Supports OpenCV 4 & Python 3+!

CASE STUDIES

# Practical Python and OpenCV

## An Introductory, Example Driven Guide to Image Processing and Computer Vision

4TH EDITION

Dr. Adrian Rosebrock

pyimagesearch

# Practical Python and OpenCV: Case Studies

**4th Edition**

Dr. Adrian Rosebrock

# COPYRIGHT



This version of the book was published on 14 December 2018.

# CONTENTS

**Author's Note:** All persons, government organizations, universities, corporations, institutions, and museums used in this book are either fictitious or used fictitiously.

## COMPANION WEBSITE & SUPPLEMENTARY MATERIAL

Thank you for picking up a copy of the 4th edition of *Practical Python and OpenCV*!

In this latest edition, I'm excited to announce the creation of a *companion website* which includes supplementary material that I could not fit inside the book.

At the end of nearly every chapter inside *Practical Python and OpenCV + Case Studies*, you'll find a link to a supplementary webpage that includes additional information, such as my commentary on methods to extend your knowledge, discussions of common error messages, recommendations on various algorithms to try, and optional quizzes to test your knowledge.

Registration to the companion website is **free** with your purchase of *Practical Python and OpenCV*.

**To create your companion website account, just use this link:**

http://pyimg.co/o1y7e

Take a second to create your account **now** so you'll have access to the supplementary materials as you work through the book.

# PREFACE

My first book, *Practical Python and OpenCV*, taught you the basics of computer vision and image processing.

Now, this Case Studies book will help you apply your knowledge to solve *actual real-world problems*.

I've written these Case Studies from the perspective of programmers who are tackling computer vision problems in their everyday lives. These characters are entirely fictional, but they are meant to convey a single common theme throughout this book – there are people are working on and solving these types of problems using computer vision every single day.

If you're looking to solve real-world problems in computer vision, such as face detection in both images and video, object tracking in video, handwriting recognition, classification, and keypoint matching, then this is the book for you.

Remember, if you have any questions or comments, or if you simply want to say hello, just send me an email at adrian@pyimagesearch.com, or visit my website at www.PyImageSearch.com and leave a comment.

I look forward to hearing from you soon!

All the best,

-Adrian Rosebrock

## PREREQUISITES

In order to make the most of these case studies, you will need to have a little bit of programming experience. All examples in this book are in the Python programming language. Familiarity with Python or other scripting languages is suggested, but not required.

A little background in machine learning and use with the scikit-learn library is also recommended; however, the examples in this book contain lots of code, so even if you are a complete beginner, do not worry! You'll find that the examples are extremely detailed and heavily documented to help you follow along.

## CONVENTIONS USED IN THIS BOOK

This book includes many code listings and terms to aid you in your journey to learn computer vision and image processing. Below are the typographical conventions used in this book:

*Italic*

Indicates key terms and important information that you should take note of. May also denote mathematical equations or formulas based on connotation.

**Bold**

Important information that you should take note of.

`Constant width`

Used for source code listings, as well as paragraphs that make reference to the source code, such as function and method names.

## USING THE CODE EXAMPLES

This book is meant to be a hands-on approach to computer vision and machine learning. The code included in this book, along with the source code distributed with this book, are free for you to modify, explore, and share as you wish.

In general, you do not need to contact me for permission if you are using the source code in this book. Writing a script that uses chunks of code from this book is totally and completely okay with me.

However, selling or distributing the code listings in this book, whether as information product or in your product's documentation, *does* require my permission.

If you have any questions regarding the fair use of the code examples in this book, please feel free to shoot me an email. You can reach me at adrian@pyimagesearch.com.

## HOW TO CONTACT ME

Want to find me online? Look no further:

| | |
|---|---|
| **Website:** | www.PyImageSearch.com |
| **Email:** | adrian@pyimagesearch.com |
| **Twitter:** | @PyImageSearch |
| **Facebook:** | PyImageSearch |
| **Google+:** | +AdrianRosebrock |
| **LinkedIn:** | Adrian Rosebrock |

# 1

# INTRODUCTION

This book is meant to be a hands-on approach to applying the basics of computer vision and image processing to solve real-world problems.

We'll start by talking with Jeremy, a college student interested in computer vision. Instead of spending his time studying for his Algorithms final exam, he instead becomes entranced by face detection.

Jeremy applies face detection to both pictures and videos, and while his final grade in Algorithms is in jeopardy, at least he learns a lot about computer vision.

We'll then chat with Laura, who works at Initech (after it burned to the ground, allegedly over a red Swingling stapler) updating bank software. She's not very challenged at her job, and she spends her nights sipping Pinot Grigio and watching *CSI* re-runs.

Sick of her job at Initech, Laura studies up on computer vision and learns how to track objects in video. Ultimately, she's able to leave her job at Initech and join their rival, Initrode, and build software used to track eye movements in

ATM cameras.

Next up, we'll stop by Hank's office. Hank and his team of programmers are consulting for the Louisiana post office, where they are tasked with building a system to accurately classify the zip codes on envelopes.

Unfortunately, Hank underbid on the job, and he's currently extremely stressed that the project will not be completed on time. If the job isn't done on time, profits will suffer and he might lose his job! Hopefully we can help Hank out.

After helping out Hank, we'll take a field trip to the New York Museum of Natural History and speak with Charles, a curator in the Botany department.

One of Charles' jobs is to classify the species of flowers in photographs. It's extremely time consuming and tedious, but the Museum pays handsomely for it.

Charles decides to create a system to *automatically* classify the species of flowers using computer vision and machine learning techniques. This approach will save him a bunch of time and allow him to get back to his research, instead of mindlessly classifying flower species.

Finally, we'll head to San Francisco to meet with Gregory, the hotshot *Wu-Tang Clan* loving entrepreneur who's working with his co-founder to create a competitor to Amazon's Flow. Flow allows users to utilize the camera on their smartphone as a digital shopping device. By simply taking a picture of a cover of a book, DVD, or video game, Flow au-

tomatically identifies the product and adds it to the user's shopping cart.

But Gregory is having a hard time getting the algorithm off the ground, and his personal savings is quickly being eaten up. Hopefully we can help Gregory before he runs out of funds!

While all these people in this book are entirely fictional, and sometimes conveyed in a quite satirical and sarcastic manner, these *are* real-world computer vision problems they are solving.

So if you want the real-world experience solving interesting problems, this is definitely the book for you.

Sound good?

Let's get started and check in with Jeremy in his college dorm room.

# 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.

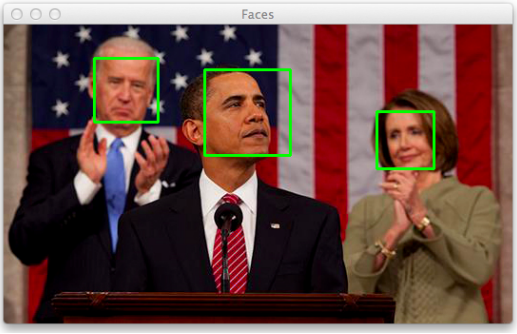


Figure 2.2: Detecting multiple faces in an image.

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.

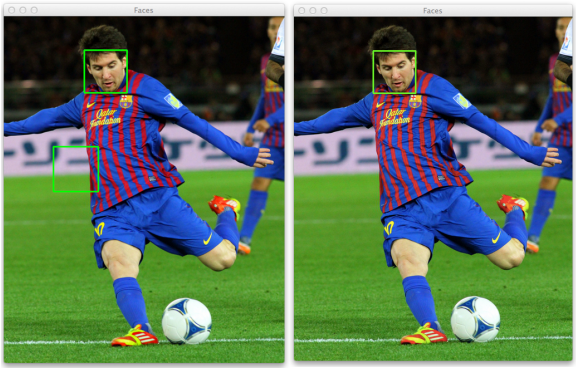


Figure 2.3: *Left:* Jeremy's code is (incorrectly) detecting two faces in the image when there is clearly only one. *Right:* Adjusting the `scaleFactor` fixes the issue – now only Lionel Messi's face is detected.

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.

