7

HISTOGRAMS

So, what exactly is a histogram? A histogram represents the distribution of pixel intensities (whether color or gray-scale) in an image. It can be visualized as a graph (or plot) that gives a high-level intuition of the intensity (pixel value) distribution. We are going to assume an RGB color space in this example, so these pixel values will be in the range of 0 to 255.

When plotting the histogram, the X-axis serves as our "bins". If we construct a histogram with 256 bins, then we are effectively counting the number of times each pixel value occurs. In contrast, if we use only 2 (equally spaced) bins, then we are counting the number of times a pixel is in the range $[0, 128)$ or $[128, 255]$. The number of pixels binned to the x-axis value is then plotted on the y-axis.

By simply examining the histogram of an image, you get a general understanding regarding the contrast, brightness,

and intensity distribution.

7.1 USING OPENCV TO COMPUTE HISTOGRAMS

Now, let's start building some histograms of our own.

We will be using the `cv2.calcHist` function to build our histograms. Before we get into any code examples, let's quickly review the function:

```
cv2.calcHist(images,channels,mask,histSize,ranges)
```

1. **images:** This is the image that we want to compute a histogram for. Wrap it as a list: `[myImage]`.
2. **channels:** This is a list of indexes, where we specify the index of the channel we want to compute a histogram for. To compute a histogram of a grayscale image, the list would be `[0]`. To compute a histogram for all three red, green, and blue channels, the channels list would be `[0,1,2]`.
3. **mask:** Remember learning about masks in Chapter 6? Well, here we can supply a mask. If a mask is provided, a histogram will be computed for *masked pixels only*. If we do not have a mask or do not want to apply one, we can just provide a value of `None`.
4. **histSize:** This is the number of bins we want to use when computing a histogram. Again, this is a list, one for each channel we are computing a histogram for. The bin sizes do not all have to be the same. Here is an example of 32 bins for each channel: `[32,32,32]`.

5. **ranges:** Here we specify The range of possible pixel values. Normally, this is $[0, 256]$ for each channel, but if you are using a color space other than RGB (such as HSV), the ranges might be different.

Next up, we'll use the `cv2.calcHist` function to compute our first histogram.

7.2 GRAYSCALE HISTOGRAMS

Now that we have an understanding of the `cv2.calcHist` function, let's write some actual code.

Listing 7.1: grayscale_histogram.py

```
1 from matplotlib import pyplot as plt
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
```

This code isn't very exciting yet. All we are doing is importing the packages we will need, setting up an argument parser, and loading our image. We'll make use of the `matplotlib` package to make plotting our histograms easier.

Listing 7.2: grayscale_histogram.py

```
13 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
14 cv2.imshow("Original", image)
```

```
15
16 hist = cv2.calcHist([image], [0], None, [256], [0, 256])
17
18 plt.figure()
19 plt.title("Grayscale Histogram")
20 plt.xlabel("Bins")
21 plt.ylabel("# of Pixels")
22 plt.plot(hist)
23 plt.xlim([0, 256])
24 plt.show()
25 cv2.waitKey(0)
```

Now things are getting a little more interesting. On **Line 13**, we convert the image from the RGB colorspace to grayscale. **Line 16** computes the actual histogram. Go ahead and match the arguments of the code up with the function documentation above. We can see that our first parameter is the grayscale image. A grayscale image has only one channel, hence we have a value of [0] for `channels`. We don't have a mask, so we set the mask value to `None`. We will use 256 bins in our histogram, and the possible values range from 0 to 256.

Finally, a call to `plt.plot()` plots our grayscale histogram, the results of which can be seen in Figure 7.1.

Not bad. How do we interpret this histogram? Well, the bins (0-255) are plotted on the x-axis. And the y-axis counts the number of pixels in each bin. The majority of the pixels fall in the range of roughly 60 to 120. Looking at the right tail of the histogram, we see very few pixels in the range 200 to 255. This means that there are very few "white" pixels in the image.

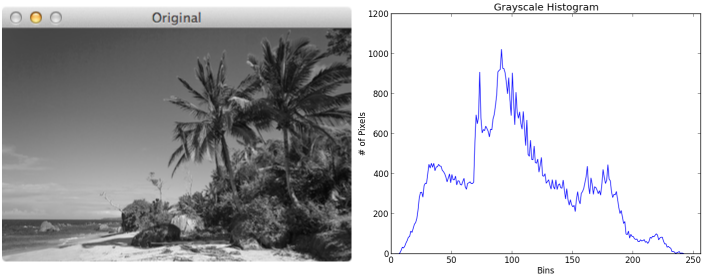

Figure 7.1: Computing a grayscale histogram of our beach image.

7.3 COLOR HISTOGRAMS

In the previous section, we explored grayscale histograms. Now let's move on to computing a histogram for each channel of the image.

