ΥΠΟΛΟΓΙΣΤΙΚΗ ΝΟΗΜΟΣΥΝΗ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ - ΤΜΗΥΠ

ΑΚΑΔΗΜΑΪΚΟ ΕΤΟΣ 2021 - 2022

ΚΥΡΙΑΖΗΣ ΑΓΑΜΕΜΝΩΝ	4400
ΠΕΡΓΑΜΗΝΕΛΗΣ ΧΡΗΣΤΟΣ	4474
ΤΣΙΑΟΥΣΗ ΘΩΜΑΗ	4510

Περιεχόμενα

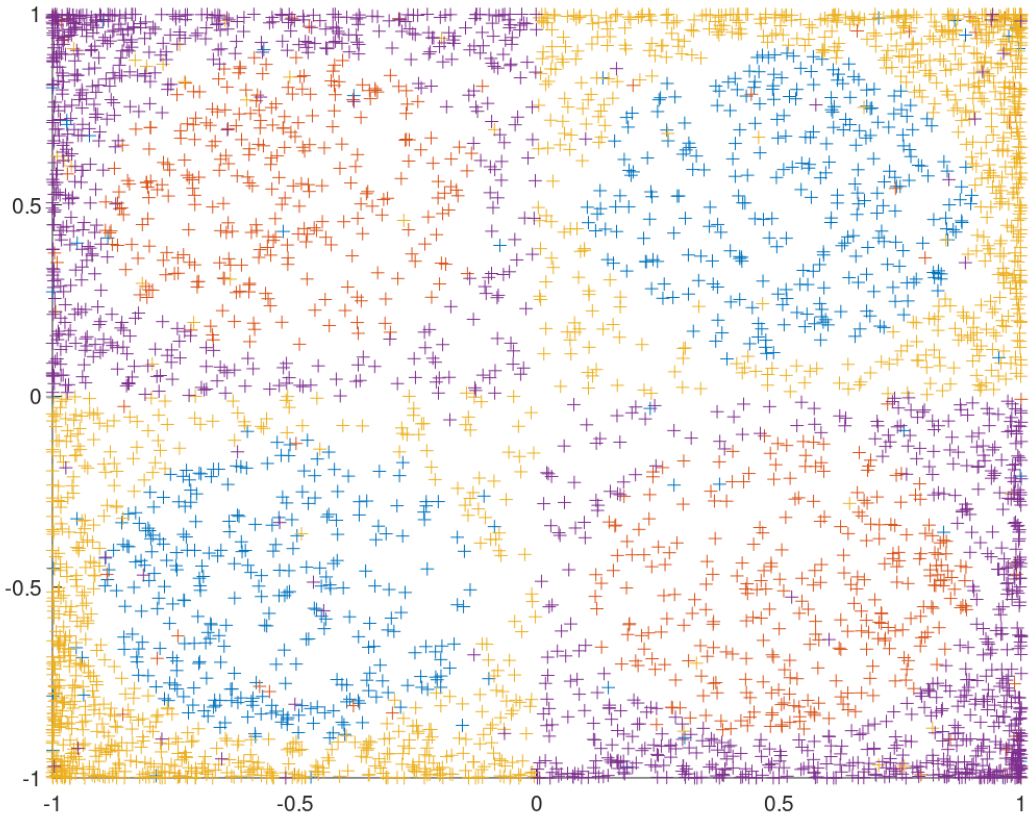
Σύνολα δεδομένων για τις ασκήσεις:

Σ1:

Για το σύνολο δεδομένων της άσκησης (1) δημιουργήσαμε 8,000 τυχαία παραδείγματα με x_1, x_2 να ανήκουν στο τετράγωνο [-1, 1]. Τα καταγράψαμε στο αρχείο data.txt με μορφή x_1,x_2,c όπου c είναι η κατηγορία που αναθέσαμε το παράδειγμα με βάση τις εξισώσεις. Επισημαίνουμε ότι προσθέσαμε θόρυβο μόνο στο σύνολο εκπαίδευσης (δηλαδή στα πρώτα 4,000 δεδομένα) με πιθανότητα 0.1. Για να πάρουμε τυχαίες τιμές στο διάστημα [-1, 1] χρησιμοποιήσαμε την συνάρτηση sin(x) με x να είναι ένας τυχαίος αριθμός rand()*. Για να πάρουμε την πιθανότητα, όπου είναι ένας αριθμός στο διάστημα [0, 1], παίρνουμε την προηγούμενη συνάρτηση και την βάζουμε σε απόλυτο (fabs(sin(x))) = (|sin(x)|).

