Machine Learning
the Fundamentals

William J. Deuschle

Harvard College
Cambridge, Massachusetts
April, 2019

Contents

Chapter 6

Clustering

In this chapter, we will explore a technique known as clustering. This represents our first foray into unsupervised machine learning techniques. Unlike the previous four chapters, where we explored techniques that assumed a dataset of inputs and targets, with the goal of eventually making predictions over unseen data, we no longer have an explicit target in mind. These techniques are instead motivated by the goal of uncovering structure in our data. Identifying clusters of similar data points is a useful and ubiquitous unsupervised technique.

6.1 Motivation

The reasons for using an unsupervised technique like clustering are broad. We often don't have a specific task in mind; rather, we are trying to uncover more information about a potentially opaque data set. This information typically indicates some measure of *distance* between our data points. While there are a variety of clustering algorithms available, the importance of this distance measurement is consistent between them.

Distance is meant to capture how 'different' two data points are from each other. Then, we can use these distance measurements to determine which data points are similar, and thus should be clustered together. A common distance measurement for two data points $\mathbf{x}$ and $\mathbf{x}'$ is given by:

$$||\mathbf{x} - \mathbf{x'}||_{L2} = \sqrt{\sum_{d=1}^{D}(\mathbf{x}_d - \mathbf{x'}_d)^2} \tag{6.1}$$

where D is the dimensionality of our data. This is known as **L2 distance**, and you can likely see the similarity to L2 regularization.

There are a variety of distance measurements available for data points living in a D-dimensional Euclidean space, but for other types of data (such as data with discrete features), we would need to select a different distance metric. Furthermore, the metrics we choose to use will have an impact on the final results of our clustering.

ML Framework Cube: Clustering

Clustering algorithms can operate on both continuous and discrete feature spaces, are fully unsupervised, and are non-probabilistic for the techniques we explore in this chapter.

Domain	***Training***	***Probabilistic***
Continuous/Discrete	Unsupervised	No

6.1.1 Applications

There are many reasons why we might separate our data by similarity. For organizational purposes, it's convenient to have different classes of data. It can be easier for a human to sift through data if it's loosely categorized beforehand. It may be a preprocessing step for an inference method; for example, by creating additional features for a supervised technique. It can help identify which features make our data points most distinct from one another. It might even provide some idea of how many distinct data types we have in our set. Here are a few specific examples of use cases for clustering:

1. Determining the number of phenotypes in a population.
2. Organizing images into folders according to scene similarity.
3. Grouping financial data as a feature for anticipating extreme market events.
4. Identifying similar individuals based on DNA sequences.

As we mentioned above, there are different methods available for clustering. In this chapter, we will explore two of the most common techniques: K-Means Clustering and Hierarchical Agglomerative Clustering. We also touch on the flavors available within each of these larger techniques.

6.2 K-Means Clustering

The high level procedure behind K-Means Clustering (known informally as k-means) is as follows:

1. Initialize cluster centers by randomly selecting points in our data set.
2. Using a distance metric of your choosing, assign each data point to the closest cluster.
3. Update the cluster centers by averaging the data points assigned to each cluster.
4. Repeat steps 1 and 2 until convergence.

In the case where we are using the L2 distance metric, this is known as *Lloyd's algorithm*, which we derive in the next section.

6.2.1 Lloyd's Algorithm

Lloyd's algorithm, named after Stuart P. Lloyd who first suggested the algorithm in 1957, optimizes our cluster assignments via a technique known as coordinate descent, which we will learn more about in later chapters.

Derivation 6.2.1 (Lloyd's Algorithm Derivation): We begin by defining a loss function for our current assignment of data points to clusters:

$$\mathcal{L}(\mathbf{X}, \{\boldsymbol{\mu}\}_{c=1}^{C}, \{\mathbf{r}\}_{n=1}^{N}) = \sum_{n=1}^{N}\sum_{c=1}^{C} r_{nc}(\mathbf{x}_n - \boldsymbol{\mu}_c)^2 \tag{6.2}$$

where $\mathbf{X}$ is our NxD data set (N is the number of data points and D is the dimensionality of our data), $\{\boldsymbol{\mu}\}_{c=1}^{C}$ is the CxD matrix of cluster centers (C is the number of clusters we chose),

and $\{\mathbf{r}\}_{n=1}^{N}$ is our NxC matrix of *responsibility vectors*. These are one-hot encoded vectors (one per data point), where the 1 is in the position of the cluster to which we assigned the nth data point.

