Cormorant Bird Counting with User Guide and Python Script Documentation

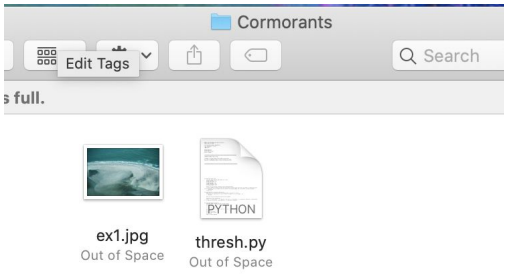
How to Use

Initial Setup:

1. Download the newest version of Python 3 to your computer from this website: https://www.python.org/downloads/.
    a. Here is a tutorial to download Python 3 for Windows: https://intellipaat.com/blog/tutorial/python-tutorial/how-to-download-and-install-python/.
    b. Here is a tutorial to download Python 3 for Mac: https://www.youtube.com/watch?v=0hGzGdRQeak.
2. Install PIL (a Python library)
3. Make a new folder called "Cormorants" and save it on your Desktop.
4. Download the file that is attached called "thresh.py" and place it inside your "Cormorants" folder.

To-do Each Time You Check a Picture:

1. Place the image in the "Cormorants" folder.

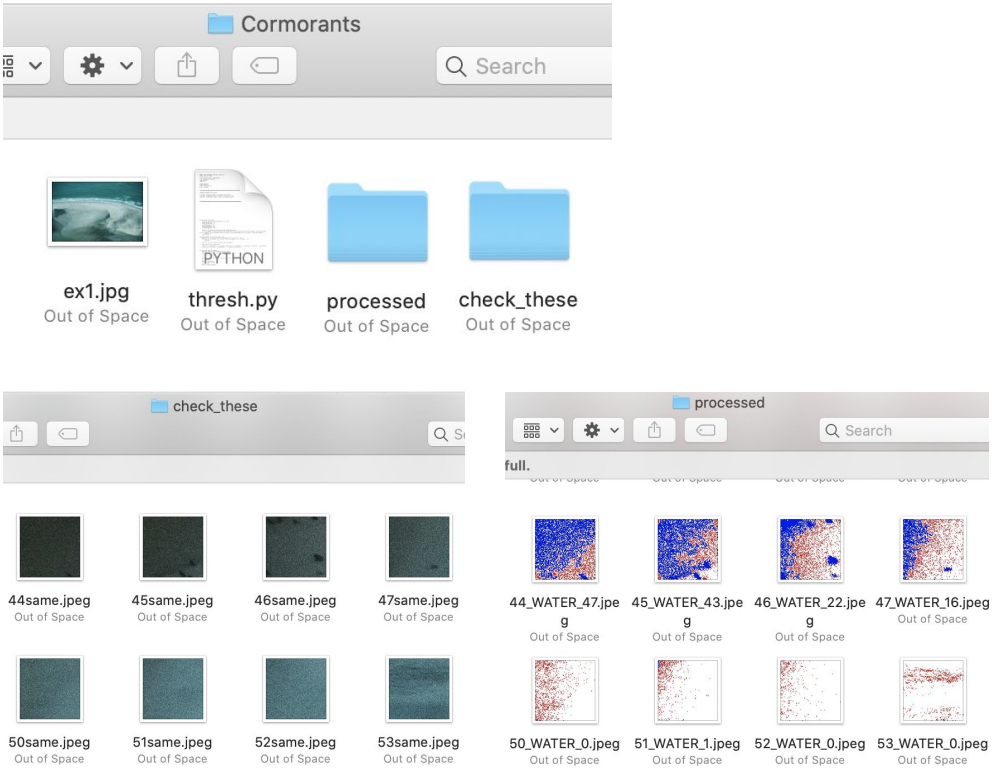


2. Open the file "thresh.py". You will have to adjust some parameters before processing each image.
    a. On line 19, there is a variable called "filename." Delete the current filename and type in your image's name, all in quotations. In the below example, the image's name is ' "ex3.jpg" ' (notice the quotations around the image name).

```
filename = "ex3.jpg"
```

    b. Next, you will run your code. This will likely take a few minutes. After the code runs fully, you should get an initial estimate for the number of birds. Unfortunately, **this estimate will be quite inaccurate**. In order to improve accuracy, you must adjust a couple other variables.

