An intro to spectral clustering

Jeremy Watt jermwatt@gmail.com - posted 7/3/13

The setup

Below is a standard dataset used to illustrate the fundamental weakness of the standard K-means clustering algorithm: that it assumes incoming data lie in convex globular clusters.

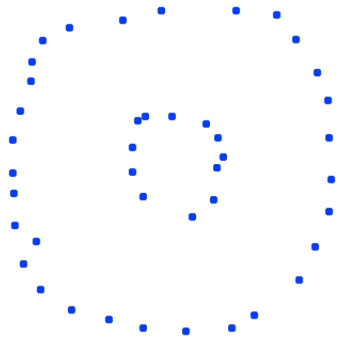

Figure 0.1: Two concentric rings of points.

You see two subgroups of data - the two concentric rings - and so if we were to cluster this set we would want to recover those two rings as our clusters. But thats not what K-means with K = 2 will do if we apply it to this dataset. Here's the type of separation we'll get with K-means on the left, and the correct separation (given by spectral clustering - the topic of these notes) on the right.

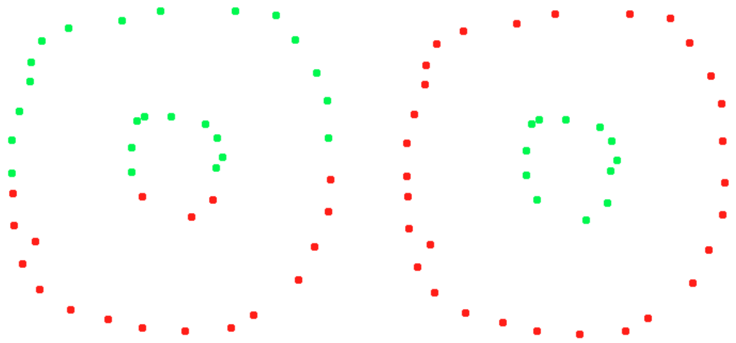

Figure 0.2: The left panel displays the result of applying K-means with K= 2 to the concentric ring (recovered clusters are color-coded). The correct clusters (returned by spectral clustering) is on the right, also color-coded.

In this set of notes we're going to learn about a clustering approach that throws out the K-means assumption that clusters fall in convex globular clusters and does something different: spectral clustering (we'll cover just the vanilla version, improved versions based on this can be found in standard papers on the subject). Spectral clustering clusters data points by treating them not as *distinct convex globular clusters* as in K-means but as *connected subgraphs*. This change in perspective allows separation of non-convex globular clusters like the concentric rings example above - something K-means just can't do. More importantly we'll see that spectral clustering can ignore sparse interconnections between arbitrarily shaped clusters of data - a characteristic that makes it an especially useful graph-based clustering technique.

1 Graphs and subgraphs

A graph is just a bunch of points (also called vertices) with edges connecting some of them to each other. These edges can be either be directed (they point from one vertex to another like a one-way street) or undirected (like a two-lane road), and are typically weighted. The weight on an edge is like a toll on a bridge connecting two cities: it represents a cost of traversing from one vertex to another by crossing that particular edge. In what follows we'll look at graphs with undirected edges. Here's an example of a graph[1].

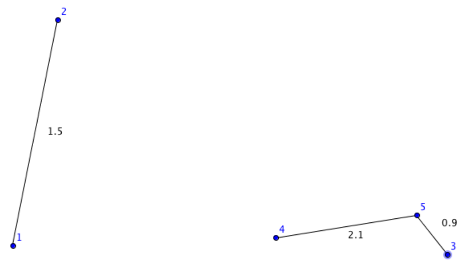

Figure 1.1: A simple graph with 5 points and a few edges connecting some of the vertices.

You usually use the notation G to refer to a graph, which consists of vertices $\mathbf{v}_1 ... \mathbf{v}_n$ and edges e_{ij} connecting the i^{th} and j^{th} vertices. In addition all of the edge weights can be recorded in a single matrix - referred to as the adjacency matrix W - corresponding to a particular graph. The $(i, j)^{th}$ entry $w_{ij} > 0$ is the weight of the connection between vertices i and j. If the i^{th} and j^{th} vertices don't share a connection you always give that connection a weight of $w_{ij} = 0$. Its also convention to give every vertex a connection to itself with a weight of 1 i.e. $w_{ii} = 1$ for all i.

So in the example illustrated above the adjacency matrix is

[1]This was drawn using the freeware GeoGebra, an awesome piece of free software for making geometric figures. You can download the software for free at their website http://www.geogebra.org/cms/en/

$$W = \begin{bmatrix} 1 & 1.5 & 0 & 0 & 0 \\ 1.5 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0.9 \\ 0 & 0 & 0 & 1 & 2.1 \\ 0 & 0 & 0.9 & 2.1 & 1 \end{bmatrix}$$

Subgraphs are just smaller connected sets of points inside G. So in the example above you can see that there are two subgraphs in G - the one with two points on the left, and the one consisting of three points on the right.

