Machine Learning - Michaelmas Term 2021
Lectures 19: Clustering

Lecturers: Phil Blunsom & Atılım Güneş Baydin

In this lecture, we will study several different algorithms for clustering—an unsupervised machine learning task in which the goal is to organise data into "clusters" in a manner that data in the same cluster is more similar to each other than data in different clusters. As is often the case in unsupervised learning tasks, there is a lot more subjectivity and we shall discuss some of these issues in the lecture. Let us keep in mind the task of clustering news articles as a running example; we can easily see that the problem of clustering is slightly ill-posed. One way to cluster the articles would be according to topics such as politics, sports, entertainment, and so on; yet another way, would be according to region, such as London, Wales, North England, Scotland, etc. In a way, both would be valid ways to form clusters and the ultimate use of the clustered data would determine which of these should be chosen. For the most part, we'll study mathematical and algorithmic aspects of clustering, assuming that the data has already been pre-processed for us in some way; however, it's worth keeping in mind that there is a fair degree of ambiguity in defining clusters.

The input data to clustering algorithms can be in one of two forms. First, the data may be given to us as N points in Euclidean space. Second, we may be given an $N \times N$ (symmetric) matrix where the $(i, j)th$ entry is a measure of similarity (or dis-similarity) between the i^{th} and j^{th} datapoint. Different algorithms work with different input formats and so it is useful to study how an input in one format can be transformed to the other.

Clustering procedures can be classified into two types based on the output produced by these algorithms. The first type are the so-called *flat* clustering procedures which output a partition of the dataset, each part representing a cluster. The second type are the *hierarchical clustering* procedures which output a tree with the original data as its leaves and each internal node representing clusters at different granularities.

1 Partition-based Clustering

We will first consider the case when the input data is N points in the D-dimensional Euclidean space. Thus, the input is $\langle \mathbf{x}_i \rangle_{i=1}^{N}$; furthermore, we will also assume that we are given the desired number of clusters k as a parameter. The goal is to output a partition $C_1, \dots, C_k$ of the set $\{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, where the C_i are non-empty, pairwise disjoint and their union is the entire dataset. We will begin by studying one of the most popular formulations of partition-based clustering—the k-means procedure—and then discuss some other variants.

1.1 The k-Means Formulation

In the k-means formulation of clustering, we assign k "means" $\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_k$, which are points in Euclidean space, each $\boldsymbol{\mu}_i$ representing one cluster. Each datapoint is then assigned to the closest mean which results in a partion of the data (ties may be broken arbitrarily). In the k-means formulation the goal is to minimize the following objective over all possible partitions $C_1, \dots, C_k$ and means $\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_k$:

$$W(C_1, \dots, C_k; \boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_k) = \sum_{j=1}^{k} \sum_{i \in C_j} \left\| \mathbf{x}_i - \boldsymbol{\mu}_j \right\|_2^2 \tag{1}$$

Algorithm 1 Lloyd's Algorithm for k-Means

```
 1: procedure $k$-MEANS($\langle \mathbf{x}_i \rangle_{i=1}^{N}$, $k$)
 2:     Initialise $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ suitably
 3:     repeat
 4:         for $j = 1, \ldots, k$ do
 5:             $C_j \leftarrow \{i \mid j = \arg\min_{j'} \left\| \mathbf{x}_i - \boldsymbol{\mu}'_j \right\|_2^2\}$        ▷ Assign points to closest means
 6:         end for
 7:         for $j = 1, \ldots, k$ do
 8:             $\boldsymbol{\mu}_j \leftarrow \frac{1}{|C_j|} \sum_{i \in C_j} \mathbf{x}_i$        ▷ Update means
 9:         end for
10:     until convergence
11: end procedure
```