Παράδειγμα: -0.186073,-0.943919,3\n

Plot:

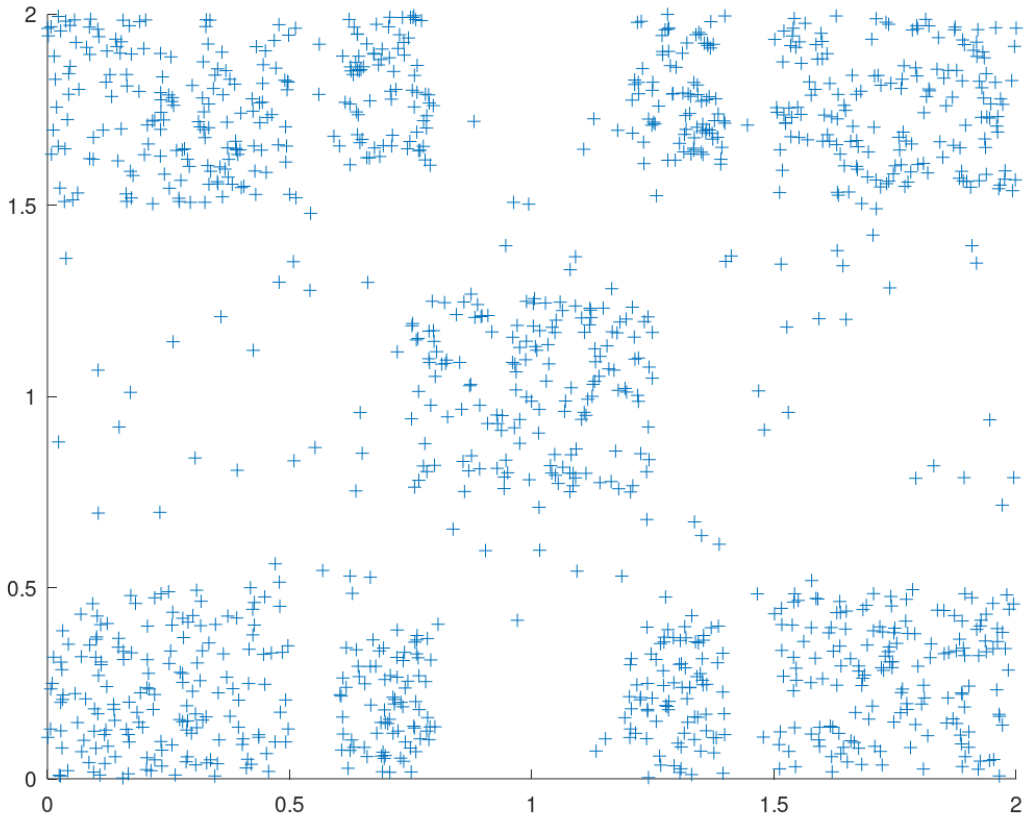


Σ2:

Για το σύνολο δεδομένων της άσκησης (2) δημιουργήσαμε 1,200 σημεία στο επίπεδο x_1, x_2. Επειδή σε αυτό το σημείο τα πεδία τιμών δεν ήτα συμμετρικά δεν μπορούσαμε να χρησιμοποιήσουμε την προηγούμενη συνάρτηση για να υπολογίσουμε τα σημεία. Έτσι δημιουργήσαμε μια καινούρια συνάρτηση τυχαίων αριθμών η οποία δέχεται μια ελάχιστη (a) και μια μέγιστη (b) τιμή. Η τιμή επιστροφής της είναι ένας τυχαίος* αριθμός στο διάστημα [a, b].

