Lecture 5: Clustering

Project matchings are currently underway.

Last time we started looking at unsupervised techniques, where we were interested in looking at the structure of the data. We weren't interested in predicting a variable, but more in the structure of the data. In ordert to describe structure, we needed a way of finding similarity/dissimilarity between points.

Clustering is a way of grouping points that are similar. It is a grouping/assignment of data points such that objects in the same cluster are:

- similar to one another
- dissimilar to objects that are in a different group.

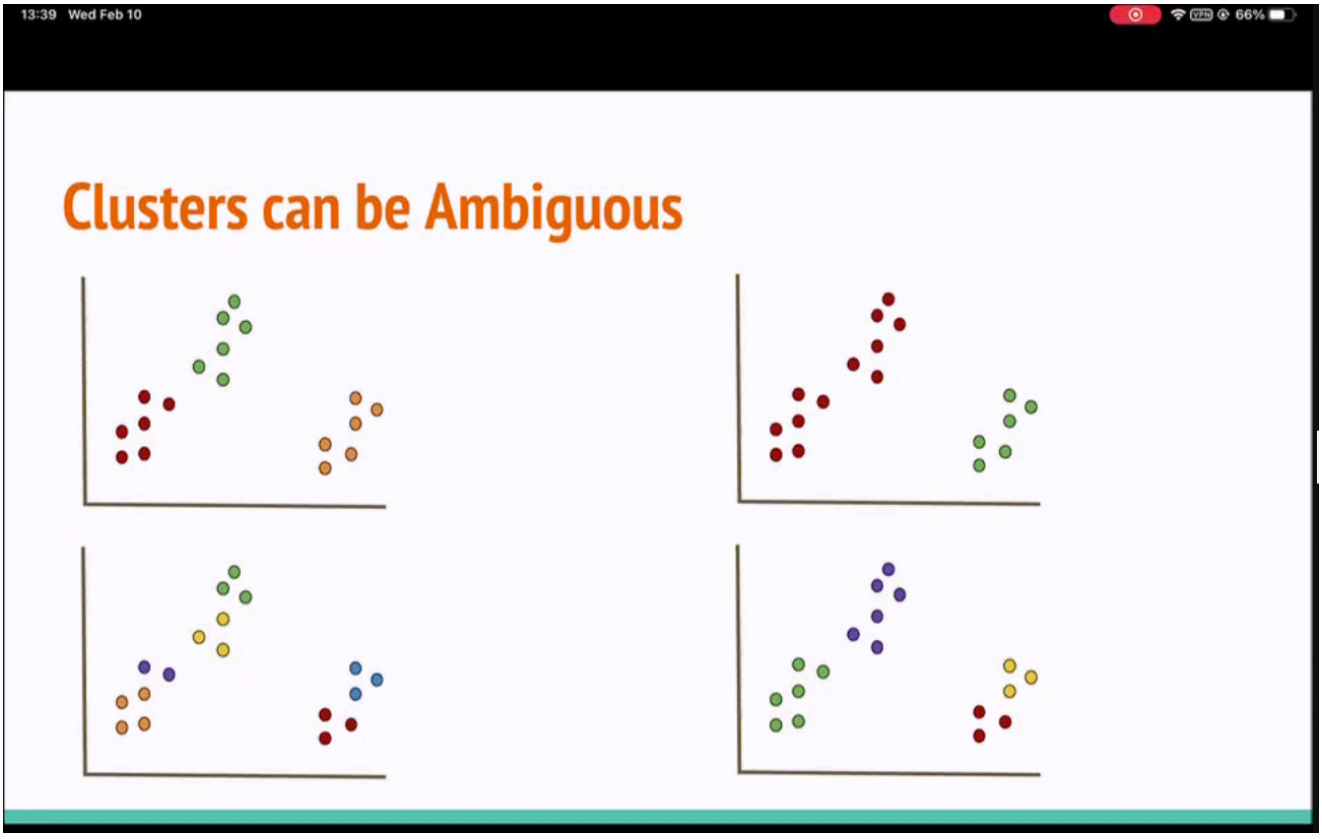
For example, we might assign a color to each group, based on some logical grouping of the points relative to each other. We don't really know what each color (grouping/cluster) means, only that the data points contained are related.

our objective then is to be able to find a clustering. We want to define a process that can give us a clustering. We need an algorithm that will assign each data point.