Listing 7.3: color_histograms.py

```
1 from __future__ import print_function
2 from matplotlib import pyplot as plt
3 import numpy as np
4 import argparse
5 import cv2
6
7 ap = argparse.ArgumentParser()
8 ap.add_argument("-i", "--image", required = True,
9     help = "Path to the image")
10 args = vars(ap.parse_args())
11
12 image = cv2.imread(args["image"])
13 cv2.imshow("Original", image)
```

Again, we'll import the packages that we'll need, utilizing `matplotlib` once more to plot the histograms.

Let's examine some code:

Listing 7.4: color_histograms.py

```
14 chans = cv2.split(image)
15 colors = ("b", "g", "r")
16 plt.figure()
17 plt.title("'Flattened' Color Histogram")
18 plt.xlabel("Bins")
19 plt.ylabel("# of Pixels")
20
21 for (chan, color) in zip(chans, colors):
22     hist = cv2.calcHist([chan], [0], None, [256], [0, 256])
23     plt.plot(hist, color = color)
24     plt.xlim([0, 256])
```

The first thing we are going to do is split the image into its three channels: blue, green, and red. Normally, we read this is red, green, blue (RGB). However, OpenCV stores the image as a NumPy array in reverse order: BGR. This is important to note. We then initialize a tuple of strings representing the colors. We take care of all this on **Lines 14-15**.

On **Lines 16-19** we set up our PyPlot figure. We'll plot the bins on the x-axis and the number of pixels placed into each bin on the y-axis.

We then reach a `for` loop on **Line 21**, where we start looping over each of the channels in the image.

Then, for each channel, we compute a histogram on **Line 22**. The code is identical to that of computing a histogram for the grayscale image; however, we are doing it for each Red, Green, and Blue channel, allowing us to characterize

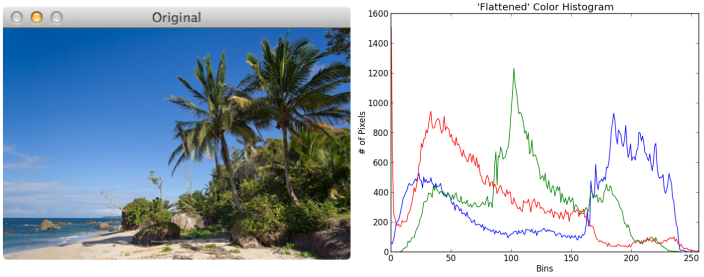

Figure 7.2: Color histograms for each Red, Green, and Blue channel of the beach image.

the distribution of pixel intensities. We add our histogram to the plot on **Line 23**.

We can examine our color histogram in Figure 7.2. We see there is a sharp peak in the green histogram around bin 100. This indicates a darker green value, from the green vegetation and trees in the beach image.

We also see a lot of blue pixels in the range 170 to 225. Considering these pixels are much lighter, we know that they are from the blue sky in our beach image. Similarly, we see a much smaller range of blue pixels in the range 25 to 50 – these pixels are much darker, and are therefore the ocean pixels in the bottom-left corner of the image.

Up until this point, we have computed a histogram for only one channel at a time. Now we move on to multi-dimensional histograms and take into consideration two

channels at a time.

I like to explain multi-dimensional histograms by using the word **AND**.

For example, we can ask a question such as, "How many pixels have a Red value of 10 **AND** a Blue value of 30?". How many pixels have a Green value of 200 **AND** a Red value of 130? By using the conjunctive **AND**, we are able to construct multi-dimensional histograms.

It's that simple. Let's check out some code to automate the process of building a 2D histogram:

Listing 7.5: color_histograms.py

```
25 fig = plt.figure()
26
27 ax = fig.add_subplot(131)
28 hist = cv2.calcHist([chans[1], chans[0]], [0, 1], None,
29     [32, 32], [0, 256, 0, 256])
30 p = ax.imshow(hist, interpolation = "nearest")
31 ax.set_title("2D Color Histogram for G and B")
32 plt.colorbar(p)
33
34 ax = fig.add_subplot(132)
35 hist = cv2.calcHist([chans[1], chans[2]], [0, 1], None,
36     [32, 32], [0, 256, 0, 256])
37 p = ax.imshow(hist, interpolation = "nearest")
38 ax.set_title("2D Color Histogram for G and R")
39 plt.colorbar(p)
40
41 ax = fig.add_subplot(133)
42 hist = cv2.calcHist([chans[0], chans[2]], [0, 1], None,
43     [32, 32], [0, 256, 0, 256])
44 p = ax.imshow(hist, interpolation = "nearest")
45 ax.set_title("2D Color Histogram for B and R")
46 plt.colorbar(p)
47
48 print("2D histogram shape: {}, with {} values".format(
```

```
49     hist.shape, hist.flatten().shape[0]))
```

Yes, this is a fair amount of code. But that's only because we are computing a 2D color histogram for each *combination* of RGB channels: Red and Green, Red and Blue, and Green and Blue.

