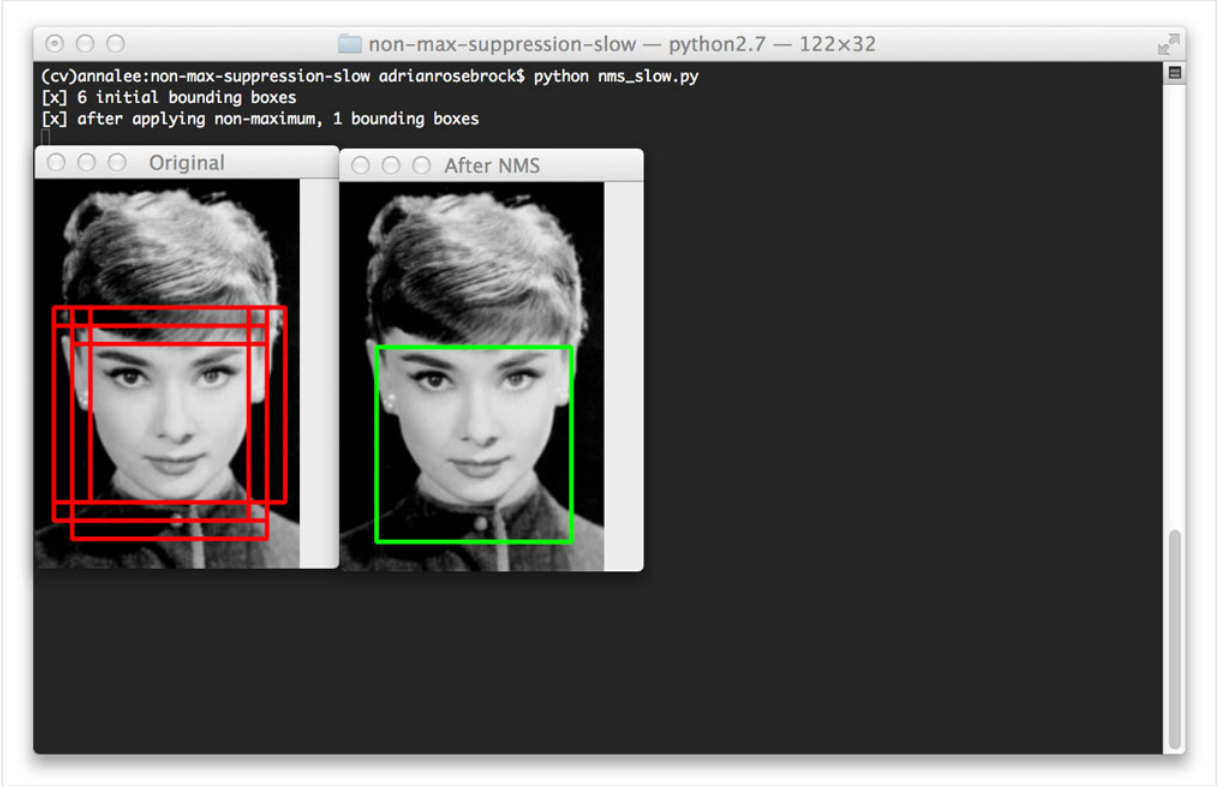
Navigation

pyimagesearch
be awesome at building image search engines

Non-Maximum Suppression for Object Detection in Python

by **Adrian Rosebrock** on November 17, 2014 in **Machine Learning**, **Tutorials**



Connecticut is cold. Very cold. Sometimes it's hard to even get out of bed in the morning. And honestly, without the aide of copious amounts of pumpkin spice lattes and the beautiful sunrise over the crisp autumn leaves, I don't think I would leave my cozy bed.

But I have work to do. And today that work includes writing a blog post about Felzenszwalb et al. method for non-maximum suppression.

If you remember, last week we discussed Histogram of Oriented Gradients for Objection Detection.