There are a few questions we need to answer:

1. What does **similar** mean?
2. how do we find a **clustering**?
3. Once we've generated a clustering, can we evaluate how good of a clustering it is? Can we quantify it? Can we compare different clustering quantitatively?

Inherently, this can be a very difficult problem - it's not easy to determine what clusertings are valid and which aren't, or if some are better than others.



There are a number of types of clustering:

- Partitional: each object belongs to precisely one cluster, and the data is partitioned in k clusters.
- Hierarchical: A set of nested clusters organized into a tree. Any point at which we decide to cut the tree generates its own cluster.
- Density-based: defined based on the local density of points - one chunk of data is quite dense, and hence that counts as a cluster.
- Soft clustering: Each point is assigned to every single cluster, with a certain probability.

We'll look at example of why we would use each of these.

Where do we use clustering in the real world?

- Anamoly detection - spam filters, fraudulent credit card uses, etc.
    - Anamolies: Normal things fall into one cluster (or set of clusters), and then outliers or anamolies are sufficiently different that they would fall into a different cluster.
    - Outlier detection - where you have data that is normally representative, but some that are so far out that they would poorly impact the final outcome.
- "Filling in the gap in the data" - maybe we have some non-values, and we want to fill them in, in a meaningful way, so we cluster points together to find what should be meaningful values for the missing points.
- Other areas, like identifying user preferences, global trends, etc.

For many reasons, this can motivate the data privacy conversation, because giving your data can possibly compromise someone else - your data might reveal trends about others that bad people can take advantage of. In general, partitioning is a simple technique, but it's a very commonly used one and can be quite powerful.

Partitional Clustering

Given n data points and we want to partition them into k clusters. What do we have to know up front? We need to know the number of data points, and we need to know the number of clusters you want. Thus, we won't really be able to find some "best partition" and use that to find k - we need to specify k up front. This can be somewhat of a limitation, since it's hard to know what k should be.

Suppose we are given all possible ways of distributing these n data points into these k buckets. How would we find the best such partition? Recall that the goal is that similar items should belong to the same cluster, and dissimilar items belong to different clusters. It would be nice if we had a general number that exists over all these buckets, and we want to boil this number down to "how good" each cluster is. Here, we can say that points that are similar should have small distances between each other - this should be true for each cluster: that all pairs of points for each cluster, sum their distances, that value should be small. We want this for every cluster, so we can sum the sums of each cluster, which would yield a total value.

A good partition is one where the total dissimilarity of points within each cluster is small.

This pattern comes up in data science a lot - where we have a cost associated with an algorithm, which we want to minimize (optimize).

Intra-cluster distance: the sum of the distances between each pair of points within a cluster. That is:

$$\sum_{k}^{K} \sum_{x_i, x_j \in C_k} d(x_i, x_j) \tag{1}$$

This is the double sum: the first is over all clusters, and the second is the sum of the distances of the points within each cluster. C_k: the k^{th} cluster. K: set of all clusters. x_i, x_j: a pair of points within a cluster C_k

Centroid: intuitively, this is the center of mass of a given function.

Question: when d is the Euclidean distance, what is the centroid (also called the center of mass) of m points $x_1, x_2, \ldots, x_m$?

Answer: It is the mean - the average of all the points.

It turns out that when the distance is Euclidean, we have this nice property that we can just look at each point's distance from the centroid. The idea is that optimizing the pairwise distances is the same thing as minimzing the sum of each point to its cluster's centroid.

$$\sum_{k}^{K} \sum_{x_i, x_j \in C_k} d(x_i, x_j)^2 = \sum_{k}^{K} |C_k| \sum_{x_i \in C_k} d(x_i, \mu_k)^2 \tag{2}$$

where $|C_k|$ is a scalar that is the size of C_k, meaning the number of points in that cluster.