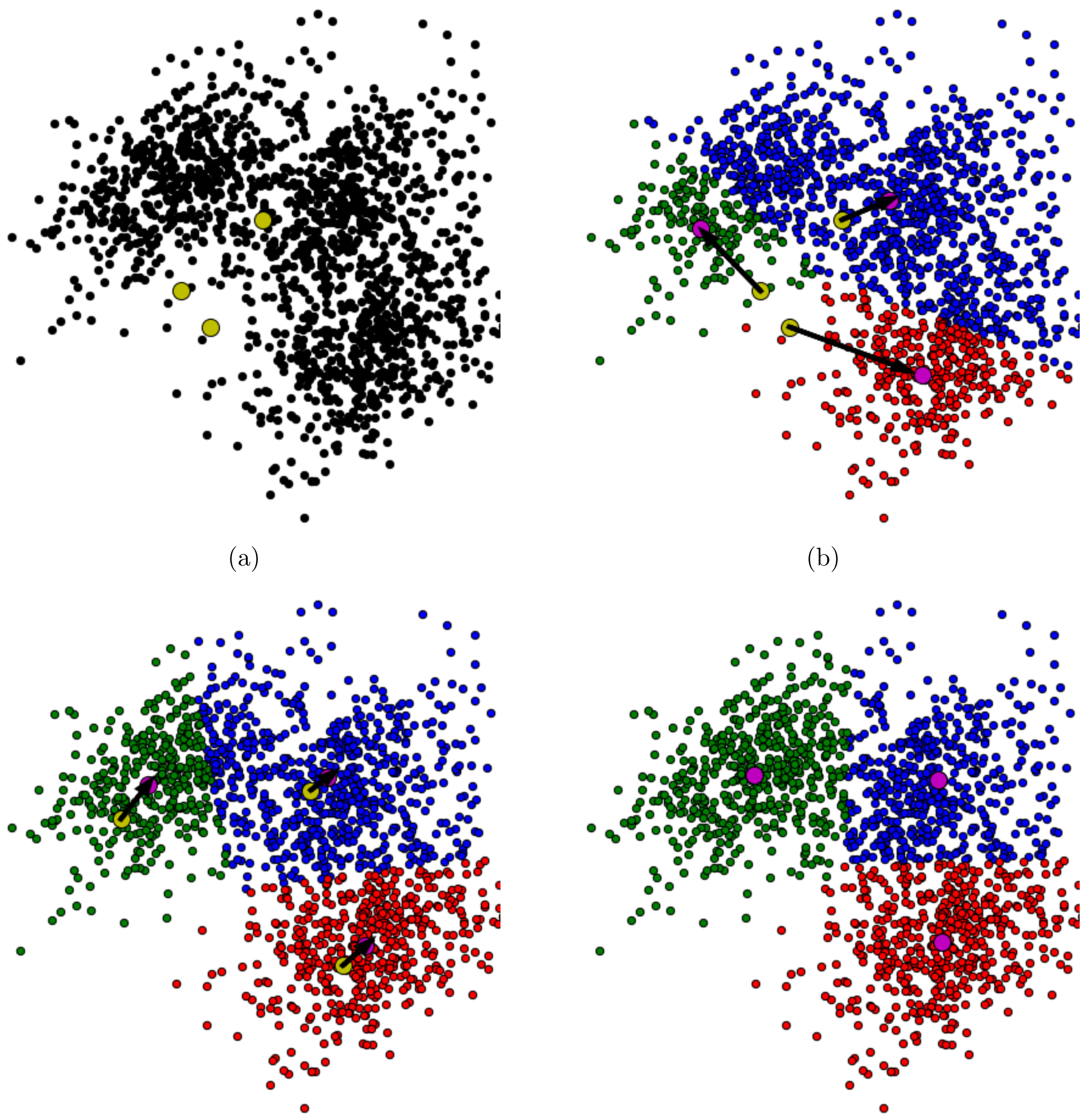
Before we discuss algorithms to optimise this objective function, let us first understand some properties of this objective. Although the objective does not explicitly require assigning each point to the nearest mean, it is evident that once the means are fixed doing so will result in the least possible value of the objective function. Clearly, putting similar points in the same cluster and picking a mean $\boldsymbol{\mu}$ close to these points is a good idea. On the other hand, there is nothing explicit in the objective that requires points in different clusters to be unlike each other. However, if k is chosen suitably (a point we will discuss later), then the desire to map all similar points to the same cluster will automatically result in assigning points that are dissimilar to different clusters.

Optimising the k-Means Objective

Unfortunately, it turns out that optimising the k-means objective function is NP-hard even when $k = 2$. Thus, in practice we have to settle for heuristics or approximation algorithms that aim to minimize the objective (1). We will study one of the popular heuristic algorithms, proposed by Lloyd (1982), which iteratively and alternately updates the partition and the means. The pseudocode is shown as Algorithm 1.

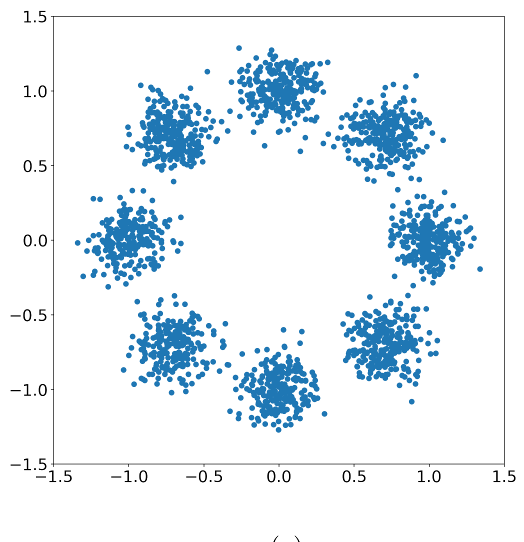
Let us focus first on the main loop of the algorithm between lines 3-10. There are two key steps inside this loop. The first assigns the datapoints to clusters. Notice that while simultaneously minimizing the objective function (1) over $C_1, \ldots, C_k, \boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ is hard, if the $\boldsymbol{\mu}_j$ are fixed, creating clusters is trivial—we merely need to assign each point to the closest mean (ties may be broken arbitrarily). Equally, once the partition given by $C_1, \ldots, C_j$ is fixed, finding the means $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ that minimise the cost is easy, we can simply take the average of all points assigned to the given part (cluster). Thus, the algorithm adopts an *alternating minimization* approach, where in each step, part of the parameters are fixed and the optimization is performed over the remaining parameters. It is worth pointing out that we don't expect this method to find the global optimum of the objective function.[1] We can however show that the method converges. To see this notice that any change to the clustering in steps 3-10 strictly decreases the value of the objective function. As there are only finitely many partitions, the loop cannot cycle forever. Unfortunately, there is no guarantee that the number of iterations is small (*e.g.,* polynomially bounded); however, in practice the method is observed to converge quickly. Figure 1 shows intermediate states at some iterations of Algorithm 1.

[1]There are more recent algorithms which guarantee fast convergence and even give some bound on the quality of the partition achieved, see *e.g.,* the k-means++ algorithm of Arthur and Vassilvitskii (2007).