Next, we adjust our responsibility vectors to minimize each data point's distance from its cluster center. Formally:

$$r_{nc} = \begin{cases} = 1 & \text{if } c = \arg\min_{c'} ||\mathbf{x}_n - \boldsymbol{\mu}_{c'}|| \\ = 0 & \text{otherwise} \end{cases} \tag{6.3}$$

After updating our responsibility vectors, we now wish to minimize our loss by updating our cluster centers $\boldsymbol{\mu}_c$. The cluster centers which minimize our loss can be computed by taking the derivative of our loss with respect to $\boldsymbol{\mu}_c$, setting equal to 0, and solving for our new cluster centers $\boldsymbol{\mu}_c$:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}_c} &= -2 \sum_{n=1}^{N} r_{nc}(\mathbf{x}_n - \boldsymbol{\mu}_c) \\ \boldsymbol{\mu}_c &= \frac{\sum_{n=1}^{N} r_{nc}\mathbf{x}_n}{\sum_{n=1}^{N} r_{nc}} \end{aligned} \tag{6.4}$$

Intuitively, this is the average of all the data points $\mathbf{x}_n$ assigned to the cluster center $\boldsymbol{\mu}_c$.

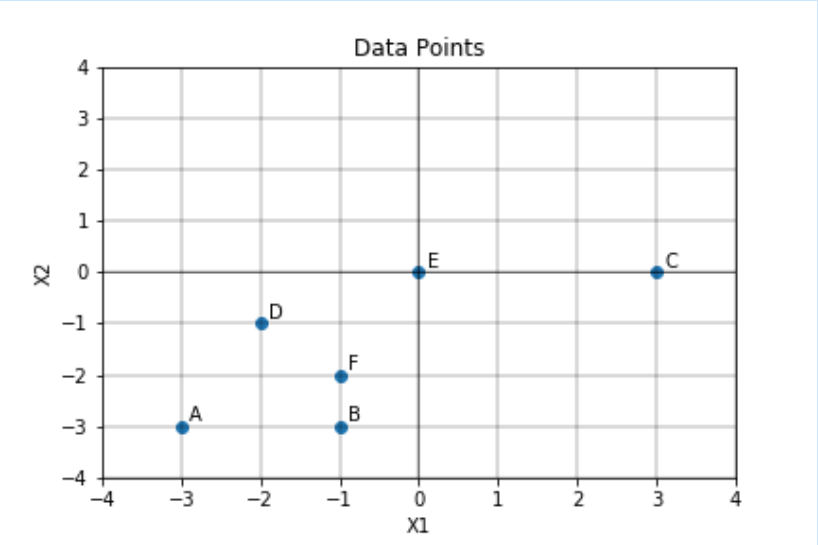
We then update our responsibility vectors based on the new cluster centers, update the cluster centers again, and continue this cycle until we have converged on a stable set of cluster centers and responsibility vectors.

Note that while Lloyd's algorithm is guaranteed to converge, it is only guaranteed to converge to a locally optimal solution. Finding the globally optimal set of assignments and cluster centers is an NP-hard problem. As a result, a common strategy is to execute Lloyd's algorithm several times with different random initializations of cluster centers, selecting the assignment that minimizes loss across the different trials. Furthermore, to avoid nonsensical solutions due to scale mismatch between features, it makes sense to standardize our data in a preprocessing step. This is as easy as subtracting the mean and dividing by the standard deviation across each feature.

6.2.2 Example of Lloyd's

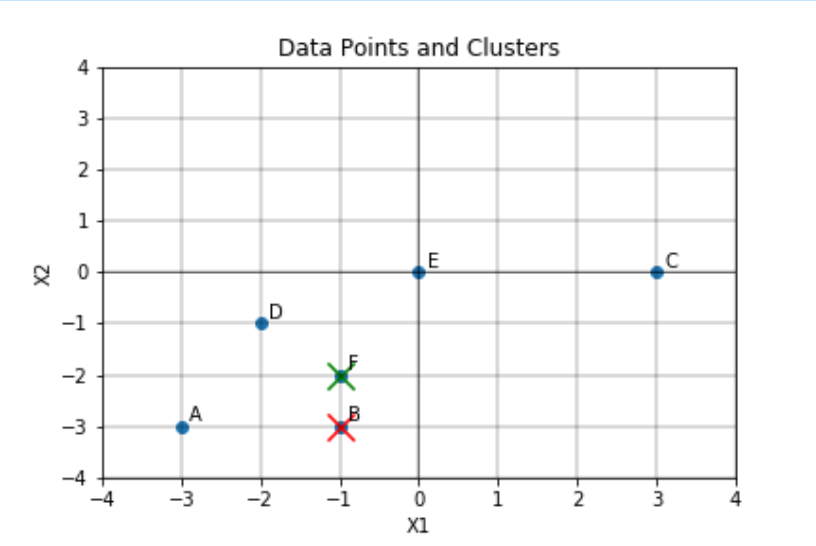
For some more clarity on exactly how Lloyd's algorithm works, let's walk through an example.