2 Finding connected subgraphs of G

Lets play a game, but we're not masochists so lets play a game thats rigged in our favor. Say we know our graph is made up of K connected subgraphs (smaller collections of vertices that are only connected to one another), but we don't know the subgraphs explicitly (i.e. we can't physically "see" the graph - it likely lives in a high dimension anyway). In addition say we also know all of the weighted connections between the points - collected in the adjacency matrix W. The game is this: we want uncover those subgraphs explicitly.

How can we go about doing this?

Well lets take the simple graph from the previous section. Notice how its adjacency matrix is block diagonal - there's a 2×2 block in the upper left and a 3×3 block in the lower right. Those blocks signify that the graph has two connected subgraphs consisting of 2 and 3 points respectively. You can just read off the subgraph membership right from these blocks of the adjacency matrix - from it you can tell that (for example) node 1 connects to node 2 and no other node, and vice versa.

Even if the nodes were numbered differently and the adjacency matrix wasn't block diagonal, you could still read off the distinct subgraph memberships right from the matrix - it would just be slightly more tedious. For example, say we switch the label 2 and 4, then the adjacency matrix would have been

$$\hat{W} = \begin{bmatrix} 1 & 0 & 0 & 1.5 & 0 \\ 0 & 1 & 0 & 0 & 2.1 \\ 0 & 0 & 1 & 0 & 0.9 \\ 1.5 & 0 & 0 & 1 & 0 \\ 0 & 2.1 & 0.9 & 0 & 1 \end{bmatrix}$$

It was easier to recognize subgraphs of G when its adjacency matrix was block diagonal - but we can still start reading down the rows (or columns) here to determine the subgraphs. Moreover, if we permute the 2^{nd} and 4^{th} rows and columns here we're back to our original W. This is true in general: the adjacency matrix corresponding to a graph with K separate and connected subgraphs is always - under mirrored permutation of its rows and columns - a block diagonal matrix with K

blocks. So regardless of how the adjacency matrix is constructed (i.e. how the nodes are ordered) the K distinct subgraphs of G can be recovered via the adjacency matrix alone.

3 A naive graph based clustering method

In a standard clustering situation the data $\mathbf{v}_1 ... \mathbf{v}_n$, doesn't come in "graph form" - we might feel safe treating the data points as nodes but it certainly doesn't come equipped edges/weights or equivalently an adjacency matrix. So we have to create an adjacency matrix for the data - usually by applying some kind of reasonable assumption regarding how points are related to each other. One common method to establish an adjacency matrix - called the $\epsilon - neighborhood$ weighting - calls for a user defined tolerance $\epsilon > 0$ and then sets entries in the adjacency matrix as

$$w_{ij} = \begin{cases} 1 & if \ \|\mathbf{v}_i - \mathbf{v}_j\|_2 < \epsilon \\ 0 & else \end{cases}$$

This connects points that are in an $\epsilon - neighborhood$ of one another with an edge with weight 1, and creates no edge to points outside the neighborhood. It seems like a reasonable way to construct a graph from a set of data.

This is one of the weighting approaches implemented in the attached MATLAB code called adjacency_toy.m. Notice that this approach could have been used on the concentric ring dataset in section 1 and we would get the same clustering provided by spectral clustering. For kicks here's a comparison of simple graph clustering using an $\epsilon - neighborhood$ scheme versus a $K - means$ approach on a smiley-face dataset.

