Week 3 - Clustering with k-means

In supervised learning tasks, we have training data (input) and known labels (output). We learn from the labeled training data and we can then apply the learned model to take other inputs and create the associated outputs.

Clustering is a unsupervised learning task. Our data has no labels. The task is to learn labels from the data – to learn structure inherent in the data. So the input in still the data and the output is still a label, but we don't know the labels before we learn them from the data.

For instance, in our work with documents, we may with to know how the documents are associated. Our inputs are a set of documents, each as a feature vector x_i. The output is a cluster labels, z_i.

What defines a cluster?
A cluster is defined by

- it's center, sometimes called the centroid
- the shape of the cluster (for instance, and ellipse).

The clustering algorithm assigns an observation xi (a document in our case) to cluster k (a document topic in our case) if xi's score under cluster k is higher than it is under other clusters. Score is often defined in terms of distance to the center of the cluster.

The k-means algorithm

0. Initialize the cluster centers (algorithm tbd).
1. Assigned observations to the closest cluster center.
2. Revise cluster centers to be the mean of the assigned observations.
3. Repeat 1 and 2 until convergence (definition TBD).

0. Initialize cluster centers.
- we choose k cluster centers, μ_1, μ_1,…, μ_k. The way we do this can vary. This will be discussed later.

1. Assigned observations to the closest cluster center.
We do this by calculating the distance of the observation to each cluster center and choosing the cluster with the minimum distance.

$$z_i \leftarrow \underset{j}{\arg min} \left\|\mu_j - x_i\right\|_2^2$$

where

z_i	the cluster label for observation i.
j	the cluster number 1..k
μ_j	the cluster j. This is a vector that represents a point in the same d dimensional space as each observation.
x_i	the observation, a d dimensional vector
$\left\|\mu_j - x_i\right\|_2^2$	calculate the distance metric between cluster center j and observation x_i. It is short hand for calculating the sum of the squared differences.
$\underset{j}{\arg min}$	return the index, j, of the minimum iteration of the function. In this case, the function is our distance calculation across all cluster centers μ_j for j = 1..k. In other words, calculate the distance between the observation x_i and all cluster centers, using j as the cluster number and return the cluster number where the distance is minimum.

2. Revise cluster centers to be the mean of the assigned observations.
Calculate the mean of all the observations and the cluster and use this as the new cluster center. Do this for all clusters.

$$\mu_j = \frac{1}{n_j} \sum_{i:z_i=j} x_i$$

where:

μ_j	the center of cluster j; a vector of dimension d
n_j	the number of observations in cluster j
x_i	observation i; a vector of dimension d
z_i	the cluster label for observation i.
$\sum_{i:z_i=j} x_i$	the sum over all observations, x_i, where the observation's cluster center, z_i, is cluster j

3. Repeat 1 and 2 until convergence (definition TBD).
Go back and reassign all observations to the new cluster centers, then recalculate the cluster centers as the mean of the new set of observations for each cluster. Do this until we detect we are done (often, until the center's position does not change within some limit).

Aside:

$\|x\|_2$	This is the L-2 norm, or Euclidean norm, for the vector x. The Euclidean norm is the most commonly used vector norm and is commonly shown as simply $\|x\|$. This is the square root of the sum of the squared elements (technically, squared absolute values).
$\|x\|_2^2$	This is the L-2 norm squared. In effect, it removes the square root, so it is the sum of the squared vector elements.

It may be worth reviewing the Euclidean distance section for the week 2 notes. Quickly; Euclidean distance in n-space between points p and q;

$$distance(p,q) = \|p-q\|_2 = \sqrt[2]{(p_1-q_1)^2+(p_2-q_2)^2+\cdots+(p_n-q_n)^2}$$

so

$$\|p-q\|_2^2 = (p_1-q_1)^2+(p_2-q_2)^2+\cdots+(p_n-q_n)^2$$

So the L-2 Norm for a difference vector is the Euclidean distance between the two vectors. The L-2 squared is the squared distance, which is simpler to calculate (because we don't take a square root) and still provides a useful distance/similarity metric.

k-means as coordinate descent

We can think of the k-means algorithm as an iterative coordinate descent algorithm. Remember that coordinate descent involves iteratively minimizing a goal value until we reach convergence. In steps 1 and 2 of k-means, we are doing minimizations;