Now that we are working with multi-dimensional histograms, we need to keep in mind the number of bins we are using. In previous examples, I've used 256 bins for demonstration purposes. However, if we used a 256 bins for each dimension in a 2D histogram, our resulting histogram would have $256 \times 256 = 65,536$ separate pixel counts. Not only is this wasteful of resources, it's not practical. Most applications use somewhere between 8 and 64 bins when computing multi-dimensional histograms. As **Lines 28 and 29** show, I am now using 32 bins instead of 256.

The most important takeaway from this code can be seen by inspecting the first arguments to the `cv2.calcHist` function. Here we see that we are passing in a list of two channels: the Green and Blue channels. And that's all there is to it.

So, how is a 2D histogram stored in OpenCV? It's actually a 2D NumPy array. Since I used 32 bins for each channel, I now have a 32×32 histogram.

How do we visualize a 2D histogram? Let's take a look at Figure 7.3 where we see three graphs. The first is a 2D color histogram for the Green and Blue channels, the second for Green and Red, and the third for Blue and Red. Shades of blue represent low pixel counts, whereas shades

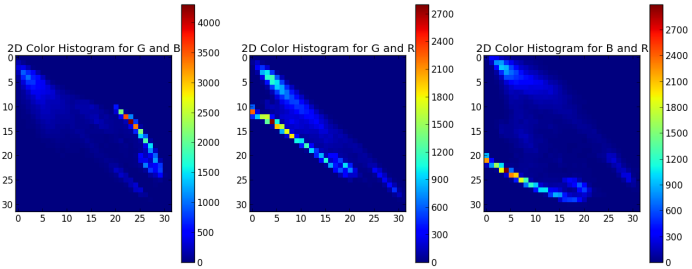

Figure 7.3: Computing 2D color histograms for each combination of Red, Green, and Blue channels.

of red represent large pixel counts (i.e., peaks in the 2D histogram). We tend to see many peaks in the Green and Blue histogram, where $x = 22$ and $y = 12$. This corresponds to the green pixels of the vegetation and trees and the blue of the sky and ocean.

Using a 2D histogram takes into account two channels at a time. But what if we wanted to account for all *three* RGB channels? You guessed it. We're now going to build a 3D histogram.

Listing 7.6: color_histograms.py

```
50 hist = cv2.calcHist([image], [0, 1, 2],
51     None, [8, 8, 8], [0, 256, 0, 256, 0, 256])
52 print("3D histogram shape: {}, with {} values".format(
53     hist.shape, hist.flatten().shape[0]))
54
55 plt.show()
```

The code here is very simple – it's just an extension of the code above. We are now computing an $8 \times 8 \times 8$ histogram for each of the RGB channels. We can't visualize this histogram, but we can see that the shape is indeed `(8,8,8)` with 512 values.

7.4 HISTOGRAM EQUALIZATION

Histogram equalization improves the contrast of an image by "stretching" the distribution of pixels. Consider a histogram with a large peak at the center of it. Applying histogram equalization will stretch the peak out towards the corner of the image, thus improving the global contrast of the image. Histogram equalization is applied to grayscale images.

This method is useful when an image contains foregrounds and backgrounds that are both dark or both light. It tends to produce unrealistic effects in photographs; however, it is normally useful when enhancing the contrast of medical or satellite images.

Regardless whether you are applying histogram equalization to a photograph, a satellite image, or an X-ray, we first need to see some code so we can understand what is going on:

Listing 7.7: equalize.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
```

Figure 7.4: *Left:* The original beach image. *Right:* The beach image after applying histogram equalization.

```
5  ap = argparse.ArgumentParser()
6  ap.add_argument("-i", "--image", required = True,
7      help = "Path to the image")
8  args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12
13 eq = cv2.equalizeHist(image)
14
15 cv2.imshow("Histogram Equalization", np.hstack([image, eq]))
16 cv2.waitKey(0)
```

Lines 1-10 handle our standard practice of importing packages, parsing arguments, and loading our image. We then convert our image to grayscale on **Line 11**.

Performing histogram equalization is done using just a single function: `cv2.equalizeHist`, which accepts a single parameter, the grayscale image we want to perform histogram equalization on. The last couple lines of code display our histogram equalized image.

The result of applying histogram equalization can be seen in Figure 7.4. On the *left*, we have our original beach image. Then, on the *right*, we have our histogram-equalized beach image. Notice how the contrast of the image has been radically changed and now spans the entire range of $[0, 255]$.

7.5 HISTOGRAMS AND MASKS

In Chapter 6, Section 6.4, I mentioned that masks can be used to focus on specific regions of an image that interest us. We are now going to construct a mask and compute color histograms for the masked region only.

First, we need to define a convenience function to save us from writing repetitive lines of code:

Listing 7.8: histogram_with_mask.py

