

K-Means

Gabor – AI – Spring, 2021

K-Means is an **unsupervised machine learning algorithm** that tries to identify K means which are most representative of the various points that are given. It is up to the user to supply the number K, which could be 4 or a higher number. Increasingly high numbers for K will likely cause a significant slowdown for the algorithm.

The algorithm itself is fairly easy to describe: Given a K, select K points (the K means) in the same space as the data points that are to be classified, where the selected points may include some or none of the data points. Now repeat the following until there is no more change:

- A) For each data point, find a closest mean. If two means are equally close, pick only one.
- B) Recompute each mean based on all those points that got associated with it in step A.

The remarkable thing about this algorithm is that it converges in practice.

Some examples of K-Means may be found at <https://blogs.oracle.com/datascience/introduction-to-k-means-clustering>. The article outlines the use of this algorithm on a data set of delivery drivers (unfortunately, a few images are missing, but the article 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 example, or a medical study trying to group subjects into predictive subgroups.

For the K-Means lab of this class, we’ll apply the algorithm to an image to reduce its size by selecting representative pixels. Suppose you have an image of $H \times W$ pixels (height by width). A pixel (point) in an image can be thought of as a combination of three different colors, red, green, and blue (RGB), with each color having 2^8 possible values, being combined in just the right way to get a single color value. A naïve approach that wanted to represent an image with n^3 distinct pixel values, where n is either 2 or 3, would take each color range and divide it into n equal segments and then use the integer closest to the midpoint of each of those segments as the pixel representative for that segment. An alternate way when $n = 3$ is to take the values 0, 127 and 256 and have them represent the color for values in the each third of the given color’s range.

The first row of images on the next page shows what happens when the image on the left is quantized according to the naïve implementation described in the prior paragraph using 8 and 27 pixels, respectively.



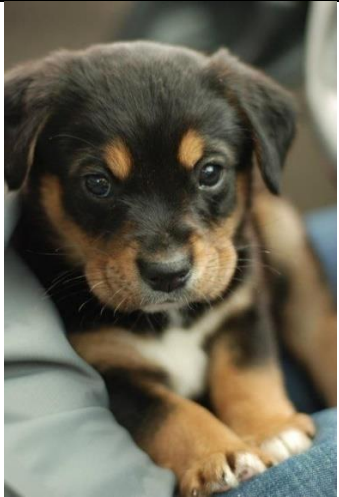


		
Original image	Naïve 8-color quantization	Naïve 27-color quantization
		
Original image	k -means 8-color quantization	k -means 27-color quantization

Image vector quantization using a naïve method in the first row and k -means in the bottom row.

The second row shows what happens with K -means (where k is 8 and 27, respectively). Clearly the last two entries on the bottom row are truer to the original than the last two entries on the top row. The most telling aspect, though, is that the Alt Text that is generated (right click an image and then select Edit Alt Text to see what is generated automatically) recognizes a dog for each image in the bottom row. However, the rightmost two in the top row don't get the coveted 'dog' label even though we know those images really are of a dog. You may notice that at the bottom of the quantized images, the color palette is shown for which colors the image is made up of.

Lab specification:

Your script will receive two arguments: an integer, $k \in [4,7]$, and the name of an image file.

Your script should read in the image file and determine some statistics about it, then run K-means on the pixels in the image to determine k representative pixels. It should then create a new image by replacing each current pixel with the representative one from its nearest mean. Finally, it should determine how many distinct regions there are in the new image using a flood fill algorithm.

Specifically, the script should print out:

Size: *height* x *width*

Pixels: #

Distinct pixel count: #

Most common pixel: (*Rval*, *Gval*, *Bval*) -> *amount*

Final means:

1: (#,#,#) => *amount*

2: (#,#,#) => *amount*

...

k : (#,#,#) => *amount*

Region counts: #, #, ... #

The final means are the means (real numbers; not the closest pixel values) arrived at via the k-means algorithm, and the number on the right of the => indicates the number of pixels in the associated mean bucket. The image should be updated (with the closest pixel to the associated mean) and saved as `img.save("kmeans/{ }.png".format(yourUserId), "PNG")`

Region counts are obtained by doing a flood fill on this resultant image. Diagonally adjacent pixels are considered adjacent for the purposes of this flood fill.

A result with a "K" generally indicates that the relevant keyword was not found (or not followed by a colon), while an "N" usually indicates that the number of numeric values on the rest of the line was not what was expected.

Setup details:

My notes say to install Pillow (and not PIL. PIL is included with Pillow). PIL stands for Python Image Library

In the code, do:

```
from PIL import Image
```

Useful image methods:

```
img = Image.open(fileName)           # Get ahold of the image
img.size()                           # A tuple
pix = img.load()                     # Live, editable image; a 2D pixel array
pix[2,3]                             # Get a pixel: The RGB tuple at posn 2,3
pix[2,3] = (255, 127, 0)             # Alter a pixel
img.save(filename, "PNG")             # Save (the altered) pix
```

The grader gives 60 seconds to carry out the specified tasks. It is likely the first implementation is not quite fast enough. The first thing one might do is to investigate what the basic algorithms is doing. How many generations does it take to converge? How many points are hopping between buckets in any given generation. Something one might do is to have a printout akin to:

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

Note that these are deltas, so that you are measuring the net influx of points into each bucket. Your k-means section is done when you have all 0s for one generation. Probably for $k = 8$, it should take about 80 generations and for $k = 27$, it may take about 300, with a lot of variance.

The above starts to give some reality to how many calculations are involved.

There are a few things one might do to increase speed.

A) Bucket the pixels. It is very likely that there are many identical pixel values. It can save a significant amount of time if one avoids redoing all the calculations for duplicated pixels. A standard way is to have the RGB tuple act as a key to a dictionary, where the values are the number of times a key appears in the image.

B) Avoid distance calculations if a point is close to its associated mean. If a pixel is associated with a particular mean, and its distance to that mean is less than half the distance of that mean to any other mean, then it is clear that that pixel stays associated with that mean. Therefore, there is no reason to compare it to any other mean, which can save a lot of calculations and comparisons.

C) Use incremental programming to speed up mean calculations. After a while, most points will settle down and relatively few points will be bouncing around. Therefore, one could use the prior mean value, along with the points that are coming and going to speed up the mean calculations.

D) A more judicious initial mean selection may hasten convergence, and also improve where the convergence gets to. One approach is to use k-means++, which is k-means where the initial means are chosen in a probabilistic way so that the initial means are relatively far away from each other.