1. Minimize z given μ

$$z_i \leftarrow \underset{j}{\arg min} \|\mu_j - x_i\|_2^2$$

2. Minimize μ given z

$$\mu_j = \frac{1}{n_j}\sum_{i:z_i=j} x_i$$

This can be re-written as a minimization of the distance from the cluster center to all observations in the cluster:

$$\mu_j = \underset{\mu}{\arg min} \sum_{i:z_i=j} \|\mu - x_i\|_2^2$$

Convergence of k-means

- NOT guaranteed to a global optimum
- we are guaranteed to converge to a local optimum, because we can cast the process as a coordinate descent. However, the quality of the minimization can vary because there are lots of local modes.

The quality of the minimization

- it is common to use the size of the change in the center (or how close it is to zero) as the metric for convergence. Once the center is 'not moving' we are done.
- the choose of initial centers can have a dramatic effect of the local minimum that we find and so it has a dramatic effect on where the clusters are and which observations are assigned to them.

Assessing convergence

How can we tell if the k-means algorithm is converging?

- Look to see if centroids stabilize (stop moving within some limit)
- Look at the cluster assignments and see if they stabilize over time.

In practice, **it is common for the algorithm to run until the cluster assignments stop changing**. Note that convergence does not say anything about the actual quality of the clustering. That requires other metrics, such as the sum of all squared distances between data points and centroids (see below: *Assessing the quality and choosing the number of clusters*).

It is also common, so that the algorithm doesn't run too long (which can cost money), to stop after a given maximum number of iterations if the algorithm has not otherwise converged.

Beware of local minima

One weakness of k-means is that it tends to get stuck in a local minimum. Run k-means multiple times with different initial centroids created using different random seeds can show the variation in heterogeneity for different initializations. So a poor initialization can lead to arbitrarily heterogeneous (bad) clusters.

One effective way to counter this tendency is to use k-means++ to provide a smart initialization (see below).

Smart initialization via k-means++

The k-means++ method tries to spread out the initial set of centroids so that they are not too close together. It is known to improve the quality of local optima and lower average runtime

(this section summarized from Wikipedia)

The k-means algorithm has at least two major theoretic shortcomings:

- First, it has been shown that the worst case running time of the algorithm is super-polynomial in the input size.[5]
- Second, the approximation found can be arbitrarily bad with respect to the objective function compared to the optimal clustering (it can lead to arbitrarily heterogeneous clusters).

The k-means++ algorithm addresses the second of these obstacles by specifying a procedure to initialize the cluster centers before proceeding with the standard k-means optimization iterations.

The intuition behind k-means++ algorithm is that spreading out the k initial cluster centers is a good thing: the first cluster center is chosen uniformly at random from the data points that are being clustered, after which each subsequent cluster center is chosen from the remaining data points with probability proportional to its squared distance from the point's closest existing cluster center.

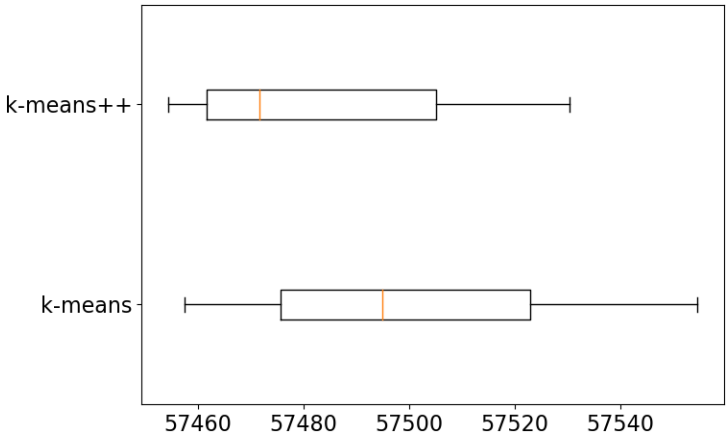
The exact algorithm is as follows:

1. Choose one center uniformly at random from among the data points.
2. For each data point x, compute D(x), the distance between x and the nearest center that has already been chosen.
3. Choose one new data point at random as a new center, using a weighted probability distribution where a point x is chosen with probability proportional to $D(x)^2$.
4. Repeat Steps 2 and 3 until k centers have been chosen.
5. Now that the initial centers have been chosen, proceed using standard k-means clustering.