(a) (b) (c) (d)

Figure 1: (a) Initial state with randomly chosen means (b) & (c) Intermediate states showing both cluster assignments and mean updates (d) Converged clusters



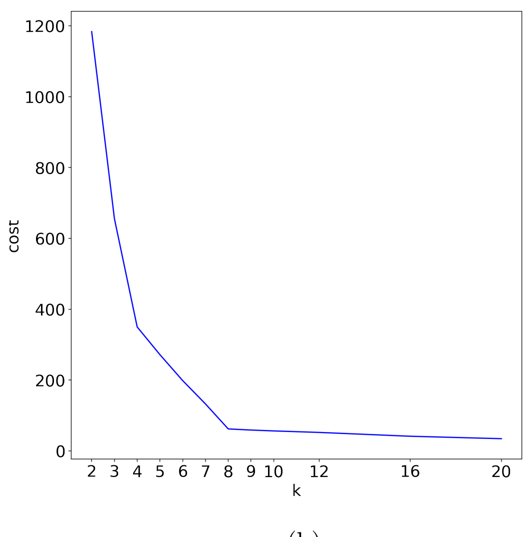


Figure 2: (a) Synthetic data with 8 clusters. (b) Plot of cost vs k for data shown in (a) . In this case, we would correctly pick 8 as the value of k as that is where the *elbow* appears.

Initialisation

Let us now focus on the initialization phase in Step 2. One reasonable way to initialize the *means* is to pick k out of the N datapoints uniformly at random. For fast convergence of the algorithm and to obtain meaningful clusters, it is important that the means are chosen to be representative of the data initially. The (random) initial choice of means does influence the final clusters significantly. In practice, it is common to run the algorithm several times and pick the partition that has the smallest value for the objective function (1).

Choosing k

Thus far, we've ignored the issue of how k, the number of desired clusters, should be chosen. Looking at the objective function, we can see that increasing the value of k will always decrease the objective function (at least assuming we could optimise it perfectly). One way to see this is to notice that splitting a cluster into two clusters, each with their own mean, can only decrease the cost. Thus, we cannot simply try to pick the value of k that results in the least value of the objective function, or else, we'll end up with trivial clusters each containing one datapoint.

Starting with 2 clusters, we would expect the objective function to drop rapidly until we reach the correct value of k and then after that the drop should be much slower. Thus, if we plot the value of the objective function as a function of k, the curve we get has a sharp bend or an *elbow*. We should pick the value of k where this happens. Such a curve and the synthetic data used to generate the curve is shown in Figure 2. (Note that there is no guarantee that the value in the plot is the least possible value for the objective function as Algorithm 1 is only a heuristic.)

1.2 Other Objective Functions

There is nothing special about using k-means when considering clustering. In the k-means formulation, the measure of dissimilarity used is the squared Euclidean distance between each datapoint and its associated mean. We can instead use the Euclidean distance (not squared) which will result in the k-median algorithm, or any other ℓ_p-norm. Furthermore, the objective function (1) considers the sum of all the *dissimilarities* in each cluster—we may instead use the maximum over all dissimilarities between data and the associated *anchor* (what we called mean earlier). In this case, the formulation is known as k-center and the *anchors* are called *centres.*

2 Transforming Input Formats

Algorithm 1 works when using data in Euclidean space. We will study other algorithms in Section 3 which work when the data is given as a matrix of (dis)similarities. As far as possible, one would like to have a wide range of algorithms to choose from. For this reason, we would like to be able to convert Euclidean data into pairwise (dis)similarities and to find Euclidean embeddings given pairwise (dis)similarities.

2.1 Dissimilarities from Euclidean Embeddings

Let us suppose each datapoint has a certain number of features denoted by $x_1, \ldots, x_D$. For each individual feature, for any possible values that can be achieved, we can define a *distance function*, $d_j(x_j, x'_j)$. We will also assign a weight w_j for feature j and consider f to be any non-decreasing function. Then given datapoints $\mathbf{x}$ and $\mathbf{x}'$, consider the following notion of dissimilarity:

$$d(\mathbf{x}, \mathbf{x}') = f\left(\sum_{j=1}^{D} w_j d_j(x_j, x'_j)\right) \tag{2}$$

Observe that setting $d_j(x_j, x'_j) = (x_j - x'_j)^2$, $w_j = 1$ and $f(z) = \sqrt{z}$, we get the familiar Euclidean distance. However, other choices could be made. It is worth pointing out two advantages of this approach: (i) We can assign different weights to different features and so in principle can decide where to place emphasis (*e.g.,* clustering by region vs clustering by topic) (ii) Actually, we do not require the original data to be in Euclidean space, even if some feature, say x_j, is categorical, provided we can define a suitable d_j, we can still use it to define dissimilarity. (The most natural choice would be $d_j(x_j, x'_j) = 0$ if $x_j = x'_j$ and 1 otherwise; this is equivalent to performing one-hot encoding and considering Euclidean distance up to a constant scaling factor.)

2.2 Multidimensional Scaling

Let us now discuss the harder case of going from a set of pairwise dissimilarities to a Euclidean embedding. Let us first assume that we have data $\mathbf{x}_1, \ldots, \mathbf{x}_N$ in Euclidean space and consider a dissimilarity matrix D, where $D_{ij} = \left\|\mathbf{x}_i - \mathbf{x}_j\right\|_2^2$, the squared Euclidean distance. We can pose the question whether it is possible to recover $\mathbf{x}_1, \ldots, \mathbf{x}_N$ given D. Clearly, there is no hope of exact recovery as any distance-preserving transformation, such as rotation, translation, etc. applied to the original data will result in the same D. Thus, we can ask whether it is possibly to find N points in some Euclidean space which will result in the same matrix D. The answer to this question is yes. It will be easier to start with a matrix M, where $M_{ij} = \mathbf{x}_i \cdot \mathbf{x}_j$. Notice that these two matrices are related as $D_{ij} = M_{ii} + M_{jj} - 2M_{ij}$. However, we can only recover M from D up to a translation. If we assume that the original data was centered, *i.e.,* $\sum_i \mathbf{x}_i = 0$, then M can be uniquely recovered from D. (As an exercise, the reader should attempt to prove this.)

