The Image Project

Ng, TJHSST Cryptography, Fall 2019

This problem set is based on a simple goal: take an image, encode it so it looks like noise, then decode it back to the original image. This is harder than it sounds.

SETUP & LEARNING IMAGE MANIPULATION:

1) If you aren't on repl.it, you need to install the Python package "PIL". Get help if you aren't comfortable figuring out how to do that. Repl.it will download this package automatically when you use it; see what happens when you copy the code in #3.

2) Download all the files from blackboard, then add them to your project folder. On repl.it, there is a cloud icon labelled "upload or drop" to the right of the run button that you can use.

3) Copy this basic code on loading and manipulating bitmap files, which you will need. Note we are using true grayscale images for this assignment (sometimes called "luminosity bitmaps"), so that each pixel only stores one value, from 0 (black) to 255 (white) instead of the usual 3 separate R, G, B values.

```
from PIL import Image

im = Image.open("Disco.bmp")     #open an image
print(im.size)                   #retrieve the length & width as a tuple
print(im.getpixel((499,341)))    #get the value of a pixel (for this project, a single number from 0 to 255)
im.putpixel((499,341),0)         #change the value of a pixel in the Image object (to keep the change, save)
im.save("DiscoModified.bmp")     #save the image object as a new file
                                 #look carefully for one black pixel in the bottom left of the R in "STAR"
```

Please note: for this project, all files have been modified to be grayscale images only, with one value for each pixel, from 0 to 255, defining only its brightness. The vast majority of images you find will be in RGB format, which you have learned about before. To convert an RGB image to a true grayscale image, use this code:

```
im = Image.open("filename").convert("L")  #"L" is the shorthand for the luminosity-only format we are using
im.save("new filename")
```

4) At this point, you should understand how to loop over every individual pixel in a grayscale image, change them one by one, and save the result. If you feel like the example code was not enough to demonstrate the necessary methods, ask for help!

ASSIGNMENT:

Please make a Word document (or similar file) where you can answer these questions and paste output images. IMPORTANT: Please paste the required images small enough that the entire document ***is no more than two pages (or one sheet of paper, double-sided)*** – no one wants you to waste a ton of printer ink printing pictures that look like static!

1) Your first challenge is to implement an easy, but spectacularly useless, encryption method – the equivalent of a Caesar cipher on the 0-255 luminosity values.
   a. Create a method that takes an Image object and an integer. It should loop over every pixel in the image, add the integer value, simplify in mod 256, and replace the pixel's previous value with the result.
   b. Run it on "Disco.bmp" with a value of 53, and add that image to your document.
   c. In your document, explain why this method would be so useless at trying to disguise the contents of an image.

2) We can clearly do better. Let's employ our most recent cipher – the Hill Cipher. Applying this to an image will require writing new methods, but they will actually be easier than encoding letters because the pixel values are already integers.
   a. Write a method that takes an Image object and a 2x2 matrix. It should read ***across*** each line of the image (not ***down***), selecting pairs of pixels (starting with (0,0) and (1,0)). Each pair of pixel values should be encoded to a new pair of numbers using the usual Hill Cipher in mod 256. Then, write the values back onto the image object in the same place, replacing the previous values. Be careful to save with a different file name than you started with!
   (Notice this means that your image needs to have an even number of pixels in each row. All of the provided images have this property; if you wish to find others, be aware.)
   b. Encrypt "SC.bmp" using the matrix [[2, 5], [3, 20]]. Add the resulting image to your document.
   c. Is this better than the Caesar shift equivalent? Is it completely successful at hiding the contents of the image?

   NOTE: Since the mod we're working in is 256, and the only prime factor of 256 is 2, any matrix with an odd determinant will be invertible. Keep this in mind.

3) Now let's go the other way.
   a. Create a decoding method to match the encode method in question 2. Make sure it decodes the image you created in question 2 back to SC.bmp. Try it with at least one other matrix aside from the given one.
   b. Look at "Mystery1.bmp". Is any part of it recognizable?
   c. It was encoded using the same matrix: [[2, 5], [3, 20]]. Decode, then add the resulting image to your document.

4) There is an augmentation to the Hill Cipher that provides much better results. It goes like this. Choose an initial matrix and also a vector of size equivalent to one row of your matrix. Encode the first values with the matrix as usual. Then, multiply the first row of the matrix by the vector (component by component), and use the resulting matrix to encode the next pair of values. Then, multiply the second row of the matrix by the vector and use the resulting matrix to encode the third pair of values. Keep alternating which row is multiplied as the encoding takes place.

   For example, if we began with [[2, 5], [3, 20]] and used the vector <3, 5>:
   - The first two pixels would be encoded with [[2, 5], [3, 20]]. Then, multiply row 1: [2 * 3, 5 * 5].
   - The next two pixels would thus be encoded with [[6, 25], [3, 20]]. Then the second row [3, 20] would be multiplied by the vector – [3 * 3, 20 * 5].
   - The next two pixels would thus be encoded with [[6, 25], [9, 100]]. Note that the top row was kept as [6, 25]. Next, that top row would be multiplied again…
   - The next two pixels would be encoded with [[18, 125], [9, 100]].
   - The next two pixels would be encoded with [[18, 125], [27, 244]]. (500 reduces to 244 in mod 256). Etc.

   This, of course, requires that you choose a vector that will continue to produce invertible matrices. With this mod, remember, any matrix with an odd determinant is invertible.

   NOTE: if you'd like to copy your initial matrix, so you can keep an original copy and a changed copy separately, you can "import copy" at the top of the Python file and then use the line "x = copy.deepcopy(y)" to make a duplicate of any object.

   a. What vectors can be used in this way, on a matrix that starts with an odd determinant, to keep producing matrices with odd determinants?
   b. Implement this algorithm, choose a vector that will work, and encode "SC.bmp" again. Add to your document.
   c. Is any part of it recognizable now? Is there any difference when different vectors are used?

5) Deciphering the above process is more straightforward than you might think. A decoding method starts with the same matrix and vector, and follows the same process, but at each step it inverts the resulting matrix and uses that to decode the next two pixels.
   a. Implement a deciphering method for the process in #4.
   b. Look at "Mystery2.bmp". Is any part of it recognizable?
   c. It was encoded using the same matrix – [[2, 5], [3, 20]] – and the vector <101, 141>. Decode it, and add the resulting image to your document.
   d. Look at "Mystery3.bmp", encoded with [[2, 5], [3, 20]], and the vector <5, 9>. Decode to verify your method works and add the resulting image to your document.

6) Much better. But now, use your matrix-and-vector encoding on "logo1.bmp" and "logo2.bmp". Even this is not fully successful with simple images! (Or complex ones with high contrast; look at "Lego.bmp").
   a. As a final challenge, use anything we've talked about all quarter (Affine, larger Hill ciphers, transposition, whatever – even any combination of the above) to make a better image encoding method than this one. You need a decode method, too - it doesn't work if it isn't decodable! Your encode method must produce an image of the same size as the original.
   b. Copy the encrypt and decrypt methods into your document, as well as outputs for "logo1.bmp" and "logo2.bmp".
   c. What makes your method better?

7) **Extension:** load RGB images, and write code to encode the R, G, and B values separately. Create examples using at least two of the above methods. Investigate what makes recognizable images and what does not, and type a paragraph describing the results of your investigation. Include your example images.

   To do this, you will probably need to look up more methods for manipulating these kinds of images in the pillow documentation at https://pillow.readthedocs.io/en/4.3.x/

8) **Extension 2:** a form of cryptography that was genuinely used in the past hides encoded information in images in the following way. (You may have seen this in PixLab in Foundations of Computer Science.)

   Step 1: Go through a standard RGB image and convert every odd R value individually to the next even number down (ie, a value of 143 becomes 142). If you code this, you will notice that the original image is practically indistinguishable from the original.

   Step 2: Encode information in some way in a black and white image exactly the same size as your original image. This can be as simple as black text on a white background.

   Step 3: Go through both images pixel by pixel. For every black pixel in the secret image, add 1 to the red value of the encoding image's corresponding pixel.

   This encodes information in a way that is almost impossible to detect with the naked eye. Your task is to implement encode and decode methods for this. Bonus if you also write a method to convert a line of text into a black and white image of the same size as another given image; again, the pillow documentation will have methods you need.

9) **Extension 3:** a one-time pad, essentially a Vigenere cipher with keyword length the same as plaintext length, is actually unbreakable if the one-time pad has truly random letter distribution. (Google "one-time pad cryptography" if you would like more information.) This can also be simulated on images.
   a. Find or create two images with identical sizes (desktop wallpaper sites might be useful for this) and use one as a one-time pad to encode the other. Does the resulting image contain anything recognizable?
   b. Make an image that is purely random that is the same size as another, and use *it* as a one-time pad to encode. Now, does the resulting image contain anything recognizable?