4. Normalize the histograms in each cell by comparing to the block of neighboring cells. This further suppresses the effect of illumination across the image.
5. Construct a one-dimensional feature vector from the information in each cell.

A fast HOG extractor is built into the Scikit-Image project, and we can try it out relatively quickly and visualize the oriented gradients within each cell (Figure 5-149):

```
In[2]: from skimage import data, color, feature
       import skimage.data

       image = color.rgb2gray(data.chelsea())
       hog_vec, hog_vis = feature.hog(image, visualise=True)

       fig, ax = plt.subplots(1, 2, figsize=(12, 6),
                              subplot_kw=dict(xticks=[], yticks=[]))
       ax[0].imshow(image, cmap='gray')
       ax[0].set_title('input image')

       ax[1].imshow(hog_vis)
       ax[1].set_title('visualization of HOG features');
```

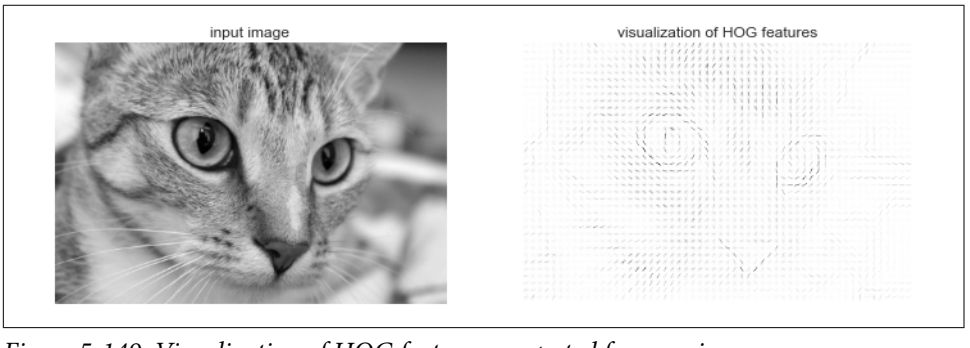

Figure 5-149. Visualization of HOG features computed from an image

HOG in Action: A Simple Face Detector

Using these HOG features, we can build up a simple facial detection algorithm with any Scikit-Learn estimator; here we will use a linear support vector machine (refer back to "In-Depth: Support Vector Machines" on page 405 if you need a refresher on this). The steps are as follows:

1. Obtain a set of image thumbnails of faces to constitute "positive" training samples.
2. Obtain a set of image thumbnails of nonfaces to constitute "negative" training samples.
3. Extract HOG features from these training samples.

4. Train a linear SVM classifier on these samples.
5. For an "unknown" image, pass a sliding window across the image, using the model to evaluate whether that window contains a face or not.
6. If detections overlap, combine them into a single window.

Let's go through these steps and try it out:

1. Obtain a set of positive training samples.

   Let's start by finding some positive training samples that show a variety of faces. We have one easy set of data to work with—the Labeled Faces in the Wild dataset, which can be downloaded by Scikit-Learn:

   ```
   In[3]: from sklearn.datasets import fetch_lfw_people
          faces = fetch_lfw_people()
          positive_patches = faces.images
          positive_patches.shape
   ```

   ```
   Out[3]: (13233, 62, 47)
   ```

   This gives us a sample of 13,000 face images to use for training.

2. Obtain a set of negative training samples.

   Next we need a set of similarly sized thumbnails that *do not* have a face in them. One way to do this is to take any corpus of input images, and extract thumbnails from them at a variety of scales. Here we can use some of the images shipped with Scikit-Image, along with Scikit-Learn's `PatchExtractor`:

   ```
   In[4]: from skimage import data, transform

          imgs_to_use = ['camera', 'text', 'coins', 'moon',
                         'page', 'clock', 'immunohistochemistry',
                         'chelsea', 'coffee', 'hubble_deep_field']
          images = [color.rgb2gray(getattr(data, name)())
                    for name in imgs_to_use]
   ```

