```
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'))
```

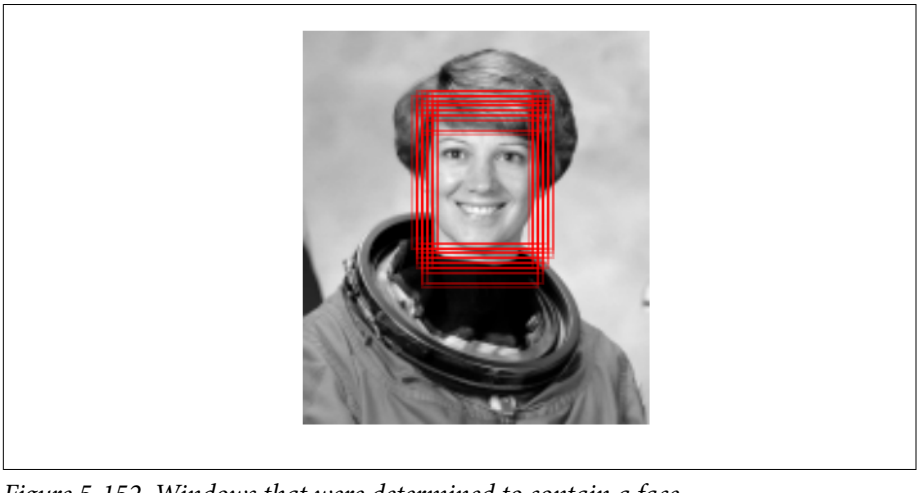

Figure 5-152. Windows that were determined to contain a face

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:

Our training set, especially for negative features, is not very complete
: The central issue is that there are many face-like textures that are not in the training set, and so our current model is very prone to false positives. You can see this if you try out the preceding algorithm on the *full* astronaut image: the current model leads to many false detections in other regions of the image.

  We might imagine addressing this by adding a wider variety of images to the negative training set, and this would probably yield some improvement. Another way to address this is to use a more directed approach, such as *hard negative mining*. In hard negative mining, we take a new set of images that our classifier has not seen, find all the patches representing false positives, and explicitly add them as negative instances in the training set before retraining the classifier.

Our current pipeline searches only at one scale
: As currently written, our algorithm will miss faces that are not approximately 62×47 pixels. We can straightforwardly address this by using sliding windows of a variety of sizes, and resizing each patch using `skimage.transform.resize` before feeding it into the model. In fact, the `sliding_window()` utility used here is already built with this in mind.

We should combine overlapped detection patches
: For a production-ready pipeline, we would prefer not to have 30 detections of the same face, but to somehow reduce overlapping groups of detections down to a single detection. This could be done via an unsupervised clustering approach (MeanShift Clustering is one good candidate for this), or via a procedural approach such as *nonmaximum suppression*, an algorithm common in machine vision.

The pipeline should be streamlined
: Once we address these issues, it would also be nice to create a more streamlined pipeline for ingesting training images and predicting sliding-window outputs. This is where Python as a data science tool really shines: with a bit of work, we could take our prototype code and package it with a well-designed object-oriented API that gives the user the ability to use this easily. I will leave this as a proverbial "exercise for the reader."

More recent advances, such as deep learning, should be considered
: Finally, I should add that HOG and other procedural feature extraction methods for images are no longer state-of-the-art techniques. Instead, many modern object detection pipelines use variants of deep neural networks. One way to think of neural networks is that they are an estimator that determines optimal feature extraction strategies from the data, rather than relying on the intuition of the user. An intro to these deep neural net methods is conceptually (and computationally!) beyond the scope of this section, although open tools like Google's

TensorFlow have recently made deep learning approaches much more accessible than they once were. As of the writing of this book, deep learning in Python is still relatively young, and so I can't yet point to any definitive resource. That said, the list of references in the following section should provide a useful place to start.

Further Machine Learning Resources

This chapter has been a quick tour of machine learning in Python, primarily using the tools within the Scikit-Learn library. As long as the chapter is, it is still too short to cover many interesting and important algorithms, approaches, and discussions. Here I want to suggest some resources for those who would like to learn more about machine learning.

Machine Learning in Python

To learn more about machine learning in Python, I'd suggest some of the following resources:

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.