To obtain some N points, say $\widetilde{\mathbf{x}}_1, \ldots, \widetilde{\mathbf{x}}_N$ from M, we consider the singular value decomposition (SVD) of M. Let $M = U\Sigma U^\mathsf{T}$ be the full SVD of M, so that $U^\mathsf{T}U = I$, and Σ is diagonal with non-negative entries. As M is square, symmetric and positive-semidefinite, the left and right singular vectors are the same and hence the use of U^T instead of V^T; both U and Σ are

$N \times N$. Since Σ has non-negative entries on the diagonal, it is possible to take its square root; let $\widetilde{\mathbf{X}} = U\Sigma^{1/2}$ (*i.e.,* we let $\widetilde{\mathbf{x}}_i$ be the i^{th} row of $U\Sigma^{1/2}$), then we see that,

$$\widetilde{\mathbf{X}}\widetilde{\mathbf{X}}^{\mathsf{T}} = U\Sigma^{1/2}\Sigma^{1/2}U^{\mathsf{T}} = M.$$

In many applications, it is easier to define a notion of similarity or dissimilarity rather than embed data in some Euclidean space. For example, when using DNA sequences we can use the Hamming distance as a measure of dissimilarity, or when using text data we may use the *cosine* kernel as a measure of similarity. Using a Mercer kernel to define similarity ensures that the resulting matrix M is positive semi-definite and hence the above derivation is valid.

In case the matrix M is not positive semi-definite, we can directly use D to solve the following optimization problem:

$$\underset{\widetilde{\mathbf{x}}_1,\ldots,\widetilde{\mathbf{x}}_N}{\operatorname{argmin}} \sum_{i \neq j} \left(\left\| \widetilde{\mathbf{x}}_i - \widetilde{\mathbf{x}}_j \right\|_2^2 - D_{ij} \right)^2 \tag{3}$$

The objective function defined in (3) is called the *stress function* as it represents the degree to which it is not possible to find a Euclidean embedding for the given dissimilarity matrix. Unfortunately, there is neither a closed form solution for the optimization problem (3) nor is it convex. In practice, one can use gradient descent to get a local optimum.

3 Hierarchical Clustering

Let us first consider a simple toy example with four news articles: one about physics, one about mathematics, one about cricket and one about football. Suppose we are given the task of partitioning this data into 3 clusters. Depending on who is asked to perform the task, we'll get clusters symbolizing ("maths", "physics", "sports"), or ("science", "cricket", "football"). We could argue that the correct number of clusters in this case is 4 which to some extent resolves this problem of ambiguity. However, even in that case, the information that "physics" and "maths" are much closer to each other than "physics" is to "cricket" is lost! The resolution to this is to use *hierarchical clustering* rather than flat clustering. The output of a hierarchical clustering algorithm is a tree (called *dendrogram*) whose leaves contain the data and each internal node represents a cluster. This is shown in Figure 3.

Algorithms for hierarchical clustering come in two varieties—divisive and agglomerative. We will mainly focus on the latter. Divisive algorithms typically recursively split the data into 2 clusters, for example using k-means with $k = 2$. Thus, divisive algorithms build the dendrograms top-down. Agglomerative algorithms build the dendrogram bottom-up, starting with N clusters each containing a single datapoint and then forming coarser clusters (higher up in the tree) by merging existing clusters. The most well-known agglomerative algorithms are the so-called *linkage algorithms* which we study in some detail.

3.1 Linkage Algorithms

Linkage algorithms take as input a dissimilarity matrix D of size $N \times N$ representing pairwise dissimilarities for all of the data. The algorithms begin by forming N clusters each containing a single datapoint. The algorithms then repeatedly merge two of the existing clusters to form a new one; thus after $N - 1$ iterations all the data is in one cluster and the dendrogram is completed.

At the bottom, when each cluster has a single datapoint, it is obvious which clusters should be merged—we simply choose the pair that is the *closest* (least dissimilarity). However, in order to continue the process, we need to define dissimilarity at cluster levels, not just at the level of individual datapoints. The linkage algorithms differ based on how this is done. We will study three different ways, called *single*, *average* and *complete* linkage.

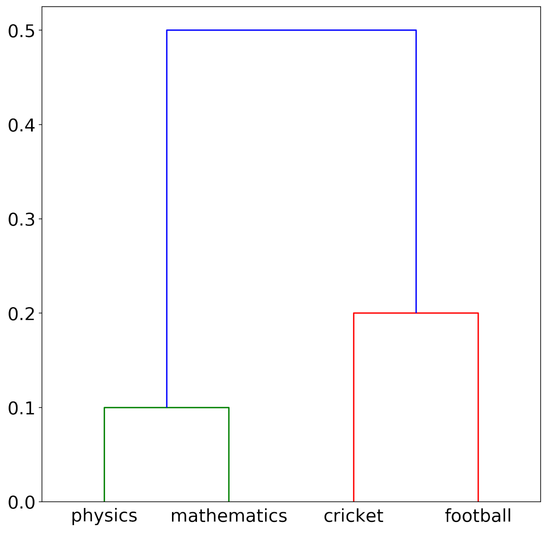


Figure 3: A dendrogram representing hierarchical clustering of news articles on cricket, football, physics and mathematics.

