6

HANDWRITING RECOGNITION WITH HOG

"I would like to find what 'teacher' taught this child penmanship in school," Hank yelled from his office on the third floor of the Louisiana post office. "And when I find this so-called 'teacher,' I would like to *kindly* remind them the importance of dotting your 'i's and crossing your 't's!"

His fist then came down on the solid oak table with a loud smack.

But his co-workers weren't surprised. This is what they would call a "mild" outburst from Hank.

Contracted by the Louisiana post office, Hank and his team of four developers have been tasked to apply computer vision and image processing to automatically recognize the zip codes on envelopes, as seen in Figure 6.1.

And thus far, it was proving to be significantly more challenging than Hank had thought, especially since his company did not bid much for the contract, and every day spent

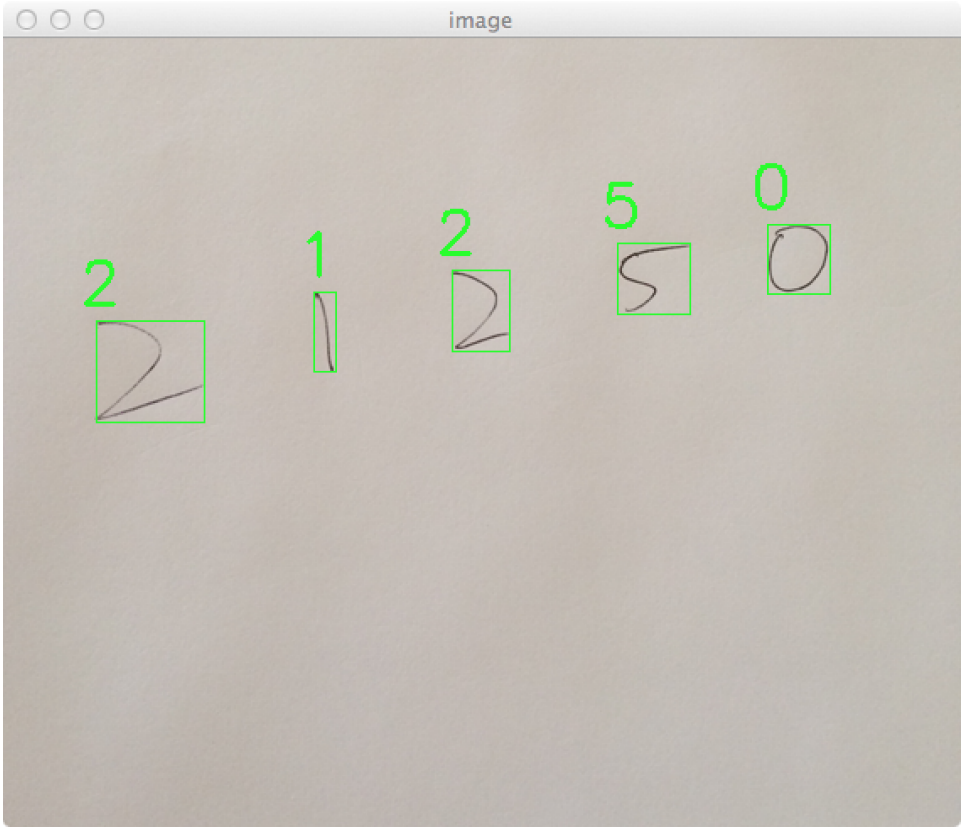

Figure 6.1: Hank's goal is to build a system that can identify handwritten digits, but he's having some trouble. Let's see if we can help him out.

working on this project only bled further into their profits.

Picking up the yellow smiley face stress ball from his desk, Hank squeezed, slowing his breathing, trying to lower his blood pressure.

His doctor warned him about getting upset like this. It wasn't good for his heart.

As a joke, Hank's co-workers had bought him a Staples Easy Button, which he kept at the corner of his desk. But all this button did was mock him. Nothing was easy. Especially recognizing the handwriting of people who clearly lacked the fundamentals of penmanship.

Hank majored in computer science back in college. He even took a few graduate level courses in machine learning before getting married to Linda, his high school sweetheart. After that, he dropped out of the masters program. Life with his new bride was more important than finishing school.

It turned out to be a good decision. They have a kid now. A house. With a white picket fence and a dog named Spot. It was the American dream.

And all of it will be ruined, the dream thrashed, morphing into an inescapable nightmare by this contract with the Louisiana post office. He just couldn't get the digit classifier to work correctly.

Taking another few seconds to calm himself, Hank thought back to his days in graduate school. He remembered a

guest lecturer who discussed an image descriptor that was very powerful for representing and classifying the content of an image.