Παράδειγμα: 1.003035,1.243751\n

Plot:



ψευδοτυχαίοι αριθμοί

Πολυεπίπεδο perceptron, άσκηση (1):

Λίγα λόγια για την άσκηση:

Η γλώσσα προγραμματισμού που επιλέξαμε να χρησιμοποιήσουμε είναι η C. Για να κατασκευάσουμε το πολυεπίπεδο perceptron χρησιμοποιήσαμε structs καθώς και ξεχωριστά header files όπου κατασκευάσαμε μόνοι μας. Τα αρχείο vector.h και vector.c ορίζουν-υλοποιούν την δομή vector_t και συναρτήσεις για αυτήν. Δεν είναι τίποτα άλλο παρά μια δομή που εκφράζει ένα διάνυσμα και συναρτήσεις για πράξεις με αυτό. Στη συνέχεια έχουμε τα αρχεία neuron.h και neuron.c στα οποία ορίζουμε-υλοποιούμε δομές και συναρτήσεις για να συνθέσουμε το perceptron.
Η δομή neuron_t περιγράφει έναν νευρώνα, η layer_t ένα επίπεδο (array από νευρώνες) και η network_t ολόκληρο το δίκτυο (array από layers). Ο κάθε νευρώνας περιέχει τα βάρη του (vector_t) καθώς και την πόλωση του.
Το κάθε επίπεδο περιέχει τους νευρώνες του, την είσοδο x για τους νευρώνες, την έξοδο o όπου προκύπτει από την συνάρτηση $g(u_i)$ *(με g την συνάρτηση ενεργοποίησης)*, το id του, το βάθος του και τέλος το διάνυσμα e όπου είναι το δ για κάθε νευρώνα στο επίπεδο *(το συγκεκριμένο χρησιμοποιείται στον αλγόριθμο backpropagation)*.
Τέλος το δίκτυο network_t περιέχει ένα διάνυσμα που αποτελεί την είσοδο του δικτύου layer0, ένα array από επίπεδα και τέλος το πλήθος των επιπέδων. **Για compile γράφετε την εντολή make στο τερματικό.**

```
struct neuron_s {
	vector_t *w;
	double w0;
};
typedef struct neuron_s neuron_t;

struct layer_s
{
	int layer_id;
	neuron_t *neurons;
	vector_t *x;
	vector_t *o;
	vector_t *e;
	int layer_depth;
};
typedef struct layer_s layer_t;

struct network_s {
	vector_t *layer0;
	layer_t *layers;
	int number_of_layers;
};
typedef struct network_s network_t;
```

Παράμετροι του πολυεπίπεδου perceptron:

Για τις παραμέτρους του πολυεπίπεδου perceptron *(αριθμού εισόδων (d), αριθμού κατηγοριών (K), αριθμού νευρώνων στα κρυμμένα επίπεδα, είδος συνάρτησης ενεργοποίησης, ρυθμού μάθησης (η) κ.α.)* κάναμε χρήση της εντολής define όπου χρησιμοποιείται στον προεπεξεργαστή της γλώσσας. Έτσι το δίκτυο είναι πλήρως παραμετροποιήσιμο χωρίς να χρειάζεται ο χειριστής να επέμβει στον κώδικα.

```
enum FUNCTION{TANH = 0, RELU = 1, LOGI = 2};
typedef double (*g)(double);

/* input size */
#define d   2
/* number of categories */
#define K   4
/* number of neurons at H1 (hidden layer 1) */
#define H1  16
/* number of neurons at H2 */
#define H2  16
/* number of neurons at H3 (if needed) */
#define H3  16
/* activation function to use at hidden layers */
#define F   TANH
/* activation function to use at the outer layer H+1 */
#define O_F TANH
/* batch size (must be divisor of N) */
#define B   40
/* epoch size */
#define N   4000
/* number of hidden layers */
#define CH  3
/* learning rate */
#define ETA 0.01
/* max accepted error */
#define ERR 0.01
/* min error difference between 2 epochs */
#define ERM 0.000001

int depth_per_layer[] = {H1,H2,H3,K};   /* amount of neurons per layer (edit
this to change the order of layers) */
g activation_functions[3] = {&tanh, &relu, &logistic};          /* list of
activation functions as pointers */
g d_activation_functions[3] = {&d_tanh, &d_relu, &d_logistic};  /* list of
activation function's derivatives as pointers */
```

Προσοχή:
Η παράμετρος K θα πρέπει να βρίσκεται πάντα στο τέλος της λίστας *depth_per_layer* αφού εκφράζει το πλήθος των κατηγοριών!

