

Saint Petersburg National Research University of Information Technologies, Mechanics and
Optics (ITMO University)
Faculty of Informational Technologies and Programming

REPORT

about laboratory work № 3

« The degree of objects similarity »

Student

Nerinovsky Arseny
(Surname, initials)

M41331
Group

Saint-Petersburg, 2018

Report

June 29, 2019

1

k -

2

1. DBSCAN k -
2. , () k - , , .

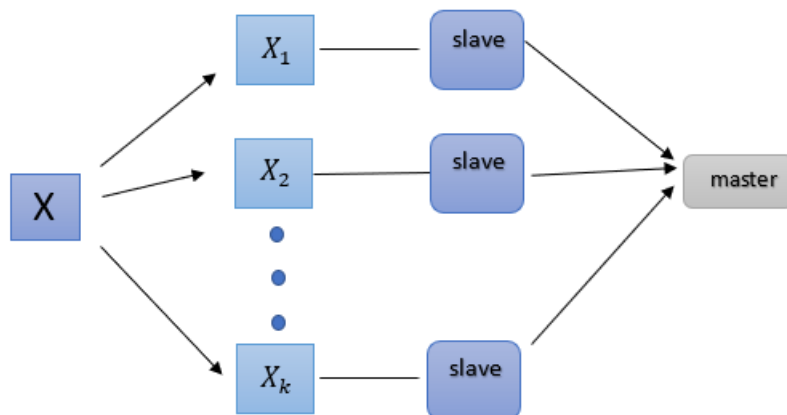
3

, , .

3.1 DBSCAN

DBSCAN (Density-based spatial clustering of applications with noise) - , . DBSCAN , .

DBSCAN , k-d tree, r-tree, vintage-point tree. $X = \{X_i, i = 1, \dots, k\}$.
DBSCAN master-slave. , . , . , .



3.2 K-

k - X K S_1, S_2, \dots, S_k , d_k , d , d K , n , re , $()$ K , $.$

4

[1]: %cat example2.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

#include <assert.h>
#include <float.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <omp.h>

#define KMEANS_NULL_CLUSTER -1

/*
 * If the algorithm doesn't converge within this number of iterations,
 * it will return with a failure error code.
 */
#define KMEANS_MAX_ITERATIONS 1000

#define kmeans_malloc(size) malloc(size)
#define kmeans_free(ptr) free(ptr)

typedef void * Pointer;

typedef enum {
    KMEANS_OK,
    KMEANS_EXCEEDED_MAX_ITERATIONS,
    KMEANS_ERROR
} kmeans_result;

typedef double (*kmeans_distance_method) (const Pointer a, const Pointer b);
```

```

typedef void (*kmeans_centroid_method) (const Pointer * objs, const int *
clusters, size_t num_objs, int cluster, Pointer centroid);

typedef struct kmeans_config
{
    /* Function returns the "distance" between any pair of objects */
    kmeans_distance_method distance_method;

    /* Function returns the "centroid" of a collection of objects */
    kmeans_centroid_method centroid_method;

    /* An array of objects to be analyzed. User allocates this array */
    /* and is responsible for freeing it. */
    /* For objects that are not capable of participating in the distance */
    /* calculations, but for which you still want included in the process */
    /* (for examples, database nulls, or geometry empties) use a NULL */
    /* value in this list. All NULL values will be returned in the */
    /* KMEANS_NULL_CLUSTER. */
    Pointer * objs;

    /* Number of objects in the preceding array */
    size_t num_objs;

    /* An array of initial centers for the algorithm */
    /* Can be randomly assigned, or using proportions, */
    /* unfortunately the algorithm is sensitive to starting */
    /* points, so using a "better" set of starting points */
    /* might be wise. User allocates and is responsible for freeing. */
    Pointer * centers;

    /* Number of means we are calculating, length of preceding array */
    unsigned int k;

    /* Maximum number of times to iterate the algorithm, or 0 for */
    /* library default */
    unsigned int max_iterations;

    /* Iteration counter */
    unsigned int total_iterations;

