Chapter 8

Data Clustering

8.1 Introduction

The purpose of clustering data is so that we can represent groups of data by an exemplar, or cluster center. This is one way to deal with data sets that would otherwise be too large to handle.

Clustering can be intuitively clear, but algorithmically difficult. For example, consider the data in the first graph in Figure 8.1. I think you would probably agree that there appears to be three natural clusterings.

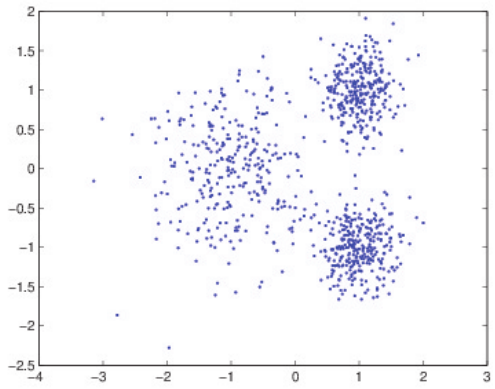


Figure 8.1: We see data that have no class labels, but there seems to be a natural separation of the data into three clumps.

A clustering then is primarily building a function that can take a data point as its input, and then output which cluster (or class) the data point belongs to.

Data may already have labels- We would call a clustering of that type to be *supervised learning*, since there are examples of "correct" labels. In our example, and in general, we'll be working with the unsupervised learning task, where the data points are given to us with no labels.

Definition: Unsupervised Clustering: Given a data set, X, whose elements are vectors $\boldsymbol{x}^{(i)} \in \mathbb{R}^n$, we want to construct a "membership function" which has its domain in $\mathbb{R}^n$ and will output the cluster index (or label). Defining this function as m, we have, for the i^{th} data point, its associated class label g_i:

$$m(\boldsymbol{x}^{(i)}) = g_i$$

where g_i is an integer from 1 to k (k being the number of clusters desired). Later we'll discuss how the algorithm might determine the value of k on its own, so for now, we'll consider k to be given.

The biggest issue with unsupervised clustering is that the problem specification is very ill-posed, meaning that there are many, many ways that one might build such a function m. For two extreme examples, consider these two membership functions:

$$m_1(\boldsymbol{x}^{(i)}) = 1$$

where i is the index for the data points. Another function that is about as useful as m_1 is the following:

$$m_2(\boldsymbol{x}^{(i)}) = i$$

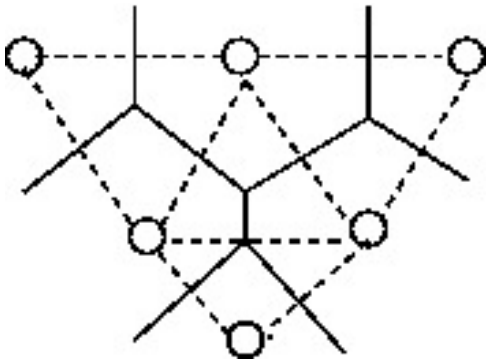
In this case, every data point is its own cluster. Neither situation is ideal, and we'll try to be somewhere in between these extremes.

In order to do determine some "best" clustering, that usually means that we'll be trying to minimize some error function, and the easiest way to do this is through *Voronoi Cells*:

Definition: Let $\left\{\boldsymbol{c}^{(i)}\right\}_{i=1}^{k}$ be points in $\mathbb{R}^n$. These points form k Voronoi Cells, where the j^{th} cell is defined as the set of points that are closer to cell j than any other cluster:

$$V_j = \left\{\boldsymbol{x} \in \mathbb{R}^n \mid \|\boldsymbol{x} - \boldsymbol{c}^{(j)}\| \leq \|\boldsymbol{x} - \boldsymbol{c}^{(i)}\|,\ i = 1, 2, \ldots, k\right\}$$

The points $\left\{\boldsymbol{c}^{(i)}\right\}_{i=1}^{k}$ are called *cluster centers*. In the uncommon occurrence that a point $\boldsymbol{x}$ lies on the border between cells, it is customary to include it in the cell whose index is smaller (although one would fashion the decision on the problem at hand). The reader might note that a Voronoi cell has a piecewise linear border as shown.



Examples in "Nature"

1. In [6], Voronoi cells are used to define the "area potentially available around a tree". That is, each tree in a stand represents the center of the cell.
2. Using a map of the campus with the emergency telephone boxes marked out and used as the centers, we could always tell where the closest phone is located.

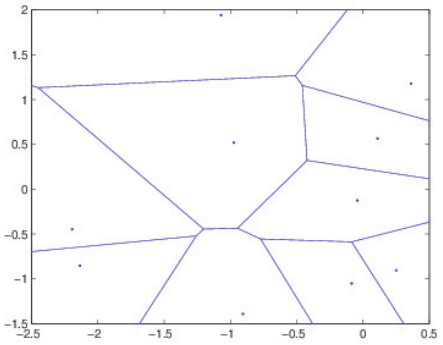
We can draw a Voronoi diagram by hand: Between neighboring cells, draw a line (that will be erased at the end), then draw the perpendicular bisectors for each line drawn. This can get complicated fairly quickly, so we will be using Matlab or Python to produce the plots. The algorithms that do these plots are very interesting (see any text in Computation Geometry) but are beyond the scope of our text.

Matlab Example:

```
X=randn(10,2);  %X is a tall matrix.
voronoi(X(:,1),X(:,2));
```

We can also have Matlab return the vertices of the Voronoi cells to plot them manually:

```
[vx,vy]=voronoi(X(:,1),X(:,2));
plot(vx,vy,'k-',X(:,1),X(:,2),'r*');
```



Python Example

In Python, we'll also build a Voronoi diagram using 10 random points in the plane. For this, well need `Voronoi` from `scipy`, and some other things for displaying the plot.

```
import matplotlib.pyplot as plt
from scipy.spatial import Voronoi, voronoi_plot_2d

A=np.random.rand(10,2)  # Note that A must be a tall matrix.
vor=Voronoi(A)

# Plot the results
fig = voronoi_plot_2d(vor)
plt.show()
```

In the algorithms that we work with, it will be convenient to have a function that will identify points within a given cluster- That is, will have the value 0 if the point is not in a given cluster, or 1 if it is. The "characteristic function" is typical for that purpose, and we'll define it as the following:

$$\chi_i(\boldsymbol{x}) = \begin{cases} 1 & \text{if } m(\boldsymbol{x}) = i \\ 0 & \text{otherwise} \end{cases}$$

One general way to measure the goodness of a clustering algorithm is to use a measure called the **distortion error** for a given cluster i, and the total distortion error[1]. Given a clustering of N points, let N_i denote the number of points in cluster i. We first define the error for cluster i, then sum them all for the overall error. Here is the definition for the distortion error for the i^{th} cluster, where we notice that if the point $\boldsymbol{x}^{(k)}$ is not in cluster i, then we simply add 0:

$$E_i = \frac{1}{N_i} \sum_{k=1}^{N} \|\boldsymbol{x}^{(k)} - \boldsymbol{c}^{(i)}\|^2 \, \chi_i(\boldsymbol{x}^{(k)})$$

so that the overall distortion error is defined to be:

$$E_{\text{total}} = \sum_{k=1}^{p} E_k$$

Now that we have a way of measuring the goodness of a clustering, we want to have an algorithm that will minimize the function.

8.2 K-means clustering

A relatively fast method to perform the clustering is the $k-$means clustering. In some circles, this is called Lloyd's algorithm, and there was an update done by Linde, Buzo and Gray to call that the Linde-Buzo-Gray (LBG) algorithm [26].

The membership function for both $k-$means and LBG is defined so that we are in cluster i if we are in the Voronoi cell defined by the i^{th} cluster center. That is,

$$m(\boldsymbol{x}) = i \text{ iff } \|\boldsymbol{x} - \boldsymbol{c}^{(i)}\| < \|\boldsymbol{x} - \boldsymbol{c}^{(j)}\| \;\; i \neq j, \;\; j = 1 : p \tag{8.1}$$

In the rare circumstance that we have a tie, we'll choose to put the point with the center whose index is smallest (but this is ad-hoc).

[1] Some authors do not square the norms, but it is more convenient for the theory, and does not change the end results.

The Euclidean Distance Matrix

You've seen that we'll need a function that will compute all of the inter-point distances for us. To be more precise, we'll need a function that will input two sets of points in $\mathbb{R}^n$, say there are M points in one set (organized in an $M \times n$ matrix), and N points in another (organized in an $N \times n$ matrix). The output will then be an $M \times N$ matrix where the (i, j) position is the distance between the point i in the first set and point j in the second set.

For Matlab/Octave, we'll write a short function to do this for us called `edm.m`. which will be available on the class website. For Python, you can compute the pairwise distances using the example below:

```
from sklearn.metrics.pairwise import euclidean_distances

X = [[0, 1], [1, 1]]
D=euclidean_distances(X, X) #D is 2 x 2 with distances.
```

The Algorithm

Keeping in mind that we'll be using Matlab for its implementation, we'll write the algorithm in matrix form.

Let X be a matrix of p data points in $\mathbb{R}^n$ (so each data point is a row, and the matrix is $p \times n$), and let C denote a matrix of k centers in $\mathbb{R}^n$ (each "center" is a row, and the matrix is $k \times n$).

At each pass of the algorithm, the membership function requires us to take $p \times k$ distance calculations, followed by p sorts (each point has to find its own cluster index), and this is where the algorithm takes most of the computational time.

K-Means Clustering

Given: p data points in $\mathbb{R}^n$ and k cluster centers (as vectors in $\mathbb{R}^n$). Then:

- Sort the data into k sets by using the membership function in Equation 8.1.
- Re-set center $\boldsymbol{c}_i$ as the centroid (mean) of the data in the i^{th} set.
- Compute the distortion error.
- Repeat until the distortion error no longer decreases (either slows to nothing or starts increasing).

A Toy Example:

Here we use the $k-$means algorithm three times on a small data set, so you can have a test set for the code that you write.

Let the data set $X^{5\times 2}$ be the matrix whose rows are given by the data points in the plane:

$$\left\{ \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix} \right\}$$

You might try drawing these on a plane. You may see a clustering of three points to the right, and a clustering of two points to the left.

We will use two cluster centers, (typically initialized randomly from the data) in this case:

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

The EDM of distances will be $D^{5\times 2}$ and is the distance between our 5 data points and 2 cluster centers.

The membership function then looks for the minimum in each row, shown as the vector M below:

$$D = \begin{bmatrix} \sqrt{2} & 1 \\ 0 & 1 \\ \sqrt{2} & \sqrt{5} \\ 1 & 0 \\ \sqrt{5} & \sqrt{8} \end{bmatrix}, \qquad M = \begin{bmatrix} 2 \\ 1 \\ 1 \\ 2 \\ 1 \end{bmatrix}$$

Resorting into two clusters,

$$\left\{ \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix} \right\} \qquad \left\{ \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$$

The cluster centers are re-set as the centroids:

$$\begin{bmatrix} -2/3 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 3/2 \end{bmatrix}$$

After one more sort and centroid calculation, we get:

$$\begin{bmatrix} -1 \\ -1/2 \end{bmatrix}, \begin{bmatrix} 2/3 \\ 4/3 \end{bmatrix}$$

which does not change afterward (so the clusters also remain the same).

The basic algorithm may have a few shortcomings. For example,

- The centers may be (or become) empty- That is, there are no data points closest to a given center. This is a good chance of this happening if you simply choose random numerical values as the initial centers, that's why we recommend choosing random *data points* as initial centers- You'll have at least one point in each cluster that way.

- We had to pre-define the number of clusters. Is it possible to construct an algorithm that will grow or shrink the number of clusters based on some performance measure? Yes, and that change is the LBG algorithm.

Matlab Note

If the "Statistics Toolbox" is available for your copy of Matlab, then `kmeans` is a built-in command.

Matlab's clustering routine expects the data to be given row-wise, so if we have p points in $\mathbb{R}^n$, then Matlab expects the data matrix to be $p \times n$.

Here's the previous example using the stats toolbox:

```
X=[1,0,-1,1,-1;2,1,0,1,-1]';
C=[0,1;1,1]';
[M,C1]=kmeans(X,2,'start',C);
```

If we didn't have an initial clustering C, the command is a lot shorter. For example, with just the data and number of clusters, we would have:

```
X=[1,0,-1,1,-1;2,1,0,1,-1]';
[M,C]=kmeans(X,2);
```

Matlab gives us several ways of initializing the clusters- See the documentation.

Exercises

1. Fill in the missing details from the example in the text: Fill in the EDM and the vector M after the second sort, and continue the example one more iteration to show that the clusters do not change. Hint: To compute the distance matrix, use the points as they were originally ordered.

2. Given p scalars, $x_1, x_2, \ldots, x_p$, show (using calculus) that the number μ that minimizes the function:

$$E(\mu) = \sum_{k=1}^{p} (x_k - \mu)^2$$

is the mean, $\mu = \bar{x}$. Hint: We think of $x_1, \ldots, x_p$ as being fixed numerical values, so E is a function of only one variable μ.

3. Generalize the last exercise so that the scalars (and μ) are now vectors in $\mathbb{R}^n$. That is, given a fixed set of p vectors in $\mathbb{R}^n$, show that the vector μ that minimizes:

$$E(\mu) = \sum_{k=1}^{p} \|\mathbf{x}^{(i)} - \mu\|^2$$

is the mean (in $\mathbb{R}^n$).

4. Write the following as a Matlab function. The abbreviation EDM is for Euclidean Distance Matrix. Good programming style: Include comments!

```
function z=edm(w,p)
%  A=edm(w,p)
%  Input:  w, number of points by dimension
%  Input:  p is number of points by dimension
%  Ouput:  Matrix z, number points in w by number pts in p
%          which is the distance from one point to another

[S,R] = size(w);
[Q,R2] = size(p);
p=p';
if (R ~= R2), error('Inner matrix dimensions do not match.'),end

z = zeros(S,Q);
if (Q<S)
  p = p';
  copies = zeros(1,S);
  for q=1:Q
    z(:,q) = sum((w-p(q+copies,:)).^2,2);
  end
else
  w = w';
  copies = zeros(1,Q);
  for i=1:S
    z(i,:) = sum((w(:,i+copies)-p).^2,1);
  end
end
z = z.^0.5;
```

5. If you don't have the `kmeans` function built-in, try writing one using the EDM function. It is fairly straightforward using the EDM function and the min function.

6. Will the cluster placement at the end of the algorithm be independent of where the clusters start? Answer this question using all the different possible pairs of initial points from our toy data set, and report your findings- In particular, did some give a better distortion error than others?

LBG modifications

We can think about ways to prune and grow the number of clusters, rather than making it a predefined quantity. Here are some suggestions for modifications you can try. This actually changes the $k-$means algorithm into the LBG algorithm.

- Split that cluster with the highest distortion measure, and continue to split clusters until the overall distortion measure is below some preset value. The two new cluster centers can be initialized a number of ways- Here is one option:
$$\boldsymbol{c}^{(i_1,i_2)} = \boldsymbol{c}^{(i)} \pm \boldsymbol{\epsilon}$$
However, this may again lead to empty clusters.

- We can prune away clusters that either have a small number of points, or whose distortion measure is smaller than some pre-set quantity. It is fairly easy to do this- Just delete the corresponding cluster center from the array.

8.3 Neural Gas

So far, the clustering algorithms we have considered use arithmetic means to place cluster centers. This works well when the data consists of "clouds" (coming from a random process, for example). This process does not work well when the data lies on some curved manifold[2]

If we imagine all of the data to be on the boundary of a circle, for example, taking an average will place the cluster centers off the circle.

Figure 8.2: If the data lies on a curved surface, then using the mean may pull the centers (stars in the graph) out of the data.

But there is something that may be exploited about the cluster centers to help us in visualizing how the data lies in its space.

It would be helpful if we knew what kinds of *structure* the data may have. For example, in Figure 8.2, it would be helpful to know that the data is actually (locally) one dimensional.

The method we will use to determine the structure of the data will be to define connections (or edges) between clusters. Consider the data in Figure 8.3. It is this kind of structure that the clustering should reflect- Which cluster centers are "neighboring" points in the data? Just as importantly, which are *not* neighbors in the data?

[2] A manifold is generally defined to be an object that looks locally like $\mathbb{R}^k$ for some k. For example, a path in three dimensions is locally 1 dimensional.

Figure 8.3: The star shaped points are the cluster centers. It would be nice to know which cluster centers belong together in a single "blob" of data, versus which cluster centers are in the other set of data.

In mathematical terminology, we want to find what is called a *topology preserving* clustering. For us, the *topology* of the clusters is defined by how the cluster centers are connected to each other (that is, so we know which cluster centers are neighbors and which are not).

More generally, we want the clustering to be what is called "topology preserving" in the sense that clusters defined to be neighbors in the cluster center topology are actually neighbors in the data, and vice versa. Before continuing, some graphical examples should help us with this concept.

In Figure 8.4, we see three topologies mapped to the data, which is a uniform distribution of points in $\mathbb{R}^2$. In the first picture, the topology of the cells is a one dimensional set. In this situation, the cluster mapping is not topology preserving, because neighboring cells in the topology are not adjacent in the plane. In the second situation, we have a three-dimensional topology mapping to the plane. In this case, neighboring data points in $\mathbb{R}^2$ are mapped to non-neighboring cells in the topology. Only in the third picture do we see that both parts of the topology preserving mapping are satisfied.

The Neural Gas Algorithm [27, 29, 28] is designed with a couple of big changes in mind (changes from k-means and LBG).

1. We don't want to use arithmetic means if possible. It is more desirable to have cluster centers that are actually embedded in with the data (like in Figure 8.3) rather than every cluster outside of the data (like in Figure 8.2).

2. We want the algorithm to build connections between the cluster centers. Hopefully this will tell us if we have separated regions of data (like in Figure 8.3).

Embedding the Clusters

To solve the first problem, we are going to endow the data with the power to attract the cluster centers. That is, when a data point $\mathbf{x}$ is chosen, we will determine which cluster center is closest (the "winner"), and we will move the cluster center towards that point. We don't want to go very far, just a slight move in that direction.

Mathematically, for a given data point $\mathbf{x}$, find the winning center, $\mathbf{c}$ and move in the direction $\mathbf{x} - \mathbf{c}$. The update rule would then look like:

$$\boldsymbol{c}_{\text{new}} = \boldsymbol{c}_{\text{old}} + h(t)(\mathbf{x} - \mathbf{c}_{\text{old}}) \tag{8.2}$$

where $h(t)$ will be a function determining how far to move. Normally, at the beginning of the clustering, h is relatively large. As the number of iterations grow larger, h gets very small (so that the movement at each iteration gets very slight).

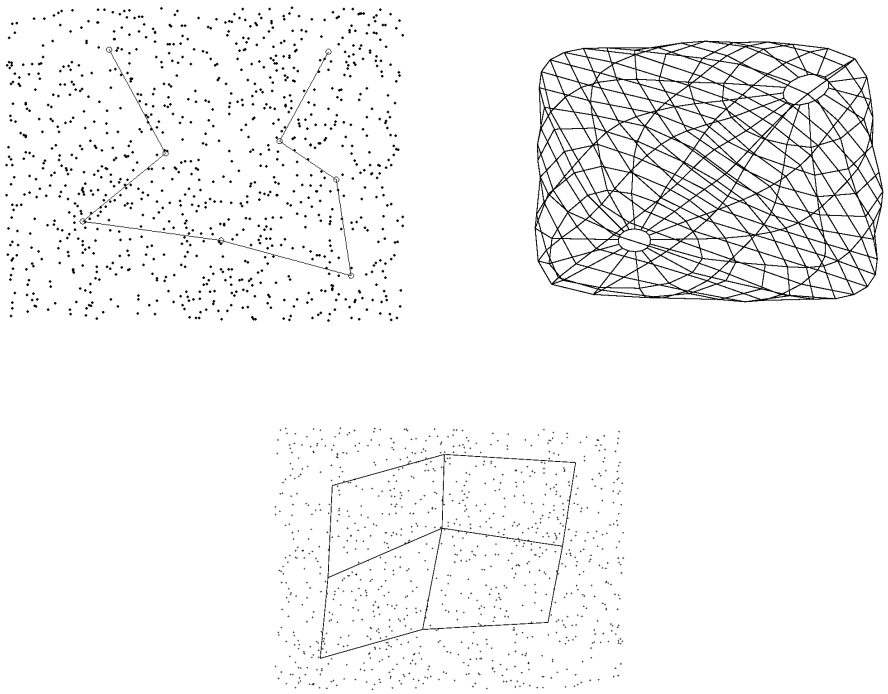


Figure 8.4: Which mapping is topology preserving? Figure 1 shows a mapping that is not topology preserving, since neighboring cells are mapped to non-neighboring points in $\mathbb{R}^2$. On the other hand, the middle clustering is not topology preserving, because neighboring points in $\mathbb{R}^2$ are mapped to non-neighboring cells in the topology. Only the third picture shows a topology preserving clustering.

This method would work very well, but it might result in a lot of empty cluster centers (centers that are never winners). This is where the concept of the "gas" comes into play- We think of the data as being in a viscous fluid, like jello.

How does this work in practice? As before, we first select a data point at random, and find the winning cluster center. We move the cluster center towards the data point. In a viscous fluid, that means that the cluster centers close to the winner will ALSO move toward the data point (just not as much). Therefore, instead of updating the single winning center using Equation 8.2, we update all centers.

The key point is that the winning center is attracted to $\mathbf{x}$ with the strongest force, then the next closest center is attracted with the next strongest force, and so on. To do this, we'll need to sort the centers by their distance to our data point $\mathbf{x}$.

You might have a negative reaction at this point- Given a lot of cluster centers, this is a lot of sorting. What is typical is that we will define a value k, and we will only update the k closest centers.

We define a way of setting this measure: We define s_i to be the number of centers closer to $\mathbf{x}$ than $\boldsymbol{c}^{(i)}$. The easiest way to compute this is to sort the centers by their distance to $\boldsymbol{x}$, then put the first k indices in a vector $\boldsymbol{v}$. So, for example, the vector $\boldsymbol{v}$ can be defined as:

$$\boldsymbol{v} = \{i_1, i_2, \ldots, i_k\}$$

so the "winning" center has index i_i. Now we define the values s:

$$s_{i_1} = 0, \quad s_{i_2} = 1, \quad s_{i_3} = 2, \quad \ldots, \quad s_{i_k} = k - 1$$

Example: Let $C = 0.1, 0.2, 0.4, 0.5$. If $x = 0.25$, then $i_1 = 2$, and $\boldsymbol{v} = \{2, 1, 3, 4\}$. The values of s: $s_2 = 0$, $s_1 = 1$, $s_3 = 2$, $s_4 = 3$.

Building Connections

Now that we have set up an algorithm that moves the centers into the data, we look at the problem of defining connections. The underlying idea is that we'll connect a winning center to its neighbor, indexed by $\boldsymbol{v}(2)$. If that connection has not been selected for a long time, that probably means that over time, the centers have drifted apart, and we should remove the connection.

Therefore, to build connections, we will use two arrays (if the number of centers is k, they are both $k \times k$), M and T.

Definition: A Connection Matrix, M, is a matrix of 1's and 0's where

$$M_{ij} = \begin{cases} 1 & \text{If cell i connected to j} \\ 0 & \text{Otherwise} \end{cases}$$

To build the connection matrix, once $\boldsymbol{v}$ is determined, then M_{i_1,i_2} is set to 1, and the age of the connection T_{i_1,i_2} is set to 0. All the other ages have 1 added to them.

Lastly, we should check to see how many iterations have passed since each edge was constructed. If too much time has passed (pre-set by some parameter), then remove those connections.

Parameter List

The number of parameters we'll use is a little large, but coding them is no problem. Here is a list of them, together with some standard values, where N is the number of data points.

- Number of data points: N
- Relative maximum distance to move, if we update the winning center: ϵ

$$\epsilon_{initial} = 0.3 \qquad \epsilon_{final} = 0.05$$

- Parameter λ also gives some control over how much to move the winner and neighbors.

$$\lambda_{initial} = 0.2N \qquad \lambda_{final} = 0.01$$

- The maximum amount of time before a connection is removed is:

$$T^m_{initial} = 0.1N \qquad T^m_{final} = 2N$$

- Maximum number of iterations: `tmax`$= 200N$.

The parameters that have an "initial" and "final" time are updated using the power rule. If the initial value is $\alpha_{initial}$ and the final value is α_{final}, then on iteration i,

$$\alpha_i = \alpha_{initial} \left(\frac{\alpha_{final}}{\alpha_{initial}} \right)^{i/\text{tmax}} \tag{8.3}$$

The Neural Gas Algorithm:

1. Select a data point $\boldsymbol{x}$ at random, and find the winner, $c^{(i_1)}$.
2. Compute $\boldsymbol{v}$ by finding the next $k-1$ closest centers.
3. Update the k centers:
$$c^{(i_k)} = c^{(i_k)} + \epsilon \exp\left(\frac{-s_{i_k}}{\lambda} \right) (\boldsymbol{x} - c^{(i_k)})$$
4. Update the Connection and Time Matrices: Set $M_{i_1,i_2} = 1$, and $T_{i_1,i_2} = 0$.
Age all connections by 1, $T_{j,k} = T_{j,k} + 1$ for all j, k.
5. Remove all old connections. Set $M_{j,k} = 0$ if $T_{j,k} \geq T^m$
6. Repeat.

2021 Note: I'm pulling all of the implementation details out of the main text and will provide it separately- These details can change a lot over the years.

8.3.1 Project: Neural Gas

This project explores one application of triangulating a data set: Obtaining a path preserving representation of a data set.

For example, suppose we have a set of points in $\mathbb{R}^2$ that represents a room. We have a robot that can traverse the room - the problem is, there are obstacles in its path. We would like a discretization of the free space so that the robot can plan the best way to move around the obstacles.

The data set is given in `obstacle1.mat`, and is plotted below. The areas where there are no points represent obstacles. **Assignment:** Use the Neural Gas clustering using $1,000$ data points and 300 cluster centers to try to represent the obstacle-free area.

8.4 DBSCAN

The method known as "Density-based spatial clustering of applications with noise", or more simply as DBSCAN[3] is called a *density-based* clustering algorithm. What makes it so distinctive from $k-$means is that it can create complex shapes for each "cluster", and secondly, it can identify "outliers", or points that shouldn't be classified as members of any cluster.

[3]Ester, Kriegel, Sander, and Xu, "A Density Based Algorithm for Discovering Clusters", KDD-96 Proceedings, 1996.

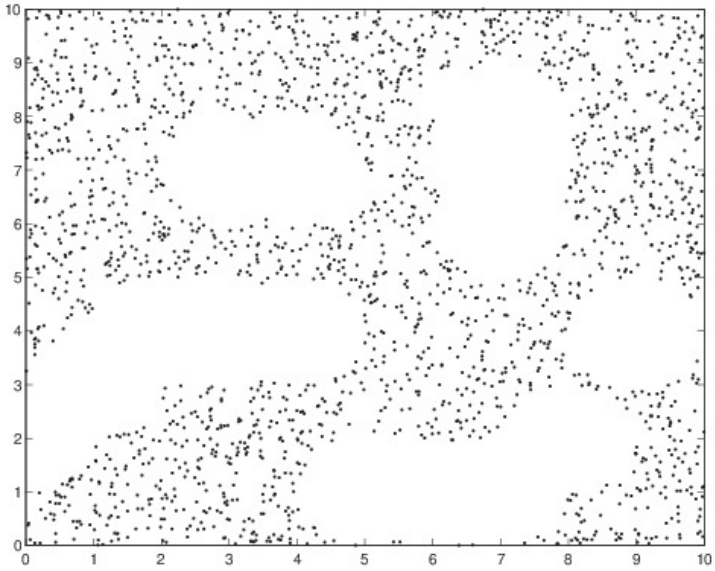


Figure 8.5: The data set that represents the "free" area. White regions indicate obstacles.

In the method, a "cluster" is defined by growing a set of points using a density criterion. That is, to be considered neighbors, a point will need to be within a given ϵ, of the center, and further, there will need to be a certain number of points also within ϵ before the set is considered a cluster. In this way, we might think of a cluster as, what some people would call, a "maximal set of density-connected points".

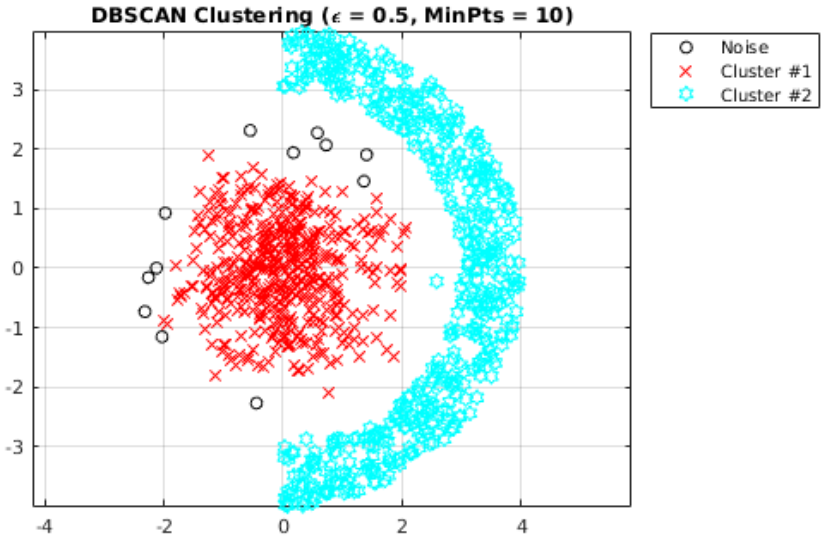
So given ϵ and a value for the smallest number of points allowed to make a cluster, `MinPts`, the algorithm separates data into three categories: A **core** point, a **border** point, or **noise**.

- A point is a **core** point if it has more than `MinPts` within ϵ.
- A point is a **border** point if if has fewer than `MinPts` within ϵ, but is still in the neighborhood of a core point.
- A **noise** point is a point that is not a core point, nor a boundary point.

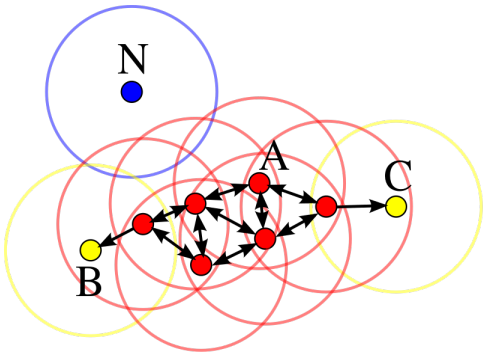
What results from the DBSCAN method is not a set of cluster centers- Rather, each "core point" is provided with an index of which cluster it belongs to. So the plot of the output is straightforward, and one such output is given in the figure to the right.

The algorithm for DBSCAN is straightforward. Here is **pseudo-code** for it:

- Compute neighbors of each point, and identify **core points.**
- Join neighboring core points into clusters.
- For each non-core point:
  - Add to a neighboring core point if possible
  - Otherwise, add to "noise".



In this diagram representing the output of the DBSCAN algorithm, we see two clusters- one indicated by red $\times$, the other cluster is formed from light blue circles (may be hard to see in grayscale), and points that are black circles represent "noise".

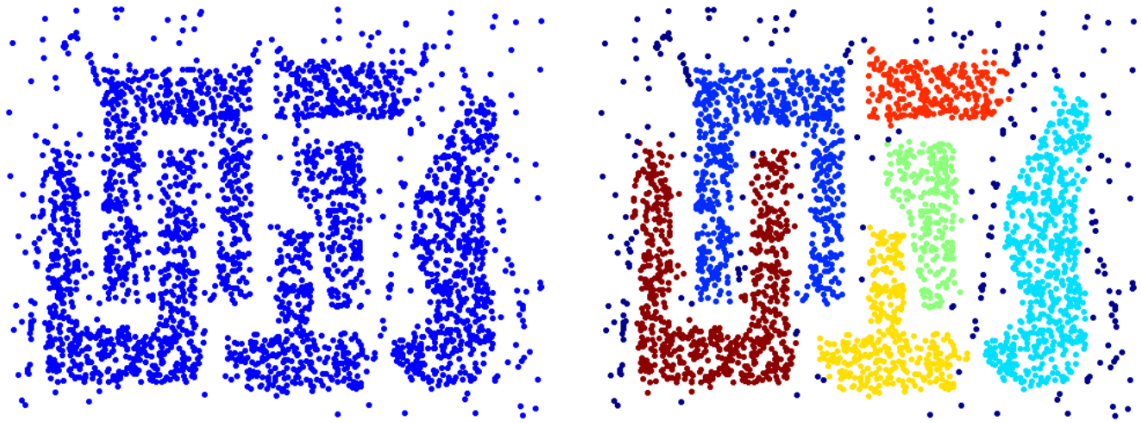


In this diagram[4], `minPts`= 4. Point A and the other red points are core points, because the area surrounding these points in an ϵ radius contain at least 4 points (including the point itself). Because they are all reachable from one another, they form a single cluster. Points B and C are not core points, but are reachable from A (via other core points) and thus belong to the cluster as well. Point N is a noise point that is neither a core point nor directly-reachable.

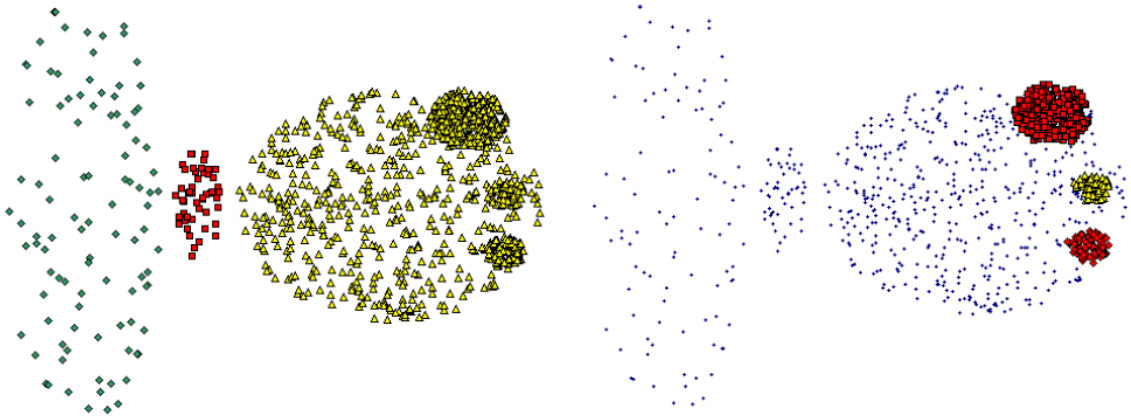
Examples and Rules of Thumb

Finally, we need to consider when DBSCAN works well, and when it doesn't (or might not) work well. DBSCAN, as we introduced it, works very well to construct odd-shaped clusters and to identify what might be noise. Here is a template example of that below[5], with the original data given to the left, and a result of DBSCAN output to the right. It seems like an intuitively good clustering.

[4]Figure and caption from https://en.wikipedia.org/wiki/DBSCAN
[5]Adapted from a talk by Prof Jing Gao of SUNY Buffalo.



So when might DBSCAN have trouble? The biggest problem with DBSCAN is that the clustering it provides can be extremely sensitive to the two parameters that the user has to provide (ϵ and `MinPts`). We'll see examples of that in the homework, but there's a nice example below[6] where the original data consists of several regions of differing size and differing density- with some very dense clusters buried inside the less dense cluster. We can see that changing the ϵ by a very small amount completely changes the nature of the clusters found. For the results on the left, the parameters were $\epsilon = 9.92$ and `MinPts=4`. On the right, same `MinPts`, but ϵ was changed to 9.75.



There are some rules of thumb for setting the parameters[7]. For example, `MinPts` should be at least twice the dimension of the data. If our data is in the plane, that would make the default value 4. It is recommended that for datasets that have a lot of noise, that are very large, that are high- dimensional, or that have many duplicates it may improve results to increase `MinPts`.

The value of ϵ is harder to estimate. Here are some considerations:

- ϵ should be as small as possible.
- Some researches say you should measure the distance to the fourth nearest neighbor for 2-dimensional data, or the distance to the k^{th} neighbor, where k is twice the dimension.
- The value might be highly problem dependent. For example, in a clustering problem using GPS, it might make physical sense to set ϵ to 1 km.

[6] Adapted from a talk by Prof Jing Gao of SUNY Buffalo.

[7] From Schubert, Sander et al in "DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN" ACM Transactions on Database Systems. 2017

And of course, there is the question of how much work should we put into the estimate, when varying these parameters may give us insight into what the point density is like in our example.

8.5 Comparisons between the algorithms

As we look back at the algorithms, each was designed to extract different information from data.

1. $k-$means:

   - The $k-$means algorithm forms Voronoi cells in the plane.

     If believe we have clusters that can be separated by the polygonal regions the the Voronoi cells produce, then $k-$means may work well.

     In our examples, $k-$means would not have worked well for something like interwoven spirals.
   - The $k-$means algorithm is fast and the output is easy to interpret (giving you templates for the data).

     We found that $k-$means did a fantastic job separating colors in the image segmentation problem, and in clustering the handwritten digits.
   - How are new points handled?

     As another point of comparison, if you're given new data points, it is easy to find which cluster the point belongs to: Take the distance to each cluster, and find the center closest to it.
   - Probably the biggest issue with using $k-$means is that the clustering you get depends on the initialization.

2. Neural Gas:

   The Neural Gas algorithm is an algorithm designed to uncover the topological structure of the data. The cluster centers here are not centers in the $k-$means sense; they do not represent a small number of template data points (like in the handwritten digits example). However, if we do want to use the cluster points as centers, we can do that.

   - The cluster points, used as cluster centers, form Voronoi cells for membership- like $k-$means.

     We have had some success in treating these as cluster centers when the data actually represents an object in lower dimensional space (something called *neural charts*[8]). In this case, we wanted to construct local coordinate systems (using a local version of the PCA), and by using Neural Gas, we were able to make sure that the origin to each coordinate system was actually embedded in the data.
   - How are new points handled? As with $k-$means, the cluster with the closest distance would give the cluster membership.
   - As with $k-$means, probably the biggest issue with Neural Gas is that the cluster points and connections change depending on the initialization (and of course with how you set the parameters).

3. DBSCAN:

   DBSCAN is an interesting algorithm in that it allows you to pick out "noisy" points. The values of the two inputs, ϵ and `MinPts` can be adjusted from anywhere between two extremes- On one extreme, every point is "noise" (ϵ too small or `MinPts` too large), or on the other extreme, the entire data set is one big cluster (ϵ too large and/or `MinPts` too small).

   - We have no cluster points per se, every point is either in a cluster (with an index provided), or is classified as noise.

[8] "Empirical Dynamical Systems Reduction II: Neural Charts", D. Hundley and M. Kirby. In *Semi-analytic methods for the Navier-Stokes Equations*, Edited by K. Coughlin, p. 65-83, 1999.

- With new data, we would either have to re-run the algorithm, or come up with some algorithm to decide which cluster it should belong to:
  - With only the original data available, we could find k closest original data points from the new point, and either use the closest point or some weighted average of the k original point classifications to determine which cluster the new point belongs to.