Example 6.1 (Lloyd's Algorithm Example): We start with a data set of size $N = 6$. Each data point is two-dimensional, with each feature taking on a value between -3 and 3. We also have a 'Red' and 'Green' cluster. Here is a table and graph of our data points, labelled A through F:

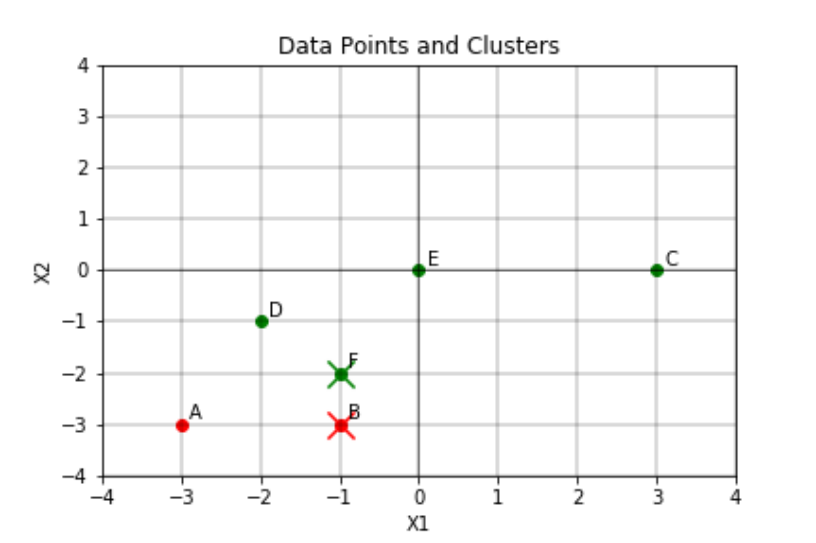


	Coordinates	Dist. to Red	Dist. to Green	Cluster Assgn.
A	(-3, -3)	n/a	n/a	n/a
B	(-1, -3)	n/a	n/a	n/a
C	(3, 0)	n/a	n/a	n/a
D	(-2, -1)	n/a	n/a	n/a
E	(0, 0)	n/a	n/a	n/a
F	(-1, -2)	n/a	n/a	n/a

Let's say we wish to have 2 cluster centers. We then randomly initialize those cluster centers by selecting two data points. Let's say we select B and F. We identify our cluster centers with a red and green 'X' respectively:

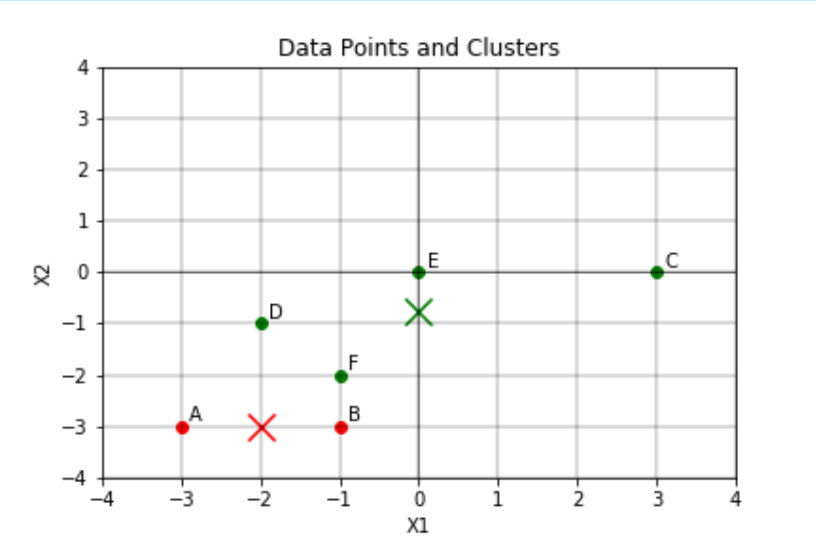


We now begin Lloyd's algorithm by assigning each data point to its closest cluster center:



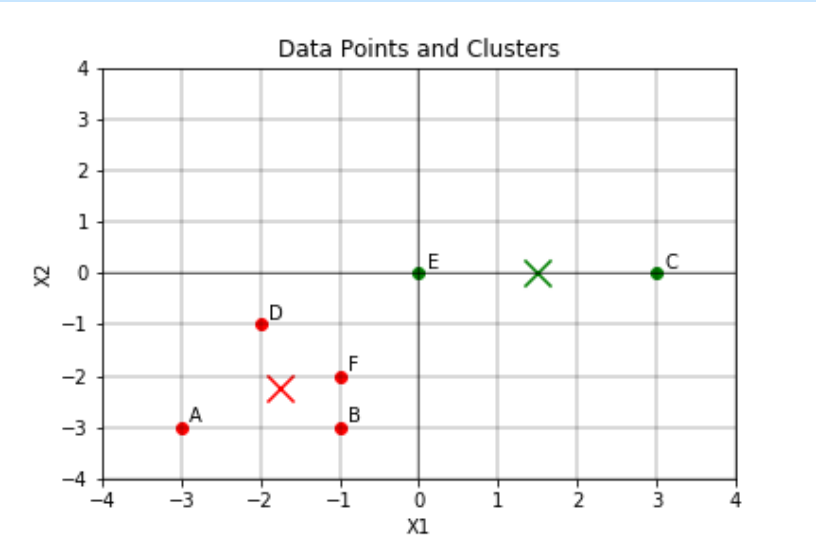
	Coordinates	Dist. to Red	Dist. to Green	Cluster Assgn.
A	(-3, -3)	n/a	n/a	Red
B	(-1, -3)	n/a	n/a	Red
C	(3, 0)	n/a	n/a	Green
D	(-2, -1)	n/a	n/a	Green
E	(0, 0)	n/a	n/a	Green
F	(-1, -2)	n/a	n/a	Green

We then update our cluster centers by averaging the data points assigned to each:



	Coordinates	Dist. to Red	Dist. to Green	Cluster Assgn.
A	(-3, -3)	2.00	2.24	Red
B	(-1, -3)	0.00	1.00	Red
C	(3, 0)	5.00	4.47	Green
D	(-2, -1)	2.24	1.41	Green
E	(0, 0)	3.16	2.24	Green
F	(-1, -2)	1.00	0.00	Green

We proceed like this, updating our cluster centers and assignments, until convergence. At convergence, we've achieved these cluster centers and assignments:



	Coordinates	Dist. to Red	Dist. to Green	Cluster Assgn.
A	(-3, -3)	1.46	5.41	Red
B	(-1, -3)	1.06	3.91	Red
C	(3, 0)	5.26	1.50	Green
D	(-2, -1)	1.27	3.64	Red
E	(0, 0)	2.85	1.50	Green
F	(-1, -2)	0.79	3.20	Red

Where our red cluster is at (-1.75, -2.25) and our green cluster is at (1.5 , 0). Note that for this random initialization of cluster centers, we deterministically identified the locally optimal set of assignments and cluster centers. For a specific initialization, running Lloyd's algorithm will always identify the same set of assignments and cluster centers. However, different initializations will produce different results

6.2.3 Number of Clusters

You may have wondered about a crucial, omitted detail: how do we choose the proper number of clusters for our data set? There doesn't actually exist a 'correct' number of clusters. The fewer clusters we have, the larger our loss will be, and as we add more clusters, our loss will get strictly smaller. That being said, there is certainly a tradeoff to be made here.

Having a single cluster is obviously useless - we will group our entire data set into the same cluster. Having N clusters is equally useless - each data point gets its own cluster.

One popular approach to identifying a good number of clusters is to perform K-Means with a varying number of clusters, and then to plot the number of clusters against the loss. Typically, that graph will look like Figure 6.1.

Notice that at $x = ...$ clusters, there appears to be a slight bend in the decrease of our loss. This is often called the **knee**, and it is common to choose the number of clusters to be where the knee occurs.

Intuitively, the idea here is that up to a certain point, adding another cluster significantly decreases the loss by more properly grouping our data points. However, eventually the benefit of adding another cluster stops being quite so significant. At this point, we have identified a natural number of groups for our data set.

6.2.4 Initialization and K-Means++

Up until now, we have assumed that we should randomly initialize our cluster centers and execute Lloyd's algorithm until convergence. We also suggested that since Lloyd's algorithm only produces a local minimum, it makes sense to perform several random initializations before settling on the most optimal assignment we've identified.

While this is a viable way to perform K-Means, there are other ways of initializing our original cluster centers that can help us find more optimal results without needing so many random initializations. One of those techniques is known as **K-Means++**.