    /* Array to fill in with cluster numbers. User allocates and frees. */
    int * clusters;
} kmeans_config;

static void
update_r(kmeans_config *config)

```

```

{
    int i;

    #pragma omp parallel for
    for (i = 0; i < config->num_objs; i++)
    {
        double distance, curr_distance;
        int cluster, curr_cluster;
        Pointer obj;

        assert(config->objs != NULL);
        assert(config->num_objs > 0);
        assert(config->centers);
        assert(config->clusters);

        obj = config->objs[i];

        /*
        * Don't try to cluster NULL objects, just add them
        * to the "unclusterable cluster"
        */
        if (!obj)
        {
            config->clusters[i] = KMEANS_NULL_CLUSTER;
            continue;
        }
        // printf("It2\n");
        /* Initialize with distance to first cluster */
        curr_distance = (config->distance_method)(obj,
config->centers[0]);
        curr_cluster = 0;

        /* Check all other cluster centers and find the nearest */
        for (cluster = 1; cluster < config->k; cluster++)
        {
            distance = (config->distance_method)(obj,
config->centers[cluster]);
            if (distance < curr_distance)
            {
                curr_distance = distance;
                curr_cluster = cluster;
            }
        }

        /* Store the nearest cluster this object is in */
        config->clusters[i] = curr_cluster;
    }
}

```

```

}

static void
update_means(kmeans_config *config)
{
    int i;

    for (i = 0; i < config->k; i++)
    {
        /* Update the centroid for this cluster */
        (config->centroid_method)(config->objs, config->clusters,
config->num_objs, i, config->centers[i]);
    }
}

kmeans_result
kmeans(kmeans_config *config)
{
    int iterations = 0;
    int *clusters_last;
    size_t clusters_sz = sizeof(int)*config->num_objs;

    assert(config);
    assert(config->objs);
    assert(config->num_objs);
    assert(config->distance_method);
    assert(config->centroid_method);
    assert(config->centers);
    assert(config->k);
    assert(config->clusters);
    assert(config->k <= config->num_objs);

    /* Zero out cluster numbers, just in case user forgets */
    memset(config->clusters, 0, clusters_sz);

    /* Set default max iterations if necessary */
    if (!config->max_iterations)
        config->max_iterations = KMEANS_MAX_ITERATIONS;

    /*
     * Previous cluster state array. At this time, r doesn't mean anything
     * but it's ok
     */
    clusters_last = kmeans_malloc(clusters_sz);

    while (1)

```

```

{
    /* Store the previous state of the clustering */
    memcpy(clusters_last, config->clusters, clusters_sz);

    update_r(config);
    update_means(config);
    /*
     * if all the cluster numbers are unchanged since last time,
     * we are at a stable solution, so we can stop here
     */
    if (memcmp(clusters_last, config->clusters, clusters_sz) == 0)
    {
        kmeans_free(clusters_last);
        config->total_iterations = iterations;
        return KMEANS_OK;
    }

    if (iterations++ > config->max_iterations)
    {
        kmeans_free(clusters_last);
        config->total_iterations = iterations;
        return KMEANS_EXCEEDED_MAX_ITERATIONS;
    }
}

kmeans_free(clusters_last);
config->total_iterations = iterations;
return KMEANS_ERROR;
}

typedef struct point
{
    double x;
    double y;
} point;

static double pt_distance(const Pointer a, const Pointer b)
{
    point *pa = (point*)a;
    point *pb = (point*)b;

    double dx = (pa->x - pb->x);
    double dy = (pa->y - pb->y);

    return dx*dx + dy*dy;
}

```

```

static void pt_centroid(const Pointer * objs, const int * clusters, size_t
num_objs, int cluster, Pointer centroid)
{
    int i;
    int num_cluster = 0;
    point sum;
    double sum_x, sum_y;
    point **pts = (point**)objs;
    point *center = (point*)centroid;

    sum_x = sum_y = 0.0;

    if (num_objs <= 0) return;

