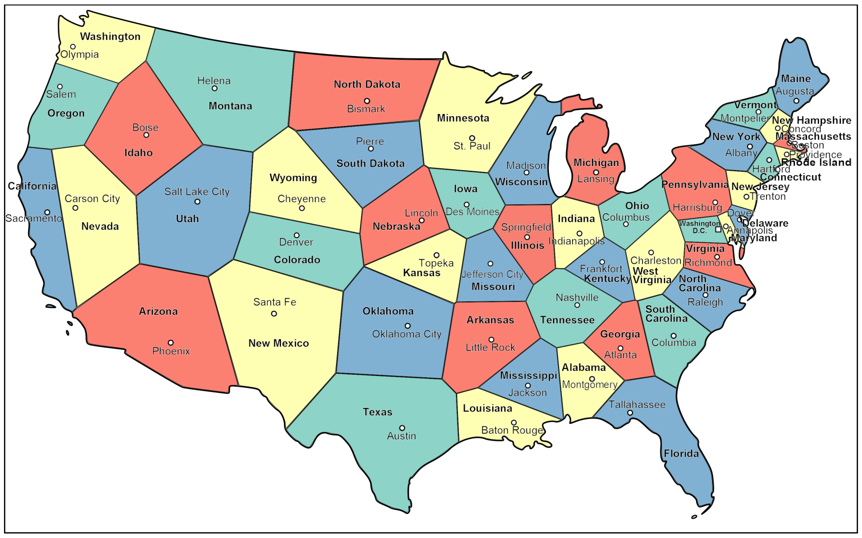
# Clustering

MATH1900: Machine Learning

Location: http://people.sc.fsu.edu/∼jburkardt/classes/ml_2019/clustering/clustering.pdf



*Using K-means to redefine the United States.*

**The Clustering Problem**

*Given data that is not labeled or organized in any way, search for natural groupings into K clusters.*

## 1 The K Means algorithm

When we are given a dataset, and we don't have any previous knowledge about how the data might be organized, we can do some simple things to try to analyze it. This includes getting statistics such as the minimum and mean, using plotting to see if we can spot patterns or structure. Sometimes, we may suspect that the data naturally has clumps or clusters. Identifying such clusters can be the start to understanding or compressing the data. A standard technique for clustering is known as the k-means algorithm.

We suppose our data is stored as an $m \times n$ numweric array, where each row contains the values of a single data item. If the data can be thought of as points in the plane, then each row would store the $x$ and $y$ coordinates of a point.

If we have $(x, y)$ data, and it does have clustering, then our eye would automatically organize a plot of the data by recognizing these clusters. We are calling the k-means algorithm to do this for us automatically, no matter how many data points, no matter how many dimensions each data point has.

The k-means algorithm is an iteration, which repeatedly updates two objects: a set of $k$ centroids, and a cluster assignment for each data item. To start the iteration, the user needs to specify the value of $k$, and initial values for the $k$ centroids, each of which should be an $n$-dimensional value like the data. After that, the updating is done automatically.

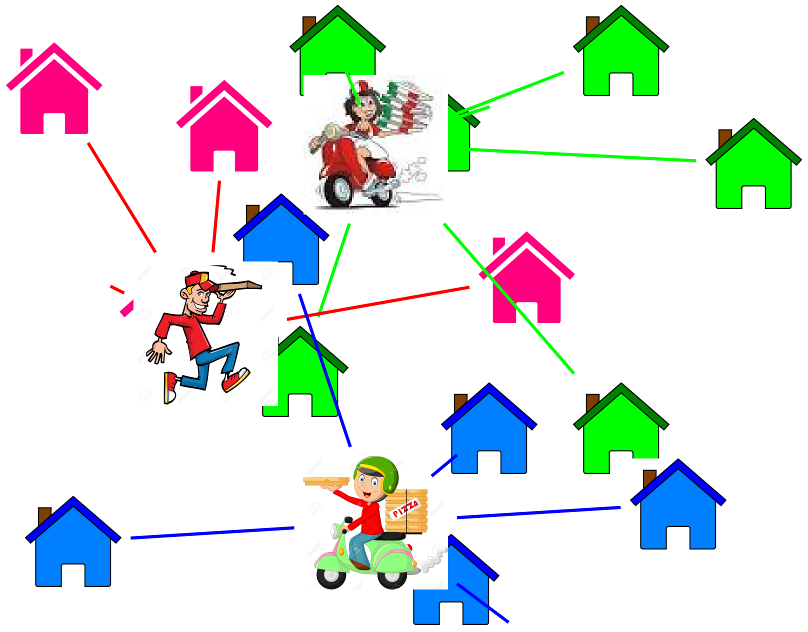
The goal of the iteration is to find a clustering that minimizes the total cost. This cost is the sum of the squared distance of each point to its cluster center. Each step of the k-means algorithm is guaranteed to reduce (or at least, never increase) this cost.

On the first step, given the user centroid values, we assign each data item to the centroid that is closest to it. On subsequent steps:

1. Replace each centroid by the average location of the cluster data;
2. Compute the cluster assignment of each data item;
3. Compute cost of this clustering;
4. If clustering cost is acceptable, or didn't decrease much, we are done;
5. If no item moved, we are done;

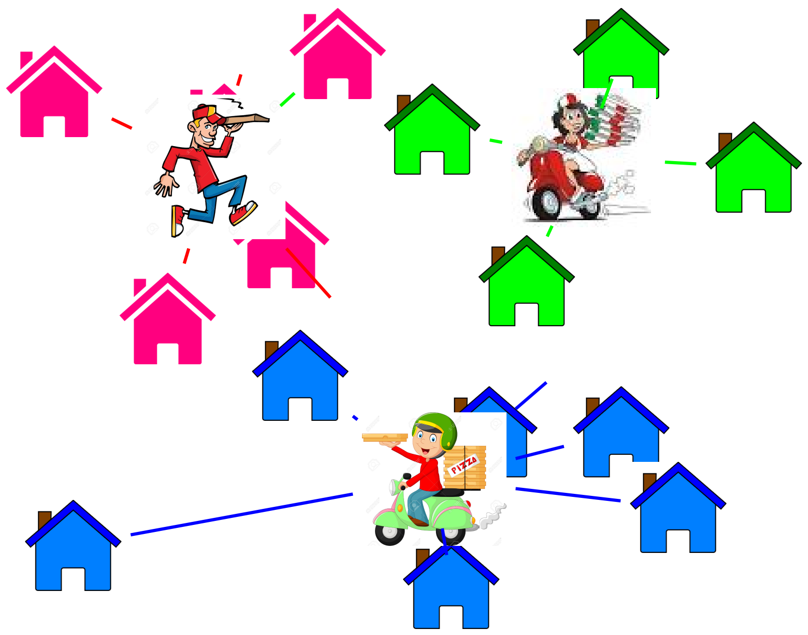
## 2 Example: The Pizza Truck Problem

The town of Grayville has three pizza trucks, which are painted red, green, and blue. By tradition, every house in Grayville has always ordered from the same truck, which then sends the delivery by scooter.



The price of gas has risen, and the owner of the pizza trucks asks a consultant if there is a way to save money.

1. Each house should be served by the nearest pizza truck. *Assign each data item to the nearest centroid)* The owner is impressed by this change, which lowers the monthly gas bill. But is that as good as we can do? It turns out that in this new system, the trucks are not well placed.
2. Each truck should be moved to the center of its service area. *(Replace each centroid by the average of its data items.)* That has got to be it, says the owner. No, because when you moved the trucks, you actually made some houses slightly closer to a different truck than the one they had been assigned.
3. Unless things settled down, go back to step #1



This simple example suggests how k-means clustering can reorganize data so that it is grouped more tightly. In this case, the regrouping simply reduces the total travel cost of the deliverers. In other cases, we expect that the grouping may reflect some meaningful fact about the data.

## 3 Example: Coding the Pizza Truck Problem

Assume that the arrays `x` and `y` contain the coordinates of each house, that `s` and `t` contain the coordinates of each truck, and that `rc`, `gc` and `bc` list the houses served by the red, green, and blue trucks respectively.

Our first improvement is to assign each house to the nearest truck. To do this, we need to compute the distance of each house to each truck, and update the assignment vectors. We can also compute the current cost.

```
rd = np.sqrt ( ( x - s[0] )**2 + ( y - t[0] )**2 )
gd = np.sqrt ( ( x - s[1] )**2 + ( y - t[1] )**2 )
bd = np.sqrt ( ( x - s[2] )**2 + ( y - t[2] )**2 )

rc = np.where ( ( rd < bd ) & ( rd < gd ) )
gc = np.where ( ( gd < bd ) & ( gd < rd ) )
bc = np.where ( ( bd < rd ) & ( bd < gd ) )

cost = sum ( bd[bc] ) + sum ( rd[rc] ) + sum ( gd[gc] )
```

Because we have reassigned some houses, it makes sense to move each truck to the center of its set of houses. We just have to average all the coordinates:

```
s[0] = np.mean ( x[rc] )
t[0] = np.mean ( y[rc] )
s[1] = np.mean ( x[gc] )
t[1] = np.mean ( y[gc] )
s[2] = np.mean ( x[bc] )
t[2] = np.mean ( y[bc] )
```

Because the trucks have moved, we need to recompute the distances and update the cost.

These two steps of reassigning houses and moving trucks are repeated until no house has to be reassigned, or the cost stops changing.

For the example problem in the illustration, here is how the cost changes:

```
0:  181.49  Initial
1:  146.29  Reassign houses
2:  117.08  Move trucks, reassign houses
3:  116.73  Move trucks, reassign houses
4:  116.73  Move trucks, reassign houses, NO CHANGE
```

You can examine a simple code for this problem in the file *pizza_kmeans.py*.

## 4 Example: The Old Faithful geyser

Timings have been made of the Old Faithful geyser, measuring the lengths of each eruption, and the pause that follows. We will work with a data file *faithful_data.txt*of $n = 272$ pairs of these measurements, hoping to find patterns.

```
  import numpy as np
  data = np.loadtxt ( 'faithful_data.txt' )
  x = data[:,0]
  y = data[:,1]
  n = len ( x )
  import matplotlib.pyplot as plt
  plt.plot ( x, y, 'b.', markersize = 15 )
  plt.show ( )
```

Listing 1: Get data and plot it

The original $x$ and $y$ data have much different scales. The plot function hides this discrepancy inside an almost square plot box. When one data axis is really 10 times larger than the other, our clustering will work much better with normalized data.

```
  xmin = np.min ( x )
  xmax = np.max ( x )
  ymin = np.min ( y )
  ymax = np.max ( y )
  x = ( x - xmin ) / ( xmax - xmin )
  s = ( y - ymin ) / ( ymax - ymin )
  plt.plot ( x, y, 'b.', markersize = 15 )
  plt.show ( )
```

Listing 2: Scale the data

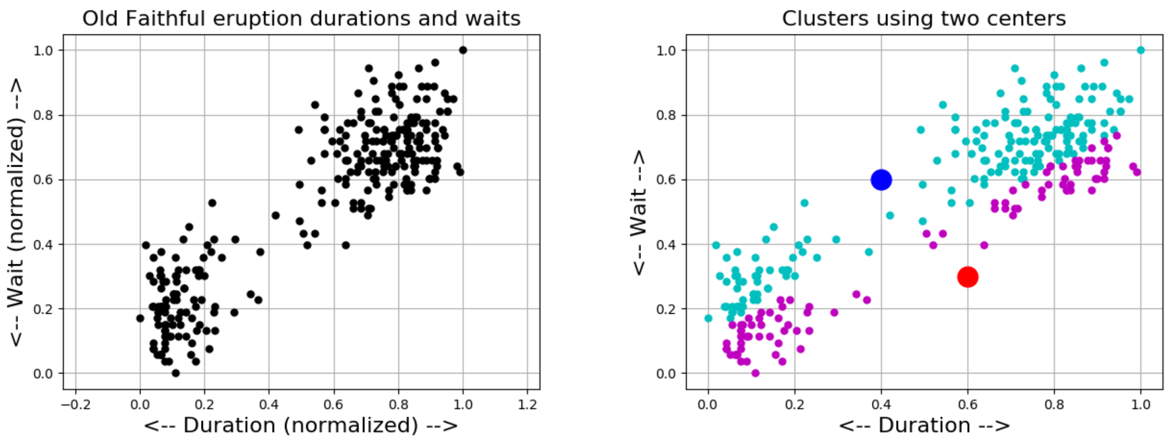
From the plot, we decide that our data naturally falls into $K = 2$ clusters. To start the k-means algorithm, we will pick two center points. They should be near the data, and not too near each other. Once we have picked the centers, we will assign each data item to the nearest center.

```
  bx = 0.4
  by = 0.6
  rx = 0.6
  ry = 0.3
  bd = np.sqrt ( ( x - bx )**2 + ( y - by )**2 )
  rd = np.sqrt ( ( x - rx )**2 + ( y - ry )**2 )
  bc = np.where ( bd < rd )
  rc = np.where ( rd <= bd )
```

Listing 3: Assign data to nearest blue or red center

```
plt.plot ( x[bc], y[bc], 'b.' )
plt.plot ( x[rc], y[rc], 'r.' )
plt.plot ( bx, by, 'bo' )
plt.plot ( rx, ry, 'ro' )
plt.show ( )
```

Listing 4: Plot our clustering



*Data, and first attempt at clustering.*

The vectors `bc` and `rc` are the indices of points that "belong" to the blue or red cluster now. We evaluate the goodness of this clustering by computing the total distance. We add the distance of every blue point to the blue center, and the distance of every red point to the red center.

```
cost = bd[bc] + rd[rc]
```

Listing 5: Evaluate cluster cost

Our first step of k-means is done.

Each subsequent step does the following:

1. Replace each centroid by the average location of the cluster data;
2. Recompute the cluster assignment of each data item;
3. Compute cost of this clustering;
4. If clustering cost is acceptable, or didn't decrease much, we are done;
5. If no item moved, we are done;

The only item we need to look at is the update of the centroids:

```
bx = np.mean ( x[bc] )
by = np.mean ( y[bc] )
rx = np.mean ( x[rc] )
ry = np.mean ( y[rc] )
```

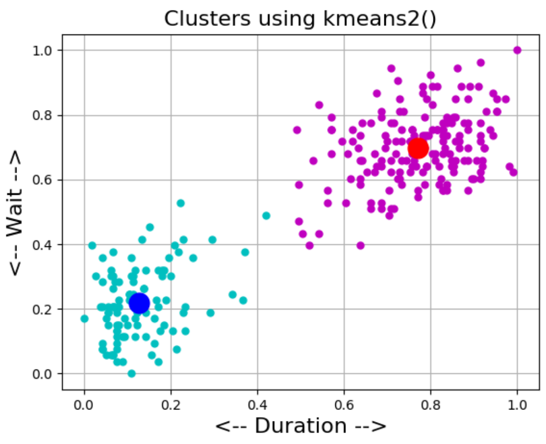
Listing 6: Updating the cluster centroids.

The k-means algorithm is an iteration, so we expect to repeat this process of updating centroids and updating cluster membership many times. However, the python library numpy provides us with a function that takes our data file and returns the centroids and cluster assignments with a single call:

```
  import numpy as np
  from scipy.cluster.vq import kmeans2
  data = loadtxt ( 'faithful_data.txt' )
# We should normalize our data here, as we did above...
  k = 2
  c, label = kmeans2 ( data, k )

  plt.plot ( data[label==0,0], data[label==0,2], 'b.' )
  plt.plot ( data[label==1,0], data[label==1,1], 'r.' )
  plt.plot ( c[0,0], c[0,1], 'bo' )
  plt.plot ( c[1,0], c[1,1], 'ro' )
  plt.show ( )
```

Here, `labels` is a vector the same length as `data`, which contains the index of the cluster assigned to each item. For our problem, it contains values of 0 or 1.



*Final clustering, courtesy of kmeans2().*

You can examine a code for this problem in the file *faithful_kmeans2.py*.

## 5 Computing Assignment #4

The file *ruspini_data.txt* contains the $(x, y)$ coordinates of 75 points, in the range $0 \leq x \leq 120, 0 \leq y \leq 160$. We want to organize the data into a small number $K$ of clusters. Make a plot of the data, and decide what a natural value for $K$ should be. This value will be more than 2, and not more than 10!
Write a Python program called *hw4.py* which

1. reads `data`, the information in *ruspini_data.txt*;
2. scales the data in column 0 and in column 1, so each column now ranges from 0 to 1;
3. prints out your choice for $k$, the number of clusters;
4. calls kmeans2() to determine a clustering;
5. plots your data using $k$ colors to display the clusters.

Email a copy of your program to Dr Schneier **mhs64@pitt.edu** before Wednesday, 9 October.