The idea behind K-Means++ is that our cluster centers will typically be spread out when we've reached convergence. As a result, it might not make sense to initialize those cluster centers in an entirely random manner. For example, Figure 6.2 would be a poor initialization.

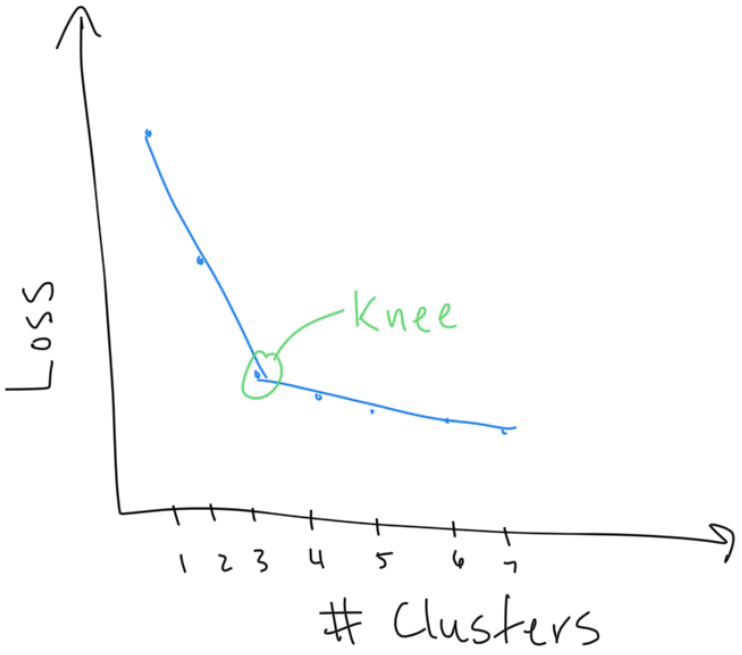


Figure 6.1: Finding the Knee.

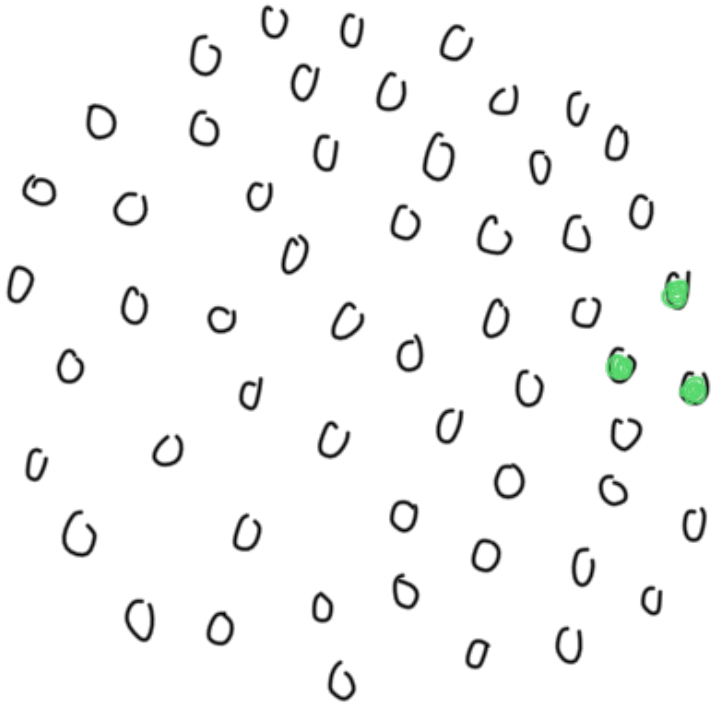


Figure 6.2: Bad Cluster Initialization.