He thought long and hard. But the name of the image descriptor just wouldn't come to him.

With a scowl, Hank looked again at the Staples Easy Button. He was tempted to pick it up and throw it through the third floor window of the post office.

But that wouldn't end well. He didn't like the irony of becoming so disgruntled in a post office that vandalism was the only method of relieving stress.

And then it came to him: *Histogram of Oriented Gradients (HOG).*

That was the name of the image descriptor!

Similar to edge orientation histograms and local invariant descriptors such as SIFT, HOG operates on the gradient magnitude of the image.

Note: Computing the gradient magnitude of an image is similar to edge detection. Be sure to see Chapter 10 of Practical Python and OpenCV *for further details on computing the gradient magnitude representation of an image.*

However, unlike SIFT, which computes a histogram over the orientation of the edges in small, localized areas of the image, HOG computes these histograms on a dense grid of uniformly-spaced cells. Furthermore, these cells can also

overlap and be contrast normalized to improve the accuracy of the descriptor.

HOG has been used successfully in many areas of computer vision and machine learning, but especially noteworthy is the detection of people in images.

In this case, Hank is going to apply the HOG image descriptor and a Linear Support Vector Machine (SVM) to learn the representation of image digits.

Luckily for Hank, the scikit-image library has already implemented the HOG descriptor, so he can rely on it when computing his feature representations.

Listing 6.1: hog.py

```
1 from skimage import feature
2
3 class HOG:
4     def __init__(self, orientations = 9, pixelsPerCell = (8, 8),
5         cellsPerBlock = (3, 3), transform = False):
6         self.orienations = orientations
7         self.pixelsPerCell = pixelsPerCell
8         self.cellsPerBlock = cellsPerBlock
9         self.transform = transform
10
11     def describe(self, image):
12         hist = feature.hog(image,
13             orientations = self.orienations,
14             pixels_per_cell = self.pixelsPerCell,
15             cells_per_block = self.cellsPerBlock,
16             transform_sqrt = self.transform)
17
18         return hist
```

Hank starts by importing the `feature` sub-package of `scikit-image` on **Line 1**. The `feature` package contains many methods to extract features from images, but perhaps

most notable is the `hog` method which Hank will utilize in his `HOG` class defined on **Line 3**.

Hank sets up `__init__` constructor on **Line 4**, requiring four parameters. The first, `orientations`, defines how many gradient orientations will be in each histogram (i.e., the number of bins). The `pixelsPerCell` parameter defines the number of pixels that will fall into each cell. When computing the HOG descriptor over an image, the image will be partitioned into multiple cells, each of size `pixelsPerCell` $\times$ `pixelsPerCell`. A histogram of gradient magnitudes will then be computed for each cell.

HOG will then normalize each of the histograms according to the number of cells that fall into each block using the `cellsPerBlock` argument.

Optionally, HOG can apply power law compression (taking the log/square-root of the input image), which can lead to better accuracy of the descriptor.

After storing the arguments for the constructor, Hank defines his `describe` method on **Line 11**, requiring only a single argument – the `image` for which the HOG descriptor should be computed.

Computing the HOG descriptor is handled by the `hog` method of the `feature` sub-package of scikit-image. Hank passes in the number of orientations, pixels per cell, cells per block, and whether or not square-root transformation should be applied to the image prior to computing the HOG descriptor.

Finally, the resulting HOG feature vector is returned to the call on **Line 18**.

Next up, Hank needs a dataset of digits that he can use to extract features from and train his machine learning model. He has decided to use a sample of the MNIST digit recognition dataset, which is a classic dataset in the computer vision and machine learning literature.

Note: I have taken the time to sample data points from the Kaggle version of the MNIST dataset. The full dataset is available here: http://www.kaggle.com/c/digit-recognizer.

The sample of the dataset consists of 5,000 data points, each with a feature vector of length 784, corresponding to the 28×28 grayscale pixel intensities of the image.

But first, he needs to define some methods to help him manipulate and prepare the dataset for feature extraction and for training his model. He'll store these dataset manipulation functions in `dataset.py`:

Listing 6.2: dataset.py

```
1 from . import imutils
2 import numpy as np
3 import mahotas
4 import cv2
5
6 def load_digits(datasetPath):
7     data = np.genfromtxt(datasetPath, delimiter = ",", dtype = "
          uint8")
8     target = data[:, 0]
9     data = data[:, 1:].reshape(data.shape[0], 28, 28)
10
11     return (data, target)
```

Hank starts by importing the packages he will need. He'll use `numpy` for numerical processing, `mahotas`, another computer vision library to aid `cv2`, and finally `imutils`, which contains convenience functions to perform common image processing tasks such as resizing and rotating images.

