K-Means and Image Vectorization

Eckel, TJHSST AI2, Spring 2022

Background & Explanation

In the previous assignment, you were introduced to *k*-means as a method of finding natural groups in a data set. We're going to extend this idea in a new direction for this assignment - we will use *k*-means to reduce an image down to a small number of individual colors, something similar to the "Posterize" button from PixLab in Foundations of Computer Science at TJ but *much* more effective.

This is a particular example of an application of *k*-means called **vector quantization**. As with last assignment, the use of "vector" here might be confusing, but you can basically think of it as equivalent to "tuple" in Python in this particular usage. If we have a data set where each data point has multiple different measures associated with it – such as, for instance, distance and speeding in the delivery driver example – then each individual data point is represented as a tuple/vector of the values of each of those measures. With that established, the idea of vector quantization is that we are in a situation where we have a whole lot of data points and we don't want to process all of them, so we'll represent the data set with a smaller number of prototypical vectors. In other words, we'll choose a certain small number of vectors that when considered together would have similar statistics to the data set as a whole. You can read a more mathematically rigorous treatment of this idea on Wikipedia if you like, just search for the "vector quantization" article.

In our case, we'll start with an image, and our data set will be the values of each pixel in the image. As you know, each pixel in an image is defined by three values – R, G, and B – that can vary from 0 to 255. So, our vectors/tuples have three values in this case. We want to choose a set of **K** specific colors where **K** is quite small. (The traditional 24-bit color space is 16,777,216 different colors; we'll want to pick something more like 8.) After the *k*-means process chooses our set of colors, we'll replace the color of each pixel in the original image with the closest color available from our new set.

As with the previous assignment, we will define the **error** at one pixel at the end of this algorithm to be the three-dimensional distance formula on the values for R, G, and B from the new value to the original value. The algorithm will use the **squared error** – ie, the same formula but without taking the square root at the end – as the measure to be minimized. The goal is to choose the **K** specific colors such that at the end the total squared error between the original pixel values and the new pixel values is as minimal as possible.

Getting Started with Implementation: Image Processing & File Access Libraries

First, you need to install the package “Pillow”. **NOT “PIL”. “Pillow”.** Do this with pip or with the package manager for either Anaconda or PyCharm. (It’s also automatically included in the base Anaconda install.)

Second, once you’ve installed “Pillow”, this code should work. “PIL” is included in the “Pillow” install. Try this:

```
import urllib.request
import io
from PIL import Image
URL = 'https://i.pinimg.com/originals/95/2a/04/952a04ea85a8d1b0134516c52198745e.jpg'
f = io.BytesIO(urllib.request.urlopen(URL).read())  # Download the picture at the url as a file object
img = Image.open(f)  # You can also use this on a local file; just put the local filename in quotes in place of f.
img.show()  # Send the image to your OS to be displayed as a temporary file
print(img.size)  # A tuple.  Note: width first THEN height.  PIL goes [x, y] with y counting from the top of the frame.
pix = img.load()  # Pix is a pixel manipulation object; we can assign pixel values and img will change as we do so.
print(pix[2,5])  # Access the color at a specific location; note [x, y] NOT [row, column].
pix[2,5] = (255, 255, 255)  # Set the pixel to white.  Note this is called on “pix”, but it modifies “img”.
img.show()  # Now, you should see a single white pixel near the upper left corner
img.save("my_image.png")  # Save the resulting image.  Alter your filename as necessary.
```

Copy/paste into PyCharm and make sure all of that is functioning, then play around with the library a bit. Get comfortable. Note that the tuple of (255, 255, 255) has R, G, B values in that order as you’d expect.

> ***Note for Macs only:*** *In past years, downloading from a URL has sometimes thrown a certificate error of some kind. Students with macs have been able to google and solve the problem; ask other students with macs to help if you encounter this, and try to find a solution. Let me know if you can’t.*

Part 1: Naïve Vector Quantization

Before we implement *k*-means on this problem, let’s see what happens if we do vector quantization without it. Specifically, let’s say we want to represent an image using a small number of colors. If we restrict R, G, and B to three values each – 0, 127, or 255 – then that makes 3 * 3 * 3 = 27 colors. If we restrict R, G, and B to two values each – only min or max, 0 or 255 – then that makes 2 * 2 * 2 = 8 colors. First, let’s code up these two possibilities.