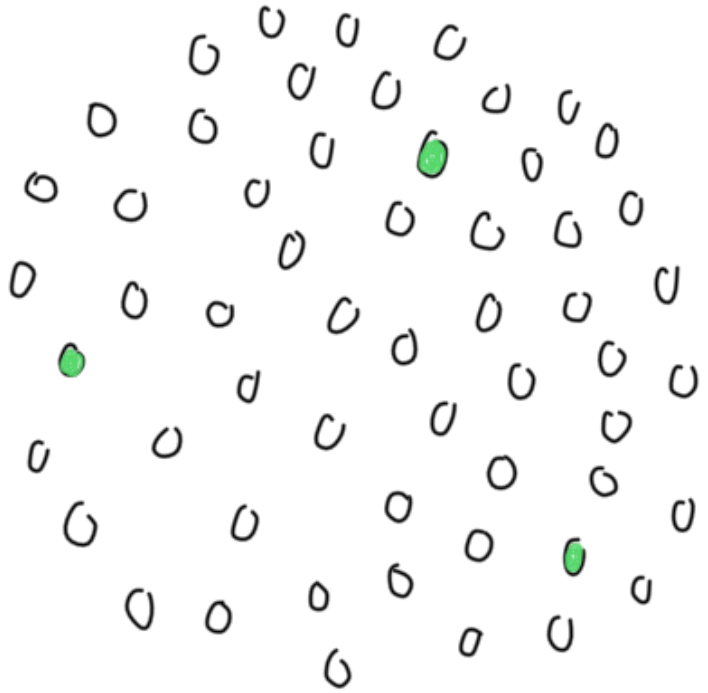


Figure 6.3: Good Cluster Initialization.

We would much rather start with a random initialization that looks like Figure 6.3.

We can use the hint that we want our cluster centers somewhat spread out to find a better random initialization. This is where the initialization algorithm presented by K-Means++ comes in.

For K-Means++, we choose the first cluster center by randomly selecting a point in our data set, same as before. However, for all subsequent cluster center initializations, we select points in our data set with probability proportional to the squared distance from their nearest cluster center. The effect of this is that we end up with a set of initializations that are relatively far from one another, as in Figure 6.3.

6.2.5 K-Medoids Alternative

Recall that in the cluster center update step (Equation 6.4) presented in the derivation of Lloyd's algorithm, we average the data points assigned to each cluster to compute the new cluster centers. Note that in some cases, this averaging step doesn't actually make sense (for example, if we have categorical variables as part of our feature set). In these cases, we can use an alternative algorithm known as **K-Medoids**. The idea behind K-Medoids is simple: instead of averaging the data points assigned to that cluster, update the new cluster center to be the data point assigned to that cluster which is most like the others.

6.3 Hierarchical Agglomerative Clustering

The motivating idea behind K-Means was that we could use a distance measurement to assign data points to a fixed number of clusters, iteratively improving our assignments and cluster locations

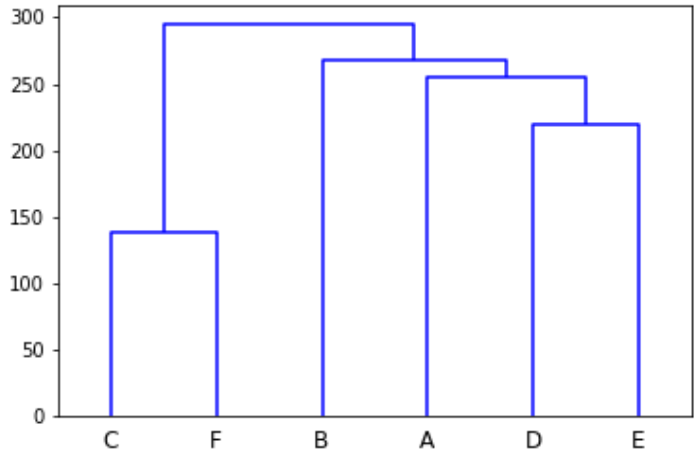


Figure 6.4: Dendrogram Example.

until convergence.

Moving on to Hierarchical Agglomerative Clustering (also known as **HAC** - pronounced 'hack'), the motivating idea is to construct a tree over our data set that describes relationships between our data. These trees are known as *dendrograms*, with an example found in Figure 6.4.

Notice that the individual data points are the leaves of our tree, and the trunk is the cluster that contains the entirety of our data set. At a high level, we construct this tree as follows:

1. Start with N clusters, one for each data point.
2. Merge the two 'closest' clusters together, reducing the number of clusters by 1. Record the distance between these two merged clusters.
3. Repeat step 2 until we're left with only a single cluster.

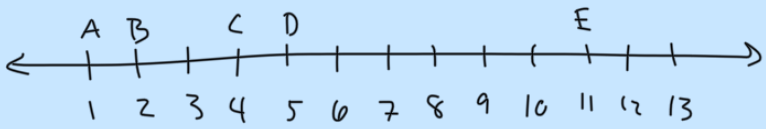
In the remainder of the chapter, we'll describe this procedure in greater detail (including what is meant by 'closest' clusters), establish the clustering information produced by the tree, and discuss how HAC differs from K-Means.