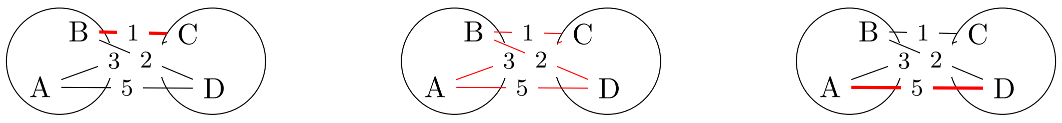
Algorithm 2 Linkage Algorithms

```
 1: procedure LINKAGE($D_{ij}, 1 \le i, j \le N$)
 2:     Initialise clusters as singletons, $C_i = \{i\}$
 3:     Initialise clusters available for merging $S = \{1, 2, \ldots, N\}$
 4:     repeat
 5:         $(j, k) = \text{argmin}_{j,k \in S}\, d(C_j, C_k)$        ▷ Pick 2 most similar clusters
 6:         $C_l = C_j \cup C_k$
 7:         $S \leftarrow (S \setminus \{j, k\}) \cup \{l\}$
 8:         Update $d(C_i, C_l)$ for each $i \in S$        ▷ Using suitable linkage rule
 9:     until $|S| = 2$
10: end procedure
```

Let C and C' be two clusters, then we can define dissimilarity $d(C, C')$ in terms of pairwise dissimilarities.

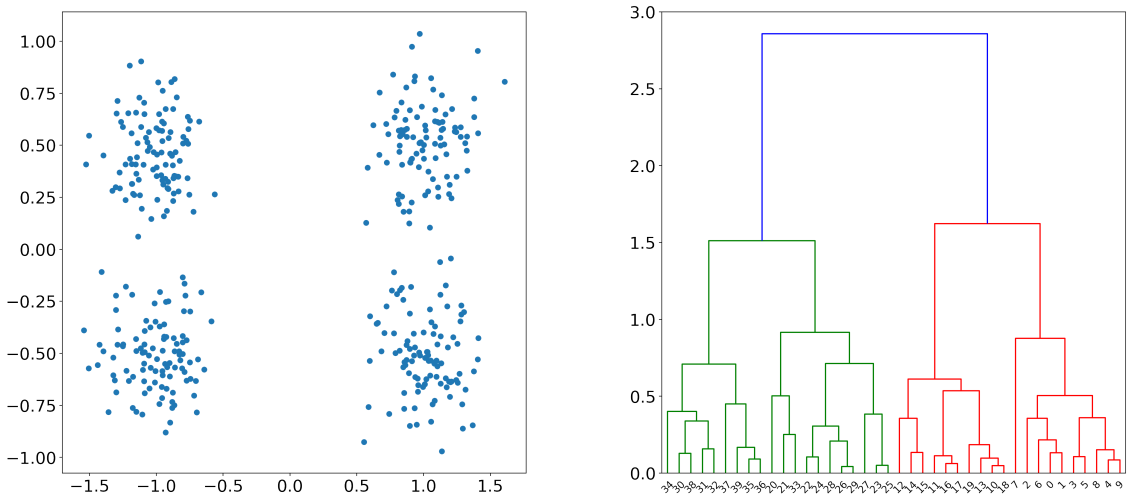
$$\begin{aligned}
&\text{Single Linkage} && d(C, C') = \min_{\mathbf{x} \in C, \mathbf{x}' \in C'} d(\mathbf{x}, \mathbf{x}') \\
&\text{Average Linkage} && d(C, C') = \frac{1}{|C| \cdot |C'|} \sum_{\mathbf{x} \in C, \mathbf{x}' \in C'} d(\mathbf{x}, \mathbf{x}') \\
&\text{Complete Linkage} && d(C, C') = \max_{\mathbf{x} \in C, \mathbf{x}' \in C'} d(\mathbf{x}, \mathbf{x}')
\end{aligned}$$

In words, single linkage defines the dissimilarity in terms of the least dissimilar elements in the two clusters, while complete linkage does so in terms of the most dissimilar elements. Average linkage simply uses the average and usually works quite well in practice. The entire pseudocode for linkage algorithms is shown as Algorithm 2 and the linkage rules are shown pictorially in Figure 4. Some data and the output of the average linkage on a subset of the data are shown in Figure 5.



(a) Single Linkage (b) Average Linkage (c) Complete Linkage

Figure 4: Linkage rules for hierarchical clustering.



(a) Data (b) Dendrogram

Figure 5: (a) Raw data and (b) Dendrogram obtained by applying the average linkage algorithm to a subset of the data.