- **27-color naïve quantization**: Take each pixel in turn. For R, G, and B individually, if the value is less than 255 // 3, make it 0. If it’s greater than 255 * 2 // 3, make it 255. Otherwise, make it 127.
- **8-color naïve quantization:** Take each pixel in turn. For R, G, and B individually, if the value is less than 128, make it 0. If it’s greater or equal, make it 255.

Save or show each result. You should see some pretty bad posterized images! (My output is later in this document for reference.)

Part 2: Implement *k*-Means

Now let's do it right.

The algorithm here is as on the previous assignment. Some important details:

- Make **K** a parameter at the top of your code that is easily modified.
- Please make sure that you start off by choosing **K** random *points in the image*, not **K** random colors from the entire color space, as your initial means. Each set of means must contain at least one pixel, and with random colors that won't always happen, but if you choose colors from the image at least each set will contain itself.
- Also, be sure you don't allow repeat colors; in the last assignment, each star was different, but in this assignment two different pixels in the same image might have the same RGB values, and if so we need to find a different pixel to use. Every starting "mean" needs to be a distinct RGB value from a pixel somewhere in the original image.
- While the algorithm is running, allow the means to have decimal values for R, G, and B. When the algorithm is *finished*, round the means to produce a set of valid integer RGB values that quantize the vector space of all of the original pixel colors.

After this, loop over the original image and replace each pixel with the closest generated mean, generating a new image using only **K** different colors.

Once this is working, run *k*-means on the same image for *k* values of 8 and 27.

Runtime for 8 should be approximately reasonable; less than a minute, perhaps two minutes at most. Runtime for 27 may very well be quite long! You'll notice it takes *many* more generations. Getting this kind of runtime is a legitimate optimization challenge. Feel free to try it yourself, or, if you like, I have placed some advice for speed on later pages of this assignment so you can read it if you want. There's also sample output to check your work.

Once your code is working, please find an image that seems interesting to work with and run all four of these processes on it – 8-color naïve, 8-color *k*-means, 27-color naïve, 27-color *k*-means – and **post your original image and all four results to the "k-means" public channel on Mattermost.** (Mattermost allows you to join any public channel; you aren't automatically in it, but should be able to find it.) It'll be neat to see everyone's images together!

Specification for *k*-Means: Image Vectorization 1

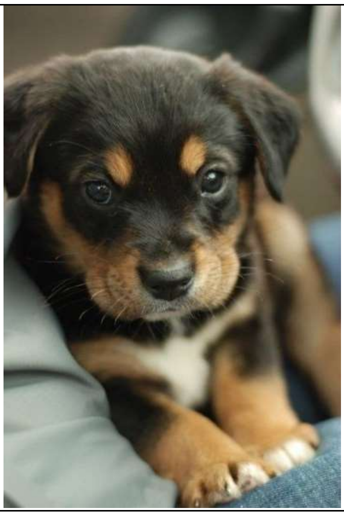
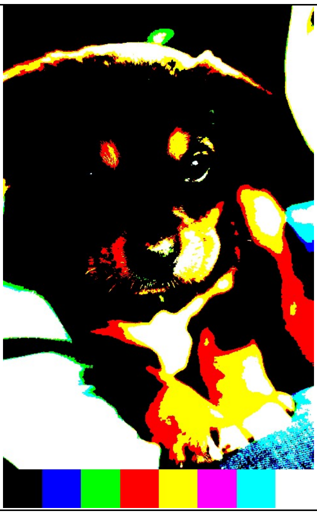
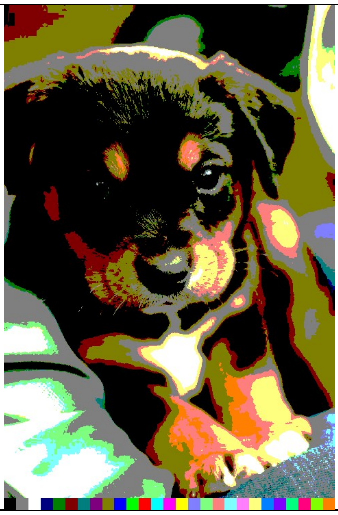
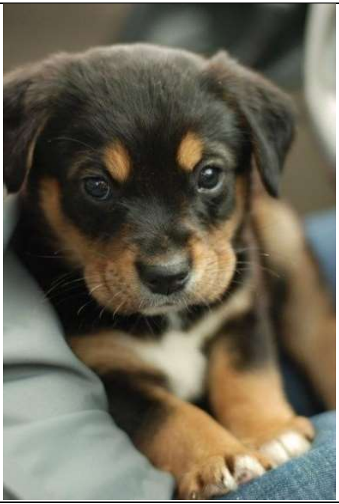
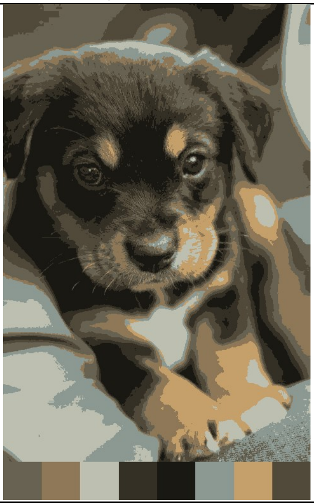
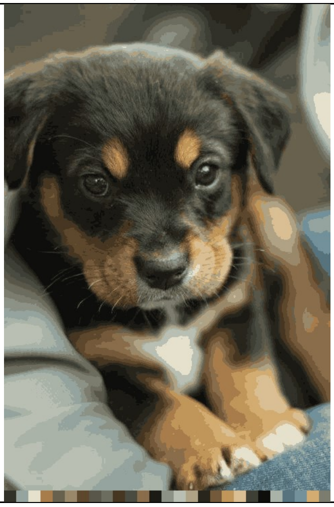
Once you've posted to Mattermost, read the specification below carefully. Make sure your code follows it, and submit **a single python script** to the link on the course website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- Your code accepts two command line arguments – a url that links to an image and a number indicating the value of k. (ie, if your code gets "https://i.pinimg.com/originals/95/2a/04/952a04ea85a8d1b0134516c52198745e.jpg 5" it will run the picture of the puppy with 5 means).
- Your code creates a file with the ***exact file name*** `"kmeansout.png"` where it saves the result of its run.
- Runtime should be less than two minutes on the puppy picture with 8 means in order to be fast enough to receive credit.
- You also must complete the task on the previous page and post your images in the public "k-means" channel to receive credit.

Sample Output

Using the puppy picture in the original URL above:

		
Original image	Naïve 8-color quantization	Naïve 27-color quantization
		
Original image	*k*-means 8-color quantization	*k*-means 27-color quantization

Look how much better *k*-means is! I'm especially impressed by the puppy's face in the last image – while you can see color banding in the blurrier areas around the outside, it's hard to tell that the face is any different at all.

As one demonstration of how good *k*-means is, whenever you paste an image into Word these days it automatically generates alt-text. It knows both of the *k*-means outputs are dogs, but it thinks the naïve 8 image is "A picture containing nature, fire, sitting" and it thinks the naïve 27 is "A picture containing looking, sitting, table, food"!

You might also notice that all of my images have a bar at the bottom with colored squares. I wanted you to see the palette available for each image, so I used PIL to generate a new image that was taller than the old one and added in the boxes at the bottom. Each box represents one of the mean color values found by the algorithm (or naïve possibilities). This is a fun little extra challenge and makes showing your results a bit more interesting; feel free to code it up if you like, before you paste your images to Mattermost. Not required, but nifty!

The command to make a new image is `Image.new(“RGB”, (width, height), 0)` if you'd like to try it. This particular call makes every pixel black. Make a new pix object on your new image then set the pixels to anything you like.

Advice for Speed

The first thing to do, if you'd like to monitor the speed of your code, is just add in a line that prints all of the movement between groups at each iteration in the algorithm. Here's the beginning of one of my 8-means runs:

```
Differences in gen 1 : [-8171, 3956, 31, 1695, -363, -1397, -10823, 15072]
Differences in gen 2 : [-1233, 178, -638, 1563, 678, -265, -9100, 8817]
Differences in gen 3 : [-1192, -470, 98, 922, 1091, 7, -4821, 4365]
Differences in gen 4 : [-1011, -667, -202, 981, 1271, 429, -2434, 1633]
Differences in gen 5 : [-723, -806, 433, 477, 1099, 610, -1373, 283]
Differences in gen 6 : [-548, -747, 334, 518, 972, 732, -877, -384]
Differences in gen 7 : [-500, -744, 267, 430, 970, 902, -609, -716]
Differences in gen 8 : [-788, -473, -54, 681, 966, 1472, -454, -1350]
Differences in gen 9 : [-851, -590, 264, 219, 1040, 1795, -417, -1460]
Differences in gen 10 : [-1230, -341, -47, 190, 1023, 3214, -712, -2097]
Differences in gen 11 : [-1503, -224, -99, 109, 694, 4190, -1164, -2003]
Differences in gen 12 : [-2031, 37, -67, 0, 740, 5426, -930, -3175]
Differences in gen 13 : [-2408, 217, -317, 0, 790, 6684, -1028, -3938]
Differences in gen 14 : [-1928, -136, -232, -55, 1030, 5646, -1637, -2688]
Differences in gen 15 : [-1482, -372, -119, -55, 1348, 3664, -1852, -1132]
```

Etc, etc.

It might be a long wait before the algorithm is finished; this is a good way to monitor progress. The algorithm is done when you get zeroes across the board.

For what it's worth, my 8-means runs on the puppy image take about 80 generations; my 27-means runs take about 300 with a lot of variance.

Some things to try if your code is slower than you'd like:

- You might try keeping track of the points that are moving into and out of each region, and only recalculate the change in the mean for each region based on those points. This avoids recalculating each mean completely every time. And, as you get towards the end, movement gets pretty slight so it saves a lot of calculation!

- Maybe you don't need to compare every pixel to every mean every time. For instance, if you compute the squared distance from a given mean to every other mean, then take the min of those, isn't it the case that if an arbitrary point's squared distance to the given mean is less than 1/4 of the aforementioned min, then it is guaranteed not to hop? That could save you a lot of comparisons.

- Your image is practically guaranteed to have enormous numbers of pixels with the same RGB values. Instead of going pixel by pixel, perhaps you can just store all the RGB values in the whole image and how many times each value appears before you begin processing. This can also save you a lot of comparisons!

- Try a different way of picking your initial means.

For what it's worth, the third bullet point is the only one I implemented and it got fast enough that I wasn't bothered. Run times for 27-mean runs are still several minutes, though. I'll let you choose to be bothered or not bothered by that as you like!

Specification for *k*-Means: Image Vectorization 2

For red credit on this assignment, we're going to make many improvements.

First, let's go ahead and do this so our results are more interesting to look at:

- Implement the color bands at the bottom of each image as I described on page 4. (For fun, you can even try to find a reasonable way of sorting them so that they appear in a rainbow order, which I did not do.)

Then, let's improve how we run the algorithm:

- Find a better way to choose the initial values that makes a meaningful impact on runtime. You should be able to show this with data. For instance, you might take a particular image and run with the old random selection and your better selection 10 times each, and show the average number of cycles is substantially lower for your method. Feel free to ask questions about this! One possibility is something called *k*-means++; feel free to Google and try that or invent your own.
- Once you've done this, send me a Mattermost message that explains what you did and what impact you had on the runtime. I'd like hard numbers; time it and tell me the difference.

Finally, let's make the resulting image better through a process called **dithering**. This is not so big a deal now but when I was a kid computer memory requirements meant that it was often impossible to assign every pixel on the screen to any color from the entire RGB color space. Instead, games and websites would pick a particular subset of the RGB color space and use only those colors to represent images – exactly the sort of thing we're doing in this assignment. **Dithering** is a process of modifying which of the chosen colors you use based on the error of the surrounding pixels. For instance, if the previous pixel was replaced by a color that was too white, we would then make this pixel more likely to be replaced by a darker color. When viewed from a distance, this has the effect of blending together and better representing the original colors.

Dithering is a well-documented process, and rather than explain it here I think I'll leave it to you to google and find a dithering algorithm and see some examples of it in action. Steinberg dithering has been a popular choice in years past but you can choose any algorithm you like.

So, for your last challenge:

- Implement dithering on your final image, once you've found the *k* RGB values you want to use to posterize your image.
- Send me a Mattermost message telling me which algorithm you used, and which websites you found information from.
- Post your results in the *k*-means channel again. Clearly state you're posting for Part 2. Now you should have 9 different images – the original plus each of the eight possible combinations of 8 or 27 colors, k-means or naïve, and dithered or non-dithered. Each one should have the color bands at the bottom.

Once you've posted to Mattermost, read the specification below carefully. Make sure your code follows it, and submit **a single python script** to the link on the course website.

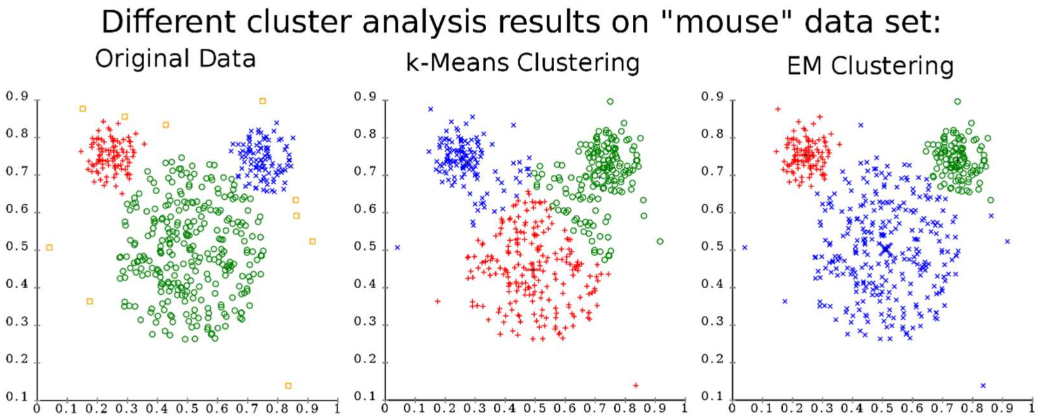
This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- Your code meets the same spec as the blue assignment, but `"kmeansout.png"` now contains a dithered result with colors across the bottom.
- Runtime should be substantially faster than your prior submission (in line with the results you reported to me on Mattermost).
- You also must complete the task on the previous page and post your images in the public "k-means" channel to receive credit.

Specification for k-Means: Image Vectorization Extension

This is an experiment; neither I nor any student of mine has ever done this, so I have no idea how it'll go. Willing to give credit for interesting failures! Just be in touch. But here's the idea.

k-means is a simple algorithm, but not necessarily the best one. In particular, *k*-means sort of relies on the idea that clusters are about the same size. Stolen from Wikipedia:



EM Clustering is one example of an algorithm with similar goals (find clusters in data) that is much harder to understand and implement but achieves better results.

You can see this pretty easily in our image processing context. Check out what happens if you run *k*-means on this image:

http://www.w3schools.com/css/trolltunga.jpg

Almost all of this image is blue or turquoise; as a result, a run of *k*-means will give you an image with 8 (or 27) shades of blue and turquoise! The little person's red shirt will vanish. A better algorithm might notice that the red colors form their own very distinct cluster and prefer that uniqueness over more common shades of more similar colors.

Feel free to either research EM Clustering further or try and imagine your own variation. I want it to run on that image with a relatively small value of *k* and still recreate the distinct red shirt. I also want you to demonstrate details on other images that your algorithm catches for an equivalent value of *k* where *k*-means does not.

Submit a document with an explanation and analysis of your results, including at least two images (the one given above and one of your own). Also submit a Python file that meets the main assignment's specification but runs your new algorithm instead.