6.3.1 HAC Algorithm

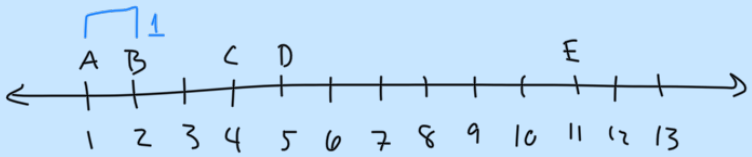
As described above, HAC proceeds by iteratively merging clusters until we're only left with a single cluster. At the start, each data point is in it's own cluster. We record the distance between the two clusters that we merge at each step, which is what allows us to construct the dendrogram in Figure 6.4. To make this process a little more clear, let's perform HAC one step at a time, constructing the dendrogram as we go.

TODO: HAC example here with four data points, probably just use the min linkage criterion.

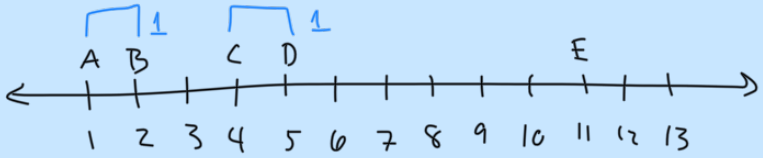
Example 6.2 (HAC Algorithm Example): Let's say we have a data set of five points A, B, C, D, E that we wish to perform HAC on. These points will simply be scalar data that we can represent on a number line. We start with 5 clusters and no connections at all:



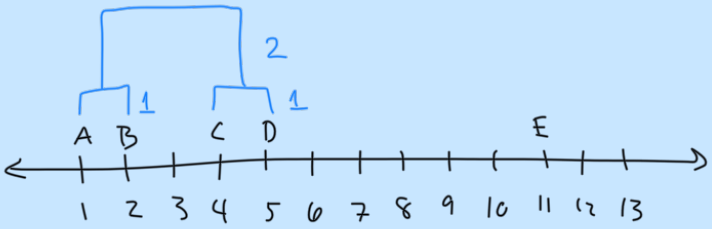
We find the closest two clusters to merge first. A and B are nearest (it's actually tied with C and D, but we can arbitrarily break these ties), so we start by merging them. Notice that we also annotate the distance between them in the tree, which in this case is 1:



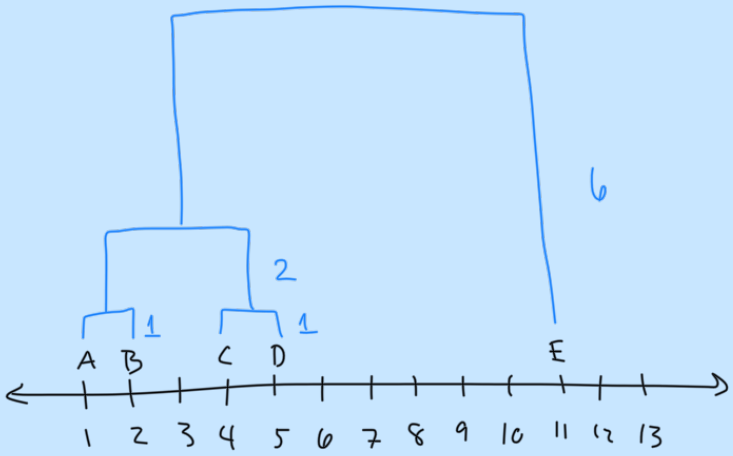
We now have four clusters: (A, B), C, D, and E. We again find the closest two clusters, which in this case is C and D:



We now have three remaining clusters: (A, B), (C, D), and E. We proceed as before, identifying the two closest clusters to be (A, B) and (C, D). Merging them:



Finally we are left with two clusters: (A, B, C, D) and E. The remaining two clusters are obviously the closest together, so we merge them:



At this point there is only a single cluster. We have constructed our tree and are finished with HAC.

Notice how the distance between two merged clusters manifests itself through the height of the dendrogram where they merge (which is why we tracked those distances as we constructed the tree). Notice also that we now have many layers of clustering: if we're only only interested in clusters whose elements are at least k units away from each other, we can 'cut' the dendrogram at that height and examine all the clusters that exist below that cut point.

Finally, there's a detail we glossed over that is very important to the different clusterings we might find using HAC. In the preceding example, we designated the distance between two clusters to be the minimum distance between any two data points in the clusters. This is what is known as the **Min-Linkage Criterion**. However, there are certainly other ways we could have computed the distance between clusters. We now turn to these different methods and the properties of clusters they produce.

6.3.2 Linkage Criterion

Here are a few of the most common linkage criteria.