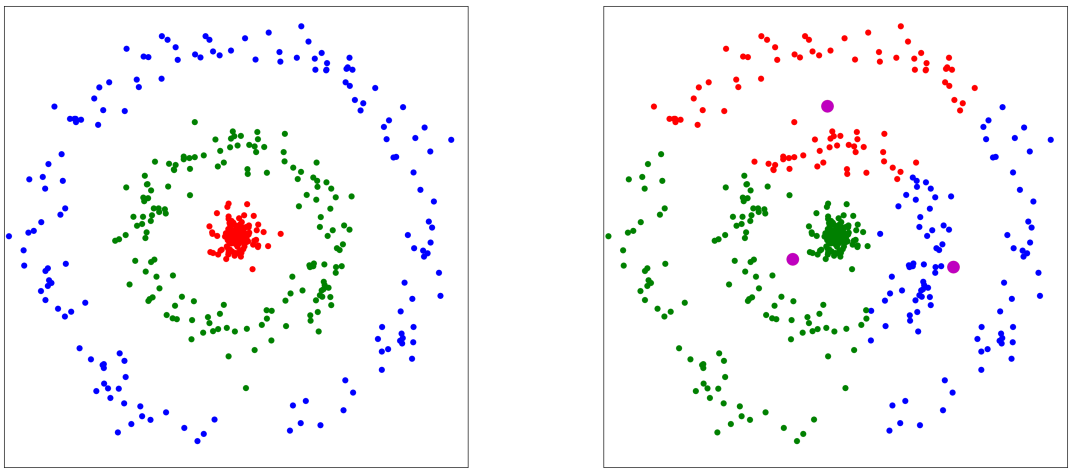
4 Spectral Clustering

We'll next consider the case when we have data in some Euclidean space, however, the clusters that we desire are non-convex. As an example, we can consider the data shown in Figure 6(a) where the desired clustering is the three different concentric circles. (Obviously, if we knew the data looked like this, we would transform it to polar co-ordinates and cluster; spectral clustering is a way to find a change of co-ordinates implicitly.) If we use the k-means formulation, the clusters we get always have convex shapes, as points are assigned to the nearest mean. On this data we get the clusters as shown in Figure 6(b). One way to get better clusters on this sort of data would be to peform non-linear dimensionality reduction, *e.g.,* using kernel PCA, followed by a simple clustering algorithm. Spectral clustering is a related, but different approach. In spectral clustering, we transform the data in Euclidean space to a weighted undirected graph and then apply graph partitioning algorithms to obtain clusters.

The first step in spectral clustering is to create a graph using the given data. We create one node for each datapoint. For some parameter k, we will connect each node to its k nearest neighbours. (Notice that since we are creating an undirected graph, this may result in some nodes having degree larger than k.) We assign a weight to each added edge. In general, we'd like the edge weight to represent "similarity", so we use a decreasing function of the Euclidean distance between the two points. A commonly used way to assign edge weights is to set,

$$w_{ij} = \exp(-\left\|\mathbf{x}_i - \mathbf{x}_j\right\|_2^2 / \sigma),$$

where σ is a width parameter. In principle, different ways of assigning weights to edges may be used.



(a) Data with three clusters as concentric circles (b) Output of k-means algorithm

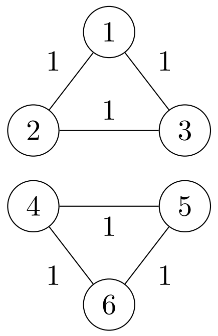
Figure 6

4.1 Spectral Graph Partitioning

Now that we have seen how to construct a graph using points in Euclidean space, our focus will be to find clusters in a graph. Algorithms for graph partitioning have been studied for a long time and we would like to exploit these techniques for the purpose of clustering. An informal definition of a cluster in a graph is a subset of nodes such that most high weight edges incident on nodes in this set remain inside the set and few high weight edges connect to nodes outside the cluster. Notice that, if the original data were well clustered in Euclidean space, this would result in good clusters in the graph, as for each datapoint most of its k nearest neighbours would be in the same cluster and furthermore they would have a higher weight (shorter distance) to each other than to points not in the same cluster.

Perhaps one of the most famous formulations for graph partitioning is min-cut, where the goal is to partition the nodes of the graph into two parts such that the cut, *i.e.,* the sum of the weights of the edges going from one part to the other, is minimised. While this sounds like a good idea for clustering, often, for graphs arising in practice, the mincut is achieved by putting one node in one part and the rest of the nodes in the other, which does not produce good clusters. It is possible to formulate multi-way cuts, balance partitioning, sparsest cut, etc., problems which in general will produce better clusters. Unfortunately, most of these formulations are NP-hard in the worst-case. Spectral graph clustering can be viewed as a relaxation of the sparsest cut(s) problem. There is a deep and beautiful theory about this which is beyond the scope of this course; the interested student is referred to the textbook by Kannan and Vempala (2009) and lecture notes by Trevisan (2014). Below, we discuss spectral graph partitioning in procedural terms and provide some intuition, rather than develop the theory around it.

Before returning to the example in Figure 6, we'll start with a much simpler example of a weighted graph. For a weighted undirected graph with N nodes, let W denote the $N \times N$ adjacency matrix, such that W_{ij} denotes the weight of the edge (i, j) ($W_{ij} = 0$ indicates that the edge is not present). As the graph is undirected, the matrix W is symmetric. We will disallow multiple edges and self-loops, so $W_{ii} = 0$ for all i. We denote by D the diagonal matrix, where $D_{ii} = \sum_j W_{ij}$, the weighted degree of node i. The matrix $L = D - W$ is called the Laplacian of the graph. The matrix L is always positive semi-definite and the smallest eigenvalue of the Laplacian is always 0 and the corresponding eigenvector is $\mathbf{1}$, the vector of all ones. A simple weighted undirected graph (disconnected with two components) along with the corresponding