This seeding method yields considerable improvement in the final error of k-means. Although **the initial selection in the algorithm takes extra time**, the k-means part itself converges very quickly after

this seeding and thus **the algorithm actually lowers the overall computation time**. The authors tested their method with real and synthetic datasets and obtained typically 2-fold improvements in speed, and for certain datasets, close to 1000-fold improvements in error.

For illustration, below is a figure that shows a box and whiskers plot for 7 randomized runs of k-means and k-means++.



Cluster Sizes for k-means and k-means++ across 7 randomized runs

The box plot shows:

- On average, k-means++ produces a better clustering than Random initialization.
- Variation in clustering quality is smaller for k-means++.

Assessing the quality and choosing the number of clusters

To assess the quality of one clustering versus another clustering for the same observations we can use the objective function of k-means; minimizing the sum of the squared distances from the observations to their cluster centers. So we are minimizing this function:

$$\sum_{j=1}^{k} \sum_{i:z_i=j} \|\mu_j - x_i\|_2^2$$

where:

$\sum_{i:z_i=j} \|\mu_j - x_i\|_2^2$ the sum of the squared distances in cluster j

$\sum_{j=1}^{k}$ the sum over all clusters

We can refer to this as the sum of the cluster heterogeneity (where clusters with larger distances are more heterogeneous – less similar).

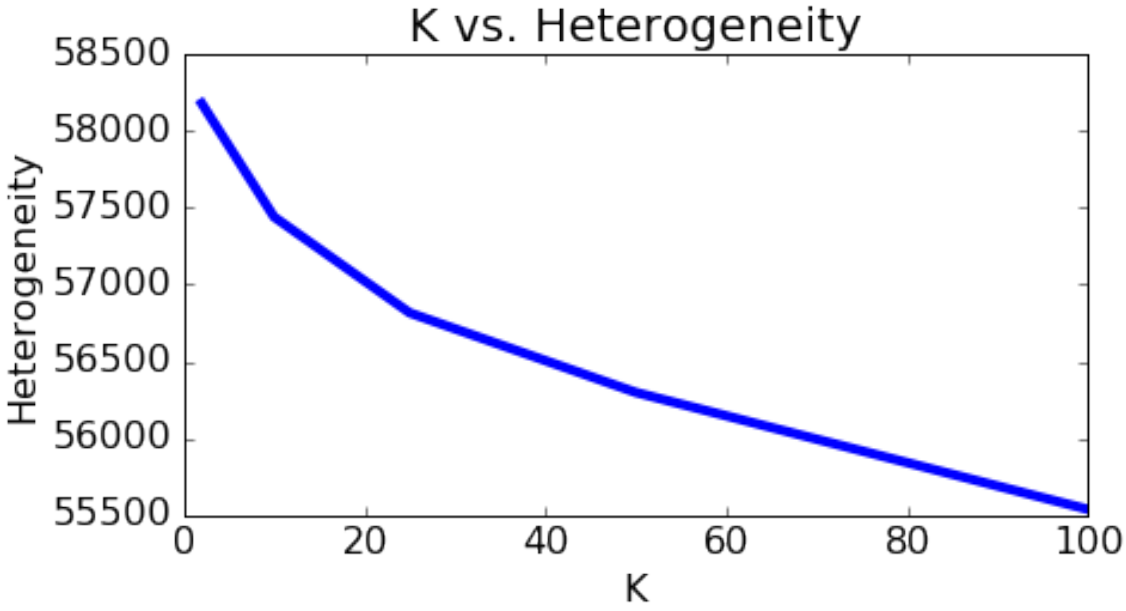
What happens as k increases?

As the number of clusters approaches the number of observations, we run into the danger of over-fitting. In the extreme case of k == N, the cluster homogeneity is zero; each of k clusters has a single element, x_i, which then is also the cluster center. So the μ_j-x_i component is zero, and so the set of summations end up as zero.