Note: Further information on the `imutils` *package can be found in Chapter 6 of* Practical Python and OpenCV *and on GitHub: http://pyimg.co/twcph.*

In order to load his dataset off disk, Hank defines the `load_digits` method on **Line 6**. This method requires only a single argument, the `datasetPath`, which is the path to where the MNIST sample dataset resides on disk.

From there, the `genfromtext` function of NumPy loads the dataset off disk and stores it as an unsigned 8-bit NumPy array. Remember, this dataset consists of pixel intensities of images – these pixel intensities will never be less than 0 and never greater than 255, thus Hank is able to use an 8-bit unsigned integer data type.

The first column of the `data` matrix contains Hank's targets (**Line 8**), which is the digit that the image contains. The target will fall into the range `[0, 9]`.

Likewise, all columns after the first one contain the pixel intensities of the image (**Line 9**). Again, these are the grayscale pixels of the digit image of size $M \times N$ and will always fall into the range `[0, 255]`.

Finally, the tuple of `data` and `target` are returned to the caller on **Line 11**.

Next up, Hank needs to perform some preprocessing on the digit images:

Listing 6.3: dataset.py

```
13 def deskew(image, width):
14     (h, w) = image.shape[:2]
15     moments = cv2.moments(image)
16 
17     skew = moments["mu11"] / moments["mu02"]
18     M = np.float32([
19         [1, skew, -0.5 * w * skew],
20         [0, 1, 0]])
21     image = cv2.warpAffine(image, M, (w, h),
22         flags = cv2.WARP_INVERSE_MAP | cv2.INTER_LINEAR)
23 
24     image = imutils.resize(image, width = width)
25 
26     return image
```

Everybody has a different writing style. While most of us write digits that "lean" to the left, some "lean" to the right. Some of us write digits at varying angles. These varying angles can cause confusion for the machine learning models trying to learn the representation of various digits.

In order to help fix some of the "lean" of digits, Hank defines the `deskew` method on **Line 13**. This function takes two arguments. The first is the `image` of the digit that is going to be deskewed. The second is the `width` that the image is going to be resized to.

Line 14 grabs the height and width of the `image`, then the `moments` of the image are computed on **Line 15**. These moments contain statistical information regarding the dis-

tribution of the location of the white pixels in the image.

The `skew` is computed based on the `moments` on **Line 17** and the warping matrix constructed on **Line 18**. This matrix `M` will be used to deskew the image.

The actual deskewing of the image take places on **Line 21** where a call to the `cv2.warpAffine` function is made. The first argument is the `image` that is going to be skewed, the second is the matrix `M` that defines the "direction" in which the image is going to be deskewed, and the third parameter is the resulting width and height of the deskewed image.

Finally, the `flags` parameter controls *how* the image is going to be deskewed. In this case, Hank uses linear interpolation.

The deskewed image is then resized on **Line 24** and is returned to the caller on **Line 26**.

In order to obtain a consistent representation of digits where all images are of the same width and height, with the digit placed at the center of the image, Hank then needs to define the extent of an image:

Listing 6.4: dataset.py

```
28 def center_extent(image, size):
29     (eW, eH) = size
30 
31     if image.shape[1] > image.shape[0]:
32         image = imutils.resize(image, width = eW)
33 
34     else:
35         image = imutils.resize(image, height = eH)
36 
37     extent = np.zeros((eH, eW), dtype = "uint8")
```

```
38      offsetX = (eW - image.shape[1]) // 2
39      offsetY = (eH - image.shape[0]) // 2
40      extent[offsetY:offsetY + image.shape[0], offsetX:offsetX +
            image.shape[1]] = image
41
42      CM = mahotas.center_of_mass(extent)
43      (cY, cX) = np.round(CM).astype("int32")
44      (dX, dY) = ((size[0] // 2) - cX, (size[1] // 2) - cY)
45      M = np.float32([[1, 0, dX], [0, 1, dY]])
46      extent = cv2.warpAffine(extent, M, size)
47
48      return extent
```

Hank defines his `center_extent` function on **Line 28**, which takes two arguments. The first is the deskewed `image` and the second is the output `size` of the image (i.e., the output width and height).

Line 31 checks to see if the width is greater than the height of the image. If this is the case, the image is resized based on its width.

Otherwise (**Line 34**), the height is greater than the width, so the image must be resized based on its height.

