2

FACE DETECTION

"Damn," Jeremy muttered, just under his breath as he took the last sip out of his can of Mountain Dew Code Red.

"I'll never get this proof right." At this point, Jeremy took aim at the waste basket across his dorm room, fired his shot, and watched as the soda can took a glancing blow off the waste basket, tumbling to the ground.

Waste basket. That's what Jeremy thought of himself right now as he studied for his undergraduate Algorithms final exam.

It's not like big-O notation is *that* complicated.

It's just deriving the proof to the master theorem that was giving him problems.

Honestly, when was he going to use the master theorem in the real world anyway?

With a sigh, Jeremy reached over to his mini-fridge, covered with Minecraft stickers, and grabbed another Moun-

tain Dew Code Red.

Popping open his soda, Jeremy opened up a new tab in his web browser and mindlessly navigated to Facebook.

As he endlessly scrolled through the sea of status updates, memes, and friends who thought they were all too clever, a notification caught his eye.

Apparently one of his friends tagged a picture of him from the party they went to last night.

At the thought of the party, Jeremy reached up and rubbed his temple. While the memory was a bit blurry, the hangover was all too real.

Jeremy clicked on the notification and viewed the picture.

Yep.

Definitely don't remember doing a shirtless run through the quad.

Just as Jeremy was about to navigate away from the photo, he noticed Facebook had drawn a rectangle around all the faces in the image, asking him to tag his friends.

How did Facebook seem to "know" where the faces were in the images?

Clearly, they were doing some sort of face detection...

...could Jeremy build a face detection algorithm of his own?

At the thought of this, Jeremy opened up `vim` and started coding away.

A few hours later, Jeremy had a fully working face detection algorithm using Python and OpenCV. Let's take a look at Jeremy's project and see what we can learn from it.

Listing 2.1: detect_faces.py

```
1 from __future__ import print_function
2 from pyimagesearch.facedetector import FaceDetector
3 import argparse
4 import cv2
5
6 ap = argparse.ArgumentParser()
7 ap.add_argument("-f", "--face", required = True,
8     help = "path to where the face cascade resides")
9 ap.add_argument("-i", "--image", required = True,
10     help = "path to where the image file resides")
11 args = vars(ap.parse_args())
12
13 image = cv2.imread(args["image"])
14 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

Lines 1-4 handles importing the packages that Jeremy needs to build his face recognition algorithm. He has defined a `FaceDetector` class in the `facedetector` module of `pyimagesearch` to keep his code neat and tidy. Then, Jeremy uses `argparse` to parse his command line arguments and `cv2` to provide him with his OpenCV bindings.

Jeremy needs two command line arguments: `--face`, which is the path to where the face classifier (more on that later) resides, and `--image`, the path to the image that contains

the faces that he wants to find.

But let's not get too far ahead of ourselves. Before we can even think about *finding* faces in an image, we first need to define a class to handle *how* we are going to find faces in an image.

Listing 2.2: facedetector.py

