JUL 7TH, 2014 | COMMENTS

Face Recognition With Python, in Under 25 Lines of Code

The following is a guest post by Shantnu Tiwari, who has worked in the low level/embedded domain for ten years. Shantnu suffered at the hands of C/C++ for several years before he discovered Python, and it felt like a breath of fresh air. He is now sharing his love (http://pythonforengineers.com/).

In this post we'll look at a surprisingly simple way to get started with face recognition using Python and the open source library OpenCV (http://opencv.org/).

Before you ask any questions in the comments section:

1. Do not skip over the blog post and try to run the code. You must understand what the code does not only to run it properly but to troubleshoot it as well.
2. Make sure to use OpenCV v2.
3. You need a working webcam for this script to work properly.
4. Review the other comments/questions as your questions have probably already been addressed.

Thank you.

OpenCV

OpenCV is the most popular library for computer vision. Originally written in C/C++, it now provides bindings for Python.

OpenCV uses machine learning algorithms to search for faces within a picture. For something as complicated as a face, there isn't one simple test that will tell you if it found a face or not. Instead, there are thousands of small patterns/features that must be matched. The algorithms break the task of identifying the face into thousands of smaller, bite-sized tasks, each of which is easy to solve. These tasks are also called classifiers (http://en.wikipedia.org/wiki/Statistical_classification).

For something like a face, you might have 6,000 or more classifiers, all of which must match for a face to be detected (within error limits, of course). But therein lies the problem: For face detection, the algorithm starts at the top left of a picture and moves down across small blocks of data, looking at each block, constantly asking, "*Is this a face? … Is this a face? … Is this a face*?" Since there are 6,000 or more tests per block, you might have millions of calculations to do, which will grind your computer to a halt.

To get around this, OpenCV uses cascades (http://docs.opencv.org/modules/objdetect/doc/cascade_classification.html). What's a cascade? The best answer can be found from the dictionary (http://dictionary.reference.com/browse/cascade): *A waterfall or series of waterfalls*

Like a series of waterfalls, the OpenCV cascade breaks the problem of detecting faces into multiple stages. For each block, it does a very rough and quick test. If that passes, it does a slightly more detailed test, and so on. The algorithm may have 30-50 of these stages or cascades, and it will only detect a face if all stages pass. The advantage is that the majority of the pictures will return negative during the first few stages, which means the algorithm won't waste time testing all 6,000 features on it. Instead of taking hours, face detection can now be done in real time.

Cascades in practice

Though the theory may sound complicated, in practice it is quite easy. The cascades themselves are just a bunch of XML files that contain OpenCV data used to detect objects. You initialize your code with the cascade you want, and then it does the work for you.

Since face detection is such a common case, OpenCV comes with a number of built-in cascades for detecting everything from faces to eyes to hands and legs. There are even cascades for non-human things. For example, if you run a banana shop and want to track people stealing bananas, this guy (http://coding-robin.de/2013/07/22/train-your-own-opencv-haar-classifier.html) has built one for that!

Installing OpenCV

First, you need to find the correct setup file for your operating system (http://opencv.org/downloads.html).

I found that installing OpenCV was the hardest part of the task. If you get strange unexplainable errors, it could be due to library clashes, 32/64 bit differences, etc. I found it easiest to just use a Linux virtual machine and install OpenCV from scratch.

Once installed, you can test whether or not it works by firing up a Python session and typing:

```
1 $ python
2 >>> import cv2
3 >>>
```

If you don't get any errors, you can move on to the next part.