Hank notes that these are important checks to make. If these checks were not made and the image was always resized on its width, then there is a chance that the height could be larger than the width, and thus would not fit into the "extent" of the image.

Hank then allocates spaces on the extent of the image on **Line 37** using the same dimensions that were passed into the function.

The `offsetX` and `offsetY` are computed on **Lines 38 and 39**. These offsets indicate the starting *(x, y)* coordinates (in

y, x order) of where the image will be placed in the `extent`.

The actual extent is set on **Line 40** using some NumPy array slicing.

The next step is for Hank to translate the digit so it is placed at the center of the image.

Hank computes the weighted mean of the white pixels in the image using the `center_of_mass` function of the `mahotas` package. This function returns the weighted *(x, y)* coordinates of the center of the image. **Line 43** then converts these *(x, y)* coordinates to integers rather than floats.

Lines 44-46 then translates the digit so that it is placed at the center of the image. More on how to translate images can be found in Chapter 6 of *Practical Python and OpenCV*.

Finally, Hank returns the centered image to the caller on **Line 48**.

Hank is now ready to train his machine learning model to recognize digits:

Listing 6.5: train.py

```
1 from sklearn.externals import joblib
2 from sklearn.svm import LinearSVC
3 from pyimagesearch.hog import HOG
4 from pyimagesearch import dataset
5 import argparse
6
7 ap = argparse.ArgumentParser()
8 ap.add_argument("-d", "--dataset", required = True,
9     help = "path to the dataset file")
10 ap.add_argument("-m", "--model", required = True,
```

```
11     help = "path to where the model will be stored")
12 args = vars(ap.parse_args())
```

He starts by importing the packages that he'll need. He'll use the `LinearSVC` model from `scikit-learn` to train a linear Support Vector Machine (SVM). He'll also import his `HOG` image descriptor and `dataset` utility functions.

Finally, `argparse` will be used to parse command line arguments, and `joblib` will be used to dump the trained model to file.

Hank's script will require two command line arguments, the first being `--dataset`, which is the path to the MNIST sample dataset residing on disk. The second argument is `--model`, the output path for his trained `LinearSVC`.

Hank is now ready to pre-process and describe his dataset:

Listing 6.6: train.py

```
12 (digits, target) = dataset.load_digits(args["dataset"])
13 data = []
14
15 hog = HOG(orientations = 18, pixelsPerCell = (10, 10),
16     cellsPerBlock = (1, 1), transform = True)
17
18 for image in digits:
19     image = dataset.deskew(image, 20)
20     image = dataset.center_extent(image, (20, 20))
21
22     hist = hog.describe(image)
23     data.append(hist)
```

The dataset consisting of the images and targets are loaded from disk on **Line 12**. The `data` list used to hold the HOG descriptors for each image is initialized on **Line 13**.

Next, Hank instantiates his `HOG` descriptor on **Line 15**, using 18 orientations for the gradient magnitude histogram, 10 pixels for each cell, and 1 cell per block. Finally, by setting `transform = True`, Hank indicates that the square-root of the pixel intensities will be computed prior to creating the histograms.

Hank starts to loop over his digit images on **Line 18**. The image is deskewed on **Line 19** and is translated to the center of the image on **Line 20**.

The HOG feature vector is computed for the pre-processed image by calling the `describe` method on **Line 22**. Finally, the `data` matrix is updated with the HOG feature vector on **Line 23**.

Hank is now ready to train his model:

Listing 6.7: train.py

```
25 model = LinearSVC(random_state = 42)
26 model.fit(data, target)
27
28 joblib.dump(model, args["model"])
```

Hank instantiates his `LinearSVC` on **Line 25** using a pseudo random state of `42` to ensure his results are reproducible. The model is then trained using the data matrix and targets on **Line 26**.

He then dumps his model to disk for later use using `joblib` on **Line 28**.

To train his model, Hank executes the following command:

Listing 6.8: train.py

```
$ python train.py --dataset data/digits.csv --model models/svm.
    cpickle
```

Now that Hank has trained his model, he can use it to classify digits in images:

Listing 6.9: classify.py

```
1 from __future__ import print_function
2 from sklearn.externals import joblib
3 from pyimagesearch.hog import HOG
4 from pyimagesearch import dataset
5 import argparse
6 import mahotas
7 import cv2
8
9 ap = argparse.ArgumentParser()
10 ap.add_argument("-m", "--model", required = True,
11     help = "path to where the model will be stored")
12 ap.add_argument("-i", "--image", required = True,
13     help = "path to the image file")
14 args = vars(ap.parse_args())
15
16 model = joblib.load(args["model"])
17
18 hog = HOG(orientations = 18, pixelsPerCell = (10, 10),
19     cellsPerBlock = (1, 1), transform = True)
```

Hank starts `classify.py` by importing the packages that he will need (**Lines 1-7**). Just as in the training phase, Hank requires the usage of `HOG` for his image descriptor and `dataset` for his pre-processing utilities.

