

K-Means Clustering on Images

My goal for this project was to take an image and use the k-means clustering algorithm, a form of unsupervised learning, to compress an image. In order to do this, I took every pixel in the image and chose a set of K distinct pixels that would correspond to K distinct colors. K would be a small number, such as 8 or 27. Then, after choosing the new colors using the k-means algorithm, every pixel in the image would be replaced with one of the K colors that was closest to the original.




Naive Approach

The first thing I did was attempt vector quantization of the image without K-means. This was called the naive approach. In this approach, we would restrict the R, B, and G values of each pixel to specific numbers. If we restricted R, B, and G to 3 colors each – 0, 127, and 255 – there would be a total of $3*3*3 = 27$ colors vs. $255*255*255 = 16,581,375$ for uncompressed RGB. If we restricted R, B, and G to 2 colors each – 0 and 255 – there would be a total of $2*2*2 = 8$ colors.

27-color approach: Using the approach with 27 colors, for each R, B, and G value of every pixel, if the value was less than $255 // 3$, it would become 0. If it was greater than $255 * 2 // 3$, it would become 255. Otherwise it would become 127.

8-color approach: Using the approach with 8 colors, for each R, B, and G value of every pixel, if the value was less than 128, it would become 0. If it was greater than or equal to 128, it would become 255.

This naive approach produced results that were very far from the original image and did not accurately represent it.






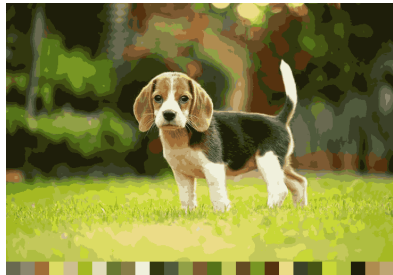
		
Original Image	Naive 8-color approach	Naive 27-color approach

We can see in these images that the naive approach without K-means produces images that are very badly posterized. At the bottom of the images are all the colors that the image uses.

K-Means Approach

I then moved on to the K-means approach. This algorithm aims to group data points together into clusters. In this case, my data was all the pixels in the image. The K value was the number of clusters, or in this case, colors, that the data would be grouped into. So, I would first choose a value for K, then I would choose K distinct random points from the dataset to be our means or centroids. I would then loop through all the other pixels and associate each one with a mean. In order to do this, I would look at the pixel's RGB values and find the squared error between that pixel's values and all of the means. I would then assign the pixel to the mean with the smallest squared error. After every pixel had been assigned to a group, I would loop through each group and find the mean of that group. Then, I would once again loop through all the pixels and assign them to a group. This will cause some of the pixels to move around to different groups. Once no pixels are moving around anymore and the groups become stable, I would stop this process. Once this process has stopped, I loop over all the pixels in the image and replace it with the closest mean, recreating the image with K colors.

This created much better results than the naive approach. I did this with the same numbers as the naive approach to compare.

		
Original Image	Naive 8-color approach	Naive 27-color approach
		
Original Image	K-Means 8-color approach	K-Means 27-color approach

We can see here that the K-means pictures look much closer to the original than the ones created using the naive approach. The colors at the bottom are all the colors used to create the image. However, one of the drawbacks of the K-means algorithm is that the runtime can vary depending on what random points are selected to be the initial centroids. I will try to optimize this runtime in the next section.

K-Means++

In order to make the runtime faster, I also implemented K-means++. This made it so that, rather than choosing random initial means/centroids, it chooses means/centroids that are further away from each other. In order to do this, it chooses the first mean/centroid randomly and then for each data point it finds the distance to the nearest, already chosen centroid. The next centroid will then be selected as the point that is the furthest away from the previous centroid. This process will continue until K centroids/means are chosen. The rest of the K-means process stays the same after this.

When I used a K value of 8 without K-means++, and ran it 10 times, the average time was 71.99 seconds. Then, when I ran it with K-means++, the average decreased to 70.63 seconds. This is a decrease of 1.88%. Then, for a K value of 27, when I ran it without K-means++, my average time was 223.94 seconds and with K-means++ it was 195.85 seconds. So, that is a decrease of 12.54%. The number of cycles to reach K stable centroids also showed a meaningful decrease. For a K value of 8, it went from an average of 132.8 cycles to 128.2, which is a decrease of 3.46%. For a K value of 27 it went from an average of 170.8 to 124.7, which is a decrease of 26.99%. The table below summarizes these findings.

Table 1: K-means and K-means++ Comparison for 10 Runs







	Times						Number of Cycles			
	k means 8	k means++ 8	k means 27	k means++ 27			k means 8	k means++ 8	k means 27	k means++ 27
	120.13577	76.88854	164.68783	214.15229			162	118	135	129
	67.77094	67.42520	164.11883	231.81959			128	148	99	77
	83.13494	77.06864	126.33075	330.19189			96	129	92	201
	80.13091	144.92413	188.49649	250.70500			170	153	124	157
	86.71255	77.96016	371.85313	196.18552			141	94	244	122
	51.33121	48.61776	223.18637	160.43571			124	124	181	134
	33.33006	36.79472	286.49675	169.38305			69	96	216	126
	63.28545	61.22433	300.54573	145.06770			156	157	255	120
	60.22856	51.52296	200.16196	74.33182			150	111	168	64
	73.87102	63.93742	213.58037	186.28374			132	152	194	117
Average	71.99314	70.63639	223.94582	195.85563		Average	132.8	128.2	170.8	124.7

Dithering

The last thing I did was implement Floyd-Steinberg dithering. Dithering is the process of approximating colors that are not in the color palette through diffusion of pixels that are available. Then, when a human looks at the image, it looks like there are more colors than there actually are. However, while dithering makes the new image look closer to the original image visually, it also has the potential to increase the size of the file, which was the case with my sample runs.

In order to implement dithering, I looped over every pixel in the image and found the closest available color. Then, I would calculate the distance between the original color in the image and the color that I have available. Once I found that error value, I would divide it and distribute it amongst the neighboring pixels. When I later get to those pixels, I would add the error to it from the previous pixels. This distributes the error amongst many different pixels and allows for that area to be perceived as the color that we want.

Using dithering, we can see that the image looks even closer to the original than with just K-means or with the naive approach. I once again used K values of 8 and 27 for easy comparison and the color bars on the bottom of the images contain all the colors used in that image.

		
Original Image	Naive 8-color approach	Naive 27-color approach
		
Original Image	K-Means 8-color approach	K-Means 27-color approach
		
Original Image	K-Means 8-color approach with dithering	K-Means 27-color approach with dithering

We can see here that the images using K-means and dithering are much better than those using just K-means or the naive approach.

I have provided a folder of all the images in the repository.