K-means

Algorithm that will minimize the cost function: Given $X = x1, x_2, \dots x_n$ our dataset, and k, find k points $\mu_1, \dots, \mu_k$ that minimizes the costs function:

$$\sum_{i}^{k} \sum_{x \in C_i} ||x - \mu_i||_2^2 \tag{3}$$

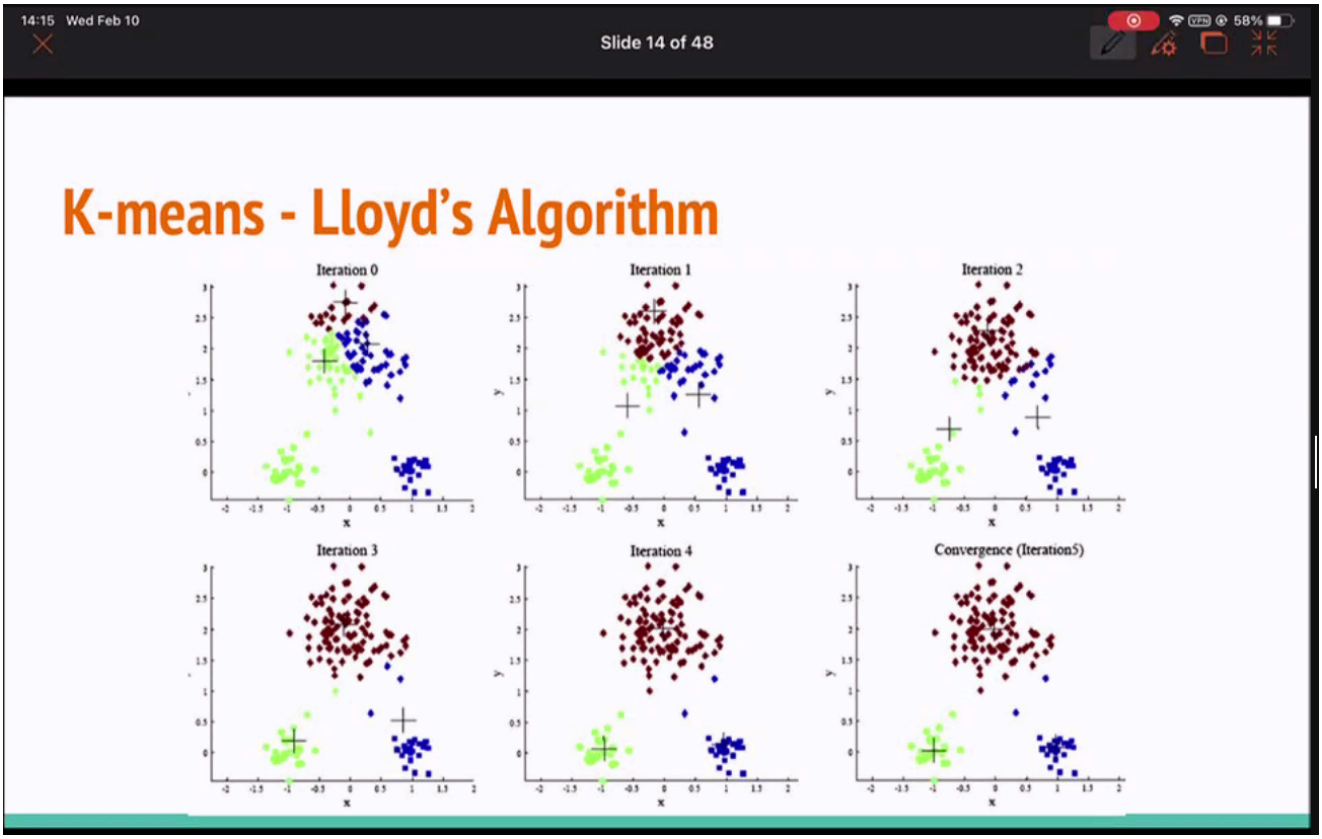
This is the Euclidean distance, also called the L2-norm, which is basically the distance between points. Here, each μ_i is the centroid of the i^{th} cluster.

It turns out that solving this problem, when x_i is more than two-dimensional, is very hard (NP-hard in fact). However, when $k = 1$ and $k = n$ this is easy. We need to find some sort of balance.

How do we generate this clustering? We can use **Lloyd's algorithm**:

1. Randomly pick k centers $\mu_1, \dots, \mu_k$
2. Assign each point to its closest center
3. Compute the new centers as the means of each cluster. For each cluster, this will almost certainly compute a new center.
4. Repeat the previous two steps until we reach a convergence. Note that when we re-computed the centers, this opens up the possibility that points will shift clusters.