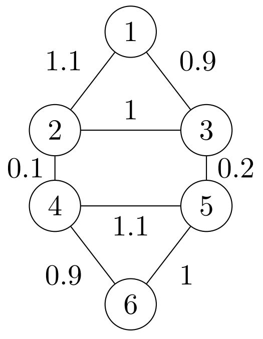


$$\mathbf{W} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

$$\mathbf{L} = \mathbf{D} - \mathbf{W} = \begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & -1 & -1 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & -1 & 2 \end{bmatrix}$$

(a) A graph with two connected components

(b) The matrices W, D and L for the graph in (a)



$$\mathbf{W} = \begin{bmatrix} 0 & 1.1 & 0.9 & 0 & 0 & 0 \\ 1.1 & 0 & 1 & 0.1 & 0 & 0 \\ 0.9 & 1 & 0 & 0 & 0.2 & 0 \\ 0 & 0.1 & 0 & 0 & 1.1 & 0.9 \\ 0 & 0 & 0.2 & 1.1 & 0 & 1 \\ 0 & 0 & 0 & 0.9 & 1 & 0 \end{bmatrix}$$

$$\mathbf{D} = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2.2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2.1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2.3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.9 \end{bmatrix}$$

$$\mathbf{L} = \mathbf{D} - \mathbf{W} = \begin{bmatrix} 2 & -1.1 & -0.9 & 0 & 0 & 0 \\ -1.1 & 2.2 & -1 & -0.1 & 0 & 0 \\ -0.9 & -1 & 2.1 & 0 & -0.2 & 0 \\ 0 & -0.1 & 0 & 2.1 & -1.1 & -0.9 \\ 0 & 0 & -0.2 & -1.1 & 2.3 & -1 \\ 0 & 0 & 0 & -0.9 & -1 & 1.9 \end{bmatrix}$$

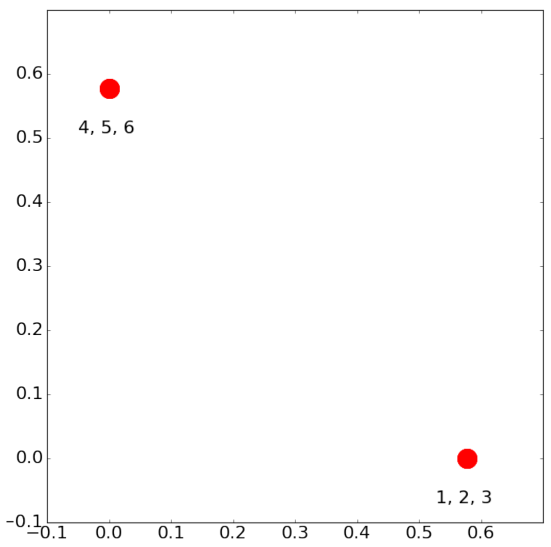
(c) A graph with two clusters

(d) The matrices W, D and L for the graph shown in (c)

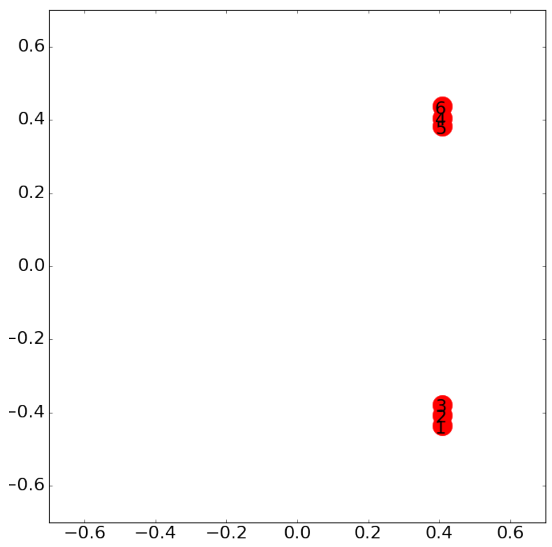
Figure 7: Two simple graphs along with their associated adjacency matrices, degree matrices and Laplacians.

Laplacian are shown in Figure 7(a) and (b).

For the graph shown in Figure 7(a) the vector $[1, 1, 1, -1, -1, -1]^T$ is also an eigenvector with eigenvalue 0. This is not an accident—for any graph with more than one connected components there will be multiple linearly independent eigenvectors with eigenvalue 0. (Exercise: Show that for a graph with k connected components there are exactly k orthogonal eigenvectors with eigenvalue 0.) What this gives us is a linear algebraic way of determining that the graph is disconnected. Let $\mathbf{v}_1, \mathbf{v}_2$ denote the two eigenvectors corresponding to the smallest eigenvalues. We observe that $\mathbf{v}_1$ and $\mathbf{v}_2$ are orthogonal as L is symmetric. We can form the matrix $\mathbf{V}_2 = [\mathbf{v}_1, \mathbf{v}_2]$ using these two vectors, which is an $N \times 2$ matrix. We can associate the i^{th} row of $\mathbf{V}_2$ with the i^{th} node to get a 2-dimensional embedding of the data. We can now cluster the data using this embedding, for example using the k-means procedure. Note that when we run an algorithm to find eigenvectors for the graph of Fig. 7(a), we may not actually get the vectors $[1, 1, 1, 1, 1, 1]^{\mathsf{T}}$ and $[1, 1, 1, -1, -1, -1]^{\mathsf{T}}$, but any two orthonormal vectors which span the same linear subspace as that spanned by these two vectors. We observe this in the scatterplot shown



(a) Scatterplot of v_1 vs v_2 for graph of Fig. 7 (a)



(b) Scatterplot of v_1 vs v_2 for graph of Fig. 7 (c)

Figure 8: Scatter plots of Euclidean embeddings obtained as part of the spectral clustering procedure applied to graphs shown in Figure 7.