```
1 import cv2
2
3 class FaceDetector:
4     def __init__(self, faceCascadePath):
5         self.faceCascade = cv2.CascadeClassifier(faceCascadePath)
6
7     def detect(self, image, scaleFactor = 1.1, minNeighbors = 5,
          minSize = (30, 30)):
8         rects = self.faceCascade.detectMultiScale(image,
9             scaleFactor = scaleFactor,
10            minNeighbors = minNeighbors, minSize = minSize,
11            flags = cv2.CASCADE_SCALE_IMAGE)
12
13        return rects
```

In order to build face recognition software, Jeremy has to use the built-in Haar cascade classifiers in OpenCV. Luckily for him, these classifiers have already been pre-trained to recognize faces!

Building our own classifier is certainly outside the scope of this case study. But if we wanted to, we would need a lot of "positive" and "negative" images. Positive images would contain images *with* faces, whereas negative images would contain images *without* faces. Based on this dataset, we could then extract features to characterize the face (or lack of face) in an image and build our own classifier. It would be a lot of work, and very time consuming, especially for someone like Jeremy, who is a computer vision

novice (and should be studying for his final exam). Luckily, OpenCV will do all the heavy lifting for him.

Anyway, these classifiers work by scanning an image from left to right, and top to bottom, at varying scale sizes. Scanning an image from left to right and top to bottom is called the "sliding window" approach.

As the window moves from left to right and top to bottom, one pixel at a time, the classifier is asked whether or not it "thinks" there is a face in the current window, based on the parameters that Jeremy has supplied to the classifier.

On **Line 1** of `facedetector.py`, Jeremy imports the `cv2` package so that he has access to his OpenCV bindings.

Then, on **Line 3** he defines his `FaceDetector` class, which will encapsulate all the necessary logic to perform face detection.

Jeremy then defines the constructor on **Line 4**, which takes a single parameter – the path to where his cascade classifier lives. This classifier is serialized as an XML file. Making a call to `cv2.CascadeClassifier` will deserialize the classifier, load it into memory, and allow him to detect faces in images.

To actually find the faces in an image, Jeremy defines the `detect` method on **Line 7**. This function takes one required parameter, the `image` that he wants to find the faces in, followed by three optional arguments. Let's take a look at what these arguments mean:

- **scaleFactor**: How much the image size is reduced at each image scale. This value is used to create the scale pyramid in order to detect faces at multiple scales in the image (some faces may be closer to the foreground, and thus be larger; other faces may be smaller and in the background, thus the usage of varying scales). A value of 1.05 indicates that Jeremy is reducing the size of the image by 5% at each level in the pyramid.

- **minNeighbors**: How many neighbors each window should have for the area in the window to be considered a face. The cascade classifier will detect multiple windows around a face. This parameter controls how many rectangles (neighbors) need to be detected for the window to be labeled a face.

- **minSize**: A tuple of width and height (in pixels) indicating the minimum size of the window. Bounding boxes smaller than this size are ignored. It is a good idea to start with `(30, 30)` and fine-tune from there.

Detecting the actual faces in the image is handled on **Line 8** by making a call to the `detectMultiScale` method of Jeremy's classifier created in the constructor of the `FaceDetector` class. He supplies his `scaleFactor`, `minNeighbors`, and `minSize`, then the method takes care of the entire face detection process for him!

The `detectMultiScale` method then returns `rects`, a list of tuples containing the bounding boxes of the faces in the image. These bounding boxes are simply the *(x, y)* location of the face, along with the width and height of the box.

Now that Jeremy has implemented the `FaceDetector` class, he can now apply it to his own images.

Listing 2.3: detect_faces.py

```
15 fd = FaceDetector(args["face"])
16 faceRects = fd.detect(gray, scaleFactor = 1.1, minNeighbors = 5,
17     minSize = (30, 30))
18 print("I found {} face(s)".format(len(faceRects)))
19
20 for (x, y, w, h) in faceRects:
21     cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
22
23 cv2.imshow("Faces", image)
24 cv2.waitKey(0)
```

On **Line 15** Jeremy instantiates his `FaceDetector` class, supplying the path to his XML classifier as the sole parameter.

Then, Jeremy detects the actual faces in the image on **Line 16** by making a call to the `detect` method.

Finally, **Line 18** prints out the number of faces found in the image.

But in order to actually draw a bounding box around the image, Jeremy needs to loop over them individually, as seen on **Line 20**. Again, each bounding box is just a tuple with four values: the x and y starting location of the face in the image, followed by the *width* and *height* of the face.

A call to `cv2.rectangle` draws a green box around the actual faces on **Line 21**.

Figure 2.1: Detecting the face of the United States president, Barack Obama.

And finally, Jeremy displays the output of his hard work on **Lines 23-24**.

To execute his face detection Python script, Jeremy fires up a shell and issues the following command:

Listing 2.4: detect_faces.py