What is convergence? This means that the points assigned as centers for each cluster are the same as the previous iteration.



Question: Will this algorithm always converge? Answer: yes. Proof by contradiction:

- Suppose that the algorithm does not converge. This implies that either:
  - The minimum cost of the function is only reached in the limit (that is, it would take an infinite number of iterations to get to the minimum)
    - This however, is impossible because we are iterating over a finite set of partitions.
  - The algorithm got stuck in a loop or a cycle
    - This again is impossible, becasue this would require that a clustering has a lower cost than itself, and we know that:
      - if old $\neq$ new, then clustering cost has improved
      - if old $=$ new, then the cost is unchanged.

Thus, by contradiction, we have proved that this algorithm will always converge.

However, note that the initial choosing of points has a large influence on the final clusters, and it is possible to end up with different clusters depending on where the initial center points are. Lloyd's algorithm does not necessarily always converge to the optimal solution.

One of the solutions here is to run Lloyd's algorithm multiple times, and choose the result with the lowest cost - however, this can still lead to bad results because it's in a probibalistic setting, and who knows how it will do? Also, we may not be able to run the algorithm many times if the data set is large enough. Instead, we'd like to try different initialization methods.

- We already discussed random initialization points - this can lead to problems if we randomyl choose points that are too close to one another.
- Farthest first traversal (FFT): pick the first point randomly, but then the second point is chosen as the further possible data point from the randomly chosen one, the third is the further from both of those, etc.
    - The problem here is this is highly sensitive to outliers.

K-means++

Initialize with a combination of the two methods.

1. Start with a random center
2. Let $D(x)$ be the distance between x and the centers selected so far. Choose the next center with probability proportional to $D(x)^a$
    1. The idea is that points further out have a higher probability of being chosen, and points closer have a lower probability of being chosen.

We do this with respect to a parameter a - think about what values a can be:

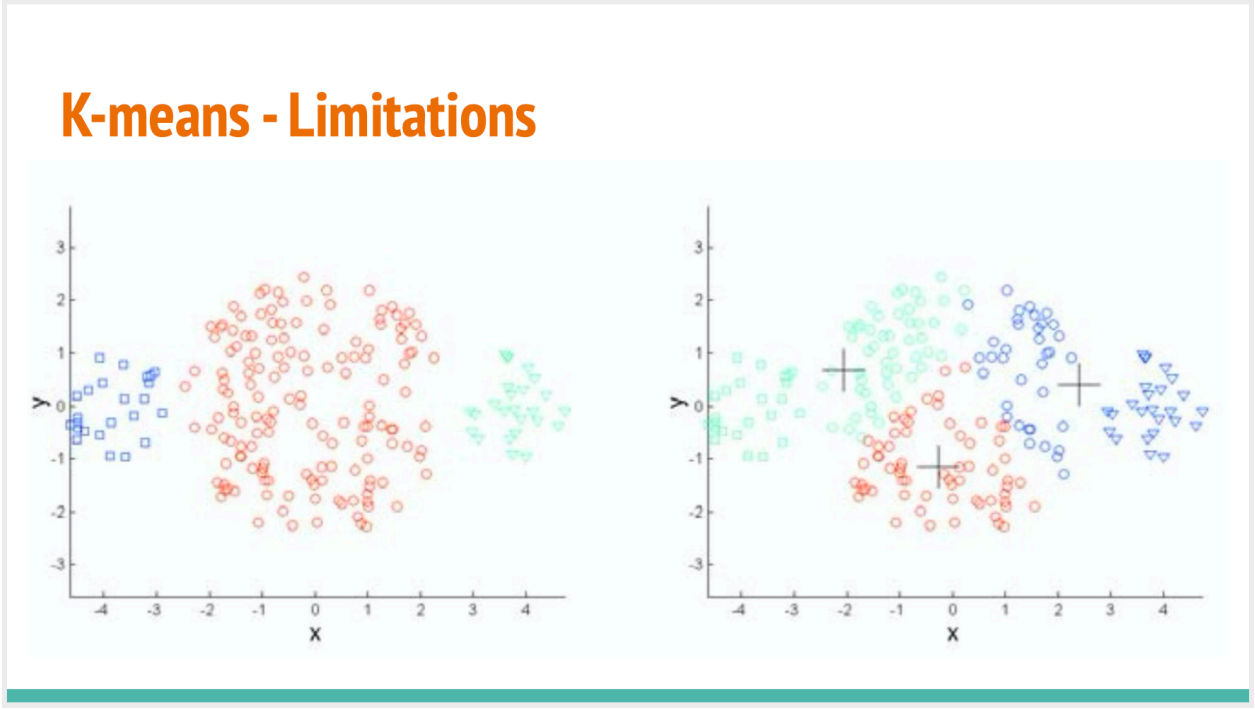
- If $a = 0$, this is random initialization
- If $a = 1$, this is Farthest First Traversla(FFT)
- if $a = 2$ - this is k-means++, where we are using the squared distance as a probability of choosing a certain point.