Φόρτωμα συνόλου εκπαίδευσης δεδομένων και κωδικοποίηση κατηγοριών:

a) Ανάγνωση από αρχείο:

Για να φορτώσουμε το σύνολο εκπαίδευσης κατασκευάσαμε μια συνάρτηση ανάγνωσης από αρχείο. Η τιμή offset που δέχεται ως όρισμα χρησιμοποιείται αργότερα για να διαβάσουμε τα επόμενα 4,000 δεδομένα που ανήκουν στο σύνολο ελέγχου αλλά δεν θα μας απασχολήσει τώρα (τίθεται σε 0).

```
void read_data_file(int offset) {
    epoch_data = (vector_t **)malloc(sizeof(vector_t *) * N);
    category_per_data = (int *)malloc(sizeof(int) * N);
    int i;
    for(i = 0; i < N; i++) {
        epoch_data[i] = zeroes(d);
    }

    double x1, x2;
    int category;

    FILE *fp = fopen("data.txt", "r");
    for(i = 0; i < offset; i++)
        fscanf(fp, "%lf,%lf,%d\n", &x1, &x2, &category);

    i = 0;
    while(fscanf(fp, "%lf,%lf,%d\n", &x1, &x2, &category) != EOF && i < N) {
        epoch_data[i]->vector[0] = x1;
        epoch_data[i]->vector[1] = x2;
        category_per_data[i] = category;
        i++;
    }
    fclose(fp);
}
```

b) Δημιουργία κατηγοριών:

Για να κωδικοποιήσουμε τις κατηγορίες του προβλήματος χρησιμοποιήσαμε κατηγοριοποίηση 1-out of-p. Για 4 κατηγορίες λοιπόν δημιουργήσαμε ένα διάνυσμα τεσσάρων θέσεων όπου μόνο μια από αυτές είχε την τιμή 1 ενώ οι υπόλοιπες την τιμή 0. Παραδείγματος χάρη αν ένα παράδειγμα ανήκει στην κατηγορία 3 το διάνυσμα που περιγράφει την κατηγορία του είναι το {0, 0, 1, 0}. Έτσι προκύπτει ο παρακάτω κώδικας:

```
void init_categories() { /* initialize categories as one-hot vectors, for
example c1 = {1,0,0,0}. c2 = {0,1,0,0} etc. */
    int i;
    categories = (vector_t **)malloc(sizeof(vector_t *) * K);
    for(i = 0; i < K; i++) {
        categories[i] = zeroes(K);
        categories[i]->vector[i] = 1;
    }
}
```

Αρχιτεκτονική δικτύου και αρχικοποίηση βαρών/πολώσεων:

Το δίκτυο κατασκευάζεται στην συνάρτηση init_network(…). Η συγκεκριμένη συνάρτηση δέχεται ως είσοδο το πλήθος των επιπέδων του δικτύου, μια λίστα που περιέχει το πλήθος των νευρώνων σε κάθε επίπεδο (κρυμμένο ή μη), και το μέγεθος της εισόδου (d). Μέσα στη μέθοδο αρχικοποιούνται οι δομές που χρησιμοποιούμε, συνδέονται τα επίπεδα του δικτύου μεταξύ τους και αρχικοποιούνται οι τιμές των βαρών στο διάστημα [-1, 1]. Τέλος η συνάρτηση επιστρέφει έναν δείκτη στη δομή network_t. Σημειώνουμε ότι ο ρυθμός μάθησης και το κατώφλι τερματισμού δίνονται με define στην main.c όπως εξηγήσαμε προηγουμένως.

```
network_t *init_network(int number_of_hidden_layers, int *layer_depth, int
input_size) {
    int i; /* layer indexer */
    int j; /* neuron indexer */
    int total_layers = number_of_hidden_layers+1; // calculate the total
number of layers
    network_t *network = (network_t *)malloc(sizeof(network_t)); // the
network is an array of layers
    network->number_of_layers = total_layers; // set the number of layers
    network->layers = (layer_t *)malloc(sizeof(layer_t)*total_layers); //
initialize the array

    network->layer0 = allocatem(input_size); /* this is the input layer for
the network */

    for(i = 0; i < total_layers; i++) {

        network->layers[i].layer_depth = layer_depth[i];
        network->layers[i].layer_id = i+1;

        /* set the input of neurons on layer 0 or connect the previous output
with the current input vector */
        network->layers[i].x = i == 0? network->layer0 : network->layers[i-
1].o;

        network->layers[i].o = zeroes(layer_depth[i]);
        network->layers[i].e = zeroes(layer_depth[i]);
        network->layers[i].neurons = (neuron_t
*)malloc(sizeof(neuron_t)*layer_depth[i]);

        for(j = 0; j < layer_depth[i]; j++) {
            network->layers[i].neurons[j].w = get_random_norm(network-
>layers[i].x->vector_size); // and set the weights vector accordingly
            network->layers[i].neurons[j].w0 = get_random_norms();
        }
    }

    return network; // finaly return a pointer to the whole structure
}
```