Like most programmers, Jeremy always enjoys a night of successful coding.

Feeling no regret and closing his laptop, Jeremy glanced at his Algorithms notes. No point in studying now. Might as well get to sleep and hope for the best tomorrow.

## Further Reading

In Chapter 2, we learned how to detect faces in images using OpenCV's pre-trained Haar cascades.

Inside the supplementary material for this chapter, I've provided suggestions to help tune Haar cascade parameters – and alternative methods that can be used to create *custom object detectors* of your own:

http://pyimg.co/8rw10

# 3

## WEBCAM FACE DETECTION

"Should have spent more time studying the master theorem", thought Jeremy as he opened the door to his dorm, throwing his bag on the ground next to a pile of dirty t-shirts, and grabbing another Mountain Dew from his mini-fridge.

At least he made it to his Algorithms exam on time. He almost slept through it. Clearly he had spent too much time working on his face detection algorithm last night.

Oh well. What's done is done. Hopefully he passed.

Reflecting on his previous night of coding, Jeremy realized something – computer vision had been on his mind non-stop. Even during his exam.

And while he still can't derive the master theorem, he was able to implement a face detection algorithm using Python and OpenCV.

But this algorithm only worked for single images, such as pictures that his friends tagged him in on Facebook.

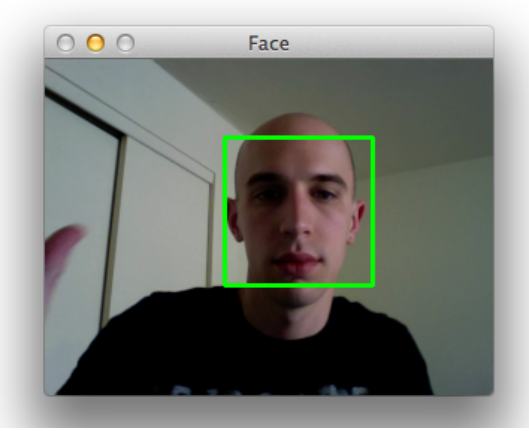


Figure 3.1: Jeremy wants to explore how to detect faces in webcam video in real-time. Let's figure out how he pulled it off.

Working with single images is all fine and good... but what about *video*?

Now that would be cool. He could extend his code to work with the built-in webcam on his laptop.

Taking another sip of Mountain Dew, Jeremy logged into his Mac, re-opened his face detection project, and smirked at himself, muttering, "Heap sort is O(n log n)."

At least he got something right on that exam.

Let's jump into some code and figure out how Jeremy managed to perform face detection real-time, as seen in Figure 3.1.

Listing 3.1: cam.py

```
1 from pyimagesearch.facedetector import FaceDetector
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("-v", "--video",
10     help = "path to the (optional) video file")
11 args = vars(ap.parse_args())
12
13 fd = FaceDetector(args["face"])
```

**Lines 1-4** handle importing the packages that Jeremy is going to use to build his real-time face detection algorithm. The `FaceDetector` class in the `facedetector` sub-package of `pyimagesearch` is just his code from last night (see Chapter 2). The `imutils` package contains convenience functions used to perform basic image operations, such as resizing.

*Note: For the astute reader, we covered the* `imutils` *package in Chapter 6 of the* Practical Python and OpenCV *book.*

To parse command line arguments, Jeremy elects to use `argparse`. And finally, `cv2` is used to bind with the OpenCV library.

Pausing a second to take another swig of Code Red, Jeremy remembered back to last night – he needed a Haar cascade classifier to find the faces in an image. The classifier is se-

rialized as an XML file which can be loaded by OpenCV. *That sounds like it better be a command line argument*, thought Jeremy, creating the `--face` argument, which points to the serialized XML cascade classifier on disk.

For debugging purposes (or if his system doesn't have a webcam), Jeremy created an optional command line argument, `--video`, which points to a video file on disk. Just in case he can't use his webcam, it would still be nice to test and debug his real-time system using a video file. In that case, all Jeremy needs to do is supply the path to his video file using the `--video` switch.

*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, which is why I am explaining how to utilize video files in OpenCV.*

Finally, the `FaceDetector` is instantiated using the path to the cascade classifier on **Line 13**.

Listing 3.2: cam.py

```
15 if not args.get("video", False):
16     camera = cv2.VideoCapture(0)
17
18 else:
19     camera = cv2.VideoCapture(args["video"])
```

Of course, Jeremy is now supporting both input video via his webcam and a file residing on disk. He needs to

create some logic to handle these cases.

**Lines 15 and 16** handle when the `--video` switch is not supplied. In this case, OpenCV will try to read video from the built-in (or USB) webcam of his laptop.

Otherwise, OpenCV is instructed to open the video file pointed to by the `--video` argument on **Lines 18 and 19**.

In either case, the `cv2.VideoCapture` function is used. Supplying an integer value of `0` instructs OpenCV to read from the webcam device, whereas supplying a string indicates that OpenCV should open the video the path points to. Supplying an invalid path will lead to a null pointer, and Jeremy obviously can't do any face detection without a valid video file.

Assuming that grabbing a reference to the video was successful, Jeremy stores this pointer in the `camera` variable.

Listing 3.3: cam.py

```
21 while True:
22     (grabbed, frame) = camera.read()
23
24     if args.get("video") and not grabbed:
25         break
26
27     frame = imutils.resize(frame, width = 300)
28     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

The next step is to start looping over all frames in the video. At the most basic level, a video is simply a sequence of images put together, implying that Jeremy can actually read these frames one at a time. The `while` loop on **Line 21** will keep looping over frames until one of two scenarios

are met: (1) the video has reached its end and there are no more frames, or (2) the user prematurely stops the execution of the script.

On **Lines 22** Jeremy grabs the next frame in the video by calling the `read()` method of `camera`. The `read` method returns a tuple of two values: the first is `grabbed`, a boolean indicating whether reading the frame was successful, and `frame`, which is the frame itself.

Jeremy takes care to handle a special case on **Lines 24 and 25**. If the video is being read from a file, and the frame was not grabbed, then the video is over, and he should break out of the infinite loop.

Otherwise, Jeremy performs a little pre-processing on **Lines 27 and 28**. The first thing he does is resize the `frame` to have a width of 300 pixels to make face detection in real time faster. Then he converts the frame to grayscale.

Smiling to himself, Jeremy realized that the hard part was now done.

All he needed to do was use his code from last night, only with a few small changes.

But his triumph was short-lived as his phone buzzed to life on his desk.

Jeremy, annoyed that he was so rudely snapped out of his programming zone, answered with a terse "Hello," agitation clearly apparent in his voice.

Immediately, his cheeks flushed.

It was his mother, asking if he had a chance to pick up the care package she sent him from the campus post office.

Horrified, Jeremy realized that he had not.

And no amount of explanation regarding studying for finals would pacify his mother. She was clearly unhappy that had forgotten about the care package that she had put so much effort into. And if Jeremy's mother was unhappy, then so was Jeremy.

After a 15-minute guilt-laden phone conversation, Jeremy finally got off the phone with a sigh.

It looked like his code would have to keep him company tonight. Hopefully, his mom wouldn't stay mad at him for long.

With an exasperated sigh, Jeremy turned back to his monitor, hit the `i` key to trigger input mode in `vim`, and got back to work:

Listing 3.4: cam.py

```
30      faceRects = fd.detect(gray, scaleFactor = 1.1,
31          minNeighbors = 5, minSize = (30, 30))
32      frameClone = frame.copy()
33
34      for (fX, fY, fW, fH) in faceRects:
35          cv2.rectangle(frameClone, (fX, fY), (fX + fW, fY + fH),
                (0, 255, 0), 2)
36
37      cv2.imshow("Face", frameClone)
38
```