```
$ python detect_faces.py --face cascades/
    haarcascade_frontalface_default.xml --image images/obama.png
```

To see the output of his hard work, check out Figure 2.1. Jeremy's script is clearly able to detect the face of United States president Barack Obama.

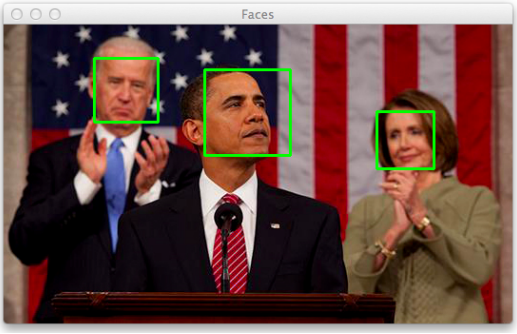

Figure 2.2: Detecting multiple faces in an image.

And since Jeremy took care to loop over the number of faces on **Line 20**, he can also conveniently detect multiple faces, as seen in Figure 2.2.

However, when Jeremy applied his script to the photo of soccer player Lionel Messi in Figure 2.3 *(left)*, he noticed something strange – his code was detecting *two* faces when there is clearly only *one* face!

Why is this? Jeremy's code was working perfectly for the other images!

The answer lies within the parameters to the `cv2.detectMutliScale` function that we discussed above. These parameters tend to be sensitive, and some parameter choices for one set of images will not work for another set of images.

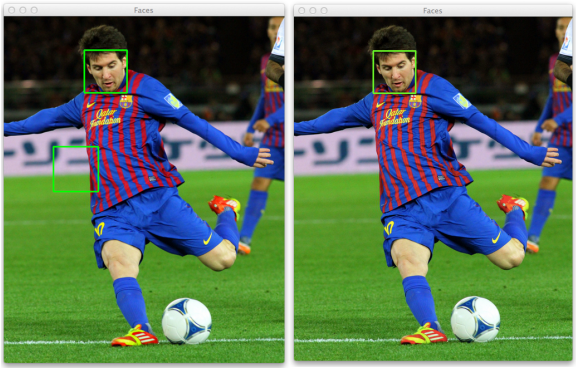

Figure 2.3: *Left:* Jeremy's code is (incorrectly) detecting two faces in the image when there is clearly only one. *Right:* Adjusting the `scaleFactor` fixes the issue – now only Lionel Messi's face is detected.

In most cases, the offending culprit will be the `scaleFactor` parameter. In other cases it may be `minNeighbors`. But as a debugging rule, start with the `scaleFactor`, adjust it as needed, and then move on to `minNeighbors`.

Taking this debugging rule into consideration, Jeremy changed his call to the `detect` method of `FaceDetector` on **Line 16**:

Listing 2.5: detect_faces.py

```
faceRects = fd.detect(gray, scaleFactor = 1.2, minNeighbors = 5,
    minSize = (30, 30))
```

The only change made was to the `scaleFactor` parameter, changing it from `1.1` to `1.2`.

But by making this simple change, we can see in Figure 2.3 *(right)* that the incorrectly labeled face has been removed and we are left with only the correctly labeled face of Lionel Messi.

Smiling contently at his accomplishments, Jeremy stole a glance at his alarm clock sitting next to his still-made bed.

3:22 am.

His Algorithms final is in less than five hours! And he still hasn't gotten a wink of sleep!

Oh well.

4

OBJECT TRACKING IN VIDEO

"Grissom should have never left the show," mused Laura, nursing her Pinot Grigio.

Another long day at Initech, and here she was, sipping her wine and watching *CSI* re-runs on Netflix.

This had become the "norm" for Laura. Coming home at 7 pm after a horribly dull day at work, she only has her TV and wine to keep her company.

With a melancholy look on her face, Laura muttered to herself, "There must be more to life than this," but another pull of her wine washed the thought away.

It's not that her life was bad. All things considered, it was actually quite good.

Her job, while boring, paid well. Money was not an issue.

The real issue was Laura lacked a sense of pride in her job. And without that pride, she did not feel complete.

Following the arsonist attack on the Initech building three years earlier (reportedly over a red Swingline stapler), the company had since rebuilt. And Laura was hired right out of college as a programmer.

Her job was to update bank software. Find the bugs. Fix them. Commit the code repository. And ship the production package.

Rinse. And. Repeat.

It was the dull monotony of the days that started to wear on Laura at first.

She quickly realized that no matter how much money she made, no matter how much was sitting in her bank account, it could not compensate for that empty feeling she had in the pit of her stomach every night – she needed a bigger challenge.

And maybe it was the slight buzz from the wine, or maybe because watching *CSI* re-runs was becoming just as dull as her job, but Laura decided that tonight she was going to make a change and work on a project of her own.

Thinking back to college, where she majored in computer science, Laura mused that the projects were her favorite part. It was the act of *creating* something that excited her.

One of her final classes as an undergraduate was a special topics course in image processing. She learned the basics of image processing and computer vision. And more impor-

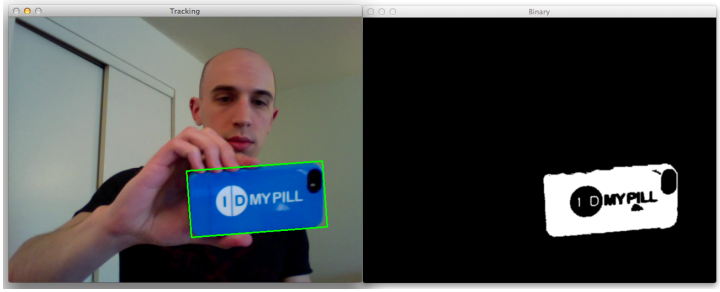

Figure 4.1: Finding and tracking an iPhone in a video. A bounding box is drawn around the tracked iPhone on the *left*, and the thresholded image is displayed on the *right*.

tantly, she really enjoyed herself when taking the class.

Pausing her *CSI* episode and refilling her wine glass, Laura reached over and grabbed her laptop.

It was time to dust off her image processing skills and build something of her own.

Object tracking in video seemed like a good place to start. Who knows where it might lead? Maybe to a better job. At least, that was Laura's train of thought, as she opened up `vim` and started coding:

Listing 4.1: track.py