The `argparse` package will once again be utilized to parse command line arguments, and `joblib` will load the trained

`LinearSVC` off disk. Finally, `mahotas` and `cv2` will be used for computer vision and image processing.

Hank requires that two command line arguments be passed to `classify.py`. The first is `--model`, the path to where the `cPickle`'d model is stored. The second, `--image`, is the path to the image that contains digits that Hank wants to classify and recognize.

The trained `LinearSVC` is loaded from disk on **Lines 16**.

Then, the `HOG` descriptor is instantiated with the exact same parameters as during the training phase on **Line 18**.

Hank is now ready to find the digits in the image so that they can be classified:

Listing 6.10: classify.py

```
21 image = cv2.imread(args["image"])
22 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
23
24 blurred = cv2.GaussianBlur(gray, (5, 5), 0)
25 edged = cv2.Canny(blurred, 30, 150)
26 (_, cnts, _) = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
       cv2.CHAIN_APPROX_SIMPLE)
27
28 cnts = sorted([(c, cv2.boundingRect(c)[0]) for c in cnts], key =
       lambda x: x[1])
```

The first step is to load the query image off disk and convert it to grayscale on **Lines 21 and 22**.

From there, the image is blurred using Gaussian blurring on **Line 24**, and the Canny edge detector is applied to find edges in the image on **Line 25**.

Finally, Hank finds contours in the edged image and sorts them from left to right. Each of these contours represents a digit in an image that needs to be classified.

Hank now needs to process each of these digits:

Listing 6.11: classify.py

```
30 for (c, _) in cnts:
31     (x, y, w, h) = cv2.boundingRect(c)
32
33     if w >= 7 and h >= 20:
34         roi = gray[y:y + h, x:x + w]
35         thresh = roi.copy()
36         T = mahotas.thresholding.otsu(roi)
37         thresh[thresh > T] = 255
38         thresh = cv2.bitwise_not(thresh)
39
40         thresh = dataset.deskew(thresh, 20)
41         thresh = dataset.center_extent(thresh, (20, 20))
42
43         cv2.imshow("thresh", thresh)
```

Hank starts looping over his contours on **Line 30**.

A bounding box for each contour is computed on **Line 31** using the `cv2.boundingRect` function, which returns the starting *(x, y)* coordinates of the bounding box, followed by the width and height of the box.

Hank then checks the width and height of the bounding box on **Line 33** to ensure it is at least seven pixels wide and twenty pixels tall. If the bounding box region does not meet these dimensions, then it is considered to be too small to be a digit.

Provided that the dimension check holds, the Region of Interest (ROI) is extracted from the grayscale image using

NumPy array slices on **Line 34**.

This ROI now holds the digit that is going to be classified. But first, Hank needs to apply some pre-processing steps.

The first is to apply Otsu's thresholding method on **Lines 36-38** (covered in Chapter 9 of *Practical Python and OpenCV*) to segment the foreground (the digit) from the background (the paper the digit was written on).

Just as in the training phase, the digit is then deskewed and translated to the center of the image on **Lines 40 and 41**.

Now, Hank can classify the digit:

Listing 6.12: classify.py

```
45          hist = hog.describe(thresh)
46          digit = model.predict([hist])[0]
47          print("I think that number is: {}".format(digit))
48
49          cv2.rectangle(image, (x, y), (x + w, y + h),
50              (0, 255, 0), 1)
51          cv2.putText(image, str(digit), (x - 10, y - 10),
52              cv2.FONT_HERSHEY_SIMPLEX, 1.2, (0, 255, 0), 2)
53          cv2.imshow("image", image)
54          cv2.waitKey(0)
```

First, Hank computes the HOG feature vector of the thresholded ROI on **Line 45** by calling the `describe` method of the `HOG` descriptor.

The HOG feature vector is fed into the `LinearSVC`'s `predict` method which classifies which digit the ROI is, based on the HOG feature vector (**Line 46**).

The classified digit is then printed to Hank on **Line 47**.

But printing the digit is not enough for Hank! He also wants to display it on the original image!