```
39      if cv2.waitKey(1) & 0xFF == ord("q"):
40          break
41
42  camera.release()
43  cv2.destroyAllWindows()
```

**Line 30** handles face detection in the same manner that Jeremy used last night. He passes in his grayscale frame and applies the `detect` method of the `FaceDetector`.

But in order to draw a bounding box of faces on his image, he decides to create a clone of his frame first on **Line 32**, just in case he needs the original frame for further preprocessing. The clone of the frame is stored in `frameClone`.

Jeremy then loops over the bounding boxes of the faces in the image and draws them using the `cv2.rectangle` function on **Lines 34-35**.

**Line 37** displays the output of his face detection algorithm.

Of course, a user might want to stop execution of the script. Instead of forcing them to `ctl+c` out of it, he checks to see if the user has pressed the `q` key on their keyboard on **Lines 39 and 40**.

Finally, the reference to his `camera` is released on **Line 42**, and any open windows created by OpenCV are closed on **Line 43**.

To test out his script, Jeremy executes the following command, supplying the path to a testing video:

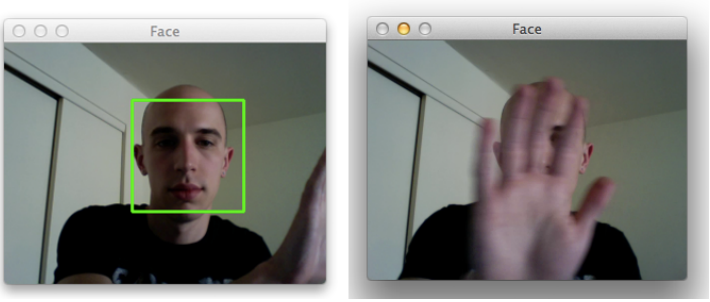


Figure 3.2: *Left:* Jeremy is able to detect a face in the webcam video. *Right:* The hand in front of the camera blocks the face, thus the face cannot be detected.

Listing 3.5: cam.py

```
$ python cam.py --face cascades/haarcascade_frontalface_default.
    xml --video video/adrian_face.mov
```

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

Listing 3.6: cam.py

```
$ python cam.py --face cascades/haarcascade_frontalface_default.
    xml
```

The results of Jeremy's hard work can be seen in Figure 3.2 *(left)*. Notice how the green bounding box is placed around the face in the image. This will happen for all frames in the image... provided there is a face in the image, of course!

If there is no a face in the image, as in Figure 3.2 *(right)*, then OpenCV will not be able to detect it! This is a simple enough concept, but is worth mentioning.

Pleased with his work, Jeremy drinks the rest of his Code Red, puts on his jacket, and heads to the door of his dorm room. Hopefully, the campus post office hasn't closed yet and he can pick up the care package from his mom.

Otherwise, he'll never hear the end of it when he comes home for Thanksgiving break...

## Further Reading

Anytime you work with video streams, you shuold consider applying *threading* to reduce I/O latency and therefore increase the number of frames you can process per second.

I detail how to access video streams using threads in the Chapter 3 supplementary material:

http://pyimg.co/rwzcr

# 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-

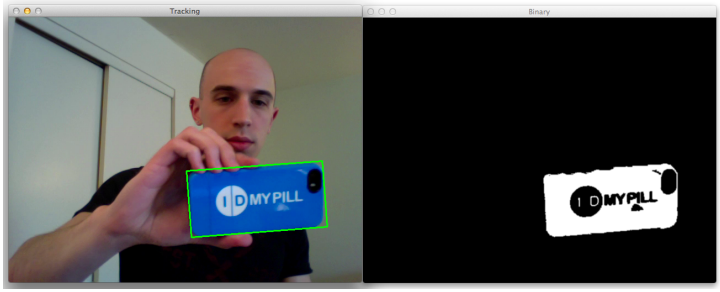


Figure 4.1: Finding and tracking an iPhone in a video. A bounding box is drawn around the tracked iPhone on the *left*, and the thresholded image is displayed on the *right*.

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

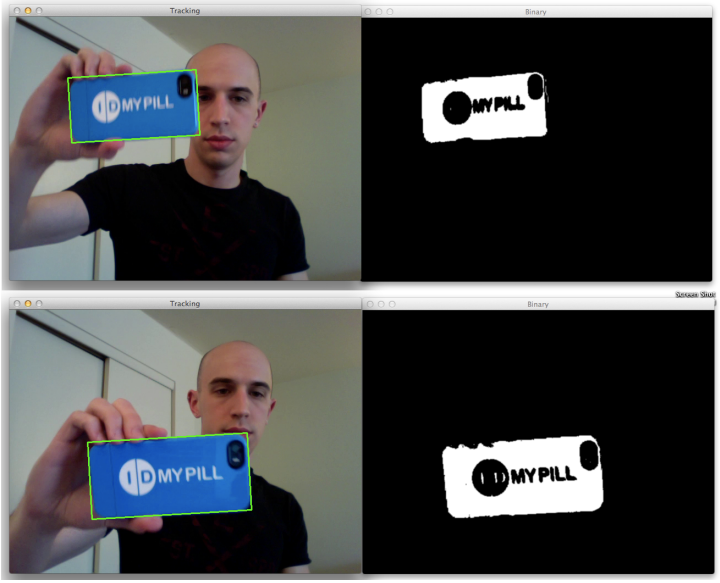


Figure 4.2: *Left:* Examples of the iPhone being successfully tracked in multiple frames of the video. *Right:* The thresholded image, with pixels falling into the `blueLower` and `blueUpper` range displayed as *white* and pixels not falling into the range as *black*.

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.

## Further Reading

A simple way to get started tracking an object in a video stream is to determine the color range of an object. To do this, we define the lower and upper boundaries of the object color in a particular color space, such as RGB, HSV, or L*a*b*.

But how do we go about actually *determining* what the lower and upper boundaries should be? The Chapter 4 supplementary material addresses this question in detail:

http://pyimg.co/bm02a

# 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

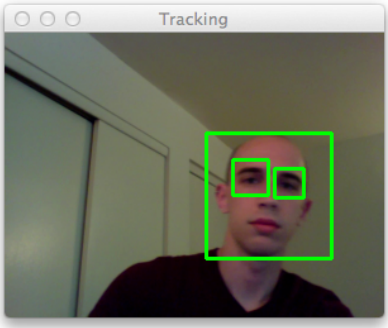


Figure 5.1: Detecting eyes in an image. First, the face must be detected. Then, the face area can be searched for eyes.

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

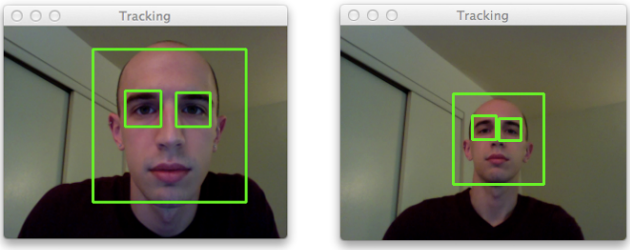


Figure 5.2: Even as the face and eyes move closer to the camera and farther away, Laura's code is still able to track them in the video

```
$ 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.

Stealing another glance at the time, Laura notices that it's 4 pm.

She made it!

Committing her code repository, Laura sits back and smiles to herself – her first day on the job and she already finished a tough project. That would definitely get the boss' attention!

This job was going to be even more rewarding than she thought.

## Further Reading

Just as we applied Haar cascades to detect and track the presence of faces in an image/video stream, we can do the same for eyes. However, now that you have detected the eyes, you might also be interested in learning how to detect pupils as well.