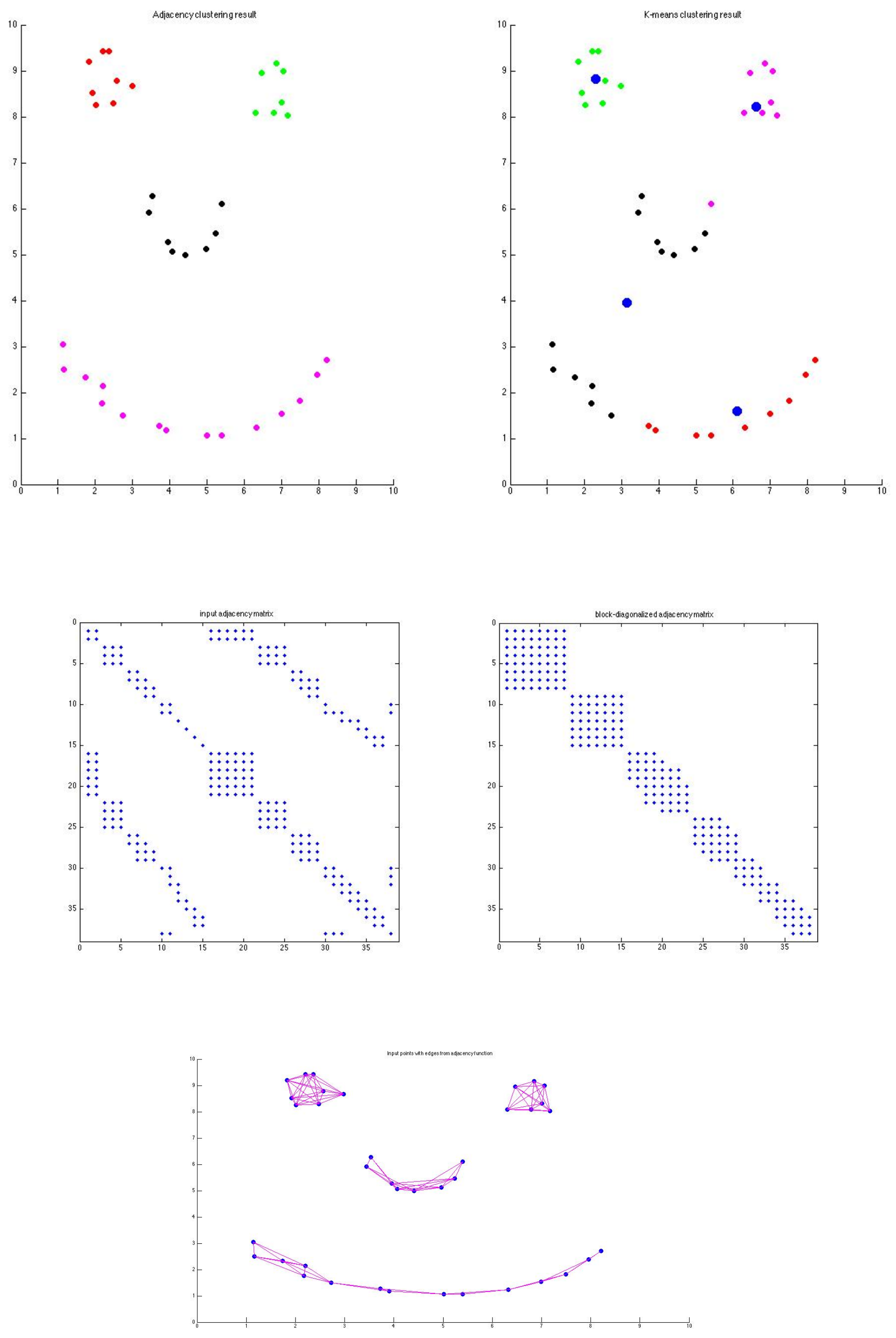

Figure 3.1: Top panel: A comparison of naive graph clustering (left plot) and K-means with K = 4 (right plot, centroids in blue) on a smiley-face dataset made using adjacency_toy.m. Clusters are color-coded and you can see that the simple graph clustering recovers the true clusters, whereas K-means does a pretty poor job. Middle panel: the initial adjacency matrix for the smiley face data (left) and adjacency matrix after block-diagonalization (right). Bottom panel: the smiley graph created using the data and the corresponding $\epsilon - neighborhood$ created edges (stretched horizontally so that the graph edges are more easily seen).

A real weakness of the $\epsilon - neighborhood$ scheme is that it can be difficult to pick a good ϵ due to the fact that data in a single dataset can cluster at different scales. For example - below I've made two clusters, the top one is tightly packed while the other is spread out (the distance between the cluster's points are all greater than the chosen ϵ). Each point in the bottom spread out cluster will be treated as a distinct cluster - with this approach or with spectral clustering - bummer.

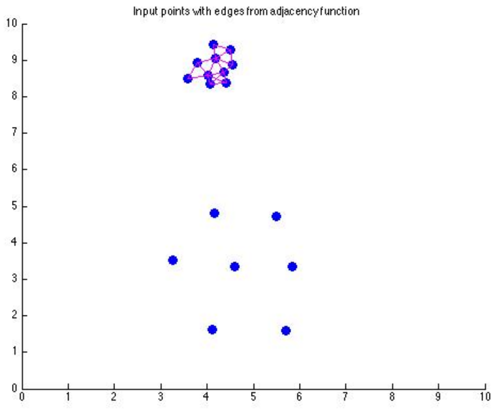

Figure 3.2: Here ϵ has been chosen so that the tight cluster on top is connected, but the points in large-scale cluster on the bottom get treated as individual clusters themselves.

A second more stable way to create an adjacency matrix is the *k nearest neighbors (knn)* approach - I've used a lowercase "k" here purposefully to differentiate between the uppercase "K" which is saved to denote the number of clusters in a method like K-means and spectral clustering. The "k" in k nearest neighbors denotes the number of edges created for each point in the dataset. In particular you create k edges to each of a datapoint's k closest neighbors, that is

$$w_{ij} = \begin{cases} 1 & if\, \mathbf{v}_j\, is\, one\, of\, the\, k\, closest\, points\, to\, \mathbf{v}_i \\ 0 & else \end{cases}$$

To make the adjacency matrix symmetric you then set each $w_{ji} = 1$ for every w_{ij} set equal to one. Here's an example - again using adjacency_toy.m - of naive graph clustering using k nearest neighbors with k = 4.