In order to do this, Hank first calls the `cv2.rectangle` function to draw a green rectangle around the current digit ROI on **Line 49**.

Then, Hank uses the `cv2.putText` method to draw the digit itself on the original image on **Line 51**. The first argument to the `cv2.putText` function is the image that Hank wants to draw on, and the second argument is the string containing what he wants to draw. In this case, the digit.

Next, Hank supplies the (x, y) coordinates of where the text will be drawn. He wants this text to appear ten pixels to the left and ten pixels above the ROI bounding box.

The fourth argument is a built-in OpenCV constant used to define what font will be used to draw the text. The fifth argument is the relative size of the text, the sixth the color of the text (green), and the final argument is the thickness of the text (two pixels).

Finally, the image is displayed to Hank on **Lines 53 and 54**.

With a twinge of anxiety, Hank executes his script by issuing the following command:

Listing 6.13: classify.py

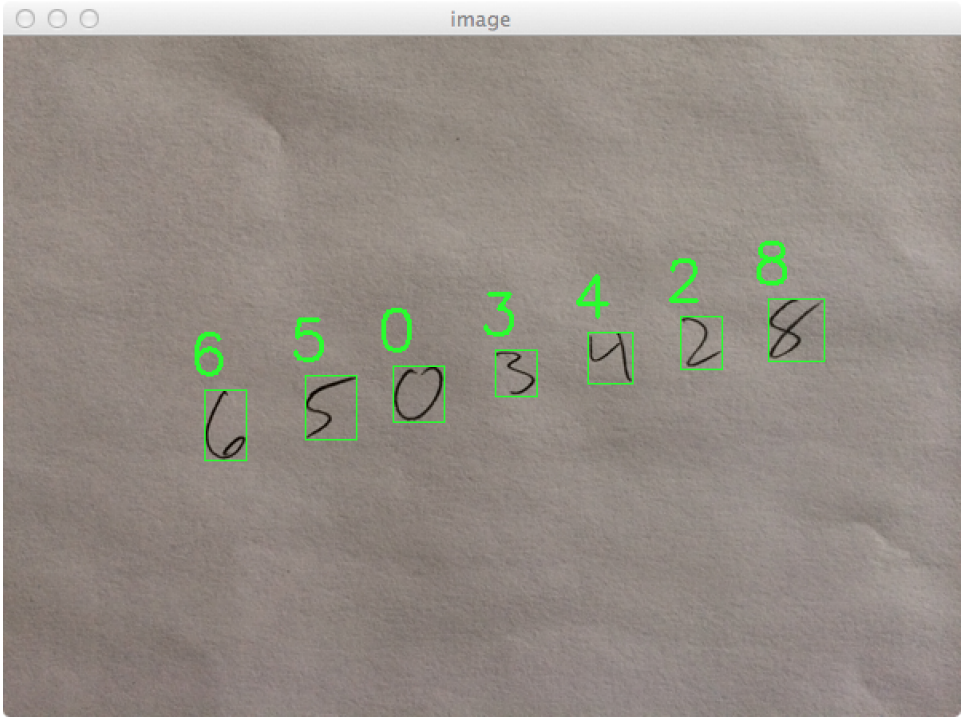

Figure 6.2: Classifying the handwritten digits of a cellphone number was a success! Hank's on the right track.

```
$ python classify.py --model models/svm.cpickle --image images/
    cellphone.png
```

The results of Hank's work can be seen in Figure 6.2, where Hank tests his code on a phone number he has written out. For each digit in the cellphone number, Hank's classifier is able to correctly label the handwritten digit!

He then moves on to another phone number, this time Apple's support line, remembering back to a few months ago when he was so angry that he threw his phone across

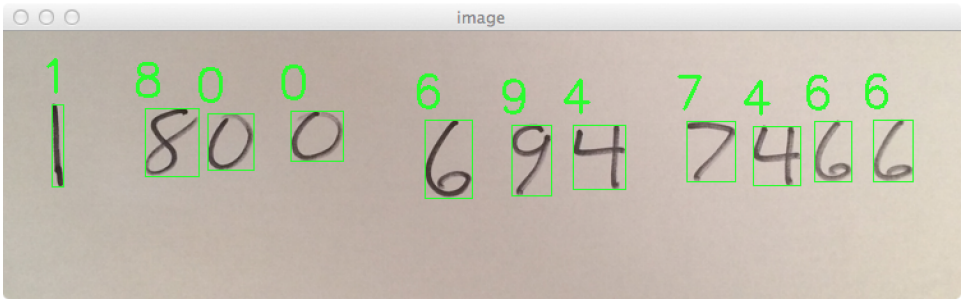

Figure 6.3: Just in case Hank needs to call Apple's technical support line, he tests his classifier on another phone number.

the room, shattering it against the wall.