Inside the supplementary material for Chapter 5, I provide a reference to one of my personal favorite pupil tracking tutorials:

http://pyimg.co/7kcen

# 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

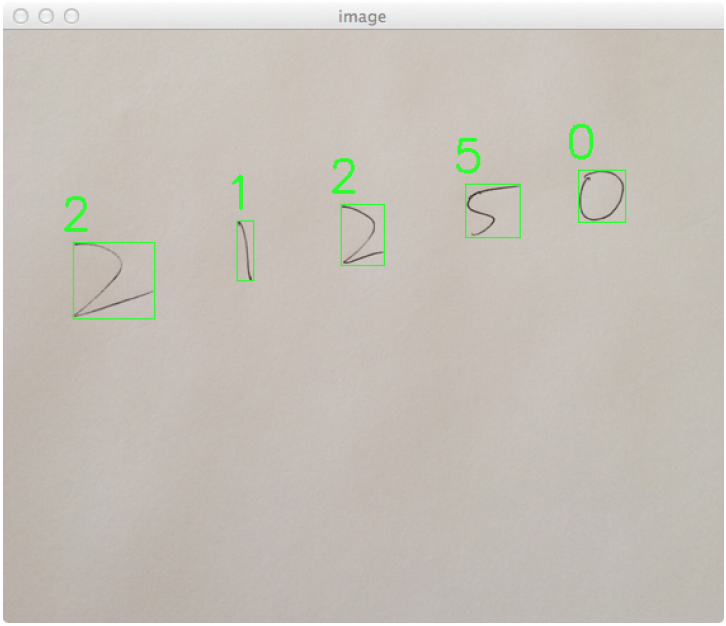


Figure 6.1: Hank's goal is to build a system that can identify handwritten digits, but he's having some trouble. Let's see if we can help him out.

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 $28 \times 28$ 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

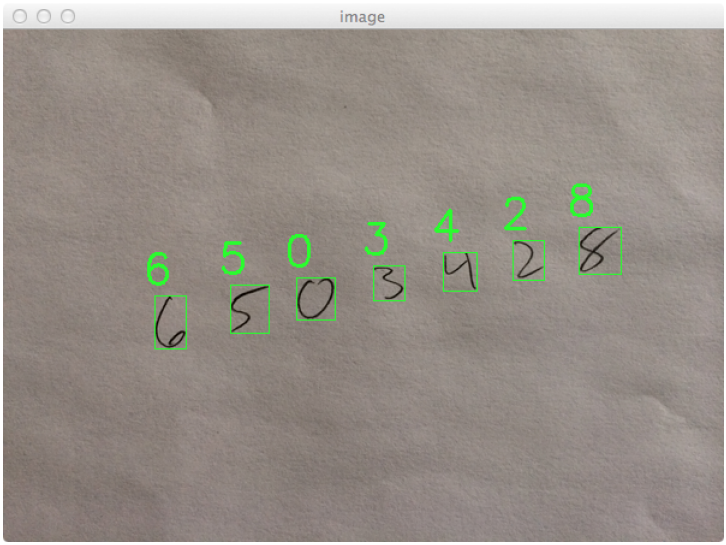


Figure 6.2: Classifying the handwritten digits of a cellphone number was a success! Hank's on the right track.

```
$ 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

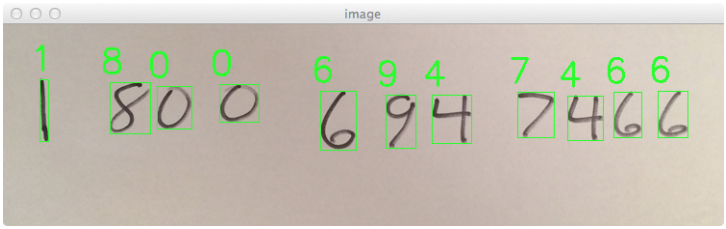


Figure 6.3: Just in case Hank needs to call Apple's technical support line, he tests his classifier on another phone number.

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.

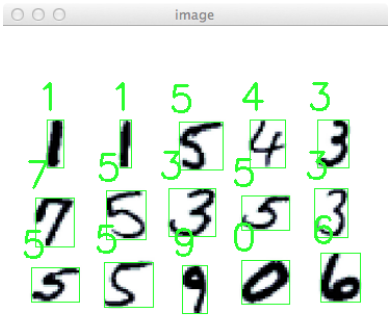


Figure 6.4: Hank evaluates his digit classifier on a sample of handwritten digits gathered from Google Image Search.

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.

## Further Reading

This chapter discussed how to recognize handwritten digits using the Histogram of Oriented Gradients image descriptor and a bit of machine learning. However, HOG is not the *only* method that can be used to recognize digits.

In fact, you can train an entire end-to-end system to recognize digits in images using only the raw pixel data – to do this, you need to apply *Deep Learning* and *Convolutional Neural Networks*:

http://pyimg.co/oelad

# 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.

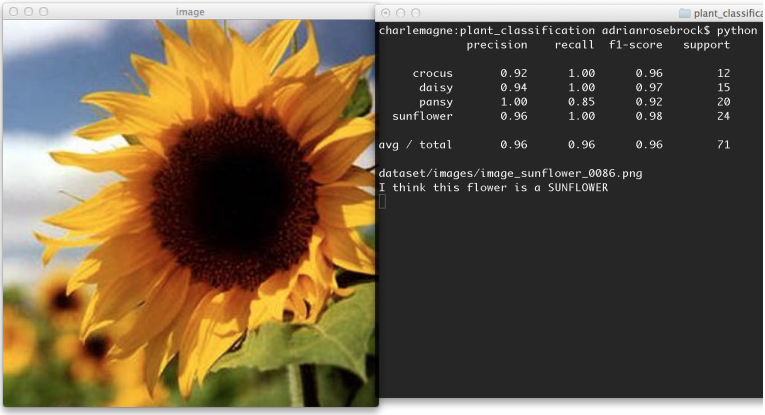


Figure 7.1: Charles' goal is to classify plant species using color histograms and machine learning. This model is able to correctly classify the image on the left as a sunflower.

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.model_selection 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-

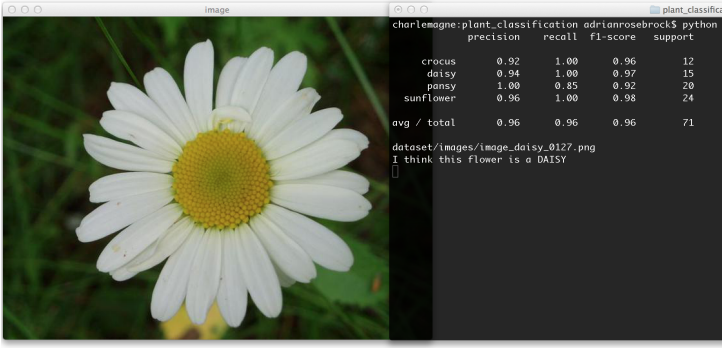


Figure 7.2: Charles has used color histograms a random forest classifier to obtain a high accuracy plant classifier.

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
```

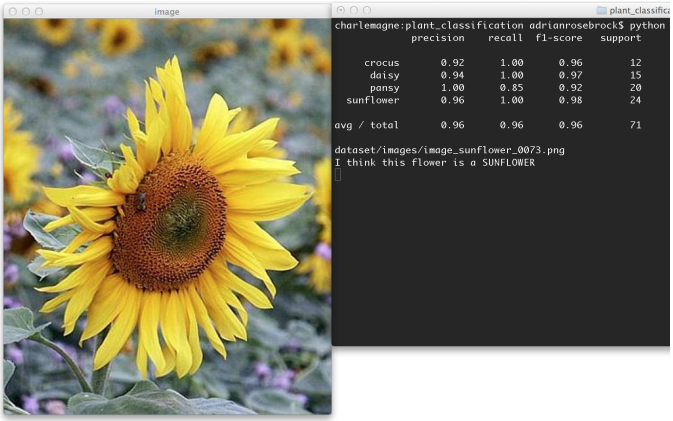


Figure 7.3: Charles' classifier correctly labels the flower as a sunflower.

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.

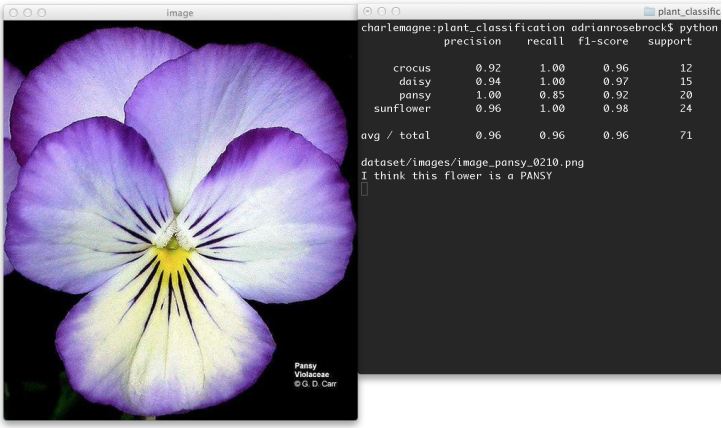


Figure 7.4: The flower is correctly labeled as a pansy.