The extreme example shows the trend; the **possible cluster heterogeneity decreases (possible similarity increases) as k increases**. When comparing any two arbitrary runs at k and k + 1, it is possible

that k clusters are less heterogeneity than k + 1 clusters; so the trend can be seen over lots of runs, but not necessarily within a small number of runs.

How do we choose k?
How do we handle this over-fitting when doing unsupervised learning? We can see from a graph of heterogeneity vs increasing k, that there is an elbow point; heterogeneity decreases rapidly at first, but then the decrease slows more and more. This is a single run and does not then show us the theoretically lowest homogeneity for a given k, but it points the way.



The plot shows that heterogeneity goes down as we increase the number of clusters. However, setting K too high may end up separating data points that are actually pretty alike. At the extreme, we can set individual data points to be their own clusters (K=N) and achieve zero heterogeneity, but separating each data point into its own cluster is hardly a desirable outcome.

Traits of good clustering:
- Documents in the same cluster should be similar.
- Documents from different clusters should be less similar.

Symptoms of bad clustering:
- Documents in a cluster have mixed content (k is too small)
- Documents with similar content are divided up and put into different clusters (k is too large).

A high value of K encourages pure clusters, but we cannot keep increasing K. For large enough K, related documents end up going to different clusters.

How much granularity do we want in our clustering? If we wanted a rough sketch of Wikipedia, we don't want too detailed clusters. On the other hand, having many clusters can be valuable when we are zooming into a certain part of Wikipedia. So **there is no fixed rule for choosing K**. The choice of K is very much about what we are trying to accomplish by doing the clustering in the first place – **it depends upon the particular application and domain**.

So the heuristic is to do enough clusterings with randomized starting values and increasing values of k in order to get a decent approximation of the lowest-possible heterogeneity (highest cluster tightness) versus

K curve. Then pick a value of K at the elbow of the curve where the heterogeneity decreases rapidly before this value of K, but then only gradually for larger values of K. This naturally trades off between trying to minimize heterogeneity, but reduce model complexity. In the heterogeneity versus K plot made above, we did not yet really see a flattening out of the heterogeneity, which might indicate that indeed K=100 is "reasonable" and we only see real overfitting for larger values of K (which are even harder to visualize using the methods we attempted above.)

It is important to understand that tiny clusters aren't necessarily bad. A tiny cluster of documents that really look like each other is preferable to a medium-sized cluster of documents with mixed content. However, having too few articles in a cluster may cause overfitting by reading too much into limited pool of training data and so separating articles that are actually alike.

k-means with text documents

Euclidean distance can be a poor metric of similarity between documents, as it unfairly penalizes long articles. For a reasonable assessment of similarity, we should disregard the length information and use length-agnostic metrics, like cosine distance. To remove length information: normalize all vectors to be unit length. Euclidean distance closely mimics cosine distance when all vectors are unit length. In particular, the squared Euclidean distance between any two vectors of length one is directly proportional to their cosine distance.

Remember from the week 2 discussion of cosine distance, the distance between vector x_i and query vector x_q;

$$\cos\theta = \left(\frac{x_i}{\|x_i\|}\right)^T \cdot \left(\frac{x_q}{\|x_q\|}\right) = \frac{{x_i}^T \cdot x_q}{\|x_i\|\|x_q\|}$$

Further, remember the discussion of Euclidean distance between two vectors:

$$distance(x_i, x_q) = \sqrt{(x_i - x_q)^2} = \sqrt{(x_i - x_q)^T \cdot (x_i - x_q)} = \|x_i - x_q\|$$

Therefore:

$$distance(x_i, x_q)^2 = \|x_i - x_q\|^2 = (x_i - x_q)^T \cdot (x_i - x_q) = \|x_i\|^2 - 2({x_i}^T x_q) + \|x_q\|^2 = 2 - 2({x_i}^T x_q)$$
$$= 2\left(1 - \frac{{x_i}^T x_q}{\|x_i\|\|x_q\|}\right)$$

This tells us that the squared Euclidean distance between any two vectors of length one is directly proportional to their cosine distance and so two unit vectors that are close in Euclidean distance are also close in cosine distance. Thus, the k-means algorithm (which naturally uses Euclidean distances) on **normalized vectors will produce the same results as clustering using cosine distance as a distance metric**.