Understanding the code

Let's break down the actual code, which you can download from the repo (https://github.com/shantnu/FaceDetect/). Grab the *face_detect.py* script, the *abba.png* pic, and the *haarcascade_frontalface_default.xml*.

```
1  # Get user supplied values
2  imagePath = sys.argv[1]
3  cascPath = sys.argv[2]
```

You first pass in the image and cascade names as command-line arguments. We'll use the Abba image as well as the default cascade for detecting faces provided by OpenCV.

```
1  # Create the haar cascade
2  faceCascade = cv2.CascadeClassifier(cascPath)
```

Now we create the cascade and initialize it with our face cascade. This loads the face cascade into memory so it's ready for use. Remember, the cascade is just an XML file that contains the data to detect faces.

```
1  # Read the image
2  image = cv2.imread(imagePath)
3  gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

Here we read the image and convert it to grayscale. Many operations in OpenCv are done in grayscale.

```
1  # Detect faces in the image
2  faces = faceCascade.detectMultiScale(
3      gray,
4      scaleFactor=1.1,
5      minNeighbors=5,
6      minSize=(30, 30),
7      flags = cv2.cv.CV_HAAR_SCALE_IMAGE
8  )
```

This function detects the actual face – and is the key part of our code, so let's go over the options.

1. The `detectMultiScale` function (http://docs.opencv.org/modules/objdetect/doc/cascade_classification.html#cascadeclassifier-detectmultiscale) is a general function that detects objects. Since we are calling it on the face cascade, that's what it detects. The first option is the grayscale image.

2. The second is the `scaleFactor`. Since some faces may be closer to the camera, they would appear bigger than those faces in the back. The scale factor compensates for this.
3. The detection algorithm uses a moving window to detect objects. `minNeighbors` defines how many objects are detected near the current one before it declares the face found. `minSize`, meanwhile, gives the size of each window.

> I took commonly used values for these fields. In real life, you would experiment with different values for the window size, scale factor, etc., until you find one that best works for you.

The function returns a list of rectangles where it believes it found a face. Next, we will loop over where it thinks it found something.

