

K-Means and Image Processing

Eckel, Independent AI Learning During Quarantine, Spring 2020

Background & Explanation

In this assignment, you'll learn the *k*-means algorithm. This is a simple example of an **unsupervised learning** algorithm in machine learning; that is, this algorithm attempts to find regularities or patterns without checking its learning against previously determined right answers. Once school is back in session, we'll be focusing on **supervised learning** algorithms that have a specified training set that includes desired outputs. But for now: *k*-means.

Specifically, *k*-means is an unsupervised learning algorithm designed to look at a lot of data and find meaningful groups. There's an article here - <https://blogs.oracle.com/datascience/introduction-to-k-means-clustering> - which outlines the use of this algorithm on a data set of delivery drivers. (The article is unfortunately missing its images now but is still comprehensible.) The algorithm is able to find groups representing rural drivers, urban drivers, and drivers that speed / drivers that drive safely just from some raw data. This is the basic idea: *k*-means is used to find meaningful clusters. In this case, "rural drivers who speed" is a meaningful cluster. You can imagine a chain store using this process to classify customers into groups for advertising purposes, for instance, or a medical study trying to group subjects into predictive subgroups.

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**, which defines a probability distribution with a finite number of prototypical vectors. The same approach can be used to select a certain number of individual items from a large data set and ensure that they are approximately representative. In our case, we'll want to choose **K** specific colors and replace each pixel in the original image with a pixel of the closest color available from our new set. When we do so, the goal is for the total squared error between the original pixel values and the new pixel values to be minimized.

As you know, each pixel in an image is defined by three values – R, G, and B – that can vary from 0 to 255. We will define the **error** of one pixel 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.

Algorithm

The *k*-means algorithm goes like this:

- 1) Specify a value of **K**.
- 2) Choose **K** random elements of the set. (In our case, the specific RGB color values at **K** random pixels from the image.) Call these the *means*, though we haven't averaged anything yet.
- 3) Loop over every single pixel and assign it to the mean that is closest (has the smallest **squared error**, as explained above).
- 4) Take each group of pixels generated in step 3 and find the actual mean of those pixels (so, the individually averaged RGB values of all pixels in that group).
- 5) Repeat steps 3 and 4. As you do so, the values of the means will change, which will cause pixels to move between groups. Keep repeating until this becomes stable; that is, until no pixel changes group. (This means you'll need to keep track of the groups and how many pixels move in or out during each round.)
- 6) When the process resolves, you've now found the **K** specific means that minimized the squared error! The algorithm isn't guaranteed to converge to optimality, but in practice it is quite robust.

After this, we'll loop over the original image and replace each pixel with the closest generated mean.

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 Pycharm package manager.

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 for Macs Only:

If you are running Mac and you are getting a certificate error when running the BytesIO thing, you should run:
/Applications/python\ 3.7/Install\ Certificates.command

You can also go into install folder manually and type certificates.command directly there.

Part 1: Naïve Vector Quantization

Ok, so 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 on the next page for reference.)

Part 2: Implement K-Means

Implement the algorithm as described on page one. Make **K** a parameter at the top of your code that is easily modified. 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. I have some sample output and advice for speed on later pages of this assignment.







Once yours 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 appropriate discussion on Piazza. I’ll have a specific question where I ask you to do this.

(Feel free to post anonymously; obviously there’s no credit here, just a chance to see various outputs & help bugfix!)

You can stop reading here if you want! The rest of this doc contains sample output, tips for speed, and many extensions.

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 – where 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. Right click on each image to edit the alt-text; look at the alt-text that Word came up with for each image. 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 Piazza. 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. One possibility is to randomly sample points and average them to produce each mean, for instance. The extensions below also mention *k*-means++, a better version of the algorithm that's designed for this purpose that you can Google.

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!

Extension Ideas

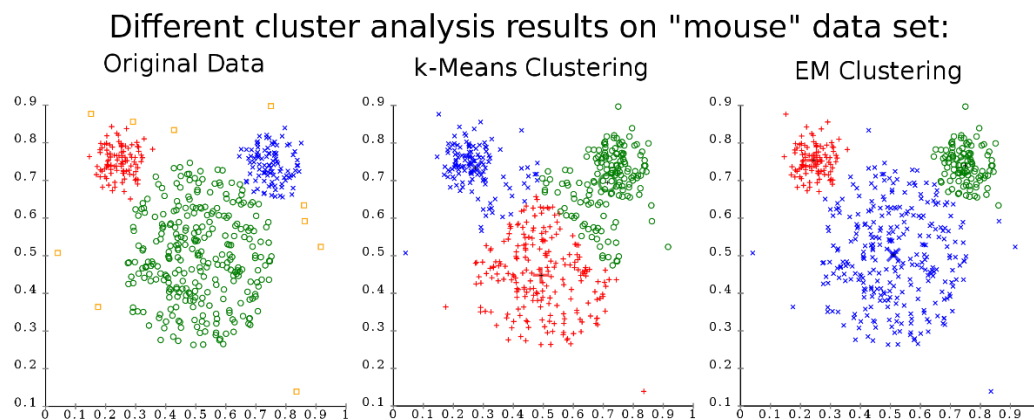
Outstanding work for your own personal satisfaction!

- Implement the color bands at the bottom of each image as I described on page 3.
- Research and implement k -means++ to make a more intelligent initial selection of starting means or find a different one you like better. Or invent your own.
- An interesting aspect of this algorithm is that we have to choose the initial value, k , ourselves. How could we decide what value is best? This is a difficult question to answer, because as we increase k , we will always get better results up to the value of the number of actually different colors in the original image. The way to answer this question is to try increasing values of k , find the total squared error of every pixel in the image, and graph the results. You'll notice that the graph has a kind of elbow-curve, a bit like an exponential function. The point where the graph turns is the right value of k .

Find the right value of k for the puppy image and post it on Piazza. Include the graph or chart you used to determine where the elbow curve was. My guess is that it's actually less than 8 – try running $k = 3$ or 4 on this image; a surprising amount of detail is preserved! But I would love to see numbers to know for sure.

Then do this on an image of your own as well.

- 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. Post any results!