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.

## Further Reading

When going through this chapter, you may have noticed that we had the *masks* for each flower, allowing us to determine *where* in the image the flower is. How did we obtain these masks? And how can you generate masks of your own for complex objects?

To find out the answers to these questions, please see the Chapter 7 supplementary material:

http://pyimg.co/oelad

# 8

## BUILDING AN AMAZON.COM COVER SEARCH

"*Wu-Tang Clan* has a bigger vocabulary than Shakespeare," argued Gregory, taking a draw of his third Marlboro menthol of the morning as the dull San Francisco light struggled to penetrate the dense fog.

This was Gregory's favorite justification for his love of the famous hip-hop group. But this morning, Jeff, his cofounder, wasn't up for arguing. Instead, Jeff was focused on integrating Google Analytics into their newly finished company website, which at the moment, consisted of nothing more than a logo and a short "About Us" section.

Five months ago, Gregory and Jeff quit their full-time jobs and created a startup focused on visual recognition. Both of them were convinced that they had the "next big idea."

So far, they weren't making a money. And they couldn't afford an office either.

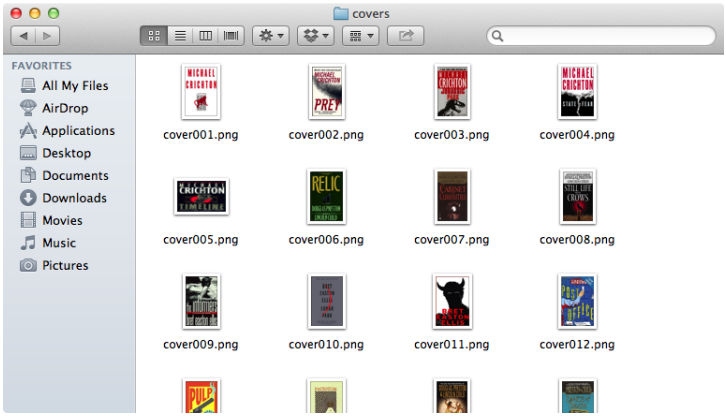


Figure 8.1: A sample of Gregory's book cover database. He has collected images for a small number of books and then plans on recognizing them using keypoint matching techniques.

But working from a San Francisco park bench next to a taco truck that serves kimchi and craft beers definitely has its upsides.

Gregory looked longingly at the taco truck. 10 am. It wasn't open yet. Which was probably a good thing. He had started to put on a few pounds after consuming five months' worth of tacos and IPAs. And he didn't have the funds to afford a gym membership to work those pesky pounds off, either.

See, Gregory's plan was to compete with the Amazon Flow app. Flow allows users to use their smartphone cam-

era as a digital shopping device. By simply taking a picture of a cover of a book, DVD, or video game, Flow automatically identifies the product and adds it to the user's shopping cart.

That's all fine and good. But Gregory wants to niche it down. He wants to cater *strictly* to the bookworms and do only book cover identification – and do it better than anyone on the market, including Amazon. That would get some attention.

With the thought of beating Amazon at their own bread and butter, Gregory opens his laptop, cracks his knuckles, slowly, one at a time, stubs his cigarette out, and gets to work.

Listing 8.1: coverdescriptor.py

```
1 import numpy as np
2 import cv2
3
4 class CoverDescriptor:
5     def __init__(self, useSIFT = False):
6         self.useSIFT = useSIFT
```

Gregory starts by importing the packages that he needs on **Lines 1 and 2**: NumPy for numerical processing and `cv2` for OpenCV bindings.

He then defines his `CoverDescriptor` class on **Line 4**, which encapsulates methods for finding keypoints in an image and then describes the area surrounding each keypoint using local invariant descriptors.

The `__init__` constructor is defined on **Line 5**, requiring one optional argument: `useSIFT`, a boolean indicating whether the SIFT keypoint detector and descriptor should be used or not.

## 8.1 KEYPOINTS, FEATURES, AND OPENCV 3 & 4

Now, before we get too far into this chapter, let's briefly discuss an important change to the organization of the OpenCV library.

In the 1st edition of *Practical Python and OpenCV + Case Studies*, our book cover identification system defaulted to using David Lowe's Difference of Gaussian (DoG) keypoint detector and the SIFT local invariant descriptor. [1]

However, the OpenCV 3.0 release brings some *very important* changes to the library – specifically to the modules that contain keypoint detectors and local invariant descriptors.

With the v3.0 release, OpenCV has moved SIFT, SURF, FREAK, and other keypoint detector and local invariant descriptor implementations into the optional `opencv_contrib` package. This move was to consolidate (1) experimental implementations of algorithms, and (2) what OpenCV calls "non-free" (i.e., patented) algorithms, which include many popular keypoint detectors and local invariant descriptors, into a 100% optional module that is not required for OpenCV to install and function. In short, if you have ever used the `cv2.FeatureDetector_create` or `cv2.DescriptorExtractor_create` functions from OpenCV 2.4.X, they are **no longer**

1 Lowe, David (1999). "Object recognition from local scale-invariant features". *Proceedings of the International Conference on Computer Vision*.

**part of OpenCV**. You can still access the free, non-patented methods such as ORB and BRISK, but if you need SIFT and SURF, you'll have to *explicitly enable* them at compile and install time.

Again, in order to obtain access to these keypoint detectors and descriptors, you will need to follow the OpenCV 3 install instructions for your appropriate operating system and Python version provided in Chapter 2 of *Practical Python and OpenCV*.

**Note:** *For more information regarding this change to OpenCV and the implications it has to keypoint detectors, local invariant descriptors, and which Python versions have access to which functions, I suggest you read through my detailed blog post which you can find here: http://pyimg.co/9vwm9.*

Since OpenCV is no longer shipped with the the SIFT module enabled automatically, we now supply a boolean to our `__init__` method called `useSIFT` with a default value of `False` to indicate that SIFT should only be used if the programmer **explicitly** wants it to be.

However, just because SIFT is not be a part of our OpenCV install doesn't mean we're out of luck – *far from it!*

OpenCV 3 has still retained many non-patented keypoint detectors and local invariant descriptors such as BRISK, ORB, KAZE, and AKAZE. Rather than using SIFT to build our book cover identification system like we did in the 1st edition of this book, we'll instead be using BRISK – the code will change slightly, but our results will still be the same.

## 8.2 IDENTIFYING THE COVERS OF BOOKS

Now that we have clarified an important difference between OpenCV 2.4.X and OpenCV 3.0, let's get back to Gregory's code:

Listing 8.2: coverdescriptor.py

```
8     def describe(self, image):
9         descriptor = cv2.BRISK_create()
10
11        if self.useSIFT:
12            descriptor = cv2.xfeatures2d.SIFT_create()
13
14        (kps, descs) = descriptor.detectAndCompute(image, None)
15        kps = np.float32([kp.pt for kp in kps])
16
17        return (kps, descs)
```

To extract both keypoints and descriptors from an image, Gregory defines the `describe` method on **Line 8**, which accepts a single parameter – the `image` in which keypoints and descriptors should be extracted from.

On **Line 9**, Gregory initializes his `descriptor` method to utilize `BRISK`. However, if he has set `useSIFT=True` in the constructor, then he re-initializes the `descriptor` to use `SIFT` on **Lines 11 and 12**.

Now that Gregory has his descriptor initialized, he makes a call to the `detectAndCompute` method on **Line 14**. As this name suggests, this method both *detects* keypoints (i.e., "interesting" regions of an image) and then *describes* and *quantifies* the region surrounding each of the keypoints. Thus, the keypoint detection is the "detect" phase, whereas the actual description of the region is the "compute" phase.

