데이터 사이언스 Assignment 3

2013010982 김영호

1. 알고리즘 – DBScan

(1). 데이터들을 스캔하면서, 각 오브젝트들에 대해

\* 오브젝트의 클러스터가 정해졌을 경우, skip

(2). 오브젝타의 클러스터가 졍해지지 않았을 경우, 이웃들 (오브젝트와의 거리가 eps보다 작은 것을 지칭)을 구함

\*이웃들의 수가 minPts보다 작은 경우, 오브젝트의 클러스터를 outlier로 설정하고 skip

(3). 이웃들의 수가 minPts보다 클 경우, 새로운 클러스터를 만들고 오브젝트를 클러스터에 포함

(4). 이웃들을 스캔하면서 각 이웃에 대해,

\* 이웃이 아웃라이어일 경우, 이웃의 클러스터를 새 클러스터에 포함

\* 이웃의 클러스터가 정해졌을 경우 skip

(5). 이웃의 이웃들의 수가 minPts보다 크면, 이웃의 이웃들을 현재의 이웃들에 추가

(6) 더 이상 스캔할 이웃이 없을 때 까지 4~5 과정을 반복한다.

2. 코드 설명

int n, eps, minPts; // n : 권장하는 클러스터 수, eps : 입실론, minPts : core가 되기 위한 가장 작은 이웃 수

int minClusterIndex; // 가장 작은 클러스터를 가리키는 수

typedef struct object {

int id; // object의 id

int type; // -1 = outlier, 0 = not defined, else = 클러스터의 id

double x\_coor; // x좌표

double y\_coor; // y좌표

}object;

std::vector<std::vector<int> > clusters; // 클러스터의 집합

std::vector<object> objects; // 오브젝트의 집합

bool areNeighbor(int i, int j) // i와 j가 이웃하는지 구하는 함수

std::vector<int> getNeighbos(int pts) // pts의 이웃들을 구하는 함수

void unionNeighbors(std::vector<int>\* originalNeighbor, std::vector<int> newNeighbor) // 기존 이웃에 새로운 이웃을 추가해주는 함수, LIFO 형식

void setMinCluster() // minClusterIndex를 지정해주는 함수

void dbScan() {

int clusterNumber = 0;

for (int i = 0; i < objects.size(); i++) { // 각 오브젝트에 대해

if (objects[i].type != 0) { // 타입이 정해졌을 경우 스킵(outlier포함)

continue;

}

std::vector<int> neighbors = getNeighbos(i); // 이웃을 구함

if (neighbors.size() + 1 < minPts) { //자신을 포함한 이웃들의 수가 minPts보다 작을 경우

objects[i].type = -1; // 아웃라이어

continue;

}

clusterNumber++; // 새로운 클러스터 id

objects[i].type = clusterNumber; // 새 클러스터에 포함

int it = -1; // 이웃vector의 index

while (it + 1 < neighbors.size()) { // 더 이상 탐색할 이웃이 없을 때 까지

it++; // 이웃들에 대해

if (objects[neighbors[it]].type == -1) { // 아웃라이어일 경우

objects[neighbors[it]].type = clusterNumber; // 클러스터에 포함

}

if (objects[neighbors[it]].type != 0) { // 타입이 정해졌을 경우

continue; // 스킵

}

objects[neighbors[it]].type = clusterNumber; // 클러스터에 포함

std::vector<int> newNeighbors = getNeighbos(neighbors[it]); // 이웃들을 구함

if (newNeighbors.size() + 1 >= minPts) { // 이웃을 포함한 이웃의 이웃들의 수가 minPts보다 크거나 같을 경우

unionNeighbors(&neighbors, newNeighbors); // 이웃의 이웃들을 이웃들에 더함

}

}

printf("input size : %lu\n", neighbors.size());

if (clusters.size() < n) { // 클러스터의 수가 n보다 작을 경우

clusters.push\_back(neighbors); // 새로운 클러스터 생성

setMinCluster(); // 가장 작은 클러스터를 지정

}

else if (clusters.size() >= n && neighbors.size() > clusters[minClusterIndex].size()) { // 클러스터의 수가 n보다 크거나 같고, 이웃들이 가장 작은 클러스터보다 클 경우

clusters[minClusterIndex] = neighbors; // 기존 가장 작은 클러스터를 지우고 새로운 클러스터 생성

setMinCluster(); // 가장 작은 클러스터를 지정

}

}

}

void printAllCluster() // 모든 클러스터들을 프린트해주는 함수