<2T>

# K-means Description

**k-means clustering** (aka kmeans, k means) is a method of vector quantization, originally from signal processing, that is popular for **cluster analysis**in data mining. k-means clustering aims to **partition** n observations into k clusters in which each observation belongs to the **cluster** with the nearest **mean**, serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells.

The problem is computationally difficult (NP-hard); however, there are efficient **heuristic algorithms** that are commonly employed and converge quickly to a local optimum. These are usually similar to the expectation-maximization algorithm for mixtures of Gaussian distributions via an iterative refinement approach employed by both k-means and Gaussian Mixture Modeling. Additionally, they both use cluster centers to model the data; however, k-means clustering tends to find clusters of comparable spatial extent, while the expectation-maximization mechanism allows clusters to have different shapes.

The algorithm has a loose relationship to the k-nearest neighbor classifier, a popular machine learning technique for classification that is often confused with k-means because of the k in the name. One can apply the 1-nearest neighbor classifier on the cluster centers obtained by k-means to classify new data into the existing clusters. This is known as nearest centroid classifier or Rocchio algorithm.

# K-means Pseudocode

//k-means, k means, kmeans algorithm

**function** k-means(*data, k*)

// initialize centroids randomly

**var** *int*num\_observations = data.size()

**var** *float* centroids[num\_observations]

get\_rand\_centroids(centroids, k)

// initialize book keeping vars

**var** *int* iterations

**var** *float* old\_centroids[num\_observations]

//k-means loop

**while** (centroids != old\_centroids AND iterations < MAX\_ITERATIONS)

old\_centroids = centroids

iterations += 1

**var** *float* labels[num\_observations]

**for**(i=0, i<num\_obersvations, i+=1)

labels[i] *=* nearest\_centroid(data[i],centroids)

centroids = update\_centroids(data, labels, k)

**return** centroids

# K-means Code

import numpy as np

# k-means, k means, kmeans algorithm

# Returns an array of centers chosen at random from data

def get\_rand\_centers(data, num\_centers):

    used = set()

    centers = np.empty(num\_centers)

    for i in range(0, num\_centers):

        t = random.randint(0, len(data)-1)

        while t in used:

            t = random.randint(0, len(data)-1)

        centers[i] = data[t]

        used.add(t)

    return centers

# Returns the 'distance' between two values

def dist(x, y):

    return (x-y)\*\*2

# Returns an updated centers array

def update\_centers(centers, clusters, data):

    num\_centers = len(centers)

    data\_size = len(data)

    temp = np.zeros(num\_centers)

    for i in range(0, data\_size):

        temp[int(clusters[i])] += data[i]

    uniq, counts = np.unique(clusters, return\_counts=True)

    centers = np.true\_divide(temp,counts)

    return centers

# Returns an array with the cluster each data point belongs to

# Checks for empty clusters and reassigned at random from data

def classify(centers, data):

    data\_size = len(data)

    cluster\_num = len(centers)

    clusters = np.empty(data\_size)

    temp = np.empty(cluster\_num)

    for i in range(0, data\_size):

        for j in range(0, cluster\_num):

            temp[j] = dist(centers[j],data[i])

        clusters[i] = np.argmin(temp)

    uniq, counts = np.unique(clusters, return\_counts=True)

    while(len(uniq) != len(centers)):

        reassign = set(range(0, len(centers))) - set(uniq)

        for i in reassign:

            centers[i] = data[random.randint(0, len(data)-1)]

        clusters = classify(centers, data)

        uniq, counts = np.unique(clusters, return\_counts=True)

    return clusters

# The k-means algorithm

# Returns an array of centers, and an array of associated variances

def k\_means(data, num\_centers):

    centers = get\_rand\_centers(data, num\_centers)

    old\_centers = None

    iterations = 0

    while(not np.array\_equal(old\_centers, centers)):

        old\_centers = np.copy(centers)

        iterations += 1

        clusters = classify(centers, data)

        centers = update\_centers(centers, clusters, data)

    return centers