Luckily for Hank (and the door), his digit classifier is once again working perfectly, as demonstrated by Figure 6.3. Again, each digit is correctly classified.

Finally, Hank tests his method against a sample of handwritten digits he gathered off Google Image Search in Figure 6.4. Without a doubt, his digit classifier is working!

With a sigh of relief, Hank realizes that the job is done. He has successfully created a computer vision pipeline to classify digits.

The contract with the post office can finally close. And maybe now he can get home at a reasonable hour to have dinner with his wife and kid.

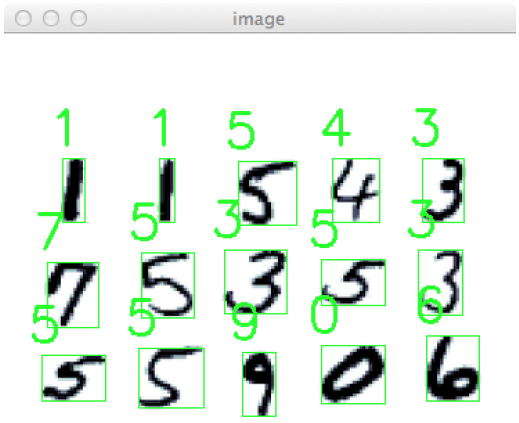

Figure 6.4: Hank evaluates his digit classifier on a sample of handwritten digits gathered from Google Image Search.

A few weeks later, Hank is less disgruntled. He's more calm at work. Even somewhat relaxed. But his co-workers still plan on buying him a new stress ball once they land their next contract.

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.

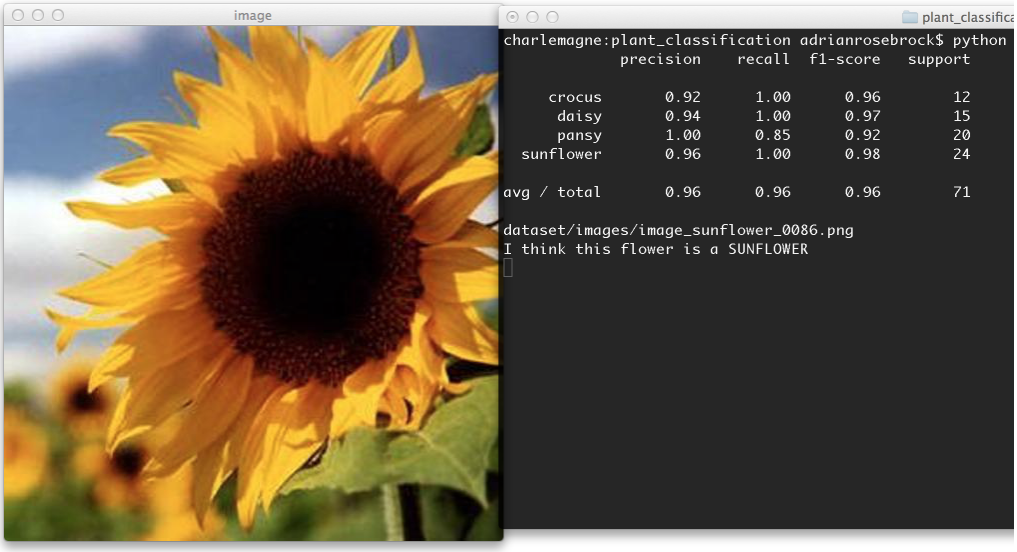

Figure 7.1: Charles' goal is to classify plant species using color histograms and machine learning. This model is able to correctly classify the image on the left as a sunflower.