Αρχικοποίηση βαρών/ πολώσεων για κάθε νευρώνα.

Υλοποίηση της συνάρτησης forward-pass:

Αρχικά η συνάρτηση forward_pass(…) που έχει υλοποιηθεί διατρέχει κάθε νευρώνα, κάθε επιπέδου και υπολογίζει το διάνυσμα εξόδου για το συγκεκριμένο επίπεδο. Το διάνυσμα εξόδου υπολογίζεται βρίσκοντας το εσωτερικό γινόμενο ανάμεσα στις τιμές εισόδου και στο διάνυσμα βαρών για τον κάθε νευρώνα.
Συγκεκριμένα $o_j = g(\sum_{i=1}^{n} w_i x_i + w_0)$ με o_j η έξοδος του j-οστού νευρώνα του επιπέδου.

```
void forward_pass(vector_t *input_vector, vector_t *output_vector) {
    int i, j; /* iterate over each neuron in every layer */
    copy(network->layer0, input_vector); /* set layer0 inputs */
    for(i = 0; i < network->number_of_layers; i++) { /* for every layer */
        for(j = 0; j < network->layers[i].layer_depth; j++) { /* iterate over
every neuron */
            if(i == network->number_of_layers-1) {
                network->layers[i].o->vector[j] =
(*activation_functions[O_F])(u(network->layers[i].x, network-
>layers[i].neurons[j].w, network->layers[i].neurons[j].w0));
            }else {
                network->layers[i].o->vector[j] =
(*activation_functions[F])(u(network->layers[i].x, network-
>layers[i].neurons[j].w, network->layers[i].neurons[j].w0));
            }
            /* and calculate the dot product
               then pass it through the desired activation function to check
if the neuron fires or not! */
        }
    }
    copy(output_vector, network->layers[network->number_of_layers-1].o);
}
```

Υλοποίηση της συνάρτησης backprob:

Η συνάρτηση backprob(…) που έχει υλοποιηθεί αρχικά καλεί την συνάρτηση d_loss η οποία υπολογίζει το δ για το επίπεδο L=H+1. Στη συνέχεια διατρέχει κάθε νευρώνα του επιπέδου L και τροποποιεί τα βάρη του κατά $\Delta w = -\eta \cdot \delta_i \cdot x_i$. Αμέσως μετά ο αλγόριθμος συνεχίζει υπολογίζοντας τα δ για κάθε κριμένο επίπεδο από το τέλος προς την αρχή. Για την διευκόλυνση του υπολογισμού του αθροίσματος $\delta_i^{(h)} = y_i^{(h)}\left(1 - y_i^{(h)}\right)\sum_{j=1}^{d_{h+1}} w_{ij}^{(h+1)}\delta_j^{(h+1)}$, χρησιμοποιήσαμε ένα διάνυσμα e το οποίο λειτουργεί σαν ενδιάμεση μνήμη μεταξύ των επιπέδων αποθηκεύοντας το δ που προκύπτει σε κάθε επίπεδο.

```
void d_loss(vector_t *actual_output, vector_t *expected_output) {
    int i;
    vector_t *diff = sub(actual_output, expected_output);
    for(i = 0; i < diff->vector_size; i++) {
        diff->vector[i] *= (*d_activation_functions[O_F])(actual_output-
>vector[i]); /* (y_hat - y) * g'(y_hat) */
        network->layers[network->number_of_layers-1].e->vector[i] += diff-
>vector[i];
    }
    for(i = 0; i < network->layers[network->number_of_layers-1].e-
>vector_size; i++) {
        network->layers[network->number_of_layers-1].e->vector[i] /=
(double)batch_size;
    }
    releasem(diff);
}
```

```
void backprop(vector_t *input_vector, vector_t *expected_output) {
    int i, j, L = network->number_of_layers-1;
    double xi;

    d_loss(network->layers[L].o, expected_output); /* calculate delta for the
outer H+1 layer */
    for(i = 0; i < network->layers[L].layer_depth; i++) { /* for each neuron
in L=H+1 */
        for(j = 0; j < network->layers[L].neurons[i].w->vector_size; j++) {
/* for each weight in this neuron */
            xi = network->layers[L].x->vector[j]; /* the xi that the wi was
multiplied with */
            network->layers[L].neurons[i].w->vector[j] -= eta * network-
>layers[L].e->vector[i] * xi;
            network->layers[L].neurons[i].w0 -= eta * network->layers[L].e-
>vector[i]; /* xi for w0 (bias) is 1 */
        }
    }

    for(i = L-1; i >= 0; i--) { /* for each hidden layer */
        for(j = 0; j < network->layers[i].layer_depth; j++) { /* for each
neuron in it */

            vector_t *wji = allocatem(network->layers[i+1].layer_depth);
            for(int n = 0; n < network->layers[i+1].layer_depth; n++) { /*
neuron indexer for L = h+1 */
                wji->vector[n] = network->layers[i+1].neurons[n].w-
>vector[j]; /* get the n-th weight from each neuron */
            }
            network->layers[i].e->vector[j] =
(*d_activation_functions[F])(network->layers[i].o->vector[j]) * dot(wji,
network->layers[i+1].e);
            releasem(wji);

            for(int q = 0; q < network->layers[i].neurons[j].w->vector_size;
q++) { /* for each weight in this neuron */
                xi = network->layers[i].x->vector[q];
                network->layers[i].neurons[j].w->vector[q] -= eta * network-
>layers[i].e->vector[j] * xi; /* xi * delta */
                network->layers[i].neurons[j].w0 -= eta * network-
>layers[i].e->vector[j]; /* xi for w0 (bias) is 1 */
            }
        }
    }
}
```

Υλοποίηση αλγορίθμου gradient descent με χρήση mini batches:

Για να υλοποιήσουμε τον αλγόριθμο gradient descent με χρήση mono batches κατασκευάσαμε την συνάρτηση train. Αρχικά η συνάρτηση χωρίζει το σύνολο όλων των δεδομένων (εποχή) σε batches μεγέθους B. Για κάθε batch η συνάρτηση εκτελεί forward pass και back propagate κάνοντας ομαδική ενημέρωση των βαρών (αν B=1 τότε έχουμε σειριακή ενημέρωση). Αφού ολοκληρωθεί το πέρασμα των 4,000 δεδομένων επιστρέφεται το συνολικό σφάλμα εποχής.

```
double train(vector_t **epoch_data, int *category_per_data) {
    int i, j, L = network->number_of_layers-1;;
    double error = 0;
    vector_t *output_vector = allocatem(K);

    for(i = 0; i < N-batch_size; i+=batch_size) {

        /* reset all deltas */
        for(int p = 0; p < network->number_of_layers; p++) {
            for(int q = 0; q < network->layers[p].e->vector_size; q++) {
                network->layers[p].e->vector[q] = 0.0f;
            }
        }

        for(j = 0; j < batch_size; j++) { /* for every input in the batch
data set */

            forward_pass(epoch_data[i+j], output_vector); /* forward pass the
data through the network */

            backprop(epoch_data[i+j], categories[category_per_data[i+j] -
1]); /* the backpropagate the error and adjust the weights */

            error += loss(output_vector, categories[category_per_data[i+j] -
1]); /* add the mean square error created for the selected batch size */
            /* the final result is the epoch error */
        }

    }
    /* release all allocated memory! */
    releasem(output_vector);
    return error/(double)N;
}
```

Υπολογισμός και εκτύπωση της ικανότητας γενίκευσης του δικτύου:

Για να προσεγγίσουμε το πόσο καλά γενικεύει το πρόβλημα το νευρωνικό δίκτυο γράφουμε σε ένα αρχείο μέσω της μεθόδου test_and_close τα αποτελέσματα που προκύπτουν από το συνόλου ελέγχου. Στη συνέχεια το αρχείο read_data.m διαβάζει τα δεδομένα από το σύνολο ελέγχου και τα δεδομένα από το παραχθέν αρχείο, βρίσκει το πλήθος των σωστών απαντήσεων μετά από σύγκριση και τέλος εκτυπώνει τον λόγο σωστών προς όλων των απαντήσεων του δικτύου. Για οπτικοποίηση των απαντήσεων κάνουμε scatter *(plot)* τα περιεχόμενα του αρχείου.