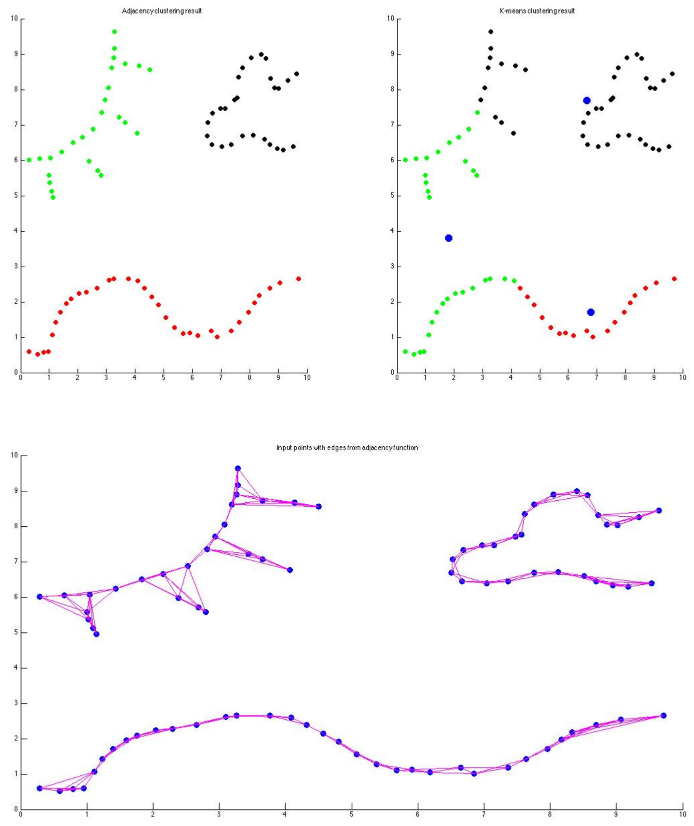

Figure 3.3: Top panel: A comparison of simple adjacency matrix clustering using k (= 4) nearest neighbors (on left) and K-means (on right - centroids are marked as blue dots). Bottom panel: the graph generated using the knn adjacency matrix (stretched horizontally so that the edges are more easily seen).

The catch with the knn approach - and most of the other approaches for generating adjacency matrices - is that typically regardless of how well you tune k you will get some sparse connections between connected subgraphs[2]. This will completely ruin the naive adjacency matrix approach for graph based clustering we've discussed above. Take the concentric ring example given in the first section - only now with the addition of one additional point. The naive graph based clustering approach will return just one cluster instead of distinguishing between the rings.

[2]Another approach to generating the adjacency matrix - the Gaussian distance approach - also majorly suffers from this problem. Here you pick a tolerance $\sigma > 0$ and using the Gaussian distance function $w_{ij} = e^{-\frac{\|\mathbf{v}_i - \mathbf{v}_j\|_2^2}{\sigma}}$ you create non-zero weights between every two points. This effectively treats the whole dataset as a completely connected graph - meaning each point is connected to every other point in the set. However those points where $\|\mathbf{v}_i - \mathbf{v}_j\|_2^2 > \sigma$ will have an exponentially shrinking weight connecting them. From what I've seen $\epsilon - neighborhood$ and especially $k - nearest\, neighbor$ schemes are more popular than this one. This seems to be due both to the fact that its harder to tune the σ parameter than the ϵ or k parameters, and because creating a completely connected graph is kind of overkill.

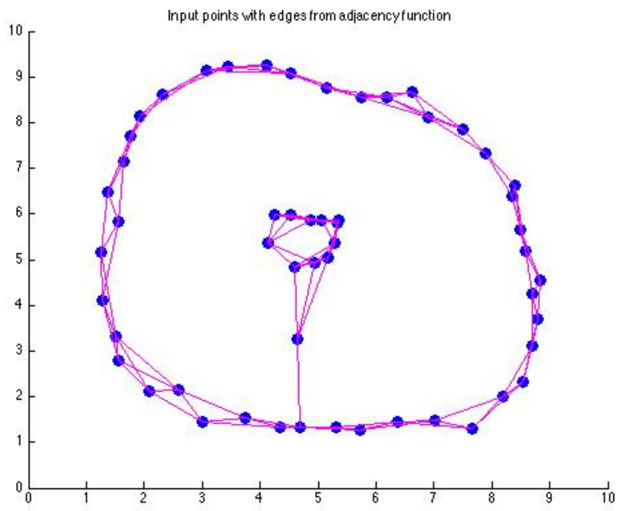

Figure 3.4: A concentric rings dataset with a single connection between the rings. The adjacency matrix corresponding to the data will tell us there is only one cluster here.

Damn - what a bummer.

This practical problem motivates the major topic of these notes: spectral clustering. Spectral clustering is a popular approach taken to cluster data that aims to circumvent the problem of sparse subgraph interconnections that just kills naive graph-based clustering, while retaining the awesome ability of graph-clustering to recover non-convex-globular shaped clusters.

4 Connected subgraphs and eigenvectors

4.1 A different perspective on subgraph recovery

Due to sparse interconnections we need another way to get at the subgraph information that is naturally encoded in the adjacency matrix of a graph with K connected components. However for now we will to assume that our graph has completely separated subgraphs - and then generalize what follows to the case where this does not hold.