```
1  from matplotlib import pyplot as plt
2  import numpy as np
3  import argparse
4  import cv2
5
6  def plot_histogram(image, title, mask = None):
7      chans = cv2.split(image)
8      colors = ("b", "g", "r")
9      plt.figure()
10     plt.title(title)
11     plt.xlabel("Bins")
12     plt.ylabel("# of Pixels")
13
14     for (chan, color) in zip(chans, colors):
15         hist = cv2.calcHist([chan], [0], mask, [256], [0, 256])
16         plt.plot(hist, color = color)
17         plt.xlim([0, 256])
```

On **Lines 1-4** we import our packages; then on **Line 6** we define `plot_histogram`. This function accepts three parameters: an `image`, the `title` of our plot, and a `mask`. The `mask` defaults to `None` if we do not have a mask for the image.

The body of our `plot_histogram` function simply computes a histogram for each channel in the image and plots it, just as in previous examples in this chapter.

Now that we have a function to help us easily plot histograms, let's move into the bulk of our code:

Listing 7.9: histogram_with_mask.py

```
18 ap = argparse.ArgumentParser()
19 ap.add_argument("-i", "--image", required = True,
20     help = "Path to the image")
21 args = vars(ap.parse_args())
22
23 image = cv2.imread(args["image"])
24 cv2.imshow("Original", image)
25 plot_histogram(image, "Histogram for Original Image")
```

Lines 18-21 parse our command line arguments. Then we load our beach image on **Line 23** and plot a histogram for each channel of the beach image on **Line 25**. The plot for our image can be seen in Figure 7.5. We will refer to this histogram again once we compute a histogram for the masked region.

Listing 7.10: histogram_with_mask.py

```
26 mask = np.zeros(image.shape[:2], dtype = "uint8")
27 cv2.rectangle(mask, (15, 15), (130, 100), 255, -1)
28 cv2.imshow("Mask", mask)
29
30 masked = cv2.bitwise_and(image, image, mask = mask)
31 cv2.imshow("Applying the Mask", masked)
```

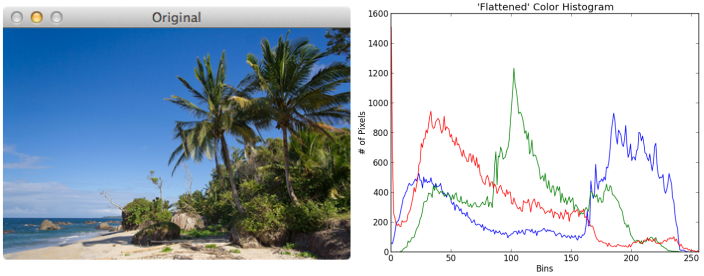

Figure 7.5: *Left:* The original beach image. *Right:* Color histograms for the red, green, and blue channels. Compare these histograms to the histograms of the masked region of blue sky in Figure 7.7.

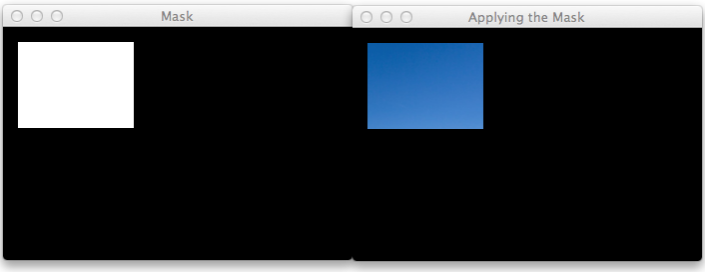

Figure 7.6: *Left:* Our rectangular mask. *Right:* Applying our mask to the beach image using a bitwise AND. Now we see only the blue sky – the rest of the image is ignored.

Now we are ready to construct a mask for the image. We define our `mask` as a NumPy array, with the same width and height as our beach image on **Line 26**. We then draw a white rectangle starting from point $(15, 15)$ to point $(130, 100)$ on **Line 27**. This rectangle will serve as our mask – only pixels in our original image belonging to the masked region will be considered in the histogram computation.

To visualize our mask, we apply a bitwise AND to the beach image (**Line 30**), the results of which can be seen in Figure 7.6. Notice how the image on the *left* is simply a white rectangle, but when we apply our mask to the beach image, we only see the blue sky (*right*).

Listing 7.11: histogram_with_mask.py

