=C).fit(X, y)
    axi.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
    plot_svc_decision_function(model, axi)
    axi.scatter(model.support_vectors_[:, 0],
                model.support_vectors_[:, 1],
                s=300, lw=1, facecolors='none');
    axi.set_title('C = {0:.1f}'.format(C), size=14)
```

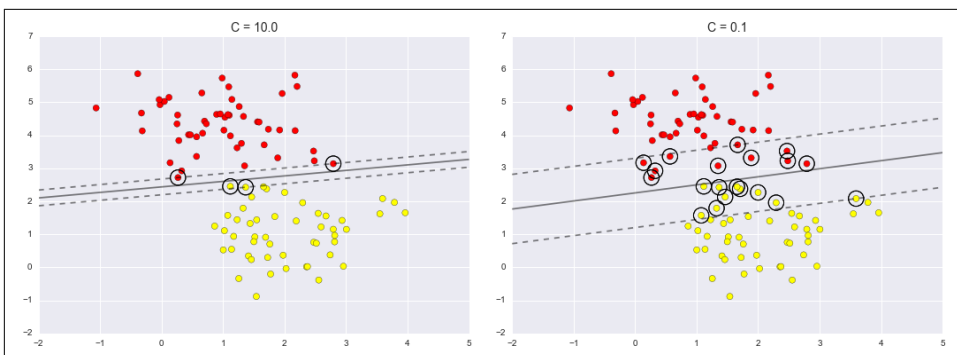


Figure 5-63. The effect of the C parameter on the support vector fit

The optimal value of the C parameter will depend on your dataset, and should be tuned via cross-validation or a similar procedure (refer back to “Hyperparameters and Model Validation” on page 359 for further information).

Example: Face Recognition

As an example of support vector machines in action, let’s take a look at the facial recognition problem. We will use the Labeled Faces in the Wild dataset, which consists of several thousand collated photos of various public figures. A fetcher for the dataset is built into Scikit-Learn:

```
In[18]: 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)
```

Let’s plot a few of these faces to see what we’re working with (Figure 5-64):

```
In[19]: 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]])
```