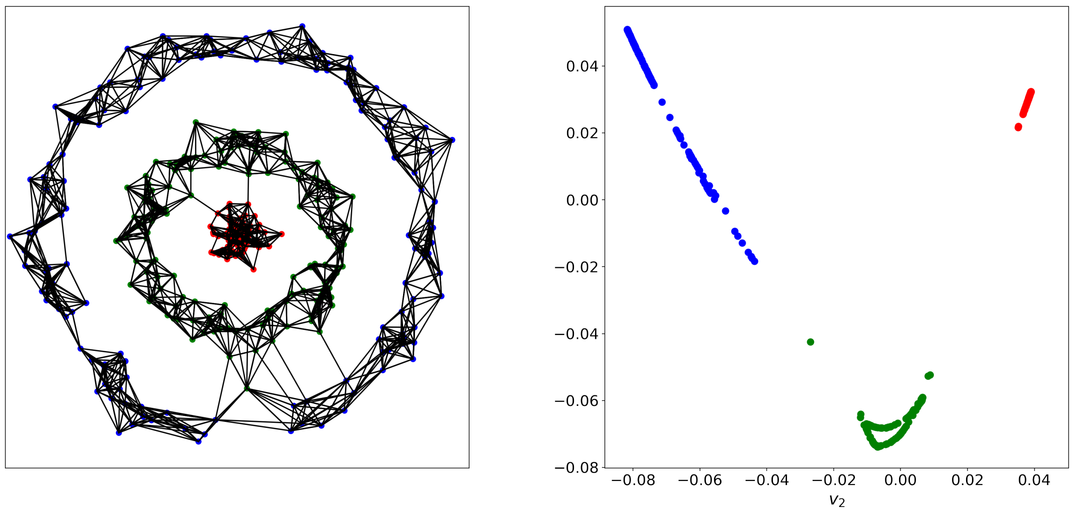
in Figure 8(a). The nodes 1, 2 and 3 are identical in their connectivity patterns and hence have the same Euclidean embedding; similarly for nodes 4, 5, 6.

Let us now consider the graph shown in Figure 7(c). The corresponding matrices W, D and L are shown in Fig. 7(d). It is no longer the case that there are two eigenvectors corresponding to the eigenvalue 0. However, if we consider the second smallest eigenvalue and the corresponding eigenvector, the resulting embedding is still useful for clustering. As we did previously, we can consider the representation $\mathbf{V}_2 = [\mathbf{v}_1, \mathbf{v}_2]$ to get a Euclidean embedding of the data with the eigenvectors corresponding to the smallest two eigenvalues. While the embeddings of the nodes 1, 2 and 3 (or 4, 5 and 6) are no longer identical, they are close to each other, as seen visually in Figure 8(b).

Algorithm 3 Spectral Clustering

```
1: procedure SPECTRAL CLUSTERING($W$, $k$)    ▷ Input graph given as weighted adjacency matrix
2:     Construct the diagonal degree matrix $D$ and Laplacian $L = D - W$.
3:     Find $\mathbf{v}_1, \ldots, \mathbf{v}_k$ the $k$ eigenvectors corresponding to the $k$ smallest eigenvalues
4:     Construct the $N \times (k-1)$ embedding matrix $\mathbf{V}_k = [\mathbf{v}_2, \ldots, \mathbf{v}_k]$
5:     Apply some clustering algorithm on $\mathbf{V}_k$, e.g., $k$-means
6: end procedure
```

In general, if we want to consider k clusters, we need to identify the eigenvectors corresponding to the k smallest eigenvalues. We will assume that the input graph is connected (otherwise, we can run the algorithm separately on each connected component). In this case, we don't need the eigenvector $\mathbf{v}_1$ corresponding to the (smallest) eigenvalue 0, as it will be (some scaling of) the vector $\mathbf{1}$. Thus, we let $\mathbf{V}_k = [\mathbf{v}_2, \mathbf{v}_3, \ldots, \mathbf{v}_k]$ be the $N \times (k-1)$ matrix which gives an embedding of the N points in $k-1$ dimensional space. We can then simply apply an algorithm such as k-means to identify the clusters. The complete spectral clustering algorithm is shown as Algorithm 3 and the generated graph and scatterplot of the Euclidean embeddings for the example in Figure 6(a) is shown in Figure 9. Thus, like kernel PCA, the first part spectral clustering can be viewed as a way to perform non-linear dimensionality reduction. If the graph



(a) Graph constructed using k-nearest neighbours for data shown in Fig. 6(a)

(b) Scatterplot of $\mathbf{v}_2$ vs $\mathbf{v}_3$ obtained after running Algorithm 3 on graph shown in (a)

Figure 9: Constructed graph and output clusters for spectral clustering.

has imbalanced degree degree distribution, it is often better to use the normalised Laplacian, $\widetilde{L} = I - D^{-1}W$.

References

David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1027–1035, 2007.

Ravindran Kannan and Santosh Vempala. *Spectral algorithms.* Foundations and Trends® in Theoretical Computer Science. Now Publishers, Inc., 2009.

S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28 (2):129–137, 1982.

Kevin P. Murphy. *Machine Learning : A Probabilistic Perspective.* MIT Press, 2012.

Luca Trevisan. Lecture notes on expansion, sparsest cut, and spectral graph theory, 2014. URL `https://people.eecs.berkeley.edu/~luca/books/expanders.pdf`.