Suppose that we are given a black box that will generate a uniform random number between 0 and N. how can we use this black box to select points with probability propotional to $D(x)^2$? Meaning, if $x = 3$, we have 9, if $x = 2$ we have 4, and if $x = 1$ we have 1. Just add up the sum of these squared distances, and use that as N - now, we want to use this random number generator such that we have a 9/14 chance of selecting the point with $x = 3$, a 4/14 chance of selecting the point with $x = 2$, and a 1/14 chance of selecting the point with $x = 1$.

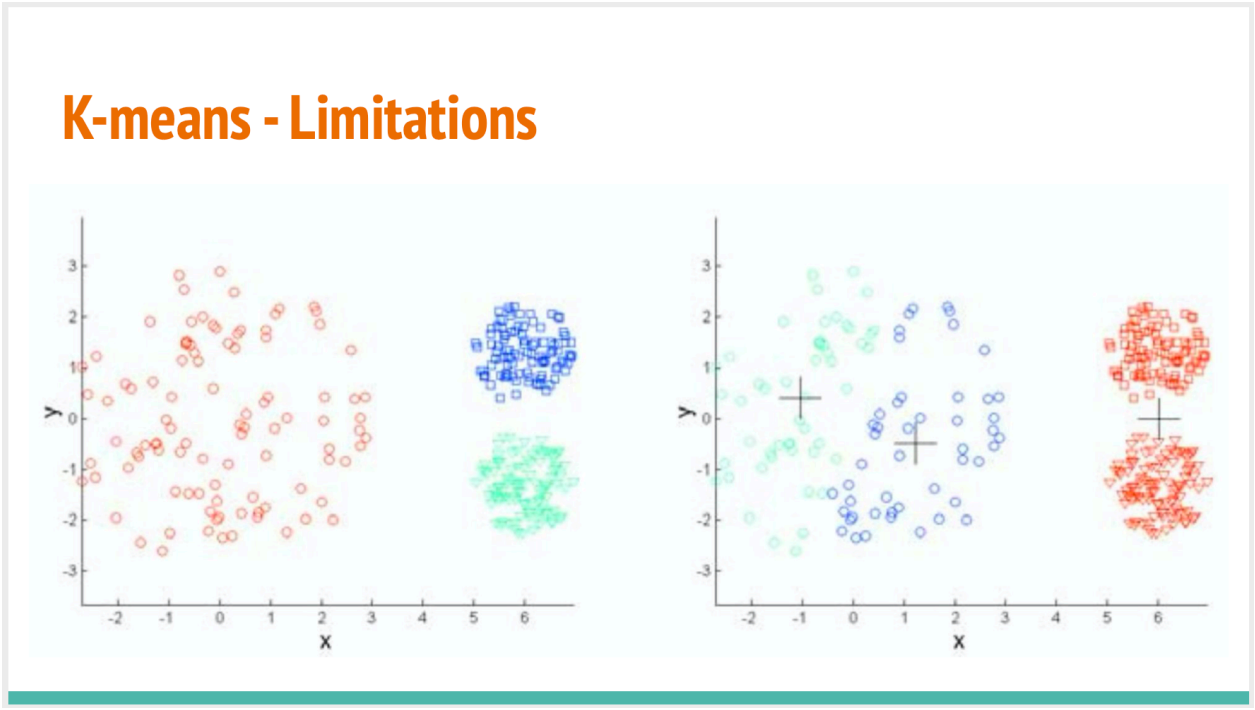
What happens tho if our black box only produces a number between 0 and 1? Can we still use the same method? Absotluly, just normalize by N.