Min-Linkage Criteria

We've already seen the Min-Linkage Criterion in action from the previous example. Formally, the criterion says that the distance $d_{C,C'}$ between each cluster pair C and C' is given by

$$d_{C,C'} = \min_{k,k'} ||\mathbf{x}_k - \mathbf{x}_{k'}|| \tag{6.5}$$

where $\mathbf{x}_k$ are data points in cluster C and $\mathbf{x}_{k'}$ are data points in cluster C'. After computing these pairwise distances, we choose to merge the two clusters that are closest together.

Max-Linkage Criterion

We could also imagine defining the distance $d_{C,C'}$ between two clusters as being the distance between the two points that are farthest apart in each cluster. This is known as the Max-Linkage Criterion.

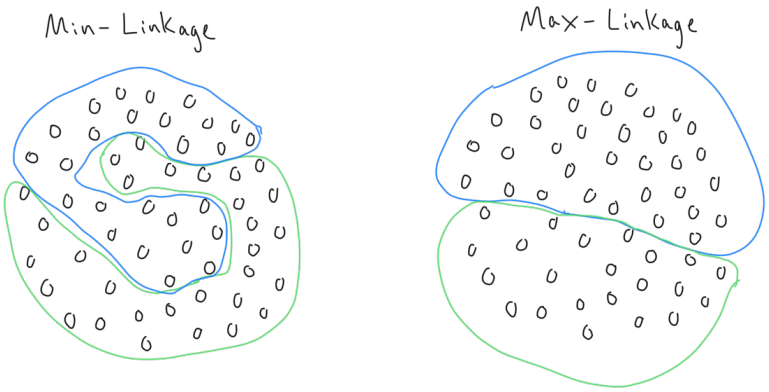


Figure 6.5: Different Linkage Criteria.

The distance between two clusters is then given by:

$$d_{C,C'} = \max_{k,k'} ||\mathbf{x}_k - \mathbf{x}_{k'}|| \tag{6.6}$$

As with the Min-Linkage Criterion, after computing these pairwise distances, we choose to merge the two clusters that are closest together.

⋆ Be careful not to confuse the linkage criterion with which clusters we choose to merge. We **always** merge the clusters that have the smallest distance between them. How we compute that distance is given by the linkage criterion.

Average-Linkage Criterion

The Average-Linkage Criterion averages the pairwise distance between each point in each cluster. Formally, this is given by:

$$d_{C,C'} = \frac{1}{KK'} \sum_{k=1}^{K} \sum_{k'=1}^{K'} ||\mathbf{x}_k - \mathbf{x}_{k'}|| \tag{6.7}$$

Centroid-Linkage Criterion

The Centroid-Linkage Criterion uses the distance between the centroid of each cluster (which is the average of the data points in a cluster). Formally, this is given by:

$$d_{C,C'} = ||\frac{1}{K} \sum_{k=1}^{K} \mathbf{x}_k - \frac{1}{K'} \sum_{k'=1}^{K'} \mathbf{x}_{k'}|| \tag{6.8}$$

Different Linkage Criteria Produce Different Clusterings

It's important to note that the linkage criterion you choose to use will influence your final clustering results. For example, the min-linkage criterion tends to produce 'stringy' clusters, while the max-linkage criterion tends to produce more compact clusters. You can see the difference between the results of these two linkage criteria in Figure 6.5.

It's useful to be aware of these different types of results when selecting your linkage criterion.

⋆ You should convince yourself of the different flavors of linkage criteria. For example, when using the min-linkage criterion, we get these 'stringy' results because we're most inclined to extend existing clusters by grabbing whichever data points are closest.

6.3.3 How HAC Differs from K-Means

Now that we're aware of two distinct clustering techniques and their variants, we consider the differences between the two methods.

First of all, there is a fundamental difference in determinism between HAC and K-Means. In general, K-Means incurs a certain amount of randomness and needs to be run multiple times to ensure a good result. On the other hand, once you've selected a linkage criterion for HAC, the clusters you calculate are deterministic. You only need to run HAC a single time.

Another difference between HAC and K-Means comes from the assumptions we make. For K-Means, we need to specify the number of clusters up front before running our algorithm, potentially using something like the knee-method to decide on the number of clusters. On the other hand, you don't need to assume anything to run HAC, which simplifies its usage. However, the downside for HAC is that when you wish to present your final clustering results, you need to decide on the max distance between elements in each cluster (so that you can cut the dendrogram).

The fact that you need to make a decision about where to cut the dendrogram means that running HAC once gives you several different clustering options. Furthermore, the dendrogram in and of itself can be a useful tool for visualizing data. We don't get the same interactivity from K-Means clustering.

⋆ We often use dendrograms to visualize evolutionary lineage.