Again, the choice between SIFT and BRISK is dependent upon your OpenCV install. If you are using OpenCV 2.4.X, take a look at the 1st edition of this book included in the download of your bundle – it will detail how to use the `cv2.FeatureDetector_create` and `cv2.DescriptorExtractor_create` functions to obtain access to SIFT, SURF, BRISK, etc. However, if you are using OpenCV 3, then you'll want to use the `BRISK_create` method unless you have taken *explicit care* to install OpenCV 3 with the `xfeatures2d` module enabled.

Regardless of whether Gregory is using SIFT or BRISK, the list of keypoints contain multiple `KeyPoint` objects which are defined by OpenCV. These objects contain information such as the *(x, y)* location of the keypoint, the size of the keypoint, and the rotation angle, amongst other attributes.

For this application, Gregory only needs the *(x, y)* coordinates of the keypoint, contained in the `pt` attribute.

He grabs the *(x, y)* coordinates for the keypoints, discarding the other attributes, and stores the points as a NumPy array on **Line 15**.

Finally, a tuple of keypoints and corresponding descriptors are returned to the calling function on **Line 17**.

At this, point Gregory can extract keypoints and descriptors from the covers of books... but how is he going to compare them?

Let's check out Gregory's `CoverMatcher` class:

Listing 8.3: covermatcher.py

```
1 import numpy as np
2 import cv2
3
4 class CoverMatcher:
5     def __init__(self, descriptor, coverPaths, ratio = 0.7,
6         minMatches = 40, useHamming = True):
7         self.descriptor = descriptor
8         self.coverPaths = coverPaths
9         self.ratio = ratio
10        self.minMatches = minMatches
11        self.distanceMethod = "BruteForce"
12
13        if useHamming:
14            self.distanceMethod += "-Hamming"
```

Again, Gregory starts off on **Lines 1 and 2** by importing the packages he will need: NumPy for numerical processing and `cv2` for his OpenCV bindings.

The `CoverMatcher` class is defined on **Line 4** and the constructor on **Line 5**. The constructor takes two required parameters and three optional ones. The two required parameters are the `descriptor`, which is assumed to be an instantiation of the `CoverDescriptor` defined above, and the path to the directory where the cover images are stored.

The three optional arguments are detailed below:

- **ratio**: The ratio of nearest neighbor distances suggested by Lowe to prune down the number of keypoints a homography needs to be computed for.
- **minMatches**: The minimum number of matches required for a homography to be calculated.

- **useHamming**: A boolean indicating whether the Hamming or Euclidean distance should be used to compare feature vectors.

The first two arguments, `ratio` and `minMatches`, we'll discuss in more detail when we get to the `match` function below. However, the third argument, `useHamming`, is especially important and worth diving into now.

You see, it is important to note that `SIFT` and `SURF` produce *real-valued* feature vectors whereas `ORB`, `BRISK`, and `AKAZE` produce *binary* feature vectors. When comparing real-valued descriptors, like `SIFT` or `SURF`, we would want to use the Euclidean distance. However, if we are using `BRISK` features which produce binary feature vectors, the Hamming distance should be used instead. Again, your choice in feature descriptor above (`SIFT` vs. `BRISK`) will also influence your choice in distance method. However, since Gregory is defaulting to `BRISK` features which produce binary feature vectors, he'll indicate that the Hamming method should be used (again, by default) on **Lines 13 and 14**.

This wasn't too exciting. Let's check out the `search` method and see how the keypoints and descriptors will be matched:

Listing 8.4: covermatcher.py

```
16     def search(self, queryKps, queryDescs):
17         results = {}
18
19         for coverPath in self.coverPaths:
20             cover = cv2.imread(coverPath)
21             gray = cv2.cvtColor(cover, cv2.COLOR_BGR2GRAY)
22             (kps, descs) = self.descriptor.describe(gray)
23
24             score = self.match(queryKps, queryDescs, kps, descs)
```

```
25            results[coverPath] = score
26
27        if len(results) > 0:
28            results = sorted([(v, k) for (k, v) in results.items()
                  if v > 0],
29                reverse = True)
30
31        return results
```

Gregory defines his `search` method on **Line 16**, requiring two arguments – the set of keypoints and descriptors extracted from the *query image*. The goal of this method is to take the keypoints and descriptors from the query image and then match them against a database of keypoints and descriptors. The entry in the database with the best "match" will be chosen as the identification of the book cover.

To store his results of match accuracies, Gregory defines a dictionary of `results` on **Line 17**. The key of the dictionary will be the unique book cover filename and the value will be the matching percentage of keypoints.

Then, on **Line 19**, Gregory starts looping over the list of book cover paths. The book cover is loaded from disk on **Line 20**, converted to grayscale on **Line 21**, and then keypoints and descriptors are extracted from it using the `CoverDescriptor` passed into the constructor on **Line 22**.

The number of matched keypoints is then determined using the `match` method (defined below) and the `results` dictionary updated on **Lines 24-25**.

Gregory does a quick check to make sure that at least some results exist on **Line 27**. Then the results are sorted in *descending* order, with book covers with more keypoint

matches placed at the top of the list.

The sorted `results` are then returned to the caller on **Line 31**.

Now, let's take a look at how Gregory matched his keypoints and descriptors:

Listing 8.5: covermatcher.py

```
33    def match(self, kpsA, featuresA, kpsB, featuresB):
34        matcher = cv2.DescriptorMatcher_create(self.
              distanceMethod)
35        rawMatches = matcher.knnMatch(featuresB, featuresA, 2)
36        matches = []
37
38        for m in rawMatches:
39            if len(m) == 2 and m[0].distance < m[1].distance *
                  self.ratio:
40                matches.append((m[0].trainIdx, m[0].queryIdx))
41
42        if len(matches) > self.minMatches:
43            ptsA = np.float32([kpsA[i] for (i, _) in matches])
44            ptsB = np.float32([kpsB[j] for (_, j) in matches])
45            (_, status) = cv2.findHomography(ptsA, ptsB, cv2.
                  RANSAC, 4.0)
46
47            return float(status.sum()) / status.size
48
49        return -1.0
```

Gregory starts by defining his `match` method on **Line 33**. This method takes four parameters, detailed below:

- **kpsA**: The list of keypoints associated with the first image to be matched.
- **featuresA**: The list of feature vectors associated with the first image to be matched.

- **kpsB**: The list of keypoints associated with the second image to be matched.
- **featuresB**: The list of feature vectors associated with the second image to be matched.

Gregory then defines his `matcher` on **Line 34** using the `cv2.DescriptorMatcher_create` function. This value will either be `BruteForce` or `BruteForce-Hamming`, indicating that he is going to compare every descriptor in `featuresA` to every descriptor in `featuresB` using either the Euclidean or Hamming distance and taking the feature vectors with the smallest distance as the "match."

The actual matching is performed on **Line 35** using the `knnMatch` function of `matcher`. The "kNN" portion of the function stands for "k-Nearest-Neighbor," where the "nearest neighbors" are defined by the smallest Euclidean distance between feature vectors. The two feature vectors with the smallest Euclidean distance are considered to be "neighbors." Both `featuresA` and `featuresB` are passed into the `knnMatch` function, with a third parameter of `2`, indicating that Gregory wants to find the two nearest neighbors for each feature vector.

The output of the `knnMatch` method is stored in `rawMatches`. However, these are not the *actual* mapped keypoints! Gregory still has a few more steps to take.

The first of which is to initialize the list of actual `matches` on **Line 36**.

From there, Gregory starts to loop over the `rawMatches` on **Line 38**.

He makes a check on **Line 39** to ensure two cases hold. The first is that there are indeed two matches. The second is to apply David Lowe's ratio test, by ensuring the distance between the first match is less than the distance of the second match, times the supplied `ratio`.

This test helps remove false matches and prunes down the number of keypoints the homography needs to be computed for, thus speeding up the entire process.

Assuming that the ratio test holds, the `matches` list is updated with a tuple of the index of the first keypoint and the index of the second keypoint.

Gregory then makes a second important check on **Line 42** – he ensures that the number of `matches` is at least the number of minimum matches. If there are not enough matches, it is not worth computing the homography since the two images (likely) do not contain the same book cover.