This method can be broken into a **6-step process**, including:

1. Sampling positive images
2. Sampling negative images

3. Training a Linear SVM
4. Performing hard-negative mining
5. Re-training your Linear SVM using the hard-negative samples
6. Evaluating your classifier on your test dataset, utilizing non-maximum suppression to ignore redundant, overlapping bounding boxes

After applying these steps you'll have an object detector that is as smooth as, well, John Coltrane:

Figure 1: My Python object detection framework applied to face detection. Even in low contrast images, faces can be easily detected.

(Note: Images utilized in this post were taken from the MIT + CMU Frontal Face Images dataset)

These are the bare minimum steps required to build an object classifier using Histogram of Oriented Gradients. Extensions to this method exist including Felzenszwalb et al.'s deformable parts model and Malisiewicz et al.'s Exemplar SVM.

However, no matter which HOG + Linear SVM method you choose, you will (with almost 100% certainty) detect multiple bounding boxes surrounding the object in the image.

For example, take a look at the image of Audrey Hepburn at the top of this post. I forked my Python framework for object detection using HOG and a Linear SVM and trained it to detect faces. Clearly, it has found Ms. Hepburns face in the image — but the detection fired a total of **six** times!

While each detection may in fact be valid, I certainty don't want my classifier to report to back to me saying that it found **six faces** when there is clearly only **one face**. Like I said, this is common "problem" when utilizing object detection methods.

In fact, I don't even want to call it a "problem" at all! It's a good problem to have. It indicates that your detector is working as expected. It would be far worse if your detector either (1) reported a false positive (i.e. detected a face where one wasn't) or (2) failed to detect a face.

To fix this situation we'll need to apply Non-Maximum Suppression (NMS), also called Non-Maxima Suppression.

When I first implemented my Python object detection framework I was unaware of a good Python implementation for Non-Maximum Suppression, so I reached out to my friend Dr. Tomasz Malisiewicz, whom I consider to be the "go to" person on the topic of object detection and HOG.

Tomasz, being the all-knowing authority on the topic referred me to two implementations in MATLAB which I have since implemented in Python. We're going to review the first method by Felzenszwalb etl al. Then, next week, we'll review the (faster) non-maximum suppression method implemented by Tomasz himself.

So without very delay, let's get our hands dirty.

Looking for the source code to this post?
Jump right to the downloads section.

OpenCV and Python versions:
This example will run on **Python 2.7/Python 3.4+** and **OpenCV 2.4.X/OpenCV 3.0+**.

Non-Maximum Suppression for Object Detection in Python

Open up a file, name it `nms.py` , and let's get started implementing the Felzenszwalb et al. method for non-maximum suppression in Python:

Non-Maximum Suppression for Object Detection in Python — Python

```python
# import the necessary packages
import numpy as np

#  Felzenszwalb et al.
def non_max_suppression_slow(boxes, overlapThresh):
    # if there are no boxes, return an empty list
    if len(boxes) == 0:
        return []

    # initialize the list of picked indexes
    pick = []

    # grab the coordinates of the bounding boxes
    x1 = boxes[:,0]
    y1 = boxes[:,1]
    x2 = boxes[:,2]
    y2 = boxes[:,3]
```

```
18
19      # compute the area of the bounding boxes and sort the bounding
20      # boxes by the bottom-right y-coordinate of the bounding box
21      area = (x2 - x1 + 1) * (y2 - y1 + 1)
22      idxs = np.argsort(y2)
```

We'll start on **Line 2** by importing a single package, NumPy, which we'll utilize for numerical processing.

From there we define our `non_max_suppression_slow` function on **Line 5**. this function accepts to arguments, the first being our set of bounding boxes in the form of *(startX, startY, endX, endY)* and the second being our overlap threshold. I'll discuss the overlap threshold a little later on in this post.

Lines 7 and 8 make a quick check on the bounding boxes. If there are no bounding boxes in the list, simply return an empty list back to the caller.

From there, we initialize our list of picked bounding boxes (i.e. the bounding boxes that we would like to keep, discarding the rest) on **Line 11**.

Let's go ahead and unpack the *(x, y)* coordinates for each corner of the bounding box on **Lines 14-17** — this is done using simple NumPy array slicing.

Then we compute the area of each of the bounding boxes on **Line 21** using our sliced *(x, y)* coordinates.