Pretend that we couldn't use the naive adjacency based approach to recovering subgraphs discussed in the previous section. Another way to recover nodes in the distinct subgraphs of G is via a sequence of smooth optimization problems that look something like

$$\underset{\mathbf{f}}{min} \sum_{i,j=1}^{n} w_{ij} \left(f_i - f_j\right)^2 \tag{4.1}$$

where $w_{ij} \geq 0$ is the $(i,j)^{th}$ weight from the adjacency matrix. The smallest the value this quantity can take is zero. The unit vector $\mathbf{f} = \mathbf{1}$ (or a scaled version of it) achieves this minimum value, but do other solutions besides this trivial one exist?

Notice what happens to the objective if you set $f_k = 1$ for for all indices k associated to vertices in a single connected component of G (call the index set for this connected component S) and $f_k = 0$ for all other vertices. This means that

$$\sum_{i,j \notin S}^{n} w_{ij} \left(f_i - f_j\right)^2 = 0$$

since $f_i = f_j = 0$ for all i, j because none of these vertices are in S. But then you also have

$$\sum_{i,j \in S}^{n} w_{ij} \left(f_i - f_j\right)^2 = 0$$

since $f_i = f_j = 1$ for all of these vertices. Awesome - we've now accounted for everything in the graph and so together this gives

$$\sum_{i,j=1}^{n} w_{ij} \left(f_i - f_j\right)^2 = \sum_{i,j \in S}^{n} w_{ij} \left(f_i - f_j\right)^2 + \sum_{i,j \notin S}^{n} w_{ij} \left(f_i - f_j\right)^2 = 0$$

This shows that for each connected component of the graph with corresponding index set S the vector $\mathbf{f}$ where

$$f_i = \begin{cases} 1 & i \in S \\ 0 & else \end{cases} \tag{4.2}$$

minimizes problem 4.1. For future reference note that we could use any constant for the non-zero support of $\mathbf{f}$, not just one, and the same principle applies. For example we could use

$$f_i = \begin{cases} \frac{1}{|S|} & i \in S \\ 0 & else \end{cases}$$

where $|S|$ is the cardinality of S. Then each $\mathbf{f}_i$ has unit norm.

For a graph with K connected components denote by $\mathbf{f}_1 ... \mathbf{f}_K$ these detection vectors. Then notice that these guys are naturally orthogonal to each other i.e.

$$\mathbf{f}_i^T \mathbf{f}_j = 0 \;\; for \; i, j = 1...K, \;\; i \neq j$$

since each has support (non-zero entries) on a distinct connected component of the graph. If you think about it for minute, these $\mathbf{f}$ vectors are *the only* solutions to 4.1 besides the unit vector $\mathbf{1}$ (or a scaled version of it).

4.2 The eigenvector recovery problems

Now lets isolate a set of problems that returns the subgraph detecting $\mathbf{f}_i$ vectors only (since these are the interesting solutions), and not the trivial unit vector. We can use the fact that the interesting solutions to 4.1 are orthogonal to each other in order to generate a sequence of related problems that return those $\mathbf{f}_i$ as solutions. This sequence of problems is

$$
\begin{aligned}
\mathbf{f}_1 &= \underset{\mathbf{f}}{argmin} \sum_{i,j=1}^{n} w_{ij} \left(f_i - f_j\right)^2 \\
\mathbf{f}_2 &= \underset{\mathbf{f}^T\mathbf{f}_1=0}{argmin} \sum_{i,j=1}^{n} w_{ij} \left(f_i - f_j\right)^2 \\
&\;\;. \\
&\;\;. \\
\mathbf{f}_K &= \underset{\substack{\mathbf{f}^T\mathbf{f}_k=0 \\ k=1\ldots K-1}}{argmin} \sum_{i,j=1}^{n} w_{ij} \left(f_i - f_j\right)^2
\end{aligned}
\tag{4.3}
$$

OK cool. But can you actually compute our vectors of interest from this set of problems? You absolutely can. Because unlike the first approach we saw to uncover the subgraphs of G - finding the right permutation matrix to diagonalize W - our optimization problems 4.1 are smooth, and where there's smoothness in the world there's always hope. However first its helpful to slightly - but very cleverly - rewrite the common objective function of these problems using the so called Laplacian matrix $L = D - W$. Here the matrix D is diagonal[3] with the i^{th} diagonal entry given by $d_i = \sum_{j=1}^{n} w_{ij}$ and with this notation the common objective can be rewritten as

$$
\sum_{i,j=1}^{n} w_{ij} \left(f_i - f_j\right)^2 = \mathbf{f}^T L \mathbf{f}
$$

You can check that real quick to make sure its right. Its also easy to check that L is symmetric and positive semi-definite[4]. Then the system of problems 4.1 we want to solve becomes

$$
\begin{aligned}
\mathbf{f}_1 &= \underset{\mathbf{f}}{argmin}\, \mathbf{f}^T L \mathbf{f} \\
\mathbf{f}_2 &= \underset{\mathbf{f}^T\mathbf{f}_1=0}{argmin}\, \mathbf{f}^T L \mathbf{f} \\
&\;\;. \\
&\;\;. \\
\mathbf{f}_K &= \underset{\substack{\mathbf{f}^T\mathbf{f}_k=0 \\ k=1\ldots K-1}}{argmin}\, \mathbf{f}^T L \mathbf{f}
\end{aligned}
\tag{4.4}
$$

Due to the symmetric positive semi-definiteness of L this kind of system of smooth optimization problems is very special, with each problem having a closed form solution. Specifically $\mathbf{f}_i$ - the solution to the i^{th} problem - is precisely the i^{th} eigenvector of L corresponding to the i^{th} zero eigenvalue of L. If you haven't seen this optimization-characterization of eigenvectors of symmetric positive-definite matrices, here's the general idea.[5]

[3] D is known as the degree matrix associated to the graph and the i^{th} diagonal entry d_i is called the degree of the i^{th} vertex, a count of how many edges that vertex is adjacent to.