```
1 import numpy as np
```

```
2 import argparse
3 import time
4 import cv2
5
6 ap = argparse.ArgumentParser()
7 ap.add_argument("-v", "--video",
8     help = "path to the (optional) video file")
9 args = vars(ap.parse_args())
10
11 blueLower = np.array([100, 67, 0], dtype = "uint8")
12 blueUpper = np.array([255, 128, 50], dtype = "uint8")
13
14 camera = cv2.VideoCapture(args["video"])
```

On **Lines 1-4**, Laura imports the packages she needs. She'll make use of NumPy for numerical processing, `argparse` for parsing command line arguments, and `cv2` for her OpenCV bindings. The `time` package is optional, but is useful if she has a very fast system that is processing the frames of a video too quickly.

Laura needs only one command line argument, `--video`, which is the path to her video file on disk. Her command line argument is parsed on **Lines 6-9**.

The object that Laura will be tracking in the video is a blue iPhone case. Since the color blue isn't prevalent in any other location in the video besides the iPhone case, she wants to track shades of blue.

In order to accomplish this color tracking, she defines the lower and upper limits of the shades of blue in the RGB color space on **Lines 11 and 12**. Remember, OpenCV represents pixels in the RGB color space, *but in reverse order*.

In this case, Laura defines colors as "blue" if they are greater than $R = 0, G = 67, B = 100$ and less than $R =$

50, $G = 128, B = 255$.

Finally, Laura opens the video file and grabs a reference to it using the `cv2.VideoCapture` function on **Line 14**. She stores this reference as `camera`.

Listing 4.2: track.py

```
16 while True:
17     (grabbed, frame) = camera.read()
18
19     if not grabbed:
20         break
21
22     blue = cv2.inRange(frame, blueLower, blueUpper)
23     blue = cv2.GaussianBlur(blue, (3, 3), 0)
```

Now that she has a reference to the video, she can start processing the frames.

Laura starts looping over the frames, one at a time, on **Line 16**. A call to the `read()` method of `camera` grabs the next frame in the video, which returns a tuple with two values. The first, `grabbed`, is a boolean indicating whether or not the frame was successfully read from the video file. The second, `frame`, is the frame itself.

She then checks to see if the frame was successfully read on **Line 19**. If the frame was not read, then she has reached the end of the video, and she can break from the loop.

In order to find shades of blue in the `frame`, Laura must make use of the `cv2.inRange` function on **Line 22**. This function takes three parameters. The first is the `frame` that she wants to check. The second is the lower threshold on RGB pixels, and the third is the upper threshold. The result

of calling this function is a thresholded image, with pixels falling within the upper and lower range set to *white* and pixels that do not fall into this range set as *black*.

Finally, the thresholded image is blurred on **Line 23** to make finding contours more accurate.

Pausing to take a pull of her Pinot Grigio, Laura contemplated the idea of quitting her job and working somewhere else.

Why spend her life working a job that wasn't challenging her?

Tabling the thought, she then went back to coding:

Listing 4.3: track.py

