CS405 Machine Learning

Lab #2 Clustering

Pre-lab

1. Read the materials about the popular algorithms for clustering (e.g. *K-means, GMM*). Get a rough idea from Wikipedia, or detail derivations from *Pattern Recognition and Machine Learning*. Try to write the *pseudo code* of these algorithms, and summarize their differences and connections.

2. Get yourselves familiar with Scikit-learn (Python), toolbox (MatLab), and see how the results of the algorithms mentioned in Q1 using existing functions.

3. What is the limitation of *K-means*?

4. What is bias and overfitting? Why *GMM* can have good performance and strike the balance between them?

Introduction

Different from classification which asks us to classify a set of data into several specific groups. Clustering requires us to put the similar data together, hold on here, make yourselves clear, after clustering, what you get is several clusters, you don't know what the clusters represent in advance.

Let's see an example. There is a group of students, we want to divide them into subgroups, so that the members in each subgroup can be as similar as possible. Obviously, the result depends on how you define "similarity". For example, the easiest way is to divide the students by sexuality. We use (M, F) to represent a student (boy is (1, 0), girl is (0, 1)). But this feature is too simple, we don't even need to do clustering. What we usually do to use more features, like height, weight, grading, etc., and map the data to N dimension vector space before clustering.

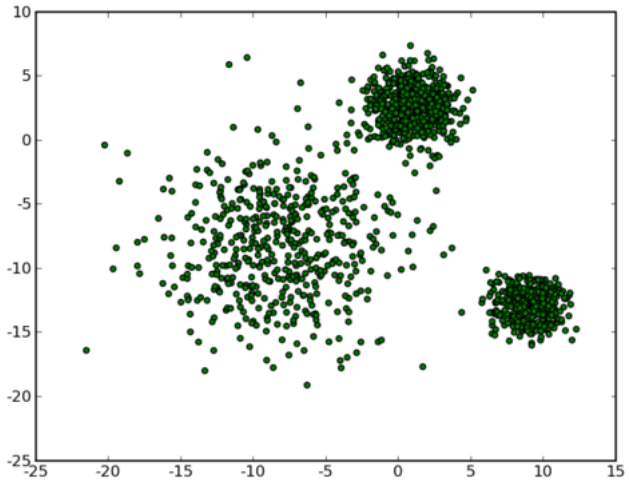

Fig.1

So now, our problem is what "similarity" is, and how to measure and qualify "similarity". To simplify the problem, assume that we already map the data set to a 2D vector space, as shown in Fig. 1. We can easily tell there are roughly 3 clusters by intuition, two are dense, one is sparse. How computer knows there are three clusters. We have the following algorithms.

K-means

The basic assumption of *K-means* is in every cluster, we can find a center, so that the distance between every point in this cluster and the center is the minimum distance compare to other centers. In math word, it can be interpreted as,
There are N data points need to be clustered into K groups, *K-means* is to minimize

$$J = \arg min \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||x_n - \mu_k||^2$$

where

$$r_{nk} = \begin{cases} 1, & point\ n\ belong\ to\ cluster\ k \\ 0, & else \end{cases}$$

And μ_k is the center of kth cluster.

It's not easy to minimize J directly, so we use an iterative refinement technique. First, we fix the center μ_k, and minimize r_{nk}. Easy to see that, if we cluster every point to their nearest center then J is the minimum. Next step is to fix n_{nk}, and minimize μ_k. Take derivative of J to μ_k, and make it zero, so that

$$\mu_k = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}}$$

which is the average of all the points in cluster k. Because for now every step we get the minimum of J, after every iteration J only can decrease to minima.

The procedure of K-means is
1. Initialize μ_k;
2. Cluster every point to their nearest center;
3. update μ_k using average $\mu_k = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}}$;
4. repeat from step 1.

Exercise 1

Implement 2-D *K-means* using Python (or MatLab).
Note: You are not allowed to use the existing function for k-means in Python (or MatLab).
Tips: if you choose to use Python, matplotlib and SciPy is needed.

K-medoids

If we see the objective minimize function of *K-means* again, we can find that K-means requires the data points in Euclidian space, and calculate the dissimilarity by Euclidian distance. For example, if the data is categorical types of data (like different kinds of dogs), there is no meaning when we calculate Euclidian distance.

In *K-medoids*, we change the Euclidian distance to a dissimilarity measure function ν

$$J = \arg min \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \nu(x_n, \mu_k)$$

A common way is to use a dissimilarity matrix D to represents ν, and D_{ij} means the difference between ith and jth cluster.

Exercise 2

Revise your code in Exercise 1 to implement *K-medoids*. Compare the results of *K-means* and *K-medoids*, and comment on the performance (from complexity, and intuition point of view).

****Exercise 3 Image Compression***

In image processing field, we always use a number to represents the value in a range. Such as, [0,1] to 0, or [1, 2] to 1. Suppose we have an 8-bit (256 levels) gray scale image whose pixel value is some real numbers from [0, 1], and the mapping function is floor(x*255). Now we want to compress the image to 4-bit. An easy way is to use x*15/255. However, if the image is not so "uniformly" distributed, then the result would be unsatisfying. For example, if the image is fully consisted of 0 and 13 (level), then the compressed image will be totally black. A practical solution is to use K-means to get k centroids, and use the value of these centroids to represent the all the points in the clusters.

Use K-means to make a simple image compression.

Gaussian Mixture Model (GMM)

GMM is another popular clustering algorithm like K-means. The difference is K-means assign each data points to a certain cluster, GMM gives the possibility of how each point assigned to every cluster, also called soft assignment.

The basic assumption of GMM is all the data follow Gaussian distribution, is other words, we want the data are generated by Gaussian Distribution. Every GMM is consisted of K Gaussian distribution, each Gaussian distribution named "component", these components are linearly combined and get the pdf of GMM,

$$p(x) = \sum_{k=1}^{K} p(k)p(x|k) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

this pdf tells us, if we want to randomly pick a point in GMM. We first randomly pick a component among K components, π_k is the probability of the kth component get chosen. After picking kth component, we just need to pick a point from this component, which is a simple Gaussian distribution.

Using GMM to do clustering is to derive $p(x)$, which is to estimate $\pi_k, \mu_k, and\ \Sigma_k$. Like the idea of k-means, we want to find a set of (π_k, μ_k, Σ_k), so that the probability can get to its maximum. To do that, we use likelihood function, and let the derivative to zero. But to avoid the arithmetic underflow of float number, we use log-likelihood here, which is

$$J = argmax \sum_{i=1}^{N} \log \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k) \right\}$$

Inspired by the process of randomly pick a point from GMM, we maximize the objective function with 2 steps and iterate.

1. estimate the possibility of each data point generated by every component. For x_i, its possibility of generated by kth component is

$$\gamma(i,k) = \frac{\pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_i|\mu_j, \Sigma_j)}$$

where μ_k, Σ_k is from last iteration (or initial value for the first time).

2. estimate the parameters of each components μ_k, Σ_k. The $\gamma(i,k)$ can be thought as the contribution kth component makes to generate data x_i. In other word, in data x_i, there is $\gamma(i,k)x_i$ generated by kth component. If we consider all the point $\gamma(1,k)x_1, \gamma(2,k)x_2, \dots, \gamma(N,k)x_N$. They also follow Gaussian distribution, so the new parameters can be calculated from

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^{N} \gamma(i,k)x_i$$

$$\Sigma_k = \frac{1}{N_k} \sum_{i=1}^{N} \gamma(i,k)(x_i - \mu_k)(x_i - \mu_k)^T$$

where $N_k = \sum_{i=1}^{N} \gamma(i,k)$

3. repeat step 1 and 2 to converge

Exercise 3

Implement a 2D GMM using Python (or MatLab).

*Spectral Clustering