[4] A matrix B is positive semi-definite if and only if $\mathbf{x}^T B \mathbf{x} \geq 0$ for all $\mathbf{x} \neq \mathbf{0}$ or equivalently if all of the eigenvalues of B are non-negative.

[5] You typically encounter this definition of eigenvectors/values in a presentation of Principle Component Analysis where you aim to dig out principle components from a sample covariance matrix.

4.3 The details

For any $n \times n$ symmetric positive definite matrix B with eigenvalues $0 \le \lambda_1 \le \lambda_2 \le \cdots \le \lambda_n$ and corresponding eigenvectors $\mathbf{y}_1 ... \mathbf{y}_n$ (these are also orthogonal since B is symmetric) you can recover the first eigenvalue/eigenvector pair from the optimization problem

$$\lambda_1 = \underset{\substack{\mathbf{f} \\ \|\mathbf{f}\|=1}}{min}\, \mathbf{f}^T B \mathbf{f}$$

where the $\mathbf{y}_1$ is the vector $\mathbf{f}$ minimizing the objective. You can show that this is true pretty easily. With the eigen-decomposition $B = Y \Lambda Y^T$ where Y is the matrix whose columns are eigenvectors and Λ is a diagonal matrix with the corresponding eigenvalues along its diagonal. Then

$$\underset{\substack{\mathbf{f} \\ \|\mathbf{f}\|=1}}{min}\, \mathbf{f}^T B \mathbf{f} = \underset{\substack{\mathbf{f} \\ \|\mathbf{f}\|=1}}{min}\, \mathbf{f}^T Y \Lambda Y^T \mathbf{f} \ge \underset{\substack{\mathbf{f} \\ \|\mathbf{f}\|=1}}{min}\, \lambda_1 \mathbf{f}^T Y Y^T \mathbf{f} = \lambda_1$$

So λ_1 is at least a lower bound on the problem. Now set $\mathbf{f} = \mathbf{y}_1$, then the objective meets the lower bound since $\mathbf{y}_1^T B \mathbf{y}_1 = \mathbf{y}_1 Y \Lambda Y^T \mathbf{y}_1 = \lambda_1 \mathbf{e}_1^T \mathbf{e}_1 = \lambda_1$ where $\mathbf{e}_1$ is the first standard basis vector. The i^{th} eigenvalue/eigenvector pair are then characterized by the related problem where you restrain the recovered vector to be orthogonal to the previous $i-1$ recovered eigenvectors ($\mathbf{f}_j = \mathbf{y}_j$ for $j = 1...i-1$)

$$\lambda_i = \underset{\substack{\mathbf{f} \perp \{\mathbf{f}_1 ... \mathbf{f}_{i-1}\} \\ \|\mathbf{f}\|=1}}{min}\, \mathbf{f}^T B \mathbf{f}$$

The symbol "$\perp$" reads "perpendicular to". Again the i^{th} eigenvector $\mathbf{y}_i$ is the argument minimizing this quantity. This is also pretty easy to show.

Since the eigenvectors $\mathbf{y}_1 ... \mathbf{y}_n$ form an orthogonal basis any $\mathbf{f}$ can be written as $\mathbf{f} = \alpha_1 \mathbf{y}_1 + \cdots + \alpha_n \mathbf{y}_n$ for some $\alpha's$. If we assume $\mathbf{f}$ is orthogonal to $\mathbf{y}_1 ... \mathbf{y}_{i-1}$ then we have $\mathbf{y}_j^T \mathbf{f} = \alpha_j = 0$ for $j = 1...i-1$. So the objective becomes $\mathbf{f}^T B \mathbf{f} = \alpha_i^2 \lambda_i + \cdots \alpha_n^2 \lambda_n$ which is minimized when $\alpha_i = 1$ and $\alpha_k = 0$ for $k > i$.

4.4 Back to the action

Now, the only visible difference between the system of problems described in "The details" section above and ours in 4.4 is that we didn't demand that the $\mathbf{f}$ vectors have norm one explicitly. But recall from the comment made above about after introducing these vectors in equation 4.2 that assuming the $\mathbf{f}$ vectors are normalized does not change our problem. So indeed our system does return the K smallest eigenvalues and corresponding eigenvectors of L. Furthermore, because we saw above that the solution to each system is in fact zero - these smallest K eigenvalues are zero as well i.e. $\lambda_1 = \lambda_2 = \cdots = \lambda_k = 0$.