```
1  print "Found {0} faces!".format(len(faces))
2
3  # Draw a rectangle around the faces
4  for (x, y, w, h) in faces:
5      cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)
```

This function returns 4 values: the x and y location of the rectangle, and the rectangle's width and height (*w*, *h*).

We use these values to draw a rectangle using the built-in `rectangle()` function.

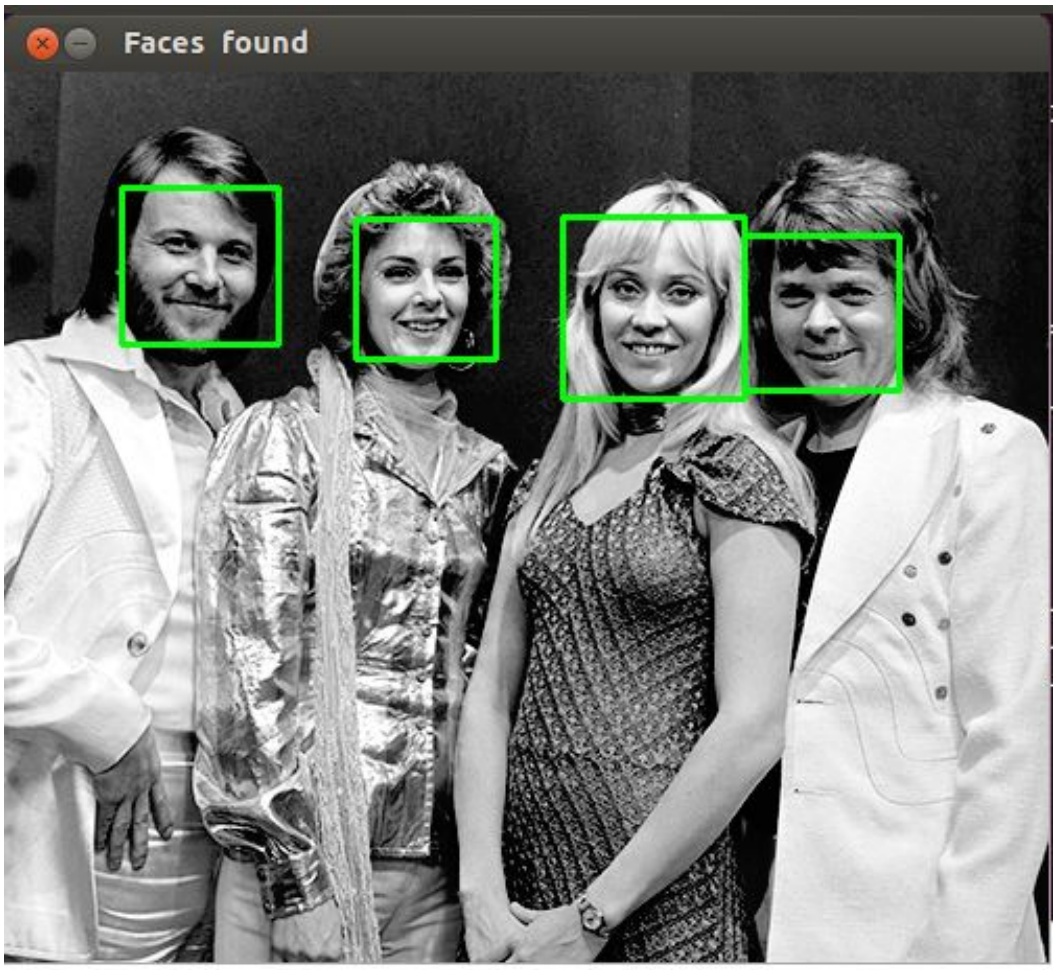
```
1  cv2.imshow("Faces found" ,image)
2  cv2.waitKey(0)
```

In the end, we display the image, and wait for the user to press a key.

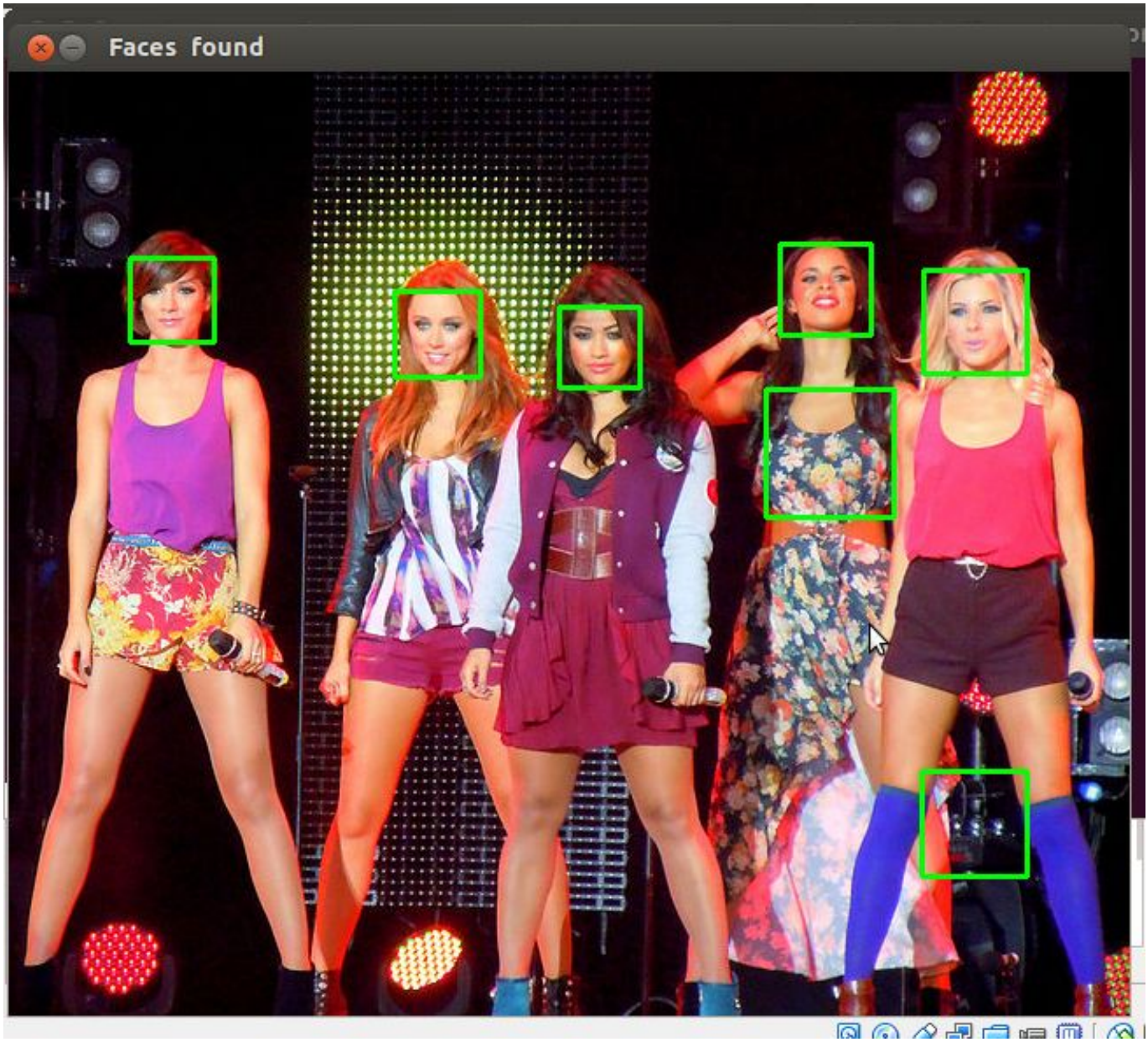
Checking the results

Let's test against the Abba photo:

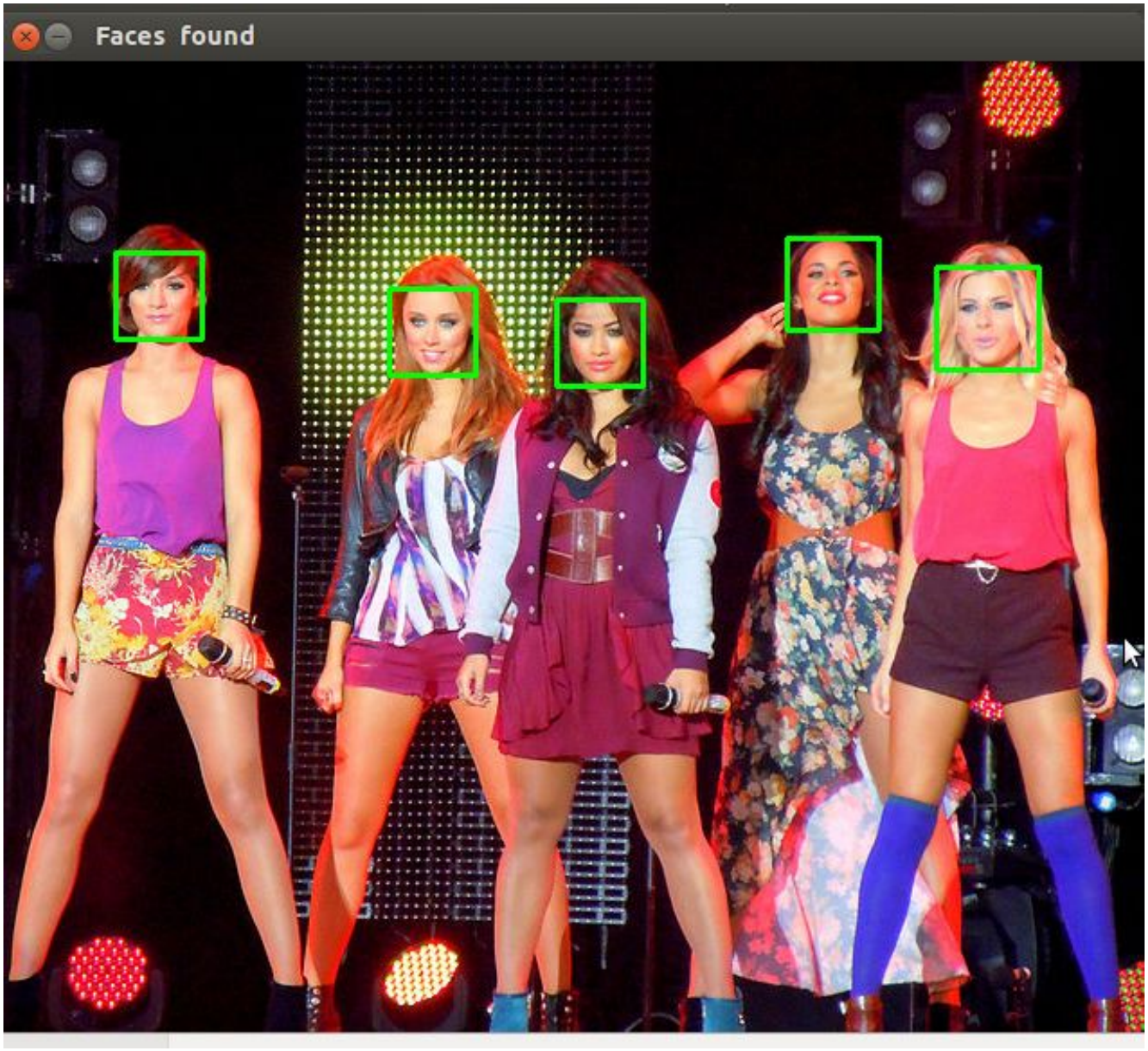
```
1  $ python face_detect.py abba.png haarcascade_frontalface_default.xml
```



That worked. How about another photo:



That ... is not a face. Let's try again. I changed the parameters and found that setting the `scaleFactor` to 1.2 got rid of the *wrong* face.



