Laboratorio 06 EDA

Pablo Luis Carazas Barrios

November 2023

1 Introduction

A la hora de organizar datos por similitud existen diferentes técnicas cada una con su ventaja y desventaja respectiva, siendo así cada una conveniente dependiendo de el objetivo el el conjunto que se tiene.

2 Optics

Optics es una variación del algoritmo DBScan, teniendo como logica el iniciar desde un punto y buscar sus puntos más cercanos para luego volver a iniciar una cuenta desde el mismo cluster así, cumpliendo como requisito ser un grupo con un minimo de integrantes y una distancia para contabilizar el rango.

3 Código

El codigo cuenta con 4 esrtucturas, un punto que contendra todas las dimensiones, un knode que va a servir para construir el kdtree, el Kdtree y la estructura Optics que se encargará de todo el procesamiento y clusterizado.

3.1 Point

Se encarga de guardar todas las dimensiones, al grupo cluster que pertenece y las distancias a los puntos más cercanos.

```
struct Point {
    vector<double> Dimensions;
    int cluster = -1;
    double rDis = NULL;
    Point(vector<double> v) {
        Dimensions = v;

    }
};
```

3.2 Knode

Se encarga de poder almacenar los puntos en un struct con la logica de un nodo para poder realizar las consultas de manera más rapida.

```
struct knode {
    Point* Punto;
    knode* children[2] = { nullptr };
    knode(Point* p) { Punto = p; }
};
```

3.3 KDtree

Esta estructura fue programada por mi y basicamente es un KDtree con las funciones basicas:

- **Set Points**: Construye un arbol con un vector de puntos.
- **Region Query**: Basicamente es un range query segun un punto dado.
- **KNearestNeighbors**: devuelve un vector de pares de los K puntos mas cercanos, de manera ordenada y con la distancia al punto dado.

```
struct kdtree {
    knode* root = nullptr;
    void Insert(Point* k) {
        if (root==nullptr) {
            root = new knode(k);
        }
        else {
            int eje = 0;
            knode* p = root;
            int tam = k->Dimensions.size();
            while (p->children[k->Dimensions[eje] >
            p->Punto->Dimensions[eje]] != nullptr) {
                p = p->children[k->Dimensions[eje] > p->Punto->Dimensions[eje]];
                eje = (eje + 1) % tam;
            }
            p->children[k->Dimensions[eje] > p->Punto->Dimensions[eje]] = new knode(k);
        }
    }

    void setPoints(vector<Point*> v) {
        for (auto& it : v) {
            Insert(it);
        }
    }
    void RegionQuery(Point* p, double R, knode* node,
    vector<Point*>& result, int depth = 0) {
        if (!node) {
            return;
        }

        int tam = p->Dimensions.size();
        int axis = depth % tam;


        double distance = 0;
        for (int i = 0; i < tam; ++i) {
            distance += pow(node->Punto->Dimensions[i] - p->Dimensions[i], 2);
        }
        distance = sqrt(distance);


        if (distance <= R) {
            result.push_back(node->Punto);
        }


        if (p->Dimensions[axis] - R <= node->Punto->Dimensions[axis]) {
```

```
        RegionQuery(p, R, node->children[0], result, depth + 1);
    }
    if (p->Dimensions[axis] + R >= node->Punto->Dimensions[axis]) {
        RegionQuery(p, R, node->children[1], result, depth + 1);
    }
}



vector<pair<double, Point*>> KNearestNeighbors(Point* target, int K) {
    priority_queue<pair<double, Point*>> pq;

    KNNHelper(root, target, K, pq, 0);


    vector<pair<double, Point*>> result;
    while (!pq.empty()) {
        result.push_back(pq.top());
        pq.pop();
    }

    reverse(result.begin(), result.end());

    return result;
}


void KNNHelper(knode* node, Point* target, int K,
priority_queue<pair<double, Point*>>& pq, int depth = 0) {
    if (!node) {
        return;
    }

    int tam = target->Dimensions.size();
    int axis = depth % tam;

    double distance = 0;
    for (int i = 0; i < tam; ++i) {
        distance += pow(node->Punto->Dimensions[i] - target->Dimensions[i], 2);
    }
    distance = sqrt(distance);

    pq.push({ distance, node->Punto });

    if (pq.size() > K) {
        pq.pop();
    }

    if (target->Dimensions[axis] < node->Punto->Dimensions[axis]) {
        KNNHelper(node->children[0], target, K, pq, depth + 1);
    }
    else {
        KNNHelper(node->children[1], target, K, pq, depth + 1);
    }

    if (abs(target->Dimensions[axis] - node->Punto->Dimensions[axis])
```

```
        < pq.top().first || pq.size() < K) {
            if (target->Dimensions[axis] >= node->Punto->Dimensions[axis]) {
                KNNHelper(node->children[0], target, K, pq, depth + 1);
            }
            else {
                KNNHelper(node->children[1], target, K, pq, depth + 1);
            }
        }
    }


};
```

3.4 Struct Optics

Esta estructura se encarga de crear el arbol y realizar el algoritmo de clusterizado, Tiene como funciones las mismas funciones y logica que nos da el pseudocodigo en la tarea.

- **Optics**: se encarga de comenzar a procesar por cercania a los puntps.
- **Update**: se encarga de actualizar la cola.
- **Opticcluster**: al igual que el pseudocodigo, se encarga de etiquetar los puntos a cada cluster.

```
struct Optics {

    kdtree* kd;
    vector<Point*> Points;
    void setp(vector<Point*> v) {
        Points = v;
    }

    vector<Point*> optics(double epsMax, int minPoints) {
        vector<Point*> ordering;
        kd = new kdtree;
        kd->setPoints(Points);
        int i = 0;

        for (auto& p : Points) {
            //cout <<"I=" << i++ << endl;
            if (findd(ordering,p)) {
                continue;
            }
            ordering.push_back(p);
            vector<Point*> neighbors;
            //neighbors.push_back(p); //not sure
            kd->RegionQuery(p, epsMax, kd->root, neighbors);
            if (neighbors.size() >= minPoints) {
                queue<Point*> toProcess;
                update(p, toProcess, ordering, epsMax, minPoints);
                int j = 0;
                while (toProcess.size() != 0) {
                    //cout <<"TAM " << toProcess.size() << endl;
                    Point* q = toProcess.front();
                    toProcess.pop();
                    update(q, toProcess, ordering, epsMax, minPoints);
                }
```

```
        }
    }
    return ordering;
}


double Distance(Point a, Point b) {
    double aux = 0;
    for (int i = 0; i < a.Dimensions.size(); i++) {
        aux = aux + pow(a.Dimensions[i] - b.Dimensions[i], 2);
    }
    return sqrt(aux);
}
void update(Point* p, queue<Point*>& priority, vector<Point*>& ordering,
double eps, int minP) {
    vector<Point*> neighbors;
    kd->RegionQuery(p, eps, kd->root, neighbors);
    //cout << "A" << endl;
    if (neighbors.size() < minP) {
        return;
    }

    for (auto*& q : neighbors) {
        if (findd(ordering,q )) {
            continue;
        }
        else {
            auto aux = kd->KNearestNeighbors(p, minP);
            double nRD = max(aux[aux.size() - 1].first, Distance(*p, *q));

            if (q->rDis == NULL) {
                q->rDis = nRD;
                priority.push(q);
            }
            else if (nRD < q->rDis) {
                q->rDis = nRD;
            }
        }
    }
}


void opticcluster(vector<Point*> ordering, double eps) {
    int current = 0;
    bool increment = false;
    for (int i = 0; i < ordering.size(); i++) {
        if (ordering[i]->rDis == NULL or ordering[i]->rDis > eps) {
            increment = true;
        }
        else {
            if (increment) {
                current++;
                ordering[i - 1]->cluster = current;
            }
            ordering[i]->cluster = current;
            increment = false;
```

```
            }
        }
    }

};
```

3.5 Graficos

Para graficar se utilizo la libreria grafica `SFML` y con funciones auxiliares se llego a graficar.

```
CircleShape createCircle(Point* p, Color c, const Vector2u& windowSize, double minX, dou
    CircleShape cir(tam);
    double scaledX = ((p->Dimensions[0] - minX) / (maxX - minX));
    double scaledY = 1.0 - ((p->Dimensions[1] - minY) / (maxY - minY));
    float x = 50 + (scaledX * (windowSize.x - 100));
    float y = 50 + (scaledY * (windowSize.y - 100));
    cir.setPosition(x, y);
    cir.setOrigin(tam / 2, tam / 2);
    cir.setFillColor(c);
    if (cen == false) {
        cir.setOutlineColor(c);
    }
    else {
        cir.setOutlineColor(Color::Black);
    }
    cir.setOutlineThickness(1);
    return cir;
}
```

4 Resultados

Se usó un dataset llamado `ABALON`, del dataset completo no se utilizó la primera columna ni la ultima, se dio un **Epsilon** de 0.05 y un **MinPoints** de 4 y se hicieron pruebas con un Kdtree y con fuerza bruta.

Kdtree	Fuerza Bruta
63.48605	250.761396

Table 1: Comparacion de Tiempo en segundos

Con esto se puede determinar que el utilizar una estructura de Datos para realizar consultas de rango en puntos multidimensionales puede reducir el tiempo de procesado en 4 veces.

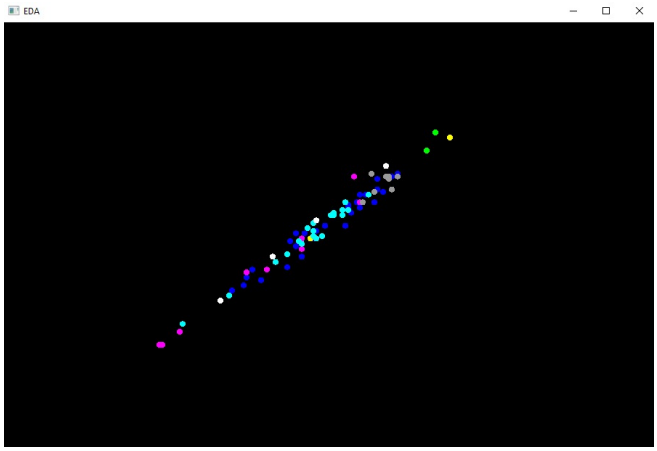

Figure 1: Grafico de los puntos de las 2 priemeras dimensiones

La rara coloracion de el grafico se debe a que ser 7 dimensiones espaciales pero solo se grafican 2 no se puede apreciar y puede que hayan puntos que parezcan lejanos en las 2 primeras dimensiones pero en otras dimensiones sean bastante cercanos.

5 Conclusiones

El algoritmo `Optics` se muestra como una versión mas optica cuando se quieren tratar conjuntos de datos de densidad variable, así teniendo la capacidad de tratar con `outliers`, a nivel de costo computacional puede ser significativamente mayor ya que trata de hacer 2 consultas de distancia por cada punto, pero el utilizar una estructura de Datos como el Kdtree puede mostrar un ahorro de tiempo de hasta 4 veces (segun esta implementacion y parametros dados), lo que singifica una gran mejora sin sacrificar el clusterizado

6 Enlaces

Github con Codigo e informe