Again, assuming that this test holds, Gregory defines two lists, `ptsA` and `ptsB` on **Lines 43 and 44**, to store the *(x, y)* coordinates for each set of matched keypoints.

Finally, Gregory can compute the homography, which is a mapping between the two keypoint planes with the same center of projection.

Effectively, this algorithm will take his `matches` and determine which keypoints are indeed a "match" and which

ones are false positives.

To accomplish this, Gregory uses the `cv2.findHomography` function and the RANSAC algorithm, which stands for Random Sample Consensus.

Without a background in linear algebra, explaining the RA- NSAC algorithm can be quite cumbersome. But in the most basic terms, RANSAC randomly samples from a set of potential matches. In this case, RANSAC randomly samples from Gregory's `matches` list. Then, RANSAC attempts to match these samples together and verifies the hypothesis of whether or not the keypoints are inliers. RANSAC continues to do this until a large enough set of `matches` are considered to be inliers. From there, RANSAC takes the set of inliers and looks for more matches.

The important takeaway is that the RANSAC algorithm is *iterative*. It randomly samples potential matches and then determines if they are indeed matches. It continues this process until a stopping criterion is reached.

The RANSAC algorithm is implemented in the `cv2.findHomography` function, which accepts four arguments. The first two are `ptsA` and `ptsB`, the *(x, y)* coordinates of the potential matches from **Lines 43 and 44**.

The third argument is the homography method. In this case, Gregory passes in `cv2.RANSAC` to indicate that he wants to use the RANSAC algorithm. Alternatively, he could have used `cv2.LMEDS`, which is the Least-Median robust method.

The final parameter is the RANSAC re-projection threshold, which allows for some "wiggle room" between keypoints. Assuming that the *(x, y)* coordinates for `ptsA` and `ptsB` are measured in *pixels*, Gregory passes in a value `4.0` to indicate that an error of `4.0` pixels will be tolerated for any pair of keypoints to be considered an inlier.

Choosing between `cv2.RANSAC` and `cv2.LMEDS` is normally dependent upon the domain of the problem. While the `cv2.LM- EDS` method has the benefit of not having to explicitly define the re-projection threshold, the downside is that it normally only works correctly when at least 50% of the keypoints are inliers.

The `cv2.findHomograpy` function returns a tuple of two values. The first is the transformation matrix, which Gregory ignores.

He is more interested in the second returned value, the `status`. The `status` variable is a list of booleans, with a value of `1` if the corresponding keypoints in `ptsA` and `ptsB` were matched, and a value of `0` if they were not.

Gregory computes the ratio of the number of inliers to the total number of potential matches on **Line 47** and returns it to the caller. A high score indicates a better "match" between two images.

Finally, if the minimum number of matches test on **Line 42** failed, **Line 49** returns a value of `-1.0`, indicating that the number of inliers could not be computed.

After all this work, Gregory can now glue all the pieces together:

Listing 8.6: search.py

```
1  from __future__ import print_function
2  from pyimagesearch.coverdescriptor import CoverDescriptor
3  from pyimagesearch.covermatcher import CoverMatcher
4  import argparse
5  import glob
6  import csv
7  import cv2
8
9  ap = argparse.ArgumentParser()
10 ap.add_argument("-d", "--db", required = True,
11     help = "path to the book database")
12 ap.add_argument("-c", "--covers", required = True,
13     help = "path to the directory that contains our book covers")
14 ap.add_argument("-q", "--query", required = True,
15     help = "path to the query book cover")
16 ap.add_argument("-s", "--sift", type = int, default = 0,
17     help = "whether or not SIFT should be used")
18 args = vars(ap.parse_args())
19
20 db = {}
21
22 for l in csv.reader(open(args["db"])):
23     db[l[0]] = l[1:]
```

On **Lines 1-7**, Gregory imports the packages he will be using. The `CoverDescriptor` will extract keypoints and local invariant descriptors from the images, whereas the `CoverMatcher` will determine how well two book covers "match."

The `argparse` package will be used to parse command line arguments, `glob` to grab the path to the book cover images, `csv` to parse the .csv database of books, and `cv2` for OpenCV bindings.

| | A | B | C |
|---|---|---|---|
| 1 | cover001.png | Michael Crichton | Next |
| 2 | cover002.png | Michael Crichton | Prey |
| 3 | cover003.png | Michael Crichton | Jurassic Park |
| 4 | cover004.png | Michael Crichton | State of Fear |
| 5 | cover005.png | Michael Crichton | Timeline |
| 6 | cover006.png | Preston and Child | Relic |
| 7 | cover007.png | Preston and Child | The Cabinent of Curiosities |
| 8 | cover008.png | Preston and Child | Still Life With Crows |
| 9 | cover009.png | Bret Easton Ellis | The Informers |
| 10 | cover010.png | Bret Easton Ellis | Lunar Park |
| 11 | cover011.png | Bret Easton Ellis | Imperial Bedrooms |
| 12 | cover012.png | Charles Bukowski | Post Office |
| 13 | cover013.png | Charles Bukowski | Pup |
| 14 | cover014.png | Charles Bukowski | Factotum |
| 15 | cover015.png | Preston and Child | The Book of the Dead |
| 16 | cover016.png | Preston and Child | Dance of Death |
| 17 | cover017.png | George Orwell | 1984 |

Figure 8.2: Gregory's CSV database of book information. The attributes, from left to right, are the unique filename of the book cover, the author of the book, and the title of the book.

**Lines 9-18** handle parsing Gregory's command line arguments. The `--db` points to the location of the book database CSV file, while `--covers` is the path to the directory that contains the book cover images. The `--query` switch is the path to Gregory's query image. Finally, the optional `--sift` switch is used to indicate whether or not the SIFT method should be used rather than the BRISK algorithm (BRISK will be used by default). The goal here is to take the query image and find the book cover in the database with the best match.

Building the database of book information is handled on **Lines 20-23**. First, the `db` dictionary is defined. Then, the book database CSV file is opened and each line looped over. The `db` dictionary is updated with the unique filename of the book as the key and the title of the book and author as the value.

Most of this code is general housekeeping and initialization. Let's take a look at something more interesting:

Listing 8.7: search.py

```
25 useSIFT = args["sift"] > 0
26 useHamming = args["sift"] == 0
27 ratio = 0.7
28 minMatches = 40
29
30 if useSIFT:
31     minMatches = 50
```

The first thing Gregory does in this code snippet is determine if the SIFT algorithm should be used in place of the (default) BRISK algorithm. If the `--sift` command line argument was supplied with a value $> 0$, we'll use SIFT;

otherwise, we'll default to BRISK on **Line 25**.

Now that the choice between BRISK and SIFT has been settled, we can also determine whether or not the Hamming distance should be used on **Line 26**. If we are using the SIFT algorithm, then we'll be extracting *real-valued* feature vectors – thus the Euclidean distance should be used. However, if we are using the BRISK algorithm, then we'll be computing *binary* feature vectors, and the Hamming distance should be used instead.

**Lines 27 and 28** then initialize the default values for Lowe's ratio test and minimum number of matches.

In the case that we are using the SIFT algorithm we, we'll add the extra constraint that more matches should be found on **Lines 30 and 31** to ensure a more accurate book cover identification.

Listing 8.8: search.py

```
32 cd = CoverDescriptor(useSIFT = useSIFT)
33 cv = CoverMatcher(cd, glob.glob(args["covers"] + "/*.png"),
34     ratio = ratio, minMatches = minMatches, useHamming =
           useHamming)
35
36 queryImage = cv2.imread(args["query"])
37 gray = cv2.cvtColor(queryImage, cv2.COLOR_BGR2GRAY)
38 (queryKps, queryDescs) = cd.describe(gray)
39
40 results = cv.search(queryKps, queryDescs)
41
42 cv2.imshow("Query", queryImage)
```

Here, Gregory has instantiated his `CoverDescriptor` on **Line 32** and his `CoverMatcher` on **Line 32**, passing in his

`CoverDescriptor` and list of book cover paths as arguments.

The query image is then loaded and converted to grayscale on **Lines 36 and 37**.