Practically speaking then, the set of problems is super easy to solve. Just perform an SVD decomposition of L and pick the K eigenvectors associated to the K zero eigenvalues of the matrix

L. Lets denote these non-unit eigenvectors in matrix form as $Y = [\mathbf{y}_1 ... \mathbf{y}_K]$. Then the indices of the non-zero support of $\mathbf{y}_i$ tell us which points belong to the i^{th} subgraph.

The K connected subgraphs of a graph G can be found by computing the K non-zero eigenvectors of the Laplacian matrix L associated to the eigenvalue $\lambda = 0$. These eigenvectors will have non-overlapping support, and the support of the i^{th} eigenvector gives the assignment of points to the i^{th} subgraph.

Lets look at an example - recall the simple 5-point graph we looked at in section 1. Computing the eigenvectors associated to the zero eigenvalues of its Laplacian matrix we'll have

$$Y = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \tag{4.5}$$

Now we can just read off - from the non-zero entries in each column - the distinct subgraph memberships.

5 Spectral clustering

So far all we've done is cover another (much more complicated way) of detecting K subgraphs, under the assumption that the subgraphs are completely disconnected. Now what if there are sparse interconnections between the subgraphs? Can the eigenvector approach discussed in the previous section be generalized? The first thing that comes to mind to generalize would be to now examine Laplacian eigenvectors associated to the K *smallest* eigenvalues (instead of zero eigenvalues, which we can't be sure L will have) - hopefully they will retain some kind of structure that can help us determine subgraphs. This is the most naive generalization you could think of, right? To check and see if it works we should run some experiments and look at the resulting eigenvector structure.

Here's an example experiment I made using the attached MATLAB file spectral_toy_2.m. A random block diagonal adjacency matrix is generated (with 3 blocks = 3 distinct fully connected subgraphs) and the three eigenvectors of the associated Laplacian are plotted. You can see that the support of each eigenvector determines a subgraph. Then a bunch of links are removed from the blocks, and a bunch of sparse connections are added between the blocks, and the three eigenvectors (associated to the smallest three eigenvalues) of the Laplacian associated to this new noisy adjacency matrix are plotted together.

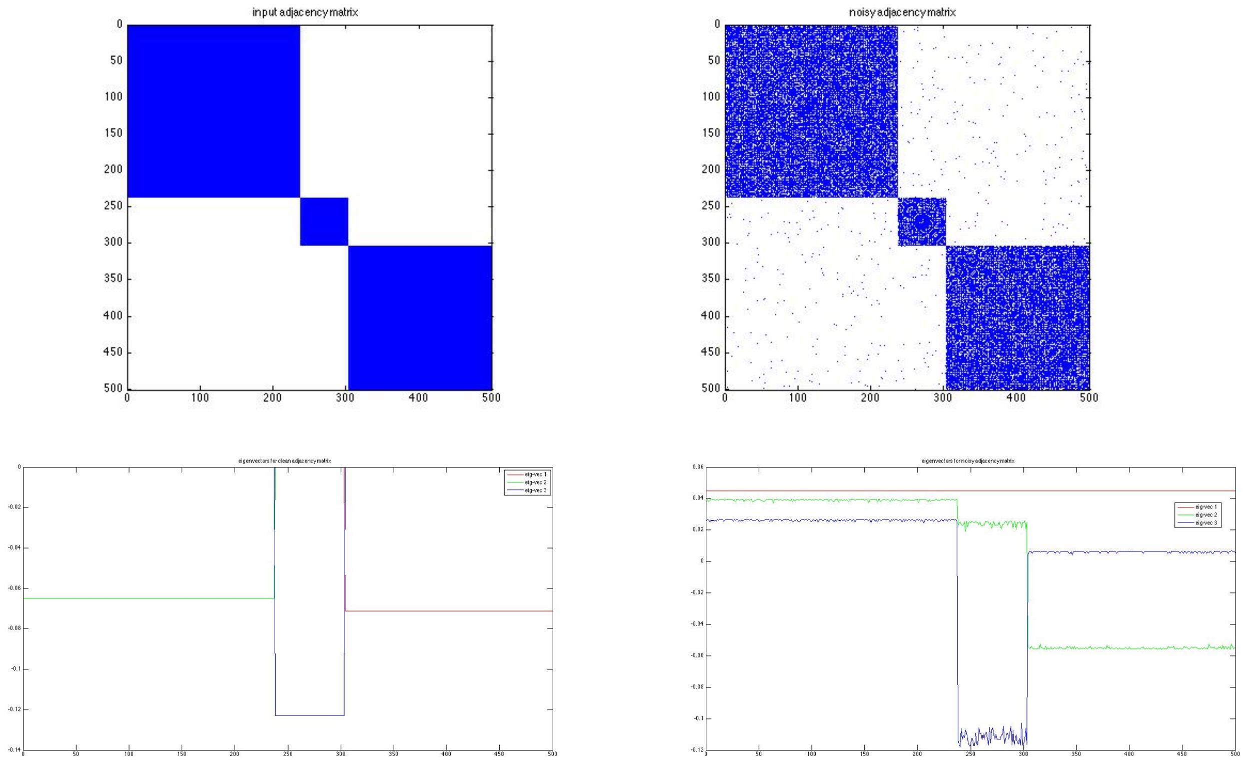

Figure 5.1: In the top left panel is a block diagonal adjacency matrix with its three subgraph-detecting Laplacian eigenvectors below it in the bottom left panel.. In the top right panel is the same adjacency matrix with sparse interconnections added, and intercluster links removed. The eigenvectors associated with the three smallest eigenvalues of the new Laplacian are plotted on the bottom right - you can see that the three underlying clusters are still identifiable using structure of these eigenvectors (but not their support).