   ```
   In[5]:
   from sklearn.feature_extraction.image import PatchExtractor

   def extract_patches(img, N, scale=1.0,
                       patch_size=positive_patches[0].shape):
       extracted_patch_size = \
       tuple((scale * np.array(patch_size)).astype(int))
       extractor = PatchExtractor(patch_size=extracted_patch_size,
                                  max_patches=N, random_state=0)
       patches = extractor.transform(img[np.newaxis])
       if scale != 1:
           patches = np.array([transform.resize(patch, patch_size)
                               for patch in patches])
       return patches
   ```

```
negative_patches = np.vstack([extract_patches(im, 1000, scale)
                              for im in images for scale in [0.5, 1.0, 2.0]])
negative_patches.shape
```

```
Out[5]: (30000, 62, 47)
```

We now have 30,000 suitable image patches that do not contain faces. Let's take a look at a few of them to get an idea of what they look like (Figure 5-150):

```
In[6]: fig, ax = plt.subplots(6, 10)
       for i, axi in enumerate(ax.flat):
           axi.imshow(negative_patches[500 * i], cmap='gray')
           axi.axis('off')
```

Figure 5-150. Negative image patches, which don't include faces

Our hope is that these would sufficiently cover the space of "nonfaces" that our algorithm is likely to see.

3. Combine sets and extract HOG features.

   Now that we have these positive samples and negative samples, we can combine them and compute HOG features. This step takes a little while, because the HOG features involve a nontrivial computation for each image:

```
In[7]: from itertools import chain
       X_train = np.array([feature.hog(im)
                           for im in chain(positive_patches,
                                           negative_patches)])
       y_train = np.zeros(X_train.shape[0])
       y_train[:positive_patches.shape[0]] = 1
```

```
In[8]: X_train.shape
```

```
Out[8]: (43233, 1215)
```

We are left with 43,000 training samples in 1,215 dimensions, and we now have our data in a form that we can feed into Scikit-Learn!

4. Train a support vector machine.

   Next we use the tools we have been exploring in this chapter to create a classifier of thumbnail patches. For such a high-dimensional binary classification task, a linear support vector machine is a good choice. We will use Scikit-Learn's `LinearSVC`, because in comparison to `SVC` it often has better scaling for large number of samples.

   First, though, let's use a simple Gaussian naive Bayes to get a quick baseline:

```
In[9]: from sklearn.naive_bayes import GaussianNB
       from sklearn.cross_validation import cross_val_score

       cross_val_score(GaussianNB(), X_train, y_train)
```

```
Out[9]: array([ 0.9408785 ,  0.8752342 ,  0.93976823])
```

   We see that on our training data, even a simple naive Bayes algorithm gets us upward of 90% accuracy. Let's try the support vector machine, with a grid search over a few choices of the C parameter:

```
In[10]: from sklearn.svm import LinearSVC
        from sklearn.grid_search import GridSearchCV
        grid = GridSearchCV(LinearSVC(), {'C': [1.0, 2.0, 4.0, 8.0]})
        grid.fit(X_train, y_train)
        grid.best_score_
```

```
Out[10]: 0.98667684407744083
```

```
In[11]: grid.best_params_
```

```
Out[11]: {'C': 4.0}
```

   Let's take the best estimator and retrain it on the full dataset:

```
In[12]: model = grid.best_estimator_
        model.fit(X_train, y_train)
```

```
Out[12]: LinearSVC(C=4.0, class_weight=None, dual=True,
              fit_intercept=True, intercept_scaling=1,
              loss='squared_hinge', max_iter=1000,
              multi_class='ovr', penalty='l2',
              random_state=None, tol=0.0001, verbose=0)
```