Next, Gregory extracts his keypoints and local invariant descriptors from the query image on **Line 38**.

In order to perform the actual matching, the `search` method of the `CoverMatcher` class is called, where Gregory supplies the query keypoints and query descriptors. A sorted list of `results` is returned, with the best book cover matches placed at the top of the list.

Finally, Gregory displays the query image to the user on **Line 42**.

Now, let's take a look at how Gregory is going to display the results to the user:

Listing 8.9: search.py

```
44 if len(results) == 0:
45     print("I could not find a match for that cover!")
46     cv2.waitKey(0)
47
48 else:
49     for (i, (score, coverPath)) in enumerate(results):
50         (author, title) = db[coverPath[coverPath.rfind("/") +
               1:]]
51         print("{}. {:.2f}% : {} - {}".format(i + 1, score * 100,
52             author, title))
53
54         result = cv2.imread(coverPath)
55         cv2.imshow("Result", result)
56         cv2.waitKey(0)
```

First, Gregory makes a check on **Line 34** to ensure that at least one book cover match was found. If a match was not found, Gregory lets the user know.

If a match was found, he then starts to loop over the `results` on **Line 49**.

The unique filename of the book is extracted, and the author and book title are grabbed from the book database on **Line 50** and displayed to the user on **Line 51**.

Finally, the actual book cover itself is loaded off disk and displayed to the user on **Lines 54-56**.

Gregory executes his script using the following command:

Listing 8.10: search.py

```
$ python search.py --db books.csv --covers covers --query queries
    /query01.png
```

The results of Gregory's script can be seen in Figure 8.3, where Gregory attempts to match the cover of Preston and Child's *Dance of Death* (*left*) to the corresponding cover in his database (*right*).

As Figure 8.3 demonstrates, the covers were successfully matched together, with over 97% of the keypoints matched as well. Indeed, out of all the 50 book covers, Gregory's code was able to correctly identify the cover of the book without a problem.

Gregory then moves on to another book cover, this time Michael Crichton's *Next* in Figure 8.4. Again, the query image is on the *left* and the successful match on the *right*.

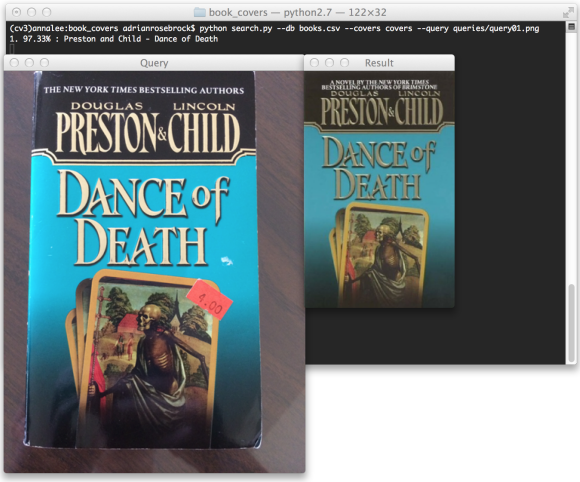


Figure 8.3: *Left:* The query image Gregory wants to match in his database. *Right:* The book cover in his database with the most matches (97.33%). The book cover has been successfully matched.


Figure 8.4: Gregory now attempts to match Crichton's *Next*. Even despite the laptop in the background and the hand covering part of the book, 73.47% of the keypoints were matched.


Figure 8.5: *Jurassic Park* is Gregory's favorite book. Sure enough, his algorithm is able to match the cover to his book database without an issue.

Despite the laptop in the background and the hand covering part of the book, 73.47% of the keypoints were matched. Once again, Gregory's algorithm has successfully identified the cover of the book!

*Jurassic Park* is Gregory's favorite book. So, why not use it as a query image?

Figure 8.5 demonstrates that Gregory's keypoint matching code can successfully match the *Jurassic Park* book cov-

ers together. Yet another successful identification!

Time for a real test.

Gregory takes a photo of *State of Fear* and uses it as his query image in Figure 8.6. Notice how the book is angled and there is text at the top of the cover (i.e., "New York Times Bestseller") that does not appear in the cover database.

Regardless, his cover match algorithm is robust enough to handle this situation and a successful match is found.

Finally, Gregory attempts to match another Preston and Child book, *The Book of the Dead*, in Figure 8.7. Despite the 30% off stickers in the upper right hand corner and substantially different lighting conditions, the two covers are matched without an issue.

"Ghostface Killah is the best member of *Wu-Tang Clan*," Gregory announced to Jeff, turning his laptop to show in the results.

But this time, Jeff could not ignore him.

Gregory had actually done it. He had created a book cover identification algorithm that was working with beautifully.

"Next stop, acquisition by Amazon!", exclaimed Jeff. "Now, let's get some tacos and beers to celebrate."


Figure 8.6: Notice how the book is not only angled in the query image, but there is also text on the cover that doesn't exist in the image database. Nonetheless, Gregory's algorithm is still easily able to identify the book.

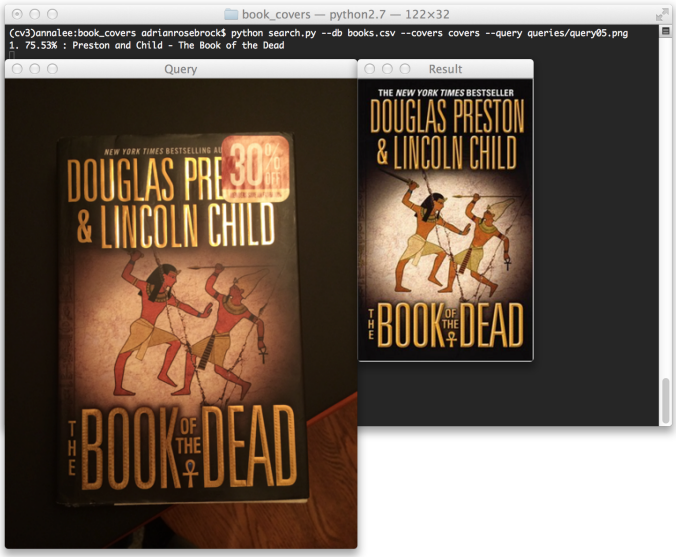


Figure 8.7: Despite the 30% off sticker and the substantially different lighting conditions, *The Book of the Dead* book cover is easily matched.

## Further Reading

Keypoint detectors and local invariant descriptors are just the start to building scalable image search engines. The problem is that such systems scale *linearly*, meaning that as more images are added to our system, *the longer it will take to perform a search*.

So, how do we overcome this issue and build image search engines that can scale to *millions* of images? The answer is to apply the bag of visual words (BOVW) model:

http://pyimg.co/8klea

# 9

# CONCLUSION

In this book, we've explored real-world computer vision problems and their solutions, including face detection, object tracking, handwriting recognition, classification, and keypoint matching.

So, you may be wondering... where do you go from here?

If you're looking to take the *next logical step* in your computer vision journey, I would encourage you to join the PyImageSearch Gurus course:

http://pyimg.co/lu2j3

This course takes a deeper dive into computer vision, image processing, and deep learning, covering more advanced algorithms and applications, including (but not limited to):

- Automatic License Plate Recognition (ANPR)
- Deep Learning and Convolutional Neural Networks
- Face Recognition
- Training Your Own Custom Object Detectors

- Scaling Image Search Engines to *millions* of Images
- Hadoop + Big Data for Computer Vision
- Image Classification + Machine Learning
- *...and much more!*

To learn more about the PyImageSearch Gurus course, and grab the course syllabus and free sample lessons, just use this link:

http://pyimg.co/lu2j3

I look forward to seeing you inside the course!

Finally, if you need help coming up with project ideas, be sure to contact me. I love talking with readers and helping out when I can. You can reach me at adrian@pyimagesearch.com.

I'll also note that I constantly post on my blog, www.PyImageSearch.com, detailing new and interesting techniques related to computer vision, image processing, and deep learning.

Be sure to sign up for my newsletter to receive exclusive tips, tricks, and hacks that I don't publish anywhere else!

Until then,

-Adrian Rosebrock