Elbow Clustering for Artificial Intelligence

Daniel Shapiro, PhD Follow
Apr 30, 2018 · 9 min read

Clustering is the process of taking a pile of unsorted stuff (your dataset) and breaking it into piles (i.e. clusters) according to the similarity of the stuff. This is an unsupervised process (no training examples needed). Perhaps the most common way of doing this is with k-means, and I will introduce a different way to do it with a Gaussian Mixture Model (GMM) and some other stuff on top of that.

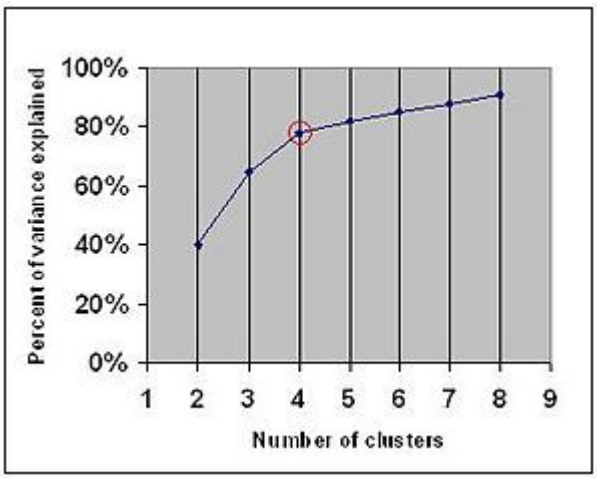
Our example today will be separating 2 kinds of picture in a dataset, where you don't really have a label or a classifier for either of these "things" in your dataset. You can see it works by just looking at the result, as we will see. I want to start off by making it clear that this method does not always work, and so it takes some fine tuning.

The first thing you need to remember to do at each major step in your artificial intelligence project is to ***LOOK AT YOUR DATA WITH YOUR EYES***. I can't emphasize strongly enough that things with k-means can go stupidly wrong if you don't see what it is doing.

Picking up where we left off in my article on image dataset collection, let's do clustering of images by perceptual hashing with k-means and GMM. If you just want to find what pictures are similar to others, there is an example right in the imagehash code to do that. What we are trying to do here is a bit more: we want to separate the data into categories (even though we don't know what the categories are). On GitHub there is an example of this type of library licensed only for academic use.

We immediately run into a problem, regardless of the library we pick: How many clusters do we want to end up with? This is not given to you by the clustering library. You need to decide, and deciding the number of clusters should be based on some logic. The way we are supposed to do this is using … **the elbow method!**

The elbow method is a weird name for a simple idea. Keep adding clusters until you see diminishing returns, and then stop. With k-means this means starting with 2 means and then 3 means, and so on until k. Same idea with GMM. When we see an elbow in the graph of explained variance versus cluster count, we back up and select the number of clusters where we see the elbow. You can see in the image below how the first two dots are the arm, the dot at 4 is the elbow, and the forearm is the points at 5 through 8.



Graph of explained variance vs cluster count, where the red circle shows the elbow. (from wikipedia)

I know: We computer people should stop naming stuff right now. Turing machine, Round robin, k-maps, it just doesn't stop.

See the figure on page 148 of "Python Machine Learning" for some more insight on how some components of a dataset can contain more variance than others (PCA). The elbow method is just as valid when deciding when to stop training during early stopping, but in that case the data is typically much noisier. Basically, we stop when it looks like it's not worth adding effort.

Why is this elbow method thing not well known? Well, I guess it is kind of subjective, and so people who want to know an exact number are weirded out. Also, we tend to have a number of clusters in mind when we cluster things, and often for stupid reasons, but it's a fact. For example, I prefer to have a number of clusters that my tiny brain understands (e.g. a small number), rather than the number that the math tells me I really need. Also, I have had cases where the number of clusters is limited by some outside influence like hardware restrictions. There may be 299,945 trading pattern types, but if my time-limited space-limited neural network on GPU can only analyze 2,048 pattern types in a useful amount of time, then I guess we're going with 2,048 clusters. There are many examples in the research literature using the elbow method. We will see in our example that the elbow method is not the only game in town.

Let's move on to doing the thing we came here to do: Cluster pictures with the elbow method using perceptual hashes! Let's sort images from a standard image dataset to see what sub-types we end up with. The goal here is to sort the images ***without*** using the ground truth labels to do the sorting (i.e. clustering). The thing we are about to do with k-means is unsupervised. It just looks at pictures and sorts them into piles. The reason for using a dataset with labels is just to help visualize after the fact what the sorting process did to the dataset. In reality you would look with your eyes at the pictures in each cluster and get an intuitive feel for what it did.

In our example today we will use the "Faces" and "airplanes" subfolders of the Caltech101 dataset. We mix the pictures like a deck of cards and put them in a big pile. Now imagine this is the output of some image scraping job.

```
from PIL import Image as IM
import imagehash
from IPython.core.display import Image, display
import glob
import numpy as np
import pandas as pd

d = {'hash' : pd.Series([], index=[]),
     'file' : pd.Series([], index=[])}

df = pd.DataFrame(d, index=[])
display(df)

type1 = glob.glob('/home/science/pHash/101_ObjectCateg
type2 = glob.glob('/home/science/pHash/101_ObjectCateg
dataset = type1+type2

counter=0.0
for i in dataset:
    try:
```

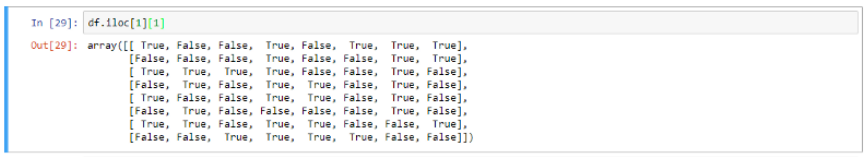
For the leaf dataset from the last article, we end up with is 4102 rows × 2 columns, telling us exactly what the hash is for each image. About 20% of the original 5,000+ images would not generate a hash, usually because the file somehow arrived damaged during the scraping process. Let's get a better sense of what just happened….

```
% done is: 84.61
% done is: 86.37
% done is: 88.14
% done is: 89.9
% done is: 91.66
% done is: 93.42
% done is: 95.19
% done is: 96.95
% done is: 98.71
```

	file	hash
0	/home/science/LeafClassify/data/train/sick/yah...	013c7f62c083b79f
1	/home/science/LeafClassify/data/train/sick/bin...	1c06020212fcffff
2	/home/science/LeafClassify/data/train/sick/yah...	00ebfff4e4e1c740
3	/home/science/LeafClassify/data/train/sick/yah...	3b09cdfcdb1b8387
4	/home/science/LeafClassify/data/train/sick/yah...	0c4f06cc6bcfdefe

Screenshot of the dataframe printing out after the hashing is completed.

We need to unwrap this sneaky thing we are calling the hash. We can compute the difference between images by subtracting the hash of two different images, but something odd pops up at this point. Even though we are able to use the subtraction operator on this hash number, it is actually NOT doing a subtract. Prove this to yourself by simply adding hashes instead of subtracting, and it will not so gracefully fail to compile. So, what is this hash thing we get at the output? Well, it's a field of boolean values for each image. Clearly the characters in the hash string are hexidecimal, and so printing one element reveals the true nature of the hash hiding underneath this pretty string: It's a bunch of binary numbers.



The binary result of the hash is hiding inside the hex string....

So, what is the subtract operation really doing when we "compare" hashes? It is doing an XOR and counting the number of bits that were not the same. Put another way, the "difference" between images is quantified as the count of the number of differences in the hash of the two images. We can prove it to ourselves using this handy little script…

```
arr = df.iloc[1][1]
arrInt = (int(str(arr), 16))
arr2 = df.iloc[2][1]
arr2Int = (int(str(arr2), 16))
print(arrInt)
print(arr2Int)
strXor = str("{0:b}".format(arrInt^arr2Int))
```

We see the following output when we run this script:

```
print(strXor.count('1'))
print(arr-arr2)
```

```
10886311195233671484
16207642963781622577
38
38
```

Counting the bits that didn't match gives the same result as "subtracting" hashes.

And so, this was quite sneaky, but now we know that the distance between hashes is not an orderable property. It changes based on which 2 images we use as inputs. What we really are saying is that we compute the hamming distance. In scipy land, this metric is #8 in this list of distance metrics.

Now, to do the clustering! We have 4 bit hex characters in our hash string, and there are 16 of the characters in each hash. That gives us 64 dimensions by which to compare images. We need to update our pandas dataframe to include 64 more columns (1 per hash feature).

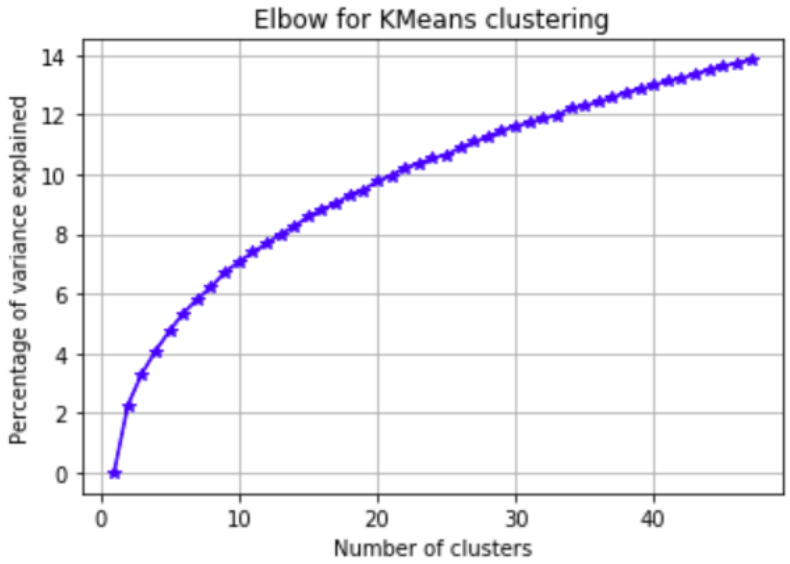
```
def explodeDims(row):
    intVal=(int(str(row['hash']), 16))
    binVal="{0:b}".format(intVal)
    strValArr=np.asarray(list(str(binVal)))
    ind=1
    for i in strValArr:
        col = 'col'+str(ind)
        row[col] = int(i)
        ind+=1
    return row

with pd.HDFStore('store.h5') as store:
    dfNew=store['df']
    dfNew=dfNew.apply(explodeDims,axis=1)
```

We end up with the 64 new columns we wanted.

	file	hash	col1	col2	col3	col4	col5	col6	col7	col8	...	col55	col56	col57	col58	cc
0	/home/science/LeafClassify/data/train/sick/yah...	a740af6a68a7a867	1	0	1	0	0	1	1	1	...	0	0	0	1	
1	/home/science/LeafClassify/data/train/sick/bin...	9713f25a9a42d93c	1	0	0	1	0	1	1	1	...	0	1	0	0	
2	/home/science/LeafClassify/data/train/sick/yah...	e0ed1d675e901b31	1	1	1	0	0	0	0	0	...	1	1	0	0	
3	/home/science/LeafClassify/data/train/sick/yah...	b948358dc6c652bb	1	0	1	1	1	0	0	1	...	1	0	1	0	
4	/home/science/LeafClassify/data/train/sick/yah...	b1221ba96a5e7745	1	0	1	1	0	0	0	1	...	1	1	0	1	

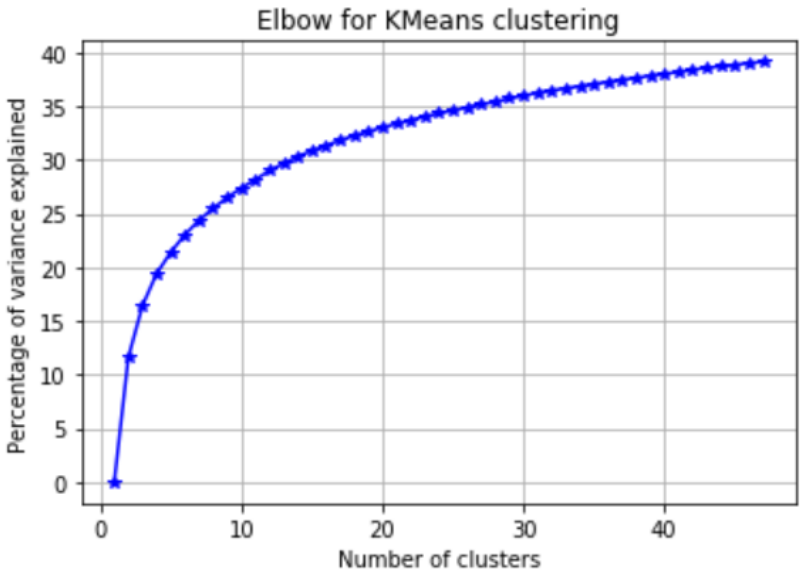
5 rows × 66 columns

Using the code from this post, we can obtain some nice graphs for the full dataset of images, including the percentage of explained variance.



This figure is for hashed generated by pHash. It shows the increase in explained variance as we add clusters.

We in the figure above that at around 8 or 9 clusters we start to lose steam. Also, since most of the variance is not explained by the model, it is not a super useful predictor.



This figure is for hashed generated by the average hash. It shows the increase in explained variance as we add clusters.

Similar to the perceptual hash figure, we see in the figure above that at around 8 or 9 clusters we start to lose steam. Much more of the variance is explained by this method than the perceptual hash in the previous figure (the top line is 40% instead of 14%). We also see a nicer

elbow in this figure. It is still not a super useful predictor, but better than before.

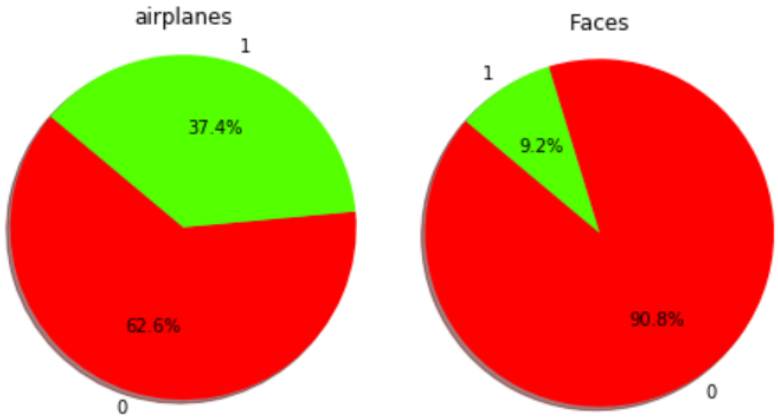
Now that we have some answers, what does it all mean? What images got clustered into each pile? Well, there are only some constrained cases where the clustering works well. Let's look at the pile of 2 kinds of images we mixed together in the beginning of the article (airplances and faces) and see if the clustering was able to unmix the two image types. Let's compare the GMM unmixing with k-means, to see which is a better unsupervised "unmixer" for this problem.

First, here is the code for K-means:

```
from IPython.display import Image
import random

clusterCount = 2

key_dict = {}

key_dict[clusterCount] = {}

for i in range(clusterCount):
    key_dict[i] = {}

#get the cluster centroids: indexing starts at 0
cluster_centroids = centroids[clusterCount]
for i in range(X.shape[0]):
    distances = [cdist([X[i]], cluster_centroids, 'euc
    final_cluster = np.argmax(distances)
    topic = dfNew.iloc[i]['file'].replace('/home/scier
    newDict = key_dict[final_cluster]
    if topic in newDict:
        newDict[topic]=newDict[topic]+1
    else:
        newDict[topic]=1
    key_dict[final_cluster] = newDict
print(key_dict)

import matplotlib.pyplot as plt
for cl in range(clusterCount):
    # Data to plot
    labels = np.asarray(list(key_dict[cl].keys()))
    sizes = np.asarray(list(key_dict[cl].values()))
    cmap = plt.cm.prism
    colors = cmap(np.linspace(0., 1., len(labels)))
    explode = (0.1, 0, 0, 0)  # explode 1st slice

    # Plot
    plt.pie(sizes, labels=labels, autopct='%1.1f%%', c
    plt.title("Cluster # "+str(cl))
    plt.axis('equal')
    plt.show()

print('================================================
#build up all the subkeys
subKeys = {}
```

```
45    for cluster in range(clusterCount):
```

The results look like this:



Cluster 0 is red, and cluster 1 is green. In the chart above, we can see that counting up all of the images containing faces, 90.8% ended up in cluster 0. Similarly 62.6% of the images with faces in them ended up in cluster 0. This is not a nice outcome.

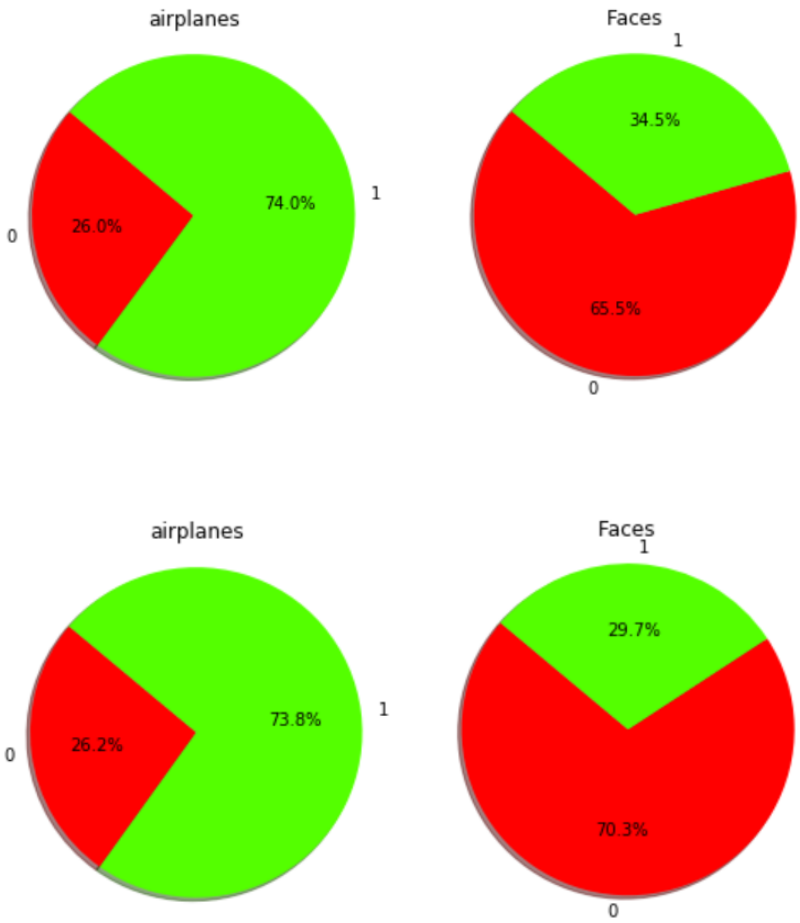
There were 800 airplane pictures and 435 faces pictures.

As we can see in the above pie charts, with 2 clusters the airplanes and faces were not separated well. We can pick 2 clusters because we know there are 2 things to separate out into piles. Stuff from both classes ended up mostly in cluster 0.With GMM we see a different story.

Let's have a look first at the code for GMM, and then see the results.

```
from sklearn.mixture import GaussianMixture

clusterCount = 2

#Predict GMM cluster membership
gm_messy = GaussianMixture(n_components=clusterCount).

key_dict = {}

key_dict[clusterCount] = {}

for i in range(clusterCount):
    key_dict[i] = {}

#populate key_dict to see what items end up in each cl
for i in range(X.shape[0]):
    topic = dfNew.iloc[i]['file'].replace('/home/scier
    final_cluster = gm_messy[i]
    newDict = key_dict[final_cluster]
    if topic in newDict:
        newDict[topic]=newDict[topic]+1
    else:
        newDict[topic]=1
    key_dict[final_cluster] = newDict


#build up all the subkeys to find what cluster each ca
subKeys = {}
for cluster in range(clusterCount):
    clusterSet = key_dict[cluster]
    for topic in clusterSet:
#         print(topic,cluster,clusterSet[topic])
        if topic in subKeys.keys():
            topicMap=subKeys[topic]
            topicMap[cluster]=clusterSet[topic]
```

And now the results from a few example runs:



So these results are a lot nicer. GMM separates the files into a folder with mostly faces in the images and another folder with mostly airplanes in the images.

We can cherry pick the results by running GMM a few times to see when the split looks good (by visual inspection of the resulting image datasets). We can also label some subset of the data and using the code above we can just look at pie charts to see how the split worked. That approach needs less data than a full blown image classifier would. There are 50 other ways of doing this as well, but count this as way 51.

Clone it!!

I have provided you with the code to play around with, and hopefully this will help you wrangle your scraped images a bit better. On a large number of classes this gets super messy, but I do have a word of advice on that. It often happens that with a lot of classes the garbage and duplicate images cluster together and help you to then delete blobs of garbage images. You can then re-cluster and see what happens. K-means sometimes works, and in this case GMM was a bit better. You

never know. Basically try lots of stuff and you will know right away when it is working.

And so, in conclusion, perceptual hashes can be used to crunch images down to a bunch of binary dimensions upon which k-means, GMM, or other unsupervised methods can arrange images into categories.

If you liked this article on clustering images with k-means and the elbow method, have a look at some of my most read past articles, like "How to Price an AI Project" and "How to Hire an AI Consultant." In addition to business-related articles, I also have prepared articles on other issues faced by companies looking to adopt deep machine learning, like "Machine learning without cloud or APIs."

Happy Coding!

-Daniel
daniel@lemay.ai ← Say hi.
Lemay.ai
1(855)LEMAY-AI

Other articles you may enjoy:

- Artificial Intelligence and Bad Data
- Artificial Intelligence: Hyperparameters
- Artificial Intelligence: Get your users to label your data