```
32 plot_histogram(image, "Histogram for Masked Image", mask = mask)
```

```
33
34 plt.show()
```

Finally, we compute a histogram for our masked image using our `plot_histogram` function and show our results (**Lines 32-34**).

We can see our masked histogram in Figure 7.7. Most red pixels fall in the range $[0, 80]$, indicating that red pixels contribute very little to our image. This makes sense, since our sky is blue. Green pixels are then present, but again, are towards the darker end of the RGB spectrum. Finally, our blue pixels fall in the brighter range and are obviously our blue sky.

Most importantly, compare our masked color histograms in Figure 7.5 to the unmasked color histograms in Figure 7.7 above. Notice how dramatically different the color histograms are. By utilizing masks, we are able to apply our computation only to the specific regions of the image that interest us – in this example, we simply wanted to examine the distribution of the blue sky.

In this chapter, you have learned all about histograms. Histograms are simple, but are used extensively in image processing and computer vision. Make sure you have a good grasp of histograms; you'll certainly be using them in the future!

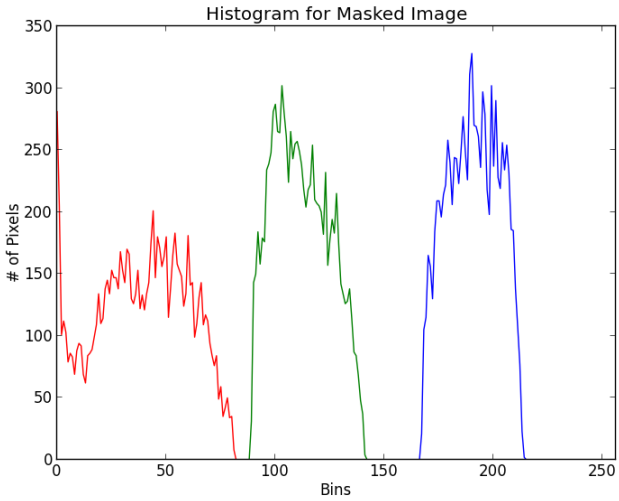

Figure 7.7: The resulting histogram of the masked image in Figure 7.6. Red contributes little to our image and is towards the darker end of the spectrum. Some lighter green values are present, and many light blue colors correspond to the sky in the image.

7

PLANT CLASSIFICATION

Stale coffee. Again.

Just think, thought Charles, *The New York Museum of Natural History*, one of the most prestigious museums in the world, and all they had to offer for their curators is stale, old coffee.

With a flick of his wrist and a sigh of dissatisfaction, Charles shoved the coffee pot back into its holder and continued down the hall to his office.

Charles had been with the museum for ten years now and quickly ascended within the ranks. And he had no doubt that his college minor in computer science was certainly a huge aid in his success.

While his more senior colleagues were struggling with Excel spreadsheets and Word documents, Charles was busy writing Python scripts and maintaining IPython Notebooks to manage and catalog his research.

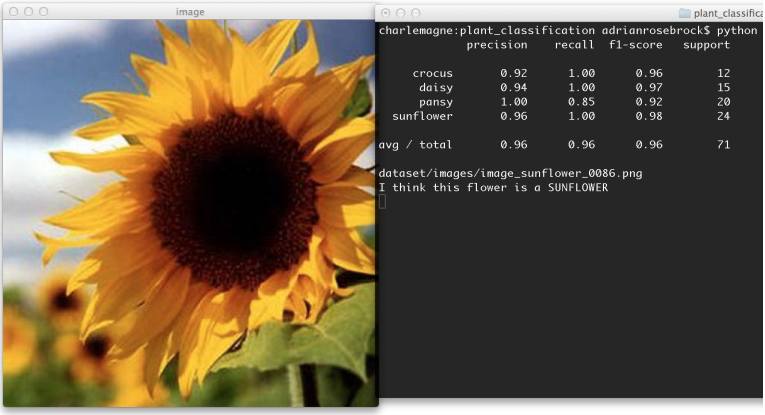

Figure 7.1: Charles' goal is to classify plant species using color histograms and machine learning. This model is able to correctly classify the image on the left as a sunflower.

His latest line of work involves the automatic classification of flower species.

Normally, identifying the species of a flower requires the eye of a trained botanist, where subtle details in the flower petals or the stem can indicate a drastically different species of flower. This type of flower classification is very time consuming and tedious.

And from the museum's point of view, it's also very expensive. Charles' time isn't cheap – and the museum was

paying him dearly for it.

But truth be told, Charles admitted to himself, he would rather be performing research, working on his manuscripts and publishing his results in the latest issue of *Museum*, although he heard Margo, the new chief editor, was a real stickler.

His heart was in his love of academia and research – not the dull task of classifying flower species day in and day out.

That's why he decided to build a classifier to *automatically* classify the species of flowers in images. A classifier like this one would save him a bunch of time and free him up to get back to his precious research.

Charles has decided to quantify the flower images using a 3D RGB histogram. This histogram will be used to characterize the color of the flower petals, which is a good starting point for classifying the species of a flower.

Let's investigate his image descriptor:

Listing 7.1: rgbhistogram.py