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-

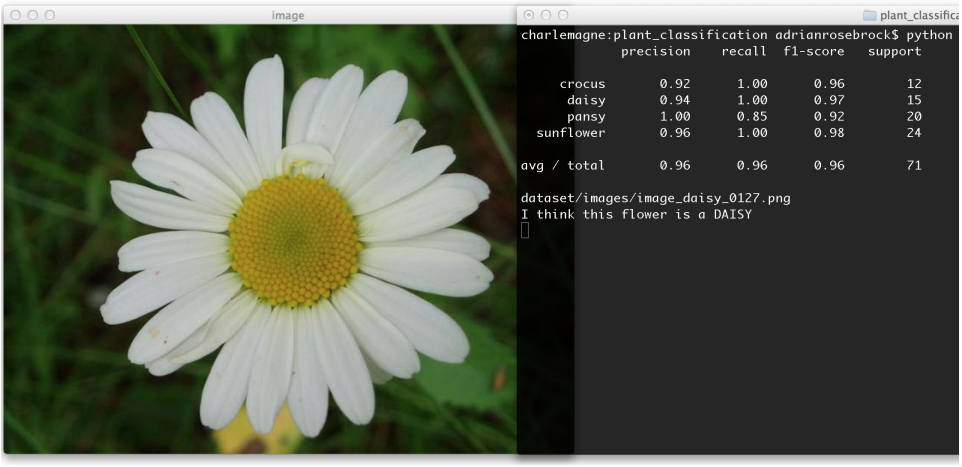

Figure 7.2: Charles has used color histograms a random forest classifier to obtain a high accuracy plant classifier.

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
```

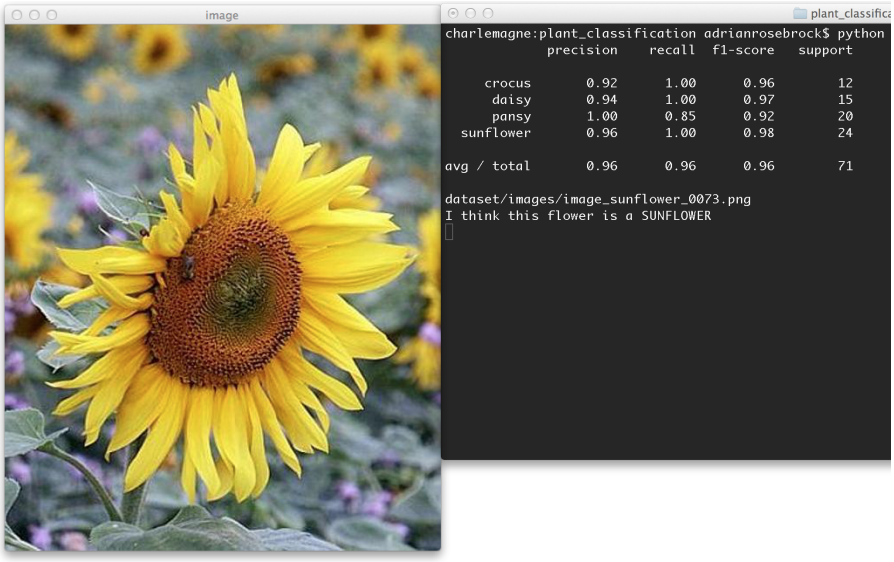

Figure 7.3: Charles' classifier correctly labels the flower as a sunflower.

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.

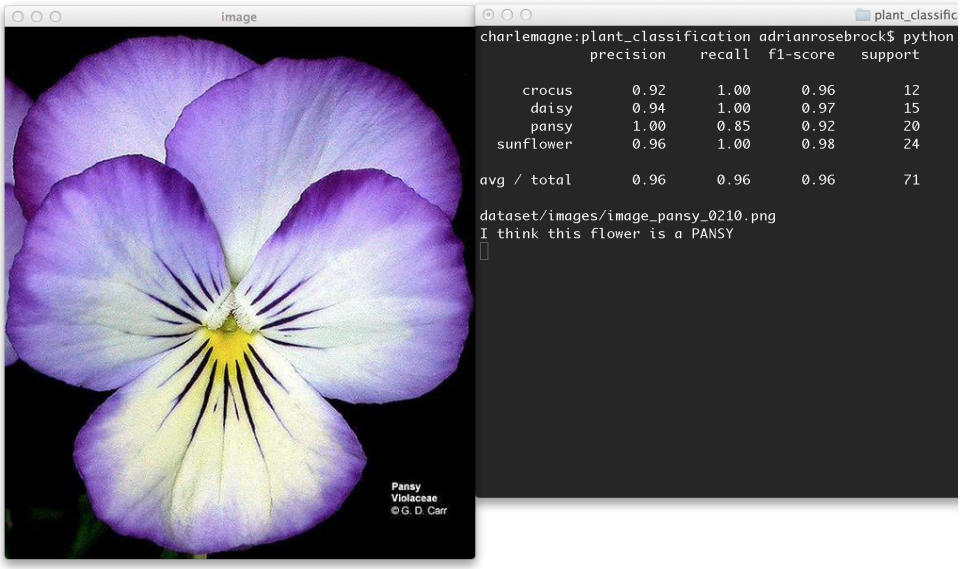

Figure 7.4: The flower is correctly labeled as a pansy.

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.