    #pragma omp parallel for reduction(+:sum_x,sum_y,num_cluster)
    for (i = 0; i < num_objs; i++)
    {
        /* Only process objects of interest */
        if (clusters[i] != cluster) continue;

        sum_x += pts[i]->x;
        sum_y += pts[i]->y;
        num_cluster+=1;
    }
    if (num_cluster)
    {
        sum_x /= num_cluster;
        sum_y /= num_cluster;
        center->x = sum_x;
        center->y = sum_y;
    }
    return;
}

int
main(int nargs, char **args)
{
    kmeans_config config;
    kmeans_result result;
    int i, j;
    int spread = 3;
    point *pts;
    point *init;
    int print_results = 0;
    unsigned long start;

    int k = 10;

```



```

srand(1234);

/* Constants */
config.k = k;
config.num_objs = 100000; // config.k * nptsincluster;
config.max_iterations = 200;
config.distance_method = pt_distance;
config.centroid_method = pt_centroid;

/* Inputs for K-means */
config.objs = calloc(config.num_objs, sizeof(Pointer));
config.centers = calloc(config.k, sizeof(Pointer));
config.clusters = calloc(config.num_objs, sizeof(int));

/* Storage for raw data */
pts = calloc(config.num_objs, sizeof(point));
init = calloc(config.k, sizeof(point));

/* Create test data! */
/* Populate with K gaussian clusters of data */

FILE *file = fopen("birch3.txt", "r");
int n = 0, x, y;
while ( fscanf(file, "    %d    %d\n", &x , &y) != EOF ) {
    pts[n].x = (double)x, pts[n].y = (double) y;
    config.objs[n] = &(pts[n]);

    n += 1;
    if( n > config.num_objs ) {
        printf("Too many objects");
        exit(1);
    }
}

/* Populate the initial means vector with random start points */
for (i = 0; i < config.k; i++)
{
    int r = floor(config.num_objs * (1.0 * rand() / RAND_MAX));
    /* Populate raw data */
    init[i] = pts[r];
    /* Pointers to raw data */
    config.centers[i] = &(init[i]);

    if (print_results)
        printf("center[%d]\t%g\t%g\n", i, init[i].x, init[i].y);
}

```

```

/* run k-means! */
start = time(NULL);
result = kmeans(&config);

printf("\n");
printf("Iteration count: %d\n", config.total_iterations);
printf("      Time taken: %ld seconds\n", (time(NULL) - start));
printf(" Iterations/sec: %.3g\n",
(1.0*config.total_iterations)/(time(NULL) - start));
printf("\n");

/* print results */
if (print_results)
{
    for (i = 0; i < config.num_objs; i++)
    {
        point *pt = (point*)(config.objs[i]);

        if (config.objs[i])
            printf("%g\t%g\t%d\n", pt->x, pt->y,
config.clusters[i]);
        else
            printf("N\tN\t%d\n", config.clusters[i]);
    }
}

free(config.objs);
free(config.clusters);
free(config.centers);

free(init);
free(pts);
}

```

5

birch3.txt <http://cs.joensuu.fi/sipu/datasets/> . , .

```

[2]: import subprocess
import os
from pathlib import Path

if Path('kmeans').exists():
    %cd kmeans/

```

```
def compile():

    _cmd = 'make -B example2'.split()
    # print(' '.join(_cmd))
    cmd = subprocess.run(_cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

    # if(cmd.stdout): print('cmd.stdout', cmd.stdout)
    if(cmd.stderr): print('cmd.stderr', cmd.stderr)

def run(env=None):
    cmd = subprocess.run('./example2', stdout=subprocess.PIPE,
→stderr=subprocess.PIPE, env=env)
```

```
[3]: compile()

for th in [1, 2, 4, 8, 16]:
    env = os.environ.copy()
    env['OMP_NUM_THREADS'] = str(th)

    print(f"Executing kmeans with {th} threads", end="\n\t")
    %timeit run(env)
    print()
```

Executing kmeans with 1 threads

488 ms ± 5.54 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Executing kmeans with 2 threads

282 ms ± 12.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Executing kmeans with 4 threads

1.48 s ± 43.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Executing kmeans with 8 threads

304 ms ± 21.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Executing kmeans with 16 threads

314 ms ± 7.97 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

6

k- . OpenMP.