Some limitations of k-means:

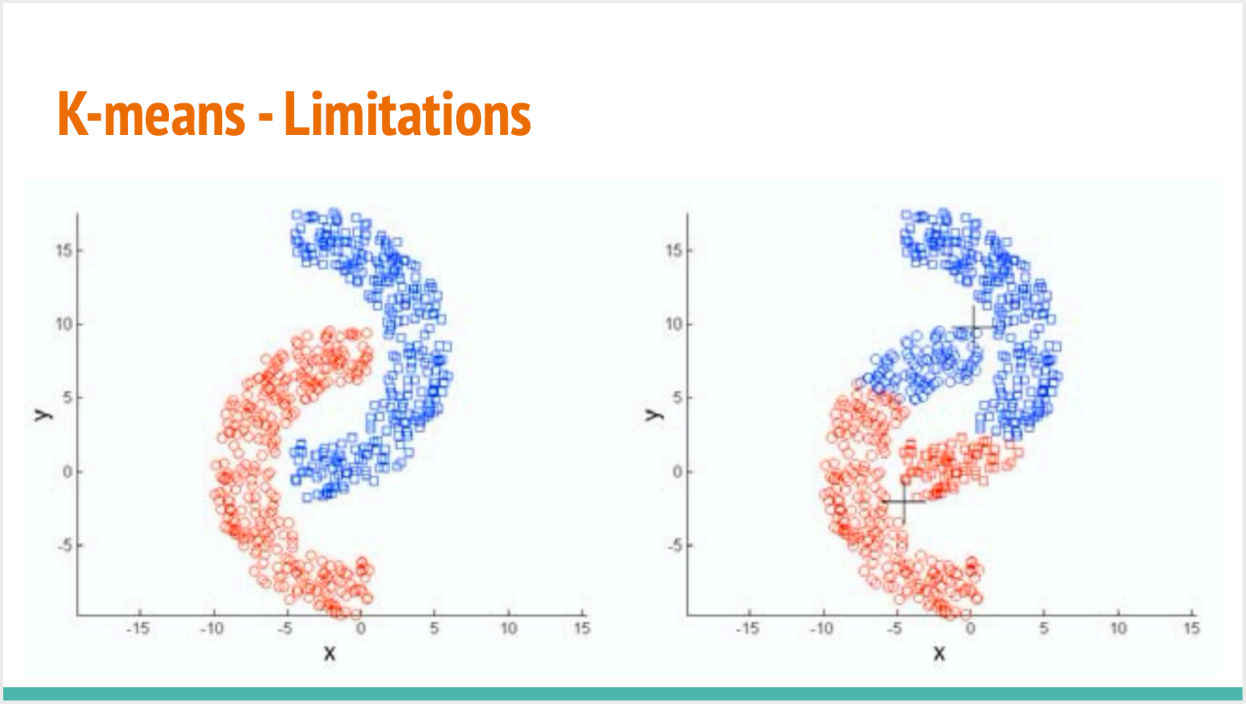
- It generally tries to find clusters of the same size: ON the left is what k-means produced, but on the right is the clustering we wanted:

- 

- It's not very good with clusters of different densities: the left is what was generated, the right is what we wanted

- 

- It tries to find globular shapes: the left is what was generated, the right is what we wanted.

- 

Choosing k

Unfortunately, there is no magic method for choosing the right k. We can however, use what is called the *elbow method*, where we iterate through different values of k. Here, we are basically looking for the points that has a good balance of cost relative to the number of buckets. It's called the elbow method because if we look at a curve of cost vs k, there is a point on the resulting chart that looks like a bent elbow. That is the point that we would want to choose, where we have a good balance between the cost and the number of buckets.

We can also use empirical or domain knowledge -we may know previously based on the specific problem how many clusters we want.

Variations:

There are a few variations on this method, including:

- K-medians - here, we use the L1 norm or the Manhattan distance for the center points, instead of the means.
- K-medoids: any distance function and the centers must be in the data set (can't just choose random values, must use actual data points as the centers)
- Weighted k-means: each point has an associated weight when using the k-means algorithm.