```
25      (_, cnts, _) = cv2.findContours(blue.copy(), cv2.
            RETR_EXTERNAL,
26          cv2.CHAIN_APPROX_SIMPLE)
27
28      if len(cnts) > 0:
29          cnt = sorted(cnts, key = cv2.contourArea, reverse = True)
                [0]
30
31          rect = np.int32(cv2.boxPoints(cv2.minAreaRect(cnt)))
32          cv2.drawContours(frame, [rect], -1, (0, 255, 0), 2)
33
34      cv2.imshow("Tracking", frame)
35      cv2.imshow("Binary", blue)
36
37      time.sleep(0.025)
38
39      if cv2.waitKey(1) & 0xFF == ord("q"):
40          break
41
42  camera.release()
43  cv2.destroyAllWindows()
```

Now that Laura has the thresholded image, she needs to find the largest contour in the image, with the assumption that the largest contour corresponds to the outline of the phone that she wants to track.

A call to `cv2.findContours` on **Line 25** finds the contours in the thresholded image. She makes sure to clone the thresholded image using the `copy()` method since the `cv2.findContour` function is destructive to the NumPy array that she passes in.

On **Line 28** Laura checks to make sure that contours were actually found. If the length of the list of contours is zero, then no regions of blue were found. If the length of the list of contours is greater than zero, then she needs to find the largest contour, which is accomplished on **Line 29**. Here, the contours are sorted in reverse order (largest first), using the `cv2.contourArea` function to compute the area of the contour. Contours with larger areas are stored at the front of the list. In this case, Laura grabs the contour with the largest area, again assuming that this contour corresponds to the outline of the iPhone.

Laura now has the outline of the iPhone, but she needs to draw a bounding box around it.

Calling `cv2.minAreaRect` computes the minimum bounding box around the contour. Then, `cv2.boxPoints` re-shapes the bounding box to be a list of points.

Note: In OpenCV 2.4.X, we would use the `cv2.cv.BoxPoints` function to compute the rotated bounding box of the contour. However, in OpenCV 3.0+, this function has been moved to `cv2`.

`boxPoints`. Both functions perform the same task, just with slightly different namespaces.

Finally, Laura draws the bounding box on **Line 32** using the `cv2.drawContours` function.

The frame with the detected iPhone is displayed on **Line 34**, and the thresholded image (pixels that fall into the lower/upper range of blue pixels) is displayed on **Line 35**.

Laura notes that **Line 37** is optional. On many newer model machines, the system may be fast enough to process > 32 frames per second. If this is the case, finding an acceptable `sleep` time will slow down the processing and bring it down to more normal speeds.

She then checks to see if the `q` key is pressed on **Lines 39-40**. If it is pressed, she breaks from the `while` loop that is continually grabbing frames from the video.

Finally, **Lines 42 and 43** destroys the reference to the `camera` and closes any windows that OpenCV has opened.

To execute her object tracking script, Laura issues the following command:

Listing 4.4: track.py