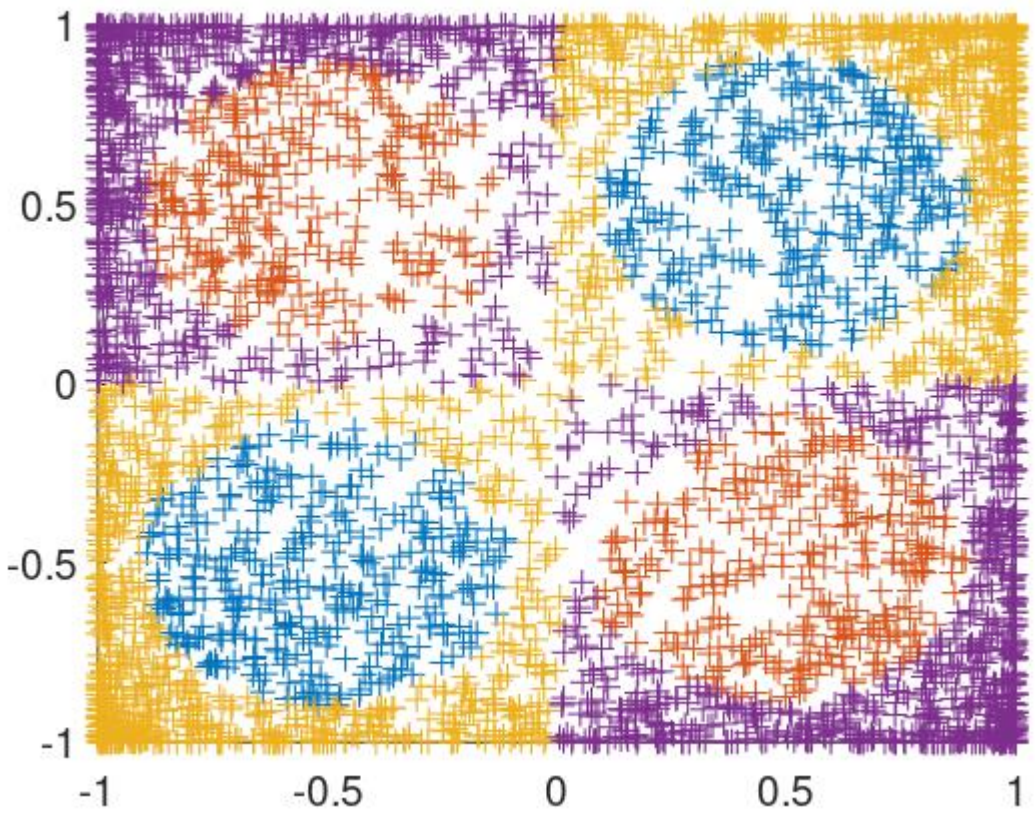
```
void test_and_close() {
    int i;
    FILE *fp = fopen("test.txt", "w"); /* empty the txt file */
    vector_t *output_vector = allocatem(K);
    read_data_file(N);
    for(i = 0; i < N; i++) {
        forward_pass(epoch_data[i], output_vector);
        fprintf(fp, "%lf,%lf,%d\n", epoch_data[i]->vector[0], epoch_data[i]-
>vector[1], max_index(output_vector)+1);
    }
    fclose(fp);
    /* release all allocated memory! */
    for(i = 0; i < N; i++) {
        releasem(epoch_data[i]);
    }
    free(epoch_data);
    free(category_per_data);
    release_network(network);
    release_categories(K);
    releasem(output_vector);
}
```