```
1 import cv2
2
3 class RGBHistogram:
4     def __init__(self, bins):
5         self.bins = bins
6
7     def describe(self, image, mask = None):
8         hist = cv2.calcHist([image], [0, 1, 2],
9             mask, self.bins, [0, 256, 0, 256, 0, 256])
10        cv2.normalize(hist, hist)
```

```
11
12          return hist.flatten()
```

Charles starts off by importing `cv2`, the only package that he'll be needing to create his image descriptor.

He then defines the `RGBHistogram` class on **Line 3** used to encapsulate how the flower images are quantified. The `__init__` method on **Line 4** takes only a single argument – a list containing the number of bins for the 3D histogram.

Describing the image will be handled by the `describe` method on **Line 7**, which takes two parameters, an `image` that the color histogram will be built from, and an optional `mask`. If Charles supplies a `mask`, then only pixels associated with the masked region will be used in constructing the histogram. This allows him to describe *only* the petals of the image, ignoring the rest of the image (i.e., the background, which is irrelevant to the flower itself).

Note: More information on masking can be found in Chapter 6 of Practical Python and OpenCV.

Constructing the histogram is accomplished on **Line 8**. The first argument is the image that Charles wants to describe. The resulting image is normalized on **Line 10** and returned as a feature vector on **Line 12**.

Note: The `cv2.normalize` function is slightly different between OpenCV 2.4.X and OpenCV 3.0. In OpenCV 2.4.X, the `cv2.normalize` function would actually return *the normalized histogram. However, in OpenCV 3.0+, `cv2.normalize` actually normalizes the histogram* within the function *and updates the second parameter (i.e., the "output") passed in. This is a sub-*

tle, but important difference to keep in mind when working with the two OpenCV versions.

Clearly, Charles took the time to read through Chapter 7 of *Practical Python and OpenCV* to understand how histograms can be used to characterize the color of an image.

Now that the image descriptor has been defined, Charles can create the code to classify what species a given flower is:

Listing 7.2: classify.py

```
1  from __future__ import print_function
2  from pyimagesearch.rgbhistogram import RGBHistogram
3  from sklearn.preprocessing import LabelEncoder
4  from sklearn.ensemble import RandomForestClassifier
5  from sklearn.cross_validation import train_test_split
6  from sklearn.metrics import classification_report
7  import numpy as np
8  import argparse
9  import glob
10 import cv2
11
12 ap = argparse.ArgumentParser()
13 ap.add_argument("-i", "--images", required = True,
14     help = "path to the image dataset")
15 ap.add_argument("-m", "--masks", required = True,
16     help = "path to the image masks")
17 args = vars(ap.parse_args())
```

Lines 1-10 handle importing the packages to build Charles' flower classifier. First, Charles imports his `RGBHistogram` used to describe each of his images, stored in the `pyimagesearch` package for organization purposes.

The `LabelEncoder` class from the scikit-learn library is imported on **Line 3**. In order to build a machine learning classifier to distinguish between flower species, Charles first needs a way to encode "class labels" associated with each

species.

Charles wants to distinguish between sunflowers, crocus, daisies, and pansies, but in order to construct a machine learning model, these species (represented as strings) need to be converted to integers. The `LabelEncoder` class handles this process.

The actual classification model Charles uses is a `RandomForestClassifier`, imported on **Line 4**. A random forest is an ensemble learning method used for classification, consisting of multiple decision trees.

For each tree in the random forest, a bootstrapped (sampling with replacement) sample is constructed, normally consisting of 66% of the dataset. Then, a decision tree is built based on the bootstrapped sample. At each node in the tree, only a sample of predictors are taken to calculate the node split criterion. It is common to use $\sqrt{n}$ predictors, where n is the number of predictors in the feature space. This process is then repeated to train multiple trees in the forest.

Note: A detailed review of random forest classifiers is outside the scope of this case study, as machine learning methods can be heavily involved.

However, if you are new at using machine learning, random forests are a good place to start, especially in the computer vision domain where they can obtain higher accuracy with very little effort.

Again, while this doesn't apply to all *computer vision classification problems, random forests are a good starting point to obtain a baseline accuracy.*

Charles then imports the `train_test_split` function from scikit-learn. When building a machine learning model, Charles needs two sets of data: a *training* set and a *testing* (or validation) set.

The machine learning model (in this case, a random forest) is trained using the *training* data and then evaluated against the *testing* data.

It is *extremely important* to keep these two sets exclusive as it allows the model to be evaluated on data points that it has not already seen. If the model has already seen the data points, then the results are biased since it has an unfair advantage!

Finally, Charles uses NumPy for numerical processing, `argparse` to parse command line arguments, `glob` to grab the paths of images off disk, and `cv2` for his OpenCV bindings.

Wow, that certainly was a lot of packages to import!

But it looks like Charles only needs two command line arguments: `--images`, which points to the directory that contains his flower images, and `--masks`, which points to the directory that contains the masks for his flowers. These masks allow him to focus only on the parts of the flower (i.e the petals) that he wants to describe, ignoring the background and other clutter that would otherwise distort the

feature vector and insert unwanted noise.

Note: The images used in this example are a sample of the Flowers 17 dataset by Nilsback and Zisserman. I have also updated their tri-maps as binary masks to make describing the images easier. More about this dataset can be found at: http://www.robots.ox.ac.uk/ vgg/data/flowers/17/index.html.

Now that all the necessary packages have been imported and the command line arguments are parsed, let's see what Charles is up to now:

Listing 7.3: classify.py