Be sure to pay close attention to **Line 22**. We apply `np.argsort` to grab the indexes of the ***sorted*** coordinates of the ***bottom-right y-coordinate*** of the bounding boxes. It is *absolutely critical* that we sort according to the bottom-right corner as we'll need to compute the overlap ratio of other bounding boxes later in this function.

Now, let's get into the meat of the non-maxima suppression function:

Non-Maximum Suppression for Object Detection in Python — Python

```
24      # keep looping while some indexes still remain in the indexes
25      # list
26      while len(idxs) > 0:
27          # grab the last index in the indexes list, add the index
28          # value to the list of picked indexes, then initialize
29          # the suppression list (i.e. indexes that will be deleted)
30          # using the last index
31          last = len(idxs) - 1
32          i = idxs[last]
33          pick.append(i)
34          suppress = [last]
```

We start looping over our indexes on **Line 26**, where we will keep looping until we run out of indexes to examine.

From there we'll grab the length of the `idx` list o **Line 31**, grab the value of the last entry in the `idx` list on **Line 32**, append the index `i` to our list of bounding boxes to keep on **Line 33**, and finally initialize our `suppress` list (the list of boxes we want to ignore) with index of the last entry of the index list on **Line 34**.

That was a mouthful. And since we're dealing with indexes into a index list it's not exactly an easy thing to explain. But definitely pause here and examine these code as it's important to understand.

Time to compute the overlap ratios and determine which bounding boxes we can ignore:

Non-Maximum Suppression for Object Detection in Python — Python

```
36          # loop over all indexes in the indexes list
37          for pos in xrange(0, last):
38              # grab the current index
39              j = idxs[pos]
40
41              # find the largest (x, y) coordinates for the start of
42              # the bounding box and the smallest (x, y) coordinates
43              # for the end of the bounding box
44              xx1 = max(x1[i], x1[j])
45              yy1 = max(y1[i], y1[j])
46              xx2 = min(x2[i], x2[j])
47              yy2 = min(y2[i], y2[j])
48
49              # compute the width and height of the bounding box
50              w = max(0, xx2 - xx1 + 1)
51              h = max(0, yy2 - yy1 + 1)
52
53              # compute the ratio of overlap between the computed
54              # bounding box and the bounding box in the area list
55              overlap = float(w * h) / area[j]
56
57              # if there is sufficient overlap, suppress the
58              # current bounding box
59              if overlap > overlapThresh:
60                  suppress.append(pos)
61
62          # delete all indexes from the index list that are in the
63          # suppression list
64          idxs = np.delete(idxs, suppress)
65
66      # return only the bounding boxes that were picked
67      return boxes[pick]
```

Here we start looping over the (remaining) indexes in the `idx` list on **Line 37**, grabbing the value of the current index on **Line 39**.

Using ***last*** entry in the `idx` list from **Line 32** and the ***current*** entry in the `idx` list from **Line 39**, we find the ***largest*** *(x, y)* coordinates for the start bounding box and the ***smallest*** *(x, y)* coordinates for the end of the bounding box on **Lines 44-47**.

Doing this allows us to find the current smallest region inside the larger bounding boxes (and hence why it's so important that we initially sort our `idx` list according to the bottom-right y-coordinate). From there, we compute the width and height of the region on **Lines 50 and 51**.

So now we are at the point where the overlap threshold comes into play. On **Line 55** we compute the `overlap`, which is a ratio defined by the area of the current smallest region divided by the area of current bounding box, where "current" is defined by the index `j` on **Line 39**.

If the `overlap` ratio is greater than the threshold on **Line 59**, then we know that the two bounding boxes sufficiently overlap and we can thus suppress the current bounding box. Common values for

`overlapThresh` are normally between 0.3 and 0.5.

Line 64 then deletes the suppressed bounding boxes from the `idx` list and we continue looping until the `idx` list is empty.

Finally, we return the set of picked bounding boxes (the ones that were not suppressed) on **Line 67**.