Αρχείο read_data.m:

```
y_hat = load("-ascii", "test.txt");
y = load("-ascii", "test_data.txt");

y_hat = transpose(y_hat);
y = transpose(y);
e = y_hat(3, :).-y(3, :);
columns(e(e == 0))/columns(e)

figure(1);
hold on;
for i = [1:1:4+1]
  scatter(y_hat(1, :)(y_hat(3, :) == i), y_hat(2, :)(y_hat(3, :) == i), "+")
endfor;
hold off;
```



Στο συγκεκριμένο διάγραμμα το δίκτυο παρουσιάζει 98.22% ποσοστό επιτυχίας στα αποτελέσματα του για το σύνολο ελέγχου που του δόθηκε. Το δίκτυο αποτελείται από 3 κρυμμένα επίπεδα με 16 νευρώνες το καθένα. Ο ρυθμός μάθησης είναι η=0.01 και η συνάρτηση ενεργοποίησης που χρησιμοποιήσαμε είναι υπερβολική εφαπτομένη. Το batch size είναι B=40.

Πρόγραμμα ομαδοποίησης με αλγόριθμο k-means, άσκηση (2):

Λίγα λόγια για την άσκηση:

Το πρόβλημα ομαδοποίησης, με χρήση του αλγορίθμου k-means, υλοποιήθηκε όπως και στην προηγούμενη άσκηση στη γλώσσα προγραμματισμού C. Τα αρχεία vector.c και vector.h, όμοια με παραπάνω, ορίζουν-υλοποιούν τη δομή vector_t και τις συναρτήσεις γι' αυτή. **Για compile γράφετε την εντολή make στο τερματικό.**

Υλοποίηση:

Στο αρχείο main.c του φακέλου k-means, αρχικά, το πρόγραμμα φορτώνει το αρχείο των δεδομένων που δημιουργήσαμε.

```
    fp = fopen("data.txt", "r");
    int i,j;
    double total_error;
    for(i = 0; i < N; i++) { /* for each point */
        data[i].cords = allocatem(P); /* allocate memory for the vector */
        data[i].cluster = 0; /* it doesn't belong to any cluster */
        fscanf(fp, "%lf,%lf\n", &data[i].cords->vector[0], &data[i].cords-
>vector[1]); /* read data from file */
    }
    fclose(fp);
```

Έπειτα, γίνεται η επιλογή των κέντρων επιλέγοντας τυχαία M από τα παραδείγματα. Ο προσδιορισμός του αριθμού των κέντρων (clusters) γίνεται στην αρχή του προγράμματος με #define M, για M = 3,5,7,9,11,13.

```
for(i = 0; i < K; i++) { /* for each cluster */
    centroids[i].cords = allocatem(P); /* allocate memory for the vector */
    centroids[i].cluster = i+1; /* assign cluster id */
    int r = rand() % N; /* pick a random point as cluser coordinates */
    centroids[i].cords->vector[0] = data[r].cords->vector[0];
    centroids[i].cords->vector[1] = data[r].cords->vector[1];
}
```

Στη συνέχεια εκτελείται ο αλγόριθμος k-means, υπολογίζοντας την Ευκλείδεια απόσταση κάθε σημείου των δεδομένων από τα centroids. Το κάθε σημείο μπαίνει στην ομάδα από την οποία έχει τη μικρότερη απόσταση από το κέντρο της. Τέλος αλλάζει την θέση των centroids βάσει την μέση απόσταση από τα δεδομένα που ανήκουν στην ομάδα τους.

```
loop:
    for(i = 0; i < N; i++) { /* for each data point */
        double min = euclidean_distance(data[i].cords, centroids[0].cords);
/* calculate the euclidean distance from the first centroid */
        int min_c = 1; /* assign the id of the first centroid */
        for(j = 0; j < K; j++) {
            double temp = euclidean_distance(data[i].cords,
centroids[j].cords); /* calculate the distance from each centroid */
            if(min > temp) { /* if we find a better distance */
                min = temp; /* change min distance value */
                min_c = j+1; /* change cluser id for this point */
            }
        }
        data[i].cluster = min_c; /* assign the fina value to the data point
*/
    }
    int terminate = 0; /* create an integer */
    for(i = 0; i < K; i++) { /* for each centroid */
        double x1_mean = 0.0f;
        double x2_mean = 0.0f;
        int counter = 0;
        for(j = 0; j < N; j++) { /* calculate the new centroid position using
mean data point coordinates */
            if(