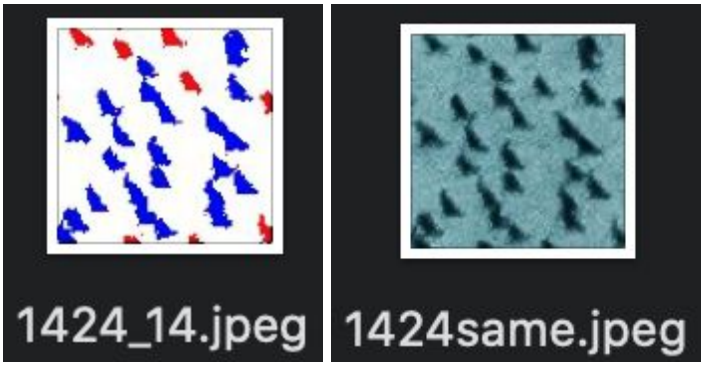
c. Once the code has run, you should see two folders in “Cormorants”: “processed” and “check_these.” Open each of these folders. Both of these folders contain 100-by-100 pixel subimages of your original image. However, “check _these” contains non-processed subimages and “processed” contains processed versions of the images in “check_these.” Example images of the folders “processed” and “check_these” are given below.



d. You must go through the images in these two folders and decide if the code has done a good job. To do this, notice that the name of the images within both folders begins with a number. The numbered image in “check_these” corresponds to the processed version of the image in “processed.” Look through at least 10 of these corresponding pairs (example in images directly above), and adhere to the following rationale:

   i. Scenario 1: The code is not counting very many birds, even in images that seem to have a lot of birds. You will notice that the processed image is colored with only red, white, and blue pixels. Blue blobs are entities that contribute to the bird count. Red blobs are entities that have the same color as birds, but were too small to add to the bird count. White pixels are the areas that are not birds. If there are birds in the “check_these” image that are colored red in the corresponding “processed” image, then the code is not counting all the birds. This indicates that the lower bound for bird size

is too high and is thus not counting some of the birds in the subimages. In the two images below, an example of this is provided.



If this is the case, then you should lower the value of "min_bird." To do this, go to the variable "min_bird" on line 20 and change it to some value that is lower (i.e., if it were initially at 30, lower it to 20). The lower image shows this line in the code. Notice that the value "60" is **not** in quotations in the code and **should not** be in quotations when you change it. Now, run the code again and **repeat step d.**

```
min_bird = 60
```

ii. Scenario 2: The code is counting blobs that are clearly not birds because they are too small. You will know this is the case if blobs that are very small and not birds are recolored blue in the "processed" images. To fix this, increase the "min_bird" variable on line 20 to some value that is higher (i.e., if it were initially at 20, increase it to 30). See the image directly above for the line to change. Notice that the value "60" is **not** in quotations in the code and **should not** be in quotations when you change it. Now, run the code again and **repeat step d.**

iii. Scenario 3: The code is counting birds well. Even in situations where birds are overlapping, the code seems to be doing well. In the two images below, an example of this case is provided. If this is the case, you can **move past step d and onto step e.**

Note: If you compare pairs of images from "processed" and "check_these" and the code is still not counting birds well, consider messing with the values of "fuzziness" (measure of extra noise from water) and "max_bird_multiplier" to get better results. Do not do this unless absolutely necessary.

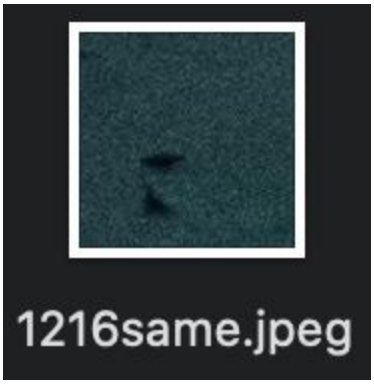
e. Now that the code is properly calibrated, run the code one last time. The code will print out the number of birds that it counted. **Record the number of birds.**

```
Run number 2036: 8383 birds in image
```

Limitations of this Method

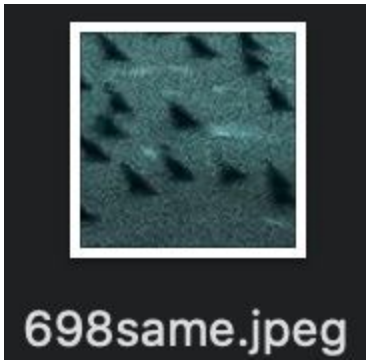
Notably, this method has trouble in a few cases:

1. Birds that are in very dark water. This recursive method uses color as the main distinguishing factor. When birds are in very dark water, they are too similar in color to the water, and will not be counted. **If you want to further improve accuracy**, you could go through and count the birds in the water and add that to the total that the code outputted. **If there is any question about whether birds in a certain section of water are counted, go into the "processed" folder and find the sub image you are concerned about. If the word "WATER" is in the image name, the birds in that image are *NOT* counted.** Below is an example of an image that wouldn't be counted for this reason.

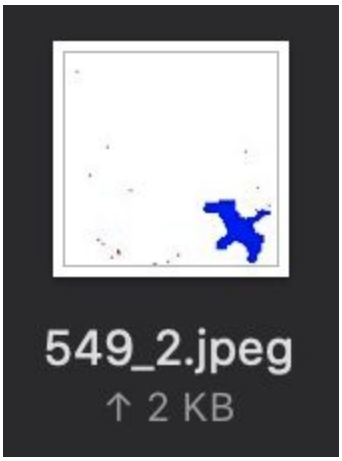


2. Birds that are in a picture that also has water in it (i.e., birds that are on the immediate shoreline). These images are too uncertain for the same reasons as in Case 1 and thus are

not used in the final bird count. Below is an example of an image that wouldn't be counted for this reason.



3. Birds with their wings spread sometimes get counted multiple times, but this doesn't affect the overall count much. In the image below this one bird is counted twice because the bird is so big that the program thinks two birds are overlapping.



Code Explanation

The code begins by dividing the inputted image into smaller 100 pixel by 100 pixel sub images. Then it increases the contrast, recolors the dark pixels (likely birds) black then red. The birds that are counted are recolored blue. The number of birds in each sub image are added to get the total number of birds in the original image. We will proceed to explain the code by defining each function and how it interacts with the other functions to count the number of birds in the image.

Header:

This is not a function but outside the functions, at the top. The user has the ability to change the values listed here including the file name and the minimum number of pixels in a bird.

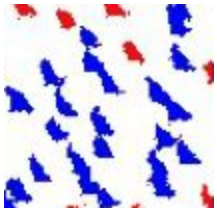
def bird_color(color_tup):

This function accepts the current pixel color being examined as a parameter called “color_tup.” Then it establishes the lower and upper bounds of RGB values for bird color. Next, it checks the red, green, and blue components of the current pixel’s color (“color_tup”) against the threshold values. If the current pixel color “color_tup” is within the bounds of the correct bird color, the function returns true. Otherwise it returns false.

def recur_bird(x, y, new_color, old_color):
This function takes advantage of a standard area fill recursive algorithm. Here let’s assume the “new_color” passed to the function is red and the “old_color” passed to the function is black. Given a single black pixel’s coordinates in a black and white image as a starting point, it recolors all other black pixels that are left, right, above, and below it red, so that all continuous black shapes are recolored red. The number of continuous black pixels (now recolored red) in a single shape is returned.

def check_bird(im3, my_counter):
First we increase the contrast of the image passed to this function. Next, the function establishes image properties pix, size, width, and height. Then, it sets starting values for a bird count, a count of continuous red blobs that are too small to be birds, and a bird dictionary. By iterating through all the pixels in the image, we populate the bird dictionary with all of the pixels that meet the color requirements to potentially be a bird according to the function “bird_color” (i.e. see if the current pixel is dark enough to be a bird). All pixels that are found to possibly be a part of a bird are recolored black and all other pixels are recolored white. What results is a black and white image with birds and some water colored black and everything else recolored white.

The next challenge is to determine what is a bird and what is likely water. In general the water is very fuzzy while the birds are distinct, roundish, black blobs. The function proceeds to look at each black pixel in the image (whose coordinates are housed in “bird_dict”) and recolor the surrounding black blob using the function “recur_bird.” “recur_bird” also returns the number of pixels that make up the black blob now recolored red. If this now red blob is made up of more pixels than the minimum number of pixels set in the “Header” by the user, then the red blob is recolored blue using the “recur_bird” function again. This is a bird.



The bird count for the current subimage is increased by one. We account for overlapping birds by dividing the number of pixels in the blob by the maximum number of pixels in a bird and then add the number of overlapping birds to the total number of birds for the current subimage. If the number of pixels in the current blob is less than the minimum number of pixels to be a bird, then we increase the count of "not_birds." By keeping track of continuous red blobs that are not big enough to be birds, we can distinguish the fuzzier water from the birds. Also, an error will be thrown and the number of birds reported in the subimage will be 0 if the program tries to identify a large mass of dark water as a bird ("RecursionError"). The image will have "REC" in its name.

Now to return the correct number of birds in the current subimage. If there are lots of collections of pixels that could be birds but many of these collections don't meet the threshold, the image probably contains water and should be discarded. The function returns 0 and the image will have "WATER" in its name. Otherwise the image is saved with the number of birds found in the image in its name. The number of birds in the current subimage is returned.

def make_white(file_path):
This function takes the file path of the image as a parameter and resizes the image so that it is divisible by 100, so that the image can be cropped into 100 pixel by 100 pixel subimages. It sets the overhanging portion of the image to white. The image below is the bottom-right corner cropped from an example larger image. The blue border was added to emphasize the added white space.

if __name__ == "__main__":
First, this function sets up new folders called "processed" and "check_these." Then the original image added by the user in "Header" is modified by the "make_white" function. The bird count is set to 0 initially and the program crops the original image into many subimages that are 100 pixels by 100 pixels in size. Each of these subimages is processed by the function "check_bird" which returns a count of birds in that particular subimage. The subimage bird count total is added to the running bird count ("num_of_birds"). The total number of birds in the original image is printed to the user.