Awesome - it does look like the structure of the eigenvectors of the noisy matrix - particularly the second and third ones - still define the three distinct subgraphs. You can run more tests like this - adding more blocks and looking at more eigenvectors - and you'll start to believe that yes, the eigenvectors still do reveal the subgraphs.

Now - how can we leverage this eigenvector structure to determine subgraph assignments? Well, scanning horizontally across from left to right on the noisy eigenvector plot above is precisely scanning down the rows of Y. Because each eigenvector is more or less constant over each subgraph you can then see that the *rows of* Y *cluster according to the subgraphs*. Therefore instead of assigning the i^{th} point to a given cluster based on the support of the i^{th} row of Y, we can analaogously cluster the rows of Y using $K - means$ and assign the point to whichever cluster the i^{th} row of Y is assigned too. This is the procedure spectral clustering follows - and it reduces to matching points using the support of Y in the case of completely distinct subgraphs.

Together these two naive generalizations of the Laplacian eigenvector approach constitute spectral clustering. Spectral clustering works as a graph–based clustering method that is fairly robust to sparse interconnections between subgraphs. This can be seen experimentally by playing around with the associated MATLAB file spectral_toy_2.m (or in an array of papers), and can be justified theoretically in a variety of waysVon Luxburg [2007]. Here's the basic spectral clustering algorithm stated officially for easy reference.

Algorithm 1 Basic spectral clustering algorithm

```
Input: $n$ data points $\mathbf{v}_1...\mathbf{v}_n$, adjacency matrix parameter, and number of clusters $K$
Output: $K$ cluster assignments
Begin
    1. Construct the adjacency matrix $W$ and graph Laplacian $L = D - W$ where $D$ is the degree matrix
    2. Compute the SVD of $L$ and retrieve $K$ the eigenvectors $Y = [\mathbf{y}_1....\mathbf{y}_K]$ associated with $L's$ smallest $K$ eigenvalues
    3. Use $K - means$ on the rows of $Y$ to determine cluster assignments
End
```

You can also try messing around with the spectral_toy.m interface to gain confidence in the robustness of this scheme and really visualize a graph with sparse interconnections - here's an example of one experiment I made. This is the same connected concentric ring dataset used I used in Figure 3.4 - explicitly I used the knn option with k = 4 and K = 2.

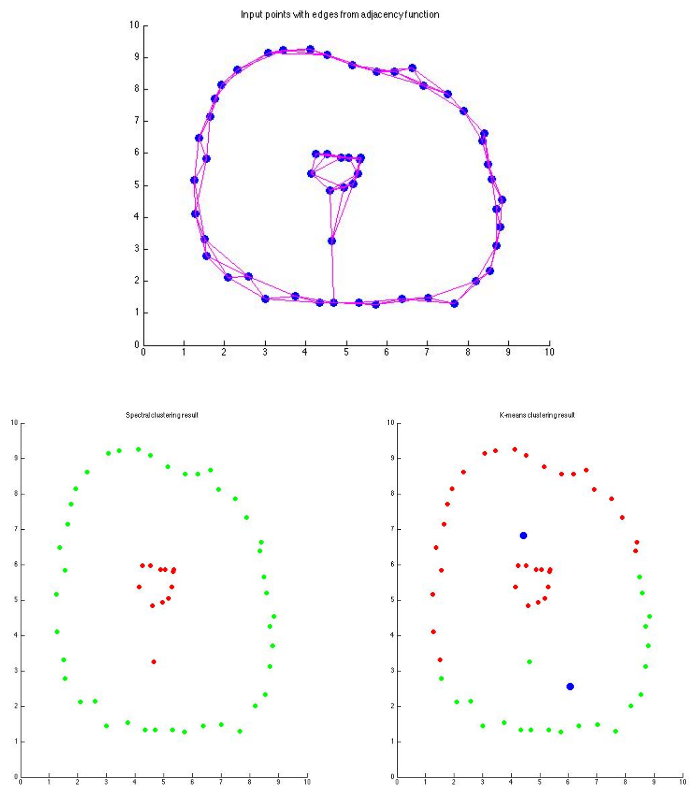

Figure 5.2: Top: the knn graph with k = 4 generated on the connected rings dataset. Bottom: a comparison of spectral clustering (left) and K-means (right) on the connected rings dataset using the knn matrix. Clusters are color-coded and the centroids are plotted in blue for K - means, and you can see that spectral clustering does a fine job. K-means does not.

Play around and make this thing fail! You can do it easily enough - just place your points so that there are enough connections between subgraphs and you'll break it. Its fun.

References

Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.