Let's go ahead and create a driver so we can execute this code and see it in action. Open up a new file, name it `nms_slow.py` , and add the following code:

Non-Maximum Suppression for Object Detection in Python — Python

```
# import the necessary packages
from pyimagesearch.nms import non_max_suppression_slow
import numpy as np
import cv2

# construct a list containing the images that will be examined
# along with their respective bounding boxes
images = [
    ("images/audrey.jpg", np.array([
    (12, 84, 140, 212),
    (24, 84, 152, 212),
    (36, 84, 164, 212),
    (12, 96, 140, 224),
    (24, 96, 152, 224),
    (24, 108, 152, 236)])),
    ("images/bksomels.jpg", np.array([
    (114, 60, 178, 124),
    (120, 60, 184, 124),
    (114, 66, 178, 130)])),
    ("images/gpripe.jpg", np.array([
    (12, 30, 76, 94),
    (12, 36, 76, 100),
    (72, 36, 200, 164),
    (84, 48, 212, 176)]))]

# loop over the images
for (imagePath, boundingBoxes) in images:
    # load the image and clone it
    print "[x] %d initial bounding boxes" % (len(boundingBoxes))
    image = cv2.imread(imagePath)
    orig = image.copy()

    # loop over the bounding boxes for each image and draw them
    for (startX, startY, endX, endY) in boundingBoxes:
        cv2.rectangle(orig, (startX, startY), (endX, endY), (0, 0, 255), 2)

    # perform non-maximum suppression on the bounding boxes
    pick = non_max_suppression_slow(boundingBoxes, 0.3)
    print "[x] after applying non-maximum, %d bounding boxes" % (len(pick))

    # loop over the picked bounding boxes and draw them
    for (startX, startY, endX, endY) in pick:
        cv2.rectangle(image, (startX, startY), (endX, endY), (0, 255, 0), 2)

    # display the images
    cv2.imshow("Original", orig)
    cv2.imshow("After NMS", image)
    cv2.waitKey(0)
```

We start by importing our `non_max_suppression_slow` function on **Line 2**. I put this function in the `pyimagesearch` package for organizational purposes, but you can put the function wherever you see fit. From there, we import NumPy for numerical processing and `cv2` for our OpenCV bindings on **Lines 3-4**.

Then, we define a list of `images` on **Line 8**. This list consists of 2-tuples, where the first entry in the tuple is a path to an image and the second entry is the list of bounding boxes. These bounding boxes were obtained from my HOG + Linear SVM classifier detecting potential "faces" at varying locations and scales. Our goal is to take the set of bounding boxes for each image and apply non-maximum suppression.

We start by looping over the image path and bounding boxes on **Line 27** and load the image on **Line 30**.

To visualize the results of non-maximum suppression in action, we first draw the original (non-suppressed) bounding boxes on **Lines 34 and 35**.

We then apply non-maximum suppression on **Line 38** and draw the picked bounding boxes on **Lines 42-43**.

The resulting images are finally displayed on **Lines 46-48**.

Non-Maximum Suppression in Action

To see the Felzenszwalb et al. non-maximum suppression method in action, download the source code and accompanying images for this post from the bottom of this page, navigate to the source code directory, and issue the following command:

Non-Maximum Suppression for Object Detection in Python — Shell

```
$ python nms_slow.py
```

First, you'll see the Audrey Hepburn image:

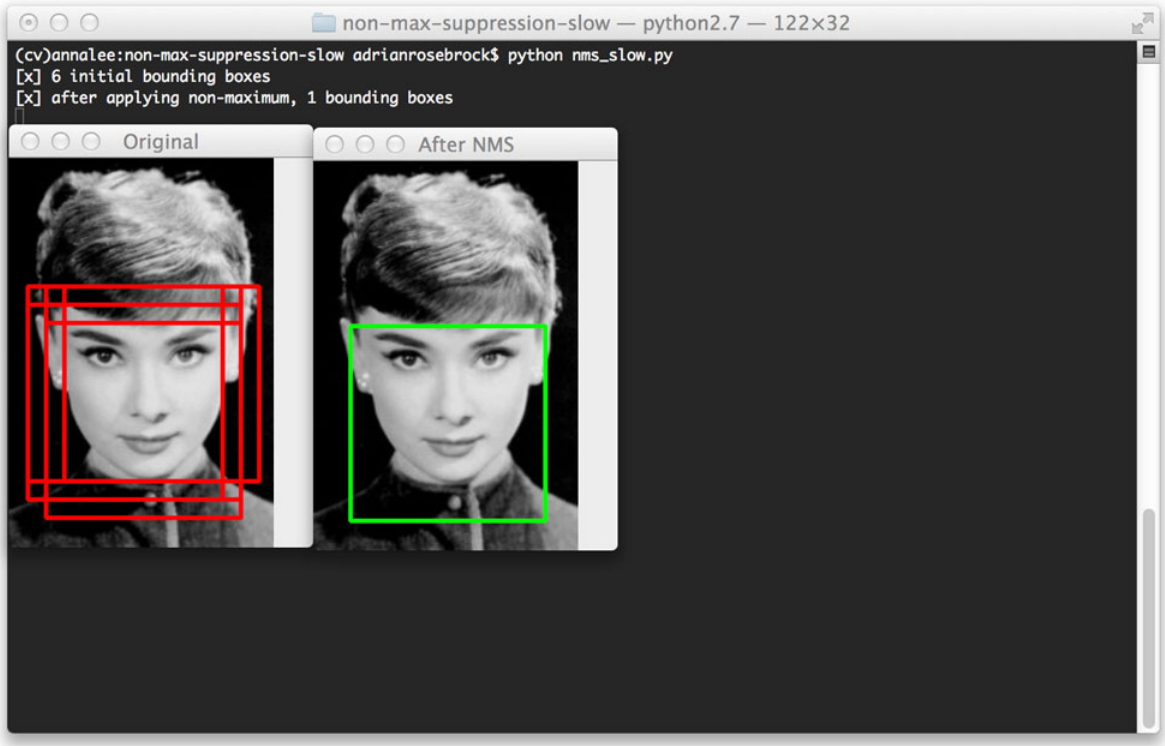


Figure 2: Our classifier initially detected **six** bounding boxes, but by applying non-maximum suppression, we are left with only **one** (the correct) bounding box.

Notice how **six** bounding boxes were detected, but by applying non-maximum suppression, we are able to prune this number down to **one**.

The same is true for the second image:

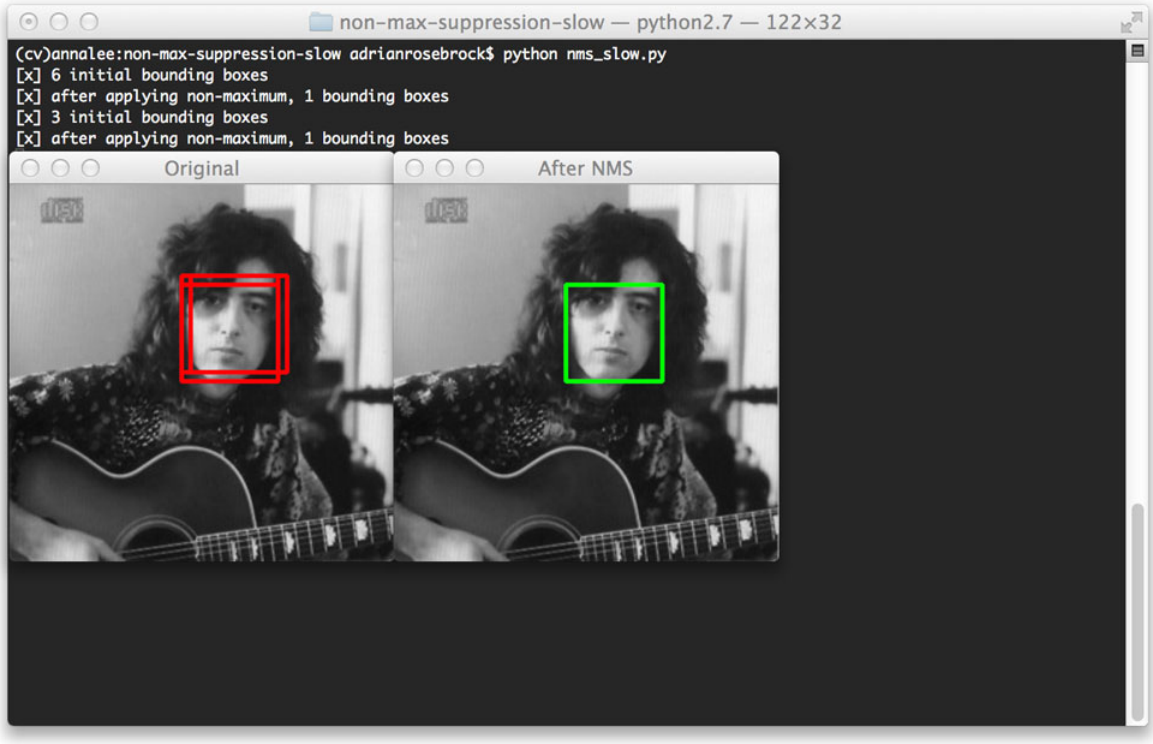


Figure 3: Initially detecting **three** bounding boxes, but by applying non-maximum suppression we can prune the number of overlapping bounding boxes down to **one**.

Here we have found **three** bounding boxes corresponding to the same face, but non-maximum suppression is about to reduce this number to **one** bounding box.

So far we have only examined images that contain one face. But what about images that contain multiple faces? Let's take a look:

Figure 4: Non-maximum suppression correctly handles when there are multiple faces, suppressing the smaller overlapping bounding boxes, but retaining the boxes that do not overlap.

Even for images that contain multiple objects, non-maximum suppression is able to ignore the smaller overlapping bounding boxes and return only the larger ones. Non-maximum suppression returns **two** bounding boxes here because the bounding boxes for each face at all. And even if they *did* overlap, do the overlap ratio does not exceed the supplied threshold of 0.3.

Summary

In this blog post I showed you how to apply the Felzenszwalb et al. method for non-maximum suppression.

When using the Histogram of Oriented Gradients descriptor and a Linear Support Vector Machine for object classification you almost *always* detect multiple bounding boxes surrounding the object you want to detect.

Instead of returning *all* of the found bounding boxes you should first apply non-maximum suppression to ignore bounding boxes that significantly overlap each other.

However, there are improvements to be made to the Felzenszwalb et al. method for non-maximum suppression.

In my next post I'll implement the method suggested by my friend Dr. Tomasz Malisiewicz which is reported to be over *100x* faster!

Be sure to download the code to this post using the form below! You'll definitely want to have it handy when we examine Tomasz's non-maximum suppression algorithm next week!

Downloads:

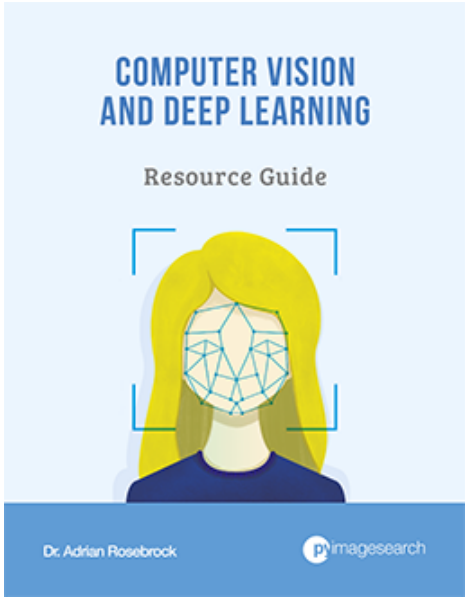
If you would like to download the code and images used in this post, please enter your email address in the form below. Not only will you get a .zip of the code, I'll also send you a **FREE 17-page Resource Guide on Computer Vision, OpenCV, and Deep Learning.** Inside you'll find my hand-picked tutorials, books, courses, and libraries to help you master CV and DL! Sound good? If so, enter your email address and I'll send you the code immediately!

Email address:

Your email address

DOWNLOAD THE CODE!

Resource Guide (it's totally free).



Enter your email address below to get my **free 17-page Computer Vision, OpenCV, and Deep Learning Resource Guide PDF**. Inside you'll find my hand-picked tutorials, books, courses, and Python libraries to help you master computer vision and deep learning!

Your email address

DOWNLOAD THE GUIDE!