What happened? Well, the first photo was taken fairly close up with a high quality camera. The second one seems to have been taken from afar and possibly from a mobile phone. This is why the `scaleFactor` had to be modified. As I said, you'll have to setup the algorithm on a case by case basis to avoid false positives.

Be warned though that since this is based on machine learning, the results will never be 100% accurate. You will get *good enough* results in most cases, but occasionally the algorithm will identify incorrect objects as faces.

The final code can be found here (https://github.com/shantnu/FaceDetect).

Extending to a webcam

So what if you want to use a webcam? OpenCV grabs each frame from the webcam and you can then detect faces by processing each frame. You will need a powerful computer, though my five year old laptop seems to cope fine, as long as I don't dance around too much.

UPDATED The next blog post is live: Face Detection in Python Using a Webcam (https://realpython.com/blog/python/face-detection-in-python-using-a-webcam/). Check it out!

Want to know more?

I will be covering this and more in my upcoming book Python for Science and Engineering, which is currently on Kickstarter (https://www.kickstarter.com/projects/513736598/python-for-science-and-engineering). I will also cover machine learning, for those who are interested in it.

Thanks!

Posted by Real Python Jul 7th, 2014 data science (/blog/categories/data-science/), opencv (/blog/categories/opencv/), python (/blog/categories/python/)

See an error in this post? Please submit a pull request on Github (https://github.com/realpython/realpython-blog).

Want to learn more? Download the Real Python course.

Download Now » $60 (https://app.simplegoods.co/i/IQCZADOY)

Or, click here (http://www.realpython.com/) to learn more about the course.

« Discover Flask, Part 2 - Creating a login page (/blog/python/introduction-to-flask-part-2-creating-a-login-page/)

Django Migrations - A Primer » (/blog/python/django-migrations-a-primer/)

Comments

Featured Comment

michaelherman Mod • 4 months ago

You can find some code improvements here >> https://gist.github.com/46bit/...

Share ›

27 Comments Real Python 1 Login