

Bisecting K-Means

The bisecting K-means algorithm is a simple development of the basic K-means algorithm that depends on a simple concept such as to acquire K clusters, split the set of some points into two clusters, choose one of these clusters to split, etc., until K clusters have been produced.

The k-means algorithm produces the input parameter, k, and division a set of n objects into k clusters so that the resulting intracluster similarity is high but the intercluster analogy is low. Cluster similarity is evaluated concerning the mean value of the objects in a cluster, which can be viewed as the cluster's centroid or center of gravity.


The original values for the means are arbitrarily authorized. These can be authorized randomly or perhaps can need the values from the first k input items themselves. The convergence component can be based on the squared error, but they are needed not to be. For instance, the algorithm is assigned to multiple clusters. Other termination methods have been locked at a fixed number of iterations. A maximum number of iterations can be involved to provide shopping even without convergence.

The Algorithm of bisecting K-Means which are as follows –

- Initialize the list of clusters to include the cluster such as all points.
- repeat
- Remove a cluster from the list of clusters.
- {Implement multiple "trial" bisections of the selected cluster.}
- for i : 1 to number of trials do
- Bisect the choose cluster utilizing basic K-means.
- end for
- Choose the two clusters from the bisection with the smallest total SSE.
- Insert these two clusters to the document of clusters.
- until the document of clusters includes K clusters.

R 코드예시:

Implementation-of-Bisecting-K-Means-and-K-Nearest-Neighbors/BKmeans_KNN.R at master · jagadeesh-h/Implementation-of-Bisecting-K-Means-and-K-Nearest-Neighbors · GitHub
Contribute to jagadeesh-h/Implementation-of-Bisecting-K-Means-and-K-Nearest-Neighbors development by creating an account on GitHub.

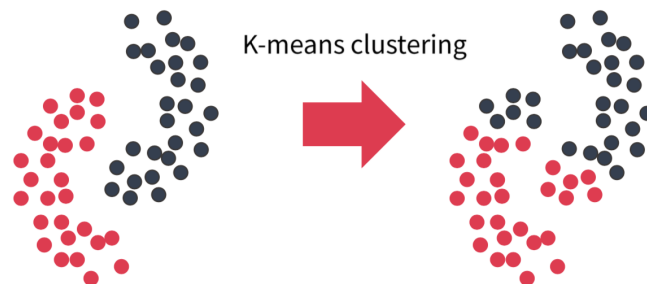
 https://github.com/jagadeesh-h/Implementation-of-Bisecting-K-Means-and-K-Nearest-Neighbors/blob/master/BKmeans_KNN.R

K-medoid Clustering

- Elbow method와 Silhouette method도 k를 찾을 수 있는 하나의 방법인 것이지 찾은 k가 절대적으로 최적이라는 것은 아니다. 그래서 군집의 갯수를 찾는 것이 상당히 어려운 문제이다. k-means clustering은 아래와 같은 단점이 존재한다.

I K-medoids clustering

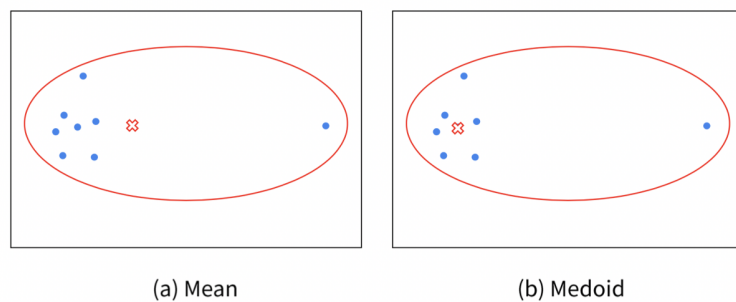
- K-means clustering의 단점
 - 초기 중심 값에 민감한 반응을 보임
 - 노이즈와 아웃라이어에 민감함
 - 군집의 개수 K를 설정하는 것에 어려움



- 위에서 언급한 k-means의 단점을 보완하기 위한 방법으로 가장 간단히 평균대신 중간점을 사용 하는 방법이다.

I K-medoids clustering

- K-medoids clustering
 - K-means clustering의 변형으로, 군집의 무게 중심을 구하기 위해 데이터의 평균 대신 중간점(medoids)을 사용 (K-means보다 이상치에 강건한 성능을 보임)
 - 아래 그림의 결과를 보면 K-medoids의 중앙점이 더 명확함 (이는 더 좋은 군집을 형성하게 될 가능성을 높임)



- k-medoid clustering도 다른 clustering과 동일하게 찾은 k가 절대적으로 최적인 k를 의미하진 않는다. 예를 들어 아래 그림에서와 같이 초승달 모양의 데이터 군집 분포는 거리에 기반한 모델인 k-means와 k-medoid clustering은 명확히 2군집으로 분류하기 어렵다.

k-means와 k-medoid 비교

I K-medoids clustering

- K-means vs K-medoids

	K-means	K-medoids
중심	군집의 평균 값	군집 내 중앙 데이터
이상치	이상치가 전체 거리 평균 값에 영향을 주어 이상치에 민감함	K-means보다 덜 민감함
계산 시간	상대적으로 적은 시간이 소요	데이터 간 모든 거리 비용을 반복하여 계산해야 하므로 상대적으로 많은 시간이 소요
파라미터	군집의 개수 k, 초기 중심점	
군집 모양	원형의 군집이 아닌 경우 군집화를 이루기 어려움(아래 그림 참조)	



예시:

```
from KMedoids import KMedoids
import matplotlib.pyplot as plt

def plot_graphs(data, k_medoids):
    colors = {0:'b*', 1:'g^', 2:'ro', 3:'c*', 4:'m^', 5:'yo', 6:'ko', 7:'w*'}
    index = 0
    for key in k_medoids.clusters.keys():
        temp_data = k_medoids.clusters[key]
        x = [data[i][0] for i in temp_data]
        y = [data[i][1] for i in temp_data]
        plt.plot(x, y, colors[index])
        index += 1
    plt.title('Cluster formations')
    plt.show()

    medoid_data_points = []
    for m in k_medoids.medoids:
        medoid_data_points.append(data[m])
    x_ = [i[0] for i in data]
    y_ = [i[1] for i in data]
    x_ = [i[0] for i in medoid_data_points]
    y_ = [i[1] for i in medoid_data_points]
    plt.plot(x, y, 'yo')
    plt.plot(x_, y_, 'r*')
    plt.title('Medoids are highlighted in red')
    plt.show()

#Example 1
data_ = [
    [5.2, 3.7], [5.1, 4.1], [5.2, 4.2], [5, 3.7], [5.3, 4], [5.5, 3.8], [5.4], [4.7, 3.7], [4.8, 4], [4.7, 3.3], [5.2, 3.3], [4.9, 3.4], [
    2.6, 1.5], [2.5, 1.2], [2.7, 1], [2.4, 1.7], [2.6, 1.5], [2.7, 1.8], [2.1, 1.1], [2.2, 1.4], [2.3, 0.8], [2.3, 1.3], [2.5, 2]
    ], [3, 3.5], [3.5, 3.8], [3.9, 3.2], [4.5, 1.9], [4.3, 1.4], [4, 2]]
n_clusters = range(2, 8)
k_medoids = [KMedoids(n_cluster=i) for i in n_clusters]
k_medoids = [k_medoid.fit(data_) for k_medoid in k_medoids]
loss = [k_medoid.calculate_distance_of_clusters() for k_medoid in k_medoids]

# Plot elbow curve (to know best cluster count)
plt.figure(figsize=(13, 8))
plt.plot(n_clusters, loss)
plt.xticks(n_clusters)
plt.xlabel('Number of Clusters')
plt.ylabel('Loss')
plt.title('Loss Vs No. Of clusters')
plt.show()

#Modeling
k_medoids = KMedoids(n_cluster=2)
k_medoids.fit(data_)
plot_graphs(data_, k_medoids);
```