5. Find faces in a new image.

   Now that we have this model in place, let's grab a new image and see how the model does. We will use one portion of the astronaut image for simplicity (see discussion of this in "Caveats and Improvements" on page 512), and run a sliding window over it and evaluate each patch (Figure 5-151):

```
In[13]: test_image = skimage.data.astronaut()
        test_image = skimage.color.rgb2gray(test_image)
        test_image = skimage.transform.rescale(test_image, 0.5)
        test_image = test_image[:160, 40:180]

        plt.imshow(test_image, cmap='gray')
        plt.axis('off');
```

Figure 5-151. An image in which we will attempt to locate a face

Next, let's create a window that iterates over patches of this image, and compute HOG features for each patch:

```
In[14]: def sliding_window(img, patch_size=positive_patches[0].shape,
                           istep=2, jstep=2, scale=1.0):
            Ni, Nj = (int(scale * s) for s in patch_size)
            for i in range(0, img.shape[0] - Ni, istep):
                for j in range(0, img.shape[1] - Ni, jstep):
                    patch = img[i:i + Ni, j:j + Nj]
                    if scale != 1:
                        patch = transform.resize(patch, patch_size)
                    yield (i, j), patch

        indices, patches = zip(*sliding_window(test_image))
        patches_hog = np.array([feature.hog(patch) for patch in patches])
        patches_hog.shape
```

```
Out[14]: (1911, 1215)
```

Finally, we can take these HOG-featured patches and use our model to evaluate whether each patch contains a face:

```
In[15]: labels = model.predict(patches_hog)
        labels.sum()
```

```
Out[15]: 33.0
```

We see that out of nearly 2,000 patches, we have found 30 detections. Let's use the information we have about these patches to show where they lie on our test image, drawing them as rectangles (Figure 5-152):

```
In[16]: fig, ax = plt.subplots()
        ax.imshow(test_image, cmap='gray')
        ax.axis('off')

        Ni, Nj = positive_patches[0].shape
        indices = np.array(indices)

        for i, j in indices[labels == 1]:
            ax.add_patch(plt.Rectangle((j, i), Nj, Ni, edgecolor='red',
                                       alpha=0.3, lw=2,
                                       facecolor='none'))
```

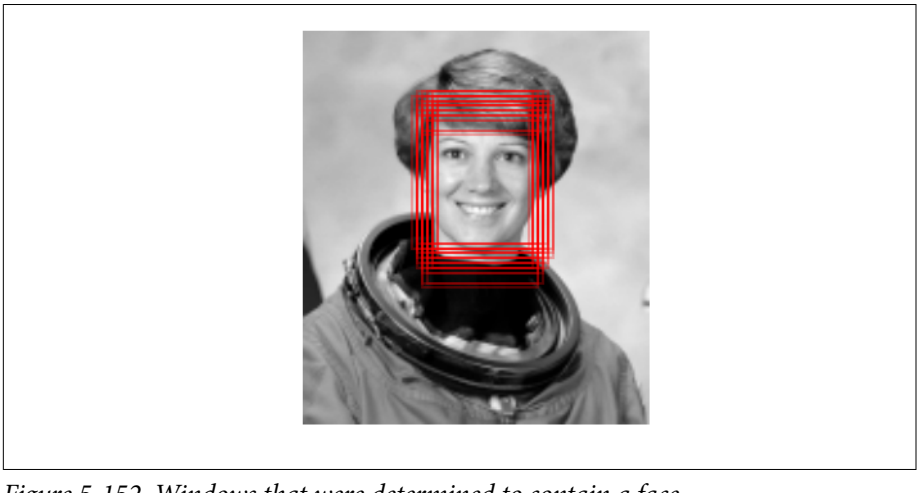

Figure 5-152. Windows that were determined to contain a face

All of the detected patches overlap and found the face in the image! Not bad for a few lines of Python.

Caveats and Improvements

If you dig a bit deeper into the preceding code and examples, you'll see that we still have a bit of work before we can claim a production-ready face detector. There are several issues with what we've done, and several improvements that could be made. In particular: