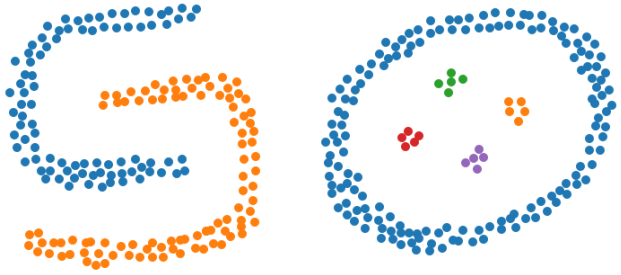
Implementing the DBSCAN clustering algorithm

Gagarine Yaikhom

In this note, we implement[1] the DBSCAN clustering algorithm. DBSCAN stands for *Density-based spatial clustering of applications with noise*. This implementation is based on the original paper by Martin Ester, Hans-Peter Kriegel, Jörg Sander and Xiaowei Xu, *A density-based algorithm for discovering clusters in large spatial databases with noise* [Proceedings of KDD-96, AAAI Press, 1996]. It will help to learn the concepts defined in this paper before studying this implementation.

The DBSCAN clustering algorithm is one of the few clustering algorithms that allows us to find clusters within clusters, like the ones shown below.



The algorithm essentially works like the flow of liquids on a terrain. It starts at a point on the terrain and flows over the terrain where there is least resistance. The resulting cluster is the area covered by the liquid. Now imagine different liquids with varying densities which cover the terrain in layers.

The first input supplied to the DBSCAN clustering algorithm is the set of data points that we wish to cluster. These can be anything, as long as we can define a distance function between the points, which is the second input. For instance, if we wish to cluster points on the two-dimensional plane using proximity as the clustering property, we use a distance function such as the Euclidean distance. On the other hand, if we wish to cluster pixels of the same intensity on an image, we use an intensity distance function. The third input configures the sensitivity of the clustering algorithm. This affects the manner in which the algorithm decides if two data points are quite similar or too different. Finally, we specify another sensitivity factor that determines whether a new cluster should start at a given data point, depending on the density of nearby data points. These will become clearer in the following sections.

Let us begin with the set of data points to cluster, denoted as D. In this implementation, we will maintain all of the supplied data points as an array. The algorithm will manipulate these data points by using the corresponding array indices. We shall parse the input supplied to the algorithm, and store the data points using the `struct point_s` data structure.

[1]Copyright 2015 Gagarine Yaikhom (MIT License). Implemented on the 3rd of August 2015 and published on the 21st of October 2015. Download the source code at: https://github.com/gyaikhom/dbscan.

In this implementation, we are using three-dimensional data points; however, we can also use any n-dimensional data points as long as a distance function can be defined between any two points in D. When the algorithms terminates successfully, `cluster_id` stores the cluster to which the data point belongs.

⟨*Data structures*⟩≡
```
typedef struct point_s point_t;
struct point_s {
    double x, y, z;
    int cluster_id;
};
```

What are the possible values of `cluster_id`? At the beginning before running the algorithm, the `cluster_id` value for all of the data points are initialised to `UNCLASSIFIED`. This essentially means that the data point needs to be looked at as it currently does not belong to any cluster. When a data point is completely isolated and could not form a cluster, it is treated as *noise*. These data point are assigned a `cluster_id` that equals `NOISE`. Any non-negative value identifies the cluster to which the data point belongs.

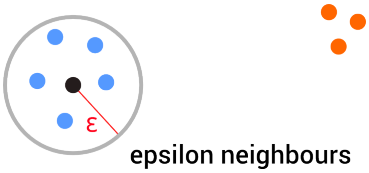
⟨*Constant definitions*⟩≡
```
#define UNCLASSIFIED -1
#define NOISE -2
```

If an `UNCLASSIFIED` data point starts a new cluster, it is considered to be a *core* data point. We shall use the following values to decide whether to start a new cluster with the data point currently under investigation, or to continue by adding the data point to the current cluster.

⟨*Constant definitions*⟩+≡
```
#define CORE_POINT 1
#define NOT_CORE_POINT 0
```

We use the following when a function should return success or failure status. The value of `FAILURE` must not coincide with the previous return values.

⟨*Constant definitions*⟩+≡
```
#define SUCCESS 0
#define FAILURE -3
```



One of the fundamental aspects of the DBSCAN algorithm is the process of finding a given data point's *epsilon neighbours*. Essentially, the epsilon neighbours of a data point p is the set of data points $S \subset D$ such that $\delta(p,q) \leq \epsilon$ for $q \in S$ and $q \neq p$. Here, δ is the distance function, and ϵ is the distance sensitivity of the algorithm, as discussed in the introduction. The figure above shows example epsilon neighbours for two-dimensional data points using the Euclidean distance.

The DBSCAN algorithm processes members of the epsilon neighbours sequentially. Hence, we will use a *single linked list* data structure to store the epsilon neighbours of a data point. As shown below, each node in the single linked list points to a data point. The `index` field stores the index of the data point in the data points array.

⟨*Data structures*⟩+≡
```
typedef struct node_s node_t;
struct node_s {
    unsigned int index;
    node_t *next;
};
```

We use the following function to create a new linked list node that points to the supplied data point.

⟨*Function definitions*⟩≡
```
node_t *create_node(unsigned int index)
{
    node_t *n = (node_t *) calloc(1, sizeof(node_t));
    if (n == NULL)
        perror("Failed to allocate node.");
    else {
        n->index = index;
        n->next = NULL;
    }
    return n;
}
```

To store the epsilon neighbours of a data point, we use the `struct epsilon_neighbours_s` data structure. This encapsulates the single linked list of nodes, and augments it with the number of nodes in the linked list. The latter is used to check if the data point against which this epsilon neighbourhood was determined is a core data point. The `head` points to the start of the single linked list; the `tail` points to the end. We use the `tail` to append nodes at the end of the single linked list.

⟨*Data structures*⟩+≡
```
typedef struct epsilon_neighbours_s epsilon_neighbours_t;
struct epsilon_neighbours_s {
    unsigned int num_members;
    node_t *head, *tail;
};
```

We use the following to append a node that points to the data point at `index`, at the end of the linked list encapsulated by the epsilon neighbours instance `en`.

⟨*Function definitions*⟩+≡
```
int append_at_end(
     unsigned int index,
     epsilon_neighbours_t *en)
{
    node_t *n = create_node(index);
    if (n == NULL) {
        free(en);
        return FAILURE;
    }
    if (en->head == NULL) {
        en->head = n;
        en->tail = n;
    } else {
        en->tail->next = n;
        en->tail = n;
    }
    ++(en->num_members);
    return SUCCESS;
}
```

We now define a function that returns the epsilon neighbours of a data point. As discussed in the previous sections, epsilon neighbours are data points that are no further than *epsilon* from the supplied data point. The distance is determined by the supplied distance function. The input data that are supplied to the `get_epsilon_neighbours` function are the `index` of the data point that we wish to find the neighbours for, the array of data points `points`, the number of data points `num_points` in the array, the `epsilon` value to use as proximity cut-off and the distance function `dist`.

⟨*Function definitions*⟩+≡

```
epsilon_neighbours_t *get_epsilon_neighbours(
    unsigned int index,
    point_t *points,
    unsigned int num_points,
    double epsilon,
    double (*dist)(point_t *a, point_t *b))
{
    epsilon_neighbours_t *en = (epsilon_neighbours_t *)
        calloc(1, sizeof(epsilon_neighbours_t));
    if (en == NULL) {
        perror("Failed to allocate epsilon neighbours.");
        return en;
    }
    for (int i = 0; i < num_points; ++i) {
        if (i == index)
            continue;
        if (dist(&points[index], &points[i]) > epsilon)
            continue;
        else {
            if (append_at_end(i, en) == FAILURE) {
                destroy_epsilon_neighbours(en);
                en = NULL;
                break;
            }
        }
    }
    return en;
}
```

The following prints the epsilon neighbours to `stdout`.

⟨*Function definitions*⟩+≡

```
void print_epsilon_neighbours(
    point_t *points,
    epsilon_neighbours_t *en)
{
    if (en) {
        node_t *h = en->head;
        while (h) {
            printf("(%lfm, %lf, %lf)\n",
                   points[h->index].x,
                   points[h->index].y,
                   points[h->index].z);
            h = h->next;
        }
    }
}
```

The following function destroys the epsilon neighbours by freeing up the container and encapsulated linked list nodes. We need this as new epsilon neighbours are created and destroyed as we process each of the data points.

As we can see, this is an inefficient way to manage memory. We should reuse already allocated nodes without freeing them. See memory management using *memory areas*, as described by Donald Knuth in *The Stanford GraphBase: A Platform for Combinatorial Computing* [New York: ACM Press, 1994]. In this implementation, we use the inefficient but simpler version.

⟨*Function definitions*⟩+≡

```
void destroy_epsilon_neighbours(epsilon_neighbours_t *en)
{
    if (en) {
        node_t *t, *h = en->head;
        while (h) {
            t = h->next;
            free(h);
            h = t;
        }
        free(en);
    }
}
```

We shall now describe the main iteration of the DBSCAN algorithm. This function takes as input the array of data points `points`, the number of data points in the array `num_points`, the `epsilon` value for proximity sensitivity, the minimum number of neighbours `minpts` required to class a data point as a core data point, and the distance function `dist`.

At each iteration, we check if the data point is `UNCLASSIFIED`. If it is, we determine if a new cluster can be created starting with this data point. The `expand` function does this. If a cluster is started, i.e., the data point was classified as a core data point, `expand` returns `CORE_POINT`; otherwise, it returns `NOT_CORE_POINT`. If `expand` returns `CORE_POINT`, it also means that the next data point should start a new cluster, as the current cluster is complete. Recall the flow-of-liquid analogy that we discussed in the introduction.

⟨*Function definitions*⟩+≡

```
void dbscan(
    point_t *points,
    unsigned int num_points,
    double epsilon,
    unsigned int minpts,
    double (*dist)(point_t *a, point_t *b))
{
    unsigned int i, cluster_id = 0;
    for (i = 0; i < num_points; ++i) {
        if (points[i].cluster_id == UNCLASSIFIED) {
            if (expand(i, cluster_id, points,
                       num_points, epsilon, minpts,
                       dist) == CORE_POINT)
                ++cluster_id;
        }
    }
}
```

Let us now look at what the `expand` function does. It takes as input the `index` of the data point that we wish to consider. In case it turns out to be a core data point, it must know the cluster that it should either start, or join as a new member. This is supplied as `cluster_id`. Then it takes the array of data points `points`, the number of data points `num_points`, the `epsilon` sensitivity, the `minpts` for classification as core data point, and finally the distance function `dist`.

The function `expand` first identifies the epsilon neighbours of the supplied data point. It then checks if there are enough neighbours to classify the data point as a core data point. If it is insufficient, the data point is marked tentatively as `NOISE` and the function returns `NO_CORE_POINT`. Future iterations due to the processing of other data points might update this classification.

On the other hand, if the data point has sufficient epsilon neighbours, we start a new cluster and continue spreading the cluster (just like the flow of liquid) as long as the sensitivity conditions are satisfied. Once all of the data points that should belong to the new cluster have been added, the function returns `CORE_POINT`, which is an indication that the next core data point should start a new cluster.

⟨*Function definitions*⟩+≡

```
int expand(
    unsigned int index,
    unsigned int cluster_id,
    point_t *points,
    unsigned int num_points,
    double epsilon,
    unsigned int minpts,
    double (*dist)(point_t *a, point_t *b))
{
    int return_value = NOT_CORE_POINT;
    epsilon_neighbours_t *seeds =
        get_epsilon_neighbours(index, points,
                               num_points, epsilon,
                               dist);
    if (seeds == NULL)
        return FAILURE;

    if (seeds->num_members < minpts)
        points[index].cluster_id = NOISE;
    else {
        points[index].cluster_id = cluster_id;
        ⟨Add epsilon neighbours to the same cluster⟩
        ⟨See how far the cluster spreads⟩
        return_value = CORE_POINT;
    }
    destroy_epsilon_neighbours(seeds);
    return return_value;
}
```

Make all of the epsilon neighbours of this data point also belong to the same cluster.

⟨*Add epsilon neighbours to the same cluster*⟩≡
```
node_t *h = seeds->head;
while (h) {
    points[h->index].cluster_id = cluster_id;
    h = h->next;
}
```

Now process all of the epsilon neighbours and see how far the cluster should spread.

⟨*See how far the cluster spreads*⟩≡
```
h = seeds->head;
while (h) {
    spread(h->index, seeds, cluster_id, points,
           num_points, epsilon, minpts, dist);
    h = h->next;
}
```

Now we present the final component of the algorithm, where we determine how far the current cluster should spread. The `spread` function takes as input the `index` of the epsilon neighbour being considered, the epsilon neighbours `seeds` of the core data point that started the current cluster, the current cluster identifier `cluster_id`, the array of data points `points`, the number of data points in the array `num_points`, the `epsilon` sensitivity, the `minpts` core data point criterion and finally the distance function `dist`.

⟨*Function definitions*⟩+≡
```
int spread(
    unsigned int index,
    epsilon_neighbours_t *seeds,
    unsigned int cluster_id,
    point_t *points,
    unsigned int num_points,
    double epsilon,
    unsigned int minpts,
    double (*dist)(point_t *a, point_t *b))
{
    epsilon_neighbours_t *spread =
        get_epsilon_neighbours(index, points,
                       num_points, epsilon,
                       dist);
    if (spread == NULL)
        return FAILURE;
    ⟨Process epsilon neighbours of neighbour⟩
    destroy_epsilon_neighbours(spread);
    return SUCCESS;
}
```

For every epsilon neighbour that the algorithm adds to the current cluster, it also processes their epsilon neighbours. If it turns out that the epsilon neighbour is itself a core data point, we process it's neighbours by appending them at the end of the current list of data points to be processed.

If the epsilon neighbour of a newly added core data point is classified as either `NOISE` or `UNCLASSIFIED`, they are added to the current cluster. Furthermore, if we encounter an `UNCLASSIFIED` data point, we append this at the end of the current core data point's epsilon neighbours `seeds`. Remember that this is the epsilon neighbours of the core point that started the current cluster. Hence, as the algorithms progresses `seeds` grows, as *directly reachable* data points are appended.

⟨*Process epsilon neighbours of neighbour*⟩≡
```
if (spread->num_members >= minpts) {
    node_t *n = spread->head;
    point_t *d;
    while (n) {
        d = &points[n->index];
        if (d->cluster_id == NOISE ||
            d->cluster_id == UNCLASSIFIED) {
            if (d->cluster_id == UNCLASSIFIED) {
                if (append_at_end(n->index, seeds)
                    == FAILURE) {
                    destroy_epsilon_neighbours(spread);
                    return FAILURE;
                }
            }
            d->cluster_id = cluster_id;
        }
        n = n->next;
    }
}
```

What we need now is the definition of a distance function. The following function implements Euclidean distance between two three-dimensional data points. This can be use for clustering based on spatial proximity.

⟨*Function definitions*⟩+≡
```
double euclidean_dist(point_t *a, point_t *b)
{
    return sqrt(pow(a->x - b->x, 2) +
            pow(a->y - b->y, 2) +
            pow(a->z - b->z, 2));
}
```

The following function parses the data points from `file`, and stores the data points inside the array `points`.

⟨*Function definitions*⟩+≡

```
unsigned int parse_input(
    FILE *file,
    point_t **points,
    double *epsilon,
    unsigned int *minpts)
{
    unsigned int num_points, i = 0;
    fscanf(file, "%lf %u %u\n",
           epsilon, minpts, &num_points);
    point_t *p = (point_t *)
        calloc(num_points, sizeof(point_t));
    if (p == NULL) {
        perror("Failed to allocate points array");
        return 0;
    }
    while (i < num_points) {
          fscanf(file, "%lf %lf %lf\n",
                 &(p[i].x), &(p[i].y), &(p[i].z));
          p[i].cluster_id = UNCLASSIFIED;
          ++i;
    }
    *points = p;
    return num_points;
}
```

The following function prints out the data points and cluster identifier for that data point to `stdout`.

⟨*Function definitions*⟩+≡

```
void print_points(
    point_t *points,
    unsigned int num_points)
{
    unsigned int i = 0;
    printf("Number of points: %u\n"
        " x     y     z     cluster_id\n"
        "-----------------------------\n"
        , num_points);
    while (i < num_points) {
          printf("%5.2lf %5.2lf %5.2lf: %d\n",
                 points[i].x,
                 points[i].y, points[i].z,
                 points[i].cluster_id);
          ++i;
    }
}
```

Finally, the main function that parses the input data points, runs the DBSCAN algorithm and then prints the clusters.

⟨*The main program*⟩≡

```
int main(void) {
    point_t *points;
    double epsilon;
    unsigned int minpts;
    unsigned int num_points =
        parse_input(stdin, &points, &epsilon, &minpts);
    if (num_points) {
        dbscan(points, num_points, epsilon,
               minpts, euclidean_dist);
        printf("Epsilon: %lf\n", epsilon);
        printf("Minimum points: %u\n", minpts);
        print_points(points, num_points);
    }
    free(points);
    return 0;
}
```

Forward declaration of functions so that we can define and explain the functions out of compilation order.

⟨*Function declarations*⟩≡
```
node_t *create_node(unsigned int index);
int append_at_end(
     unsigned int index,
     epsilon_neighbours_t *en);
epsilon_neighbours_t *get_epsilon_neighbours(
    unsigned int index,
    point_t *points,
    unsigned int num_points,
    double epsilon,
    double (*dist)(point_t *a, point_t *b));
void print_epsilon_neighbours(
    point_t *points,
    epsilon_neighbours_t *en);
void destroy_epsilon_neighbours(epsilon_neighbours_t *en);
void dbscan(
    point_t *points,
    unsigned int num_points,
    double epsilon,
    unsigned int minpts,
    double (*dist)(point_t *a, point_t *b));
int expand(
    unsigned int index,
    unsigned int cluster_id,
    point_t *points,
    unsigned int num_points,
    double epsilon,
    unsigned int minpts,
    double (*dist)(point_t *a, point_t *b));
int spread(
    unsigned int index,
    epsilon_neighbours_t *seeds,
    unsigned int cluster_id,
    point_t *points,
    unsigned int num_points,
    double epsilon,
    unsigned int minpts,
    double (*dist)(point_t *a, point_t *b));
double euclidean_dist(point_t *a, point_t *b);
double adjacent_intensity_dist(point_t *a, point_t *b);
unsigned int parse_input(
    FILE *file,
    point_t **points,
    double *epsilon,
    unsigned int *minpts);
void print_points(
    point_t *points,
    unsigned int num_points);
```

⟨*Include libraries*⟩≡
```
#include <limits.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
```

⟨*dbscan*⟩≡
```
⟨Include libraries⟩
⟨Constant definitions⟩
⟨Data structures⟩
⟨Function declarations⟩
⟨Function definitions⟩
⟨The main program⟩
```