```
$ python track.py --video video/iphonecase.mov
```

The output of the script can be seen in Figure 4.2.

On the *left*, both figures show an individual frame of the movie with the iPhone successfully found and tracked. The *right* image shows the thresholded image, with pixels

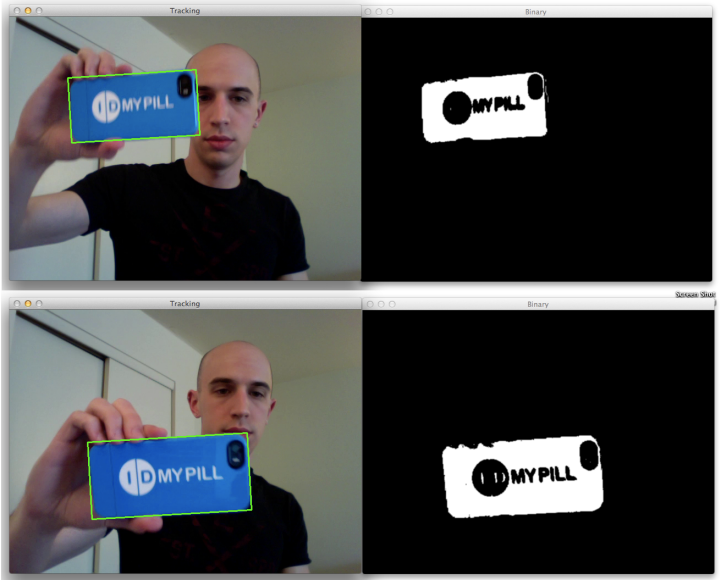

Figure 4.2: *Left:* Examples of the iPhone being successfully tracked in multiple frames of the video. *Right:* The thresholded image, with pixels falling into the `blueLower` and `blueUpper` range displayed as *white* and pixels not falling into the range as *black*.

falling into the `blueLower` and `blueUpper` range displayed as *white* and pixels not falling into the range as *black*.

The very next day, Laura walked into Initech and gave her two weeks notice – no more working a job that didn't challenge her. Laura wanted more out of life.

And she found it.

Only a month after leaving Initech, she was approached by their rival, Initrode. They were looking for someone to do eye tracking on their ATM.

Ecstatic, Laura accepted the job – and received a higher salary than she did at Initech. But at this point, the money didn't matter. The satisfaction of working a job she enjoyed was all the payment she needed.

Laura doesn't need her glass (or two) of Pinot Grigio at night anymore. But she still likes her *CSI* re-runs. As she dreamily drifts off to the glow of Grissom's face on TV, she notes that the re-runs are somehow less boring now that she is working a job she actually likes.

5

EYE TRACKING

"Unbelievable," thought Laura.

Only a month ago she had been working at Initech, bored out of her mind, updating bank software, completely unchallenged.

Now, after a few short weeks, she was at Initech's competitor, Initrode, doing something she loved – working with computer vision.

It all started a month ago when she decided to put down that glass of Pinot Grigio, open up her laptop, and learn a new skill.

In just a single night she was able to write a Python script to find and track objects in video. She posted her code to an OpenCV forum website, where it gained a lot of attention.

Apparently, it caught the eye of one of the Initrode research scientists, who promptly hired Laura as a computer vision developer.

Now, sitting at her desk, her first assignment is to create a computer vision system to track eyes in video. And the boss needs it done by the end of the day.

That's definitely a tall order.

But, Laura admitted to herself, she didn't feel stressed.

That morning she was browsing the same OpenCV forums she posted her object tracking code to and came across a guy named Jeremy – apparently he had done some work in face recognition, which is the first step of eye tracking.

Ecstatic, Laura downloaded Jeremy's code and started hacking:

Listing 5.1: eyetracker.py

```
1  import cv2
2
3  class EyeTracker:
4      def __init__(self, faceCascadePath, eyeCascadePath):
5          self.faceCascade = cv2.CascadeClassifier(faceCascadePath)
6          self.eyeCascade = cv2.CascadeClassifier(eyeCascadePath)
7
8      def track(self, image):
9          faceRects = self.faceCascade.detectMultiScale(image,
10             scaleFactor = 1.1, minNeighbors = 5,
11             minSize = (30, 30),
12             flags = cv2.CASCADE_SCALE_IMAGE)
13         rects = []
```

The first thing Laura does is import the `cv2` package on **Line 1** so that she has access to her OpenCV bindings.

Then, Laura defines her `EyeTracker` class on **Line 3** and the constructor on **Line 4**. Her `EyeTracker` class takes two

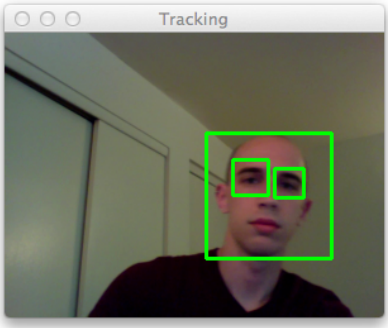

Figure 5.1: Detecting eyes in an image. First, the face must be detected. Then, the face area can be searched for eyes.

arguments: `faceCascadePath` and `eyeCascadePath`. The first is the path to the built-in face cascade classifier in OpenCV. The second is the path to the eye cascade classifier.

Note: For more information on how the OpenCV cascade classifiers work, refer to Chapter 2 where I provide a detailed review.

She then loads both classifiers off disk using the `cv2.CascadeClassifier` function on **Lines 5 and 6**.

From there, Laura defines the `track` method which is used to find the eyes in the image. This method takes only a single parameter – the `image` that contains the face and eyes she wants to track.

Laura then calls the `detectMultiScale` method (**Line 9**) of her `faceCascade` classifier. This method returns to her the bounding box locations (i.e., the x, y, width, and height) of each face in the image.

She then initializes a list of rectangles that will be used to contain the face and eye rectangles in the image.

Note: The parameters to `detectMultiScale` *are hard-coded into the* `EyeTracker` *class. If you are having trouble detecting faces and eyes in your own images, you should start by exploring these parameters. See Chapter 2 for a detailed explanation of these parameters and how to tune them for better detection accuracy.*

Not a bad start. Now that Laura has the face regions in the image, let's see how she can use them to find the eyes:

Listing 5.2: eyetracker.py

```
15          for (fX, fY, fW, fH) in faceRects:
16              faceROI = image[fY:fY + fH, fX:fX + fW]
17              rects.append((fX, fY, fX + fW, fY + fH))
18
19              eyeRects = self.eyeCascade.detectMultiScale(faceROI,
20                  scaleFactor = 1.1, minNeighbors = 10,
21                  minSize = (20, 20),
22                  flags = cv2.CASCADE_SCALE_IMAGE)
23
24              for (eX, eY, eW, eH) in eyeRects:
25                  rects.append(
26                      (fX + eX, fY + eY, fX + eX + eW, fY + eY + eH))
27
28          return rects
```

Laura starts looping over the x, y, width, and height location of the faces on **Line 15**.

She then extracts the face Region of Interest (ROI) from the image on **Line 16** using NumPy array slicing. The `faceROI` variable now contains the bounding box region of the face.

Finally, she appends the *(x, y)* coordinates of the rectangle to the list of `rects` for later use.

Now she can move on to eye detection.

This time, she makes a call to the `detectMultiScale` method of the `eyeCascade` on **Line 19**, giving her a list of locations in the image where eyes appear.

Laura uses a much larger value of `minNeighbors` on **Line 20** since the eye cascade tends to generate more false-positives than other classifiers.

Note: Again, these parameters are hard-coded into the `EyeTracker` *class. If you apply this script to your own images and video, you will likely have to tweak them a bit to obtain optimal results. Start with the* `scaleFactor` *variable and then move on to* `minNeighbors`.

Then, Laura loops over the bounding box regions of the eyes on **Line 24**, and updates her list of bounding box rectangles on **Line 25**.

Finally, the list of bounding boxes is returned to the caller on **Line 28**.

After saving her code, Laura takes a second to look at the time – 3 pm!

She worked straight through lunch!

But at least the hard part is done. Time to glue the pieces together by creating `eyetracking.py`:

Listing 5.3: eyetracking.py

```
1 from pyimagesearch.eyetracker import EyeTracker
2 from pyimagesearch import imutils
3 import argparse
4 import cv2
5
6 ap = argparse.ArgumentParser()
7 ap.add_argument("-f", "--face", required = True,
8     help = "path to where the face cascade resides")
9 ap.add_argument("-e", "--eye", required = True,
10     help = "path to where the eye cascade resides")
11 ap.add_argument("-v", "--video",
12     help = "path to the (optional) video file")
13 args = vars(ap.parse_args())
14
15 et = EyeTracker(args["face"], args["eye"])
```

First, Laura imports her necessary packages on **Lines 1-4**. She'll use her custom `EyeTracker` class to find faces and eyes in images. She'll also use `imutils`, a set of image manipulation convenience functions to help her resize her images. Finally, she'll use `argparse` for command line parsing and `cv2` for her OpenCV bindings.

On **Lines 7-13**, Laura parses her command line arguments: `--face`, which is the path to her face cascade classifier, and `--eye`, which is the path to her eye cascade classifier.

For debugging purposes (or if her system doesn't have a webcam), Laura creates an optional command line argument, `--video`, which points to a video file on disk.

Note: If you are using the Ubuntu VirtualBox virtual machine I have created to aid you in your computer vision studies (more information is available here: https://www.pyimagesearch.com/practical-python-opencv/), you will not be able to access a webcam if your system is equipped with one. Handling webcams is outside the capability of VirtualBox, hence why I am explaining how to utilize video files in OpenCV.

Finally, Laura instantiates her `EyeTracker` class on **Line 15** using the paths to her face and eye classifiers, respectively.

Listing 5.4: eyetracking.py

```
17 if not args.get("video", False):
18     camera = cv2.VideoCapture(0)
19
20 else:
21     camera = cv2.VideoCapture(args["video"])
22
23 while True:
24     (grabbed, frame) = camera.read()
25
26     if args.get("video") and not grabbed:
27         break
```

Lines 17 and 18 handle if a video file is not supplied – in this case, the `cv2.VideoCapture` function is told to use the webcam of the system.

Otherwise, if a path to a video file *was* supplied (**Line 20 and 21**), then the `cv2.VideoCapture` function opens the

video file and returns a pointer to it.

Laura starts looping over the frames of the video on **Line 23**. A call to the `read` method of the `camera` grabs the next frame in the video. A tuple is returned from the `read` method, containing (1) a boolean indicating whether or not the frame was successfully read, and (2), the `frame` itself.

Then, Laura makes a check on **Lines 26-27** to determine if the video has run out of frames. This check is only performed if the video is being read from file.

Now that Laura has the current frame in the video, she can perform face and eye detection:

Listing 5.5: eyetracking.py

```
29      frame = imutils.resize(frame, width = 300)
30      gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
31
32      rects = et.track(gray)
33
34      for rect in rects:
35          cv2.rectangle(frame, (rect[0], rect[1]),
36              (rect[2], rect[3]), (0, 255, 0), 2)
37
38      cv2.imshow("Tracking", frame)
39
40      if cv2.waitKey(1) & 0xFF == ord("q"):
41          break
42
43  camera.release()
44  cv2.destroyAllWindows()
```

In order to make face and eye detection faster, Laura first resizes the image to have a width of 300 pixels on **Line 29**.

She then converts it to grayscale on **Line 30**. Converting to grayscale tends to increase the accuracy of the cascade classifiers.

A call is made to the `track` method of her `EyeTracker` on **Line 32** using the current `frame` in the video. This method then returns a list of `rects`, corresponding to the faces and eyes in the image.

On **Line 34** she starts looping over the bounding box rectangles and draws each of them using the `cv2.rectangle` function, where the first argument is the `frame`, the second the starting *(x, y)* coordinates of the bounding box, the third the ending *(x, y)* coordinates, followed by the color of the box (green), and the thickness (2 pixels).

Laura then displays the `frame` with the detected faces and eyes on **Line 38**.

A check is made on **Lines 41 and 42** to determine if the user pressed the `q` key. If the user did, then the frame loop is broken out of.

Finally, a cleanup is performed on **Lines 43 and 44**, where the camera pointer is released and all windows created by OpenCV are closed.

Laura executes her script by issuing the following command:

Listing 5.6: cam.py

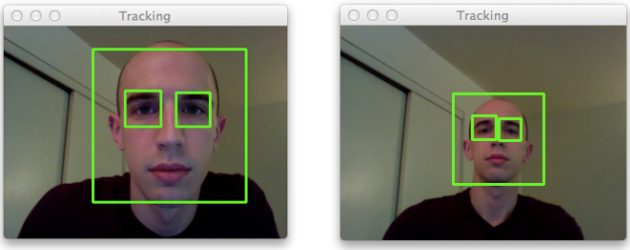

Figure 5.2: Even as the face and eyes move closer to the camera and farther away, Laura's code is still able to track them in the video

```
$ python eyetracking.py --face cascades/
    haarcascade_frontalface_default.xml --eye cascades/
    haarcascade_eye.xml --video video/adrian_eyes.mov
```

Of course, if she wanted to use her webcam, she would use this command, omitting the `--video` switch:

Listing 5.7: cam.py

```
$ python eyetracking.py --face cascades/
    haarcascade_frontalface_default.xml --eye cascades/
    haarcascade_eye.xml
```

To see the output of Laura's hard work, take a look at Figure 5.2.

Notice how even as the face and eyes move closer to the camera and then farther away, Laura's code is still able to track them in the video without an issue.