Course reader: *Data clusters*

- Data can be drawn from *clusters*. Clusters may be in space (like stars forming galaxies) or items from a questionnaire (like age and previous programming experience), or any other dimensions.
- Clusters can be defined based on distance to a centroid, local proximity, or other features. There is no single optimal clustering method (that should come as no surprise by this point of the course!), because different kinds of data can have different kinds of clusters.
- Therefore, algorithms to cluster data is an important and active area of research in machine learning and statistics.
- The goal of this section is to show how to simulate clusters in any number of dimensions. In the videos I show 2D and 3D because those can be easily visualized, but the principle extends to any number of dimensions.
- The purpose of simulating clustered data is to test clustering algorithms on the simulated data so you can learn about those algorithms and what their advantages/limitations are.
- Although this course is not focused on analysis algorithms, the k-means clustering method is illustrated. kmeans works by minimizing *intraclass distance* while maximizing *interclass distance*. Basically, this means that cluster centers are selected such that data points in the cluster are close to the center and also far away from other cluster centers.
- The kmeans algorithm works well for data that have a linear distance-based clustering. You'll see in the exercises that because kmeans is a stochastic algorithm, the results can change at each iteration.
- It's difficult to know how many clusters to extract. You'll see this in the exercises as well.

Exercises

1. Because you know the ground-truth, you can assess the accuracy of the results of kmeans. From the line `data = [a; b; c];` you know that the first 1/3 of data points are in group "A," the next third in group "B," and so on. The first output of the `kmeans` function is called `groupidx` in the code. That variable contains a group assignment for each data point (arbitrarily coded as 1, 2, or 3). Regardless of the numerical value, the first third of the numbers should have the same value, and same for the second and third thirds of numbers. Figure out a way to compute the accuracy of kmeans.

2. Now that you can compute accuracy, test how the accuracy depends on the blur parameter (standard deviation of the data points around the centroid).

3. K-means clustering is an iterative procedure with random weights. It starts by guessing cluster centroids randomly and then moving them around a bit to try improve the balance between within-cluster and across-cluster distances. Such an approach is useful because it helps prevent the algorithm from getting stuck in a minimum or overfitting noise. But it can create some ambiguity. Run k-means clustering on the 2D data (using k=3 clusters) many times with the same random data. Does the assignment of data point to cluster change on each iteration? How about which group is given which integer value (color)?

4. How did I know to use k=3 clusters? I knew because I created the data. But with real data, you don't know how many clusters there really are (otherwise you wouldn't need to do the analysis!). Run kmeans many times for k=2, 3, 4, 5, and 6. What happens to the results for repeated runs using the wrong vs. correct number of clusters? Think of an algorithm that could be used to determine an appropriate number of clusters. (Note that implementing the algorithm is option; just try to think about how such an algorithm should work based on your simulations and observations.)

5. The code generates only symmetric clusters, meaning that the spatial variance is the same in all directions around the cluster centroid. Modify the code to create asymmetric clusters by having the variance along one dimension be half the variance along the other dimension. Does that affect the clustering results?

Answers

1. One possible solution below, which assumes that the within-category accuracy is at least 50% for all groups.

PYTHON

```
acc = np.zeros(k)
for i in range(k):
# indices of this group
idx = np.arange( i*nPerClust+1 , i*nPerClust+nPerClust)

# compute accuracy assuming >50% accuracy
acc[i] = np.mean( groupidx[idx]==np.round(np.mean(groupidx[idx])) )

print('Total accuracy is %g%%' %np.round(100*np.mean(acc),2))
```

MATLAB

```
acc = zeros(k,1);
for i=1:k
   % indices of this group
   idx = (i-1)*nPerClust+1 : (i-1)*nPerClust+nPerClust;

   % compute accuracy assuming >50% accuracy
   acc(i) = mean( groupidx(idx)==round(mean(groupidx(idx))) );
end

disp([ 'Total accuracy is ' num2str(100*mean(acc)) '%' ])
```

2. No big shocker here — accuracy decreases when the blur increases. For bonus points, you can program this in a loop to create a plot of overall accuracy as a function of the blur parameter.

3. The answer depends partly on the blur parameter (let's assume you have it between 1 and 2). There are occasional data points that get swapped around into different clusters, but they are generally stable (again, depending on cluster centroid distance and blur). On the other hand, the assignment of number/color to group is totally random and changes on each iteration.

4. For any cluster number other than k=3, you can re-run the clustering a few times and get wildly different results. That's your clue that the number of clusters is wrong. An algorithm to estimate the correct number of clusters would be something that runs kmeans a bunch of times per k, then finds the best centroids from those runs ("best" means minimizing intracluster distances and maximizing intercluster distances), then onto the next k, and so on. Finally, the optimal k can be taken as the k with the best distances.

5. Apply a stretch parameter to one of the dimensions, like this:
`a = [ A[0]+np.random.randn(nPerClust)*blur ,`

```
A[1]+np.random.randn(nPerClust)*blur*.5 ]

a = [ A(1)+randn(nPerClust,1)*n A(2)+.5*randn(nPerClust,1)*n ];
```