```
19 imagePaths = sorted(glob.glob(args["images"] + "/*.png"))
20 maskPaths = sorted(glob.glob(args["masks"] + "/*.png"))
21
22 data = []
23 target = []
24
25 desc = RGBHistogram([8, 8, 8])
26
27 for (imagePath, maskPath) in zip(imagePaths, maskPaths):
28     image = cv2.imread(imagePath)
29     mask = cv2.imread(maskPath)
30     mask = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)
31
32     features = desc.describe(image, mask)
33
34     data.append(features)
35     target.append(imagePath.split("_")[-2])
```

On **Lines 19 and 20**, Charles uses `glob` to grab the paths of his images and masks, respectively. By passing in the directory containing his images, followed by the wild card `*.png`, Charles is able to quickly construct his list of image paths.

Lines 22 and 23 simply initialize Charles' data matrix and list of class labels (i.e., the species of flowers).

Line 25 then instantiates his image descriptor – a 3D RGB color histogram with 8 bins per channel. This image descriptor will yield an $8 \times 8 \times 8 = 512$-dimensional feature vector used to characterize the color of the flower.

On **Line 27** Charles starts to loop over his images and masks. He loads the image and mask off disk on **Line 28 and 29**, and then converts the mask to grayscale on **Line 30**.

Applying his 3D RGB color histogram on **Line 32** yields his feature vector, which he then stores in his `data` matrix on **Line 34**.

The species of the flower is then parsed out and the list of targets updated on **Line 35**.

Now Charles can apply his machine learning method:

Listing 7.4: classify.py

```
37 targetNames = np.unique(target)
38 le = LabelEncoder()
39 target = le.fit_transform(target)
40
41 (trainData, testData, trainTarget, testTarget) = train_test_split
       (data, target,
42     test_size = 0.3, random_state = 42)
43
44 model = RandomForestClassifier(n_estimators = 25, random_state =
       84)
45 model.fit(trainData, trainTarget)
46
47 print(classification_report(testTarget, model.predict(testData),
48     target_names = targetNames))
```

First, Charles encodes his class labels on **Lines 37-39**. The `unique` method of NumPy is used to find the unique species names, which are then fed into the `LabelEncoder`. A call to `fit_transform` "fits" the unique species names into integers, a category for each species, and then "transforms" the strings into their corresponding integer classes. The `target` variable now contains a list of integers, one for each data point, where each integer maps to a flower species name.

From there, Charles must construct his training and testing split on **Line 41**. The `train_test_split` function takes care of the heavy lifting for him. He passes in his `data` matrix and list of targets, specifying that the test dataset should be 30% of the size of the entire dataset. A pseudo random state of `42` is used so that Charles can reproduce his results in later runs.

The `RandomForestClassifier` is trained on **Line 44 and 45** using `25` decision trees in the forest. Again, a pseudo random state is explicitly used so that Charles' results are reproducible.

Line 47 then prints out the accuracy of his model using the `classification_report` function. Charles passes in the actual testing targets as the first parameter and then lets the model predict what it thinks the flower species are for the testing data. The `classification_report` function then compares the *predictions* to the *true targets* and prints an accuracy report for both the overall system and each individ-

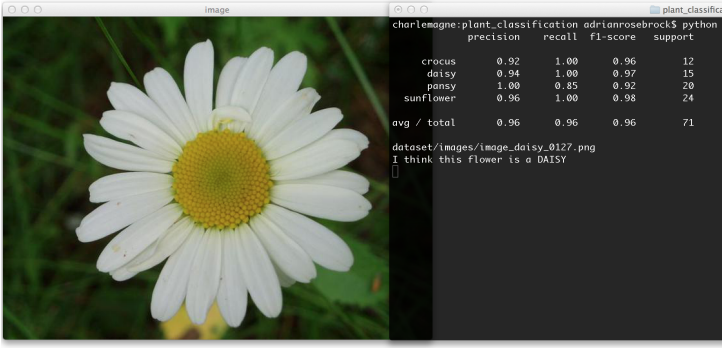

Figure 7.2: Charles has used color histograms a random forest classifier to obtain a high accuracy plant classifier.

ual class label.

The results of Charles' experiment can be seen in Figure 7.2.

His random forest classifier is able to classify a crocus correctly 100% of the time, a daisy 100% of the time, a pansy 90% of the time, and a sunflower 96% of the time.

Not bad for using just a color histogram for his features!

To investigate the classification further, Charles defines the following code:

Listing 7.5: classify.py

```
50 for i in np.random.choice(np.arange(0, len(imagePaths)), 10):
```

```
51     imagePath = imagePaths[i]
52     maskPath = maskPaths[i]
53
54     image = cv2.imread(imagePath)
55     mask = cv2.imread(maskPath)
56     mask = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)
57
58     features = desc.describe(image, mask)
59
60     flower = le.inverse_transform(model.predict([features]))[0]
61     print(imagePath)
62     print("I think this flower is a {}".format(flower.upper()))
63     cv2.imshow("image", image)
64     cv2.waitKey(0)
```

On **Line 50**, he randomly picks 10 different images to investigate, then he grabs the corresponding image and mask paths on **Lines 51 and 52**.

The `image` and `mask` are then loaded on **Lines 54 and 55** and the `mask` converted to grayscale on **Line 56**, just as in the data preparation phase.

He then extracts his feature vector on **Line 58** to characterize the color of the flower.

His random forest classifier is queried on **Line 60** to determine the species of the flower, which is then printed to console and displayed on screen on **Lines 61-64**.

Charles executes his flower classifier by issuing the following command:

Listing 7.6: classify.py

```
$ python classify.py --images dataset/images --masks dataset/
    masks
```

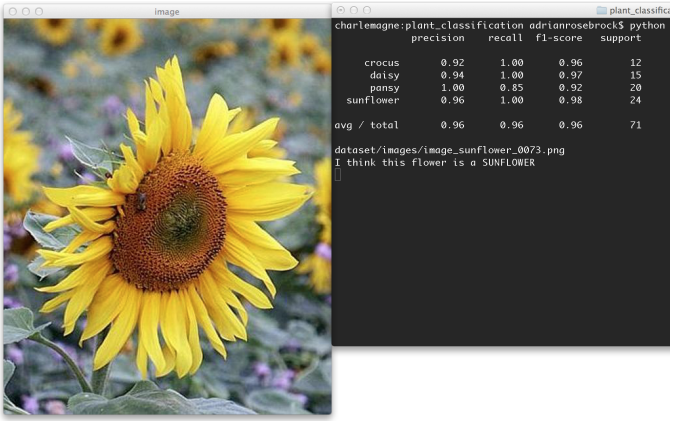

Figure 7.3: Charles' classifier correctly labels the flower as a sunflower.

First, his script trains a random forest classifier on the color histograms of the flowers and then is evaluated on a set of testing data.

Then, Figure 7.3 and Figure 7.4 display a sampling of his results. In each case, his classifier is able to correctly classify the species of the flower.

Satisfied with his work, Charles decides it's time for that cup of coffee.

And not the stale junk in the museum cafeteria either.

No, a triumphant day like this one deserves a splurge.

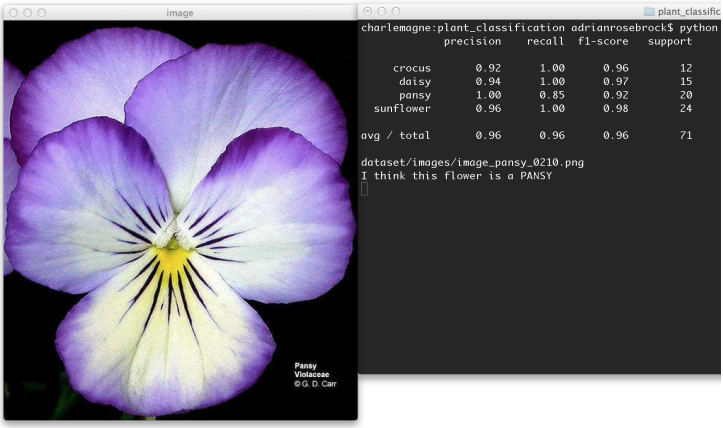

Figure 7.4: The flower is correctly labeled as a pansy.

Grabbing his coat off the back of his chair, he heads out the door of his office, dreaming about the taste of a peppermint mocha from the Starbucks just down the street.