**K Means Clustering**

Usually involves classifying objects with a feature vector, which is a vector of attributes.

Table

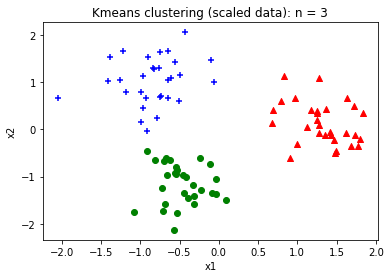
Description automatically generated with medium confidence

We can plot these objects in their feature-vector space, and see what groups our data might self-segregate into. Note it’s important to scale your data first (i.e., center and scale data so that a given column entry, xi, goes to (xi – μ)/σ, where μ and σ are the average and std of the values in that column. Consider following data of three clusters (identified by marker type) and the K Means algorithm run on it, for n = 3 clusters.

A diagram of a number of dots

Description automatically generated with medium confidence

So the algorithm identified red, blue and green clusters. Green looks okay, but we would’ve separated the red and blue into vertical groups, not horizontal groups. It did it this way, however, because of the x1 scale vs. the x2 scale. Remember (see below) it’s trying to minimize dissimilarity. And since the x1 scale is so large, the variation in the x1 coordinate almost wholly accounts for the dissimilarity present. The variation in x2 is so small, in comparison, that we might as well just flatten all the data onto the x1 axis and have no x2 axis at all. In that case, it makes sense that if we are to split the data into three groups, we would split it right in the middle (along the x-axis) of the “+” and “o” groups. Furthermore, we can see that if we were to do Kmeans with n = 2, it would just combine the “+” and “o” groups into one. And if we were to do an elbow plot (see below), it would probably show n = 2 as the most natural number of groups. But clearly there should really be three groups present. We can see this because when we look at the graph, we aren’t paying attention to the absolute numbers along the axes, just the relative distances. We can achieve this numerically by normalizing the data along each axis.



When we do, we get what we were expecting rather.

**Math**

A routine for finding these is to presume some number of classes/clusters, k. This is somewhat arbitrary, perhaps needs to be informed by theory. k = 1 would be uninformative, as would k = N (the # of data points). So we’re obviously going for something in between. Data here seems to indicate presence of 4 classes.

Chart, scatter chart

Description automatically generated

Then our goal is to find the k cluster coordinates (not typically an actual data point), ck, that satisfy the following conditions. These coordinates identify the average coordinate of the cluster. First, let S stand for the set of all points. And let Sk be the cluster, or set of points, *e* ∈ *Sk*, that are closest to ck (yes for any bunch of points, there will a set which is *closer* to a given ck than the other ck’s). Then let the variability of the cluster Sk be defined as:



This is also, mathematically, the inertia of each cluster, assuming each point has a mass of 1. And the dissimilarity overall, of S, be defined as the sum of all the clusters’ variabilities (or inertias).



And our goal is then to find the points which minimize the *total* variability, i.e., the dissimilarity. Consider the examples below. The red ck’s are the proposed cluster coordinates. So we might suppose that K = 1, in which case we’d just get something like:

A diagram of a number of dots

Description automatically generated

Or we might propose K = 4, and then we’d get something like this:

Chart, scatter chart

Description automatically generated

Or we could propose K = 5, and get:

A diagram of red and blue dots

Description automatically generated

We’ll consider how to determine which K to use in a bit. But do note that as K increases, the dissimilarity will go to zero. For instance if we take K to be the number of data points, then dissimilarity will definitely be zero. So it doesn’t do any good to look for the K that has the lowest disimilarity overall.

**Routine**

For a given K, we typically use a greedy algorithm to determine the points within a cluster. Start with random centroid coordinates {ck}. Put the points closest to ck in Sk. Compute the centroids of the clusters, and move ck there. Then recalculate the points closest to {ck}, recalculate the centroids, and move {ck} to the new centroids, etc. When the centroids stop moving, you’ve arrived at your result. This will often minimize dissimilarity, but sometimes you will have found a ‘local’ minimum. So it’s best to run the algorithm a few times and pick the best of the runs (lowest overall dissimilarity). Here’s an example of the output of a code he provided.

Chart, scatter chart

Description automatically generated

Note that we won’t necessarily be able to draw a non-squiggly-looking surface between all the clusters.

**Is there an optimal number of clusters?**

Like with the Hierarchical clustering method, our algorithm doesn’t really tell us how many clusters there are/should be. So if you didn’t know the number of clusters you should have beforehand, then what? I would think you’d run the progroam for k = 1, 2, 3, etc. And what? Wouldn’t it make sense to define dissimilarity as the sum of the variabilities within the centroid group + variabilities of the centroids themselves? Oh, well see Design Matrices – T test file Appendix 2. There we see that this is the same as the total variability of the entire data set about its global mean. This is of course a constant, independent of the number of clusters. So our proposed prescription will not work. StatQuest guy says that if we plot the reduction in variation = dissimilarity(k=0) – dissimilarity(k=k) vs. k, then we can identify the optimum number of k’s as the k\* where there’s a ‘kink’ in the graph.

Chart, line chart

Description automatically generated

and we can identify the optimal number of clusters, k, by the location of the ‘kink’ in the graph.

**Liabilities**

First, a non-liability. Kmeans is much faster than Hierarchical Clustering. But here’s a note of caution. From the datacamp video, got this graph,

A screenshot of a graph

Description automatically generated

It illustrates that the k-means clustering algorithm failed to capture three distitnct groups in this dataset, while the hierarchical clustering guy did get them. This is because the kmeans algorithm prefers to group clusters into sets of relatively equal sizes. Can sort of see how the dissimilarity is smaller for the grouping on the left. Related to this, and looking at the next file, it looks like it might not be the best at catching nested groups, like this:

Bubble chart

Description automatically generated

Apparently DBScan is better at this. And I’d think Hierarchical Clustering would be too. Another point to mention is that the kmeans clustering algorithm starts by randomly picking cluster centers and then refining these guesses. As such, its result might depend on this random guess. It might be good to change the numpy.random.seed(#) value at the top of the code, to see the effect that changing the initial guesses might have. If the groups are fairly well separated, then the algorithm should be stable.

**Prediction**

We can use this algorithm to predict the identity of a new sample. The algorithm would find which centroid is closest to our given data point, and assign the point to that respective cluster.

**Exploring the Model and Hyperparameters**

Here’s a set of 2D data.

x1 x2 marker

0 -5.581006 2.453801 +

1 -4.775744 6.493566 +

2 -2.498410 3.883308 +

3 -7.721780 4.778397 +

4 -4.800134 2.625327 +

5 -5.095982 2.983548 +

6 -5.712459 3.620277 +

7 -7.177118 4.828715 +

8 -5.702035 6.645507 +

9 0.176800 6.832048 +

10 -6.011302 8.433696 +

11 -6.168618 6.038520 +

12 0.657432 -4.065840 o

13 -0.835175 -4.746274 o

14 -2.262751 -3.368424 o

15 -2.410987 -4.609493 o

16 -5.672407 -3.274687 o

17 -5.086841 -5.121301 o

18 -6.273988 -4.602747 o

19 -0.343871 -3.356554 o

20 -2.165032 -3.279266 o

21 -1.564511 -4.390938 o

22 -0.995685 -6.681225 o

23 -3.409054 -1.810766 o

24 1.648725 -0.563825 ^

25 3.404706 -1.865709 ^

26 2.892774 0.711521 ^

27 2.889918 -1.585560 ^

28 4.892444 2.111042 ^

29 5.665094 3.221986 ^

30 3.398538 1.512025 ^

31 2.619147 2.509115 ^

32 3.591971 -2.887745 ^

33 4.067365 1.741439 ^

34 5.401179 1.658608 ^

35 2.007351 -0.467921 ^

I markered the points according to their classes. And the class coordinates were taken from a normal distribution with a mean (centroid) and standard deviation (unique to each class). Then I colored the points according to their classes as well, and plotted below:

A diagram of a plot of classes

Description automatically generated

The clustering below is done with scipy, though sklearn could’ve been used just well.

**Hyperparameter: n\_clusters**

So there’s really just one hyperparameter, and that’s the number of clusters you want to use. So I’ll just plot a few different n’s,

A diagram of a number of red dots and green dots

Description automatically generated A diagram of a number of dots

Description automatically generated

A diagram of a number of dots

Description automatically generated A diagram of a number of dots

Description automatically generated

A diagram of different colored dots

Description automatically generated A diagram of a number of dots and points

Description automatically generated with medium confidence

When we were doing the hierarchical mapping, we could visually work out the best number of clusters by seeing which one had the longest ‘legs’ in the dendrogram. Here we use an elbow plot, i.e., a plot of dissimilarity vs. n.

A graph with a line

Description automatically generated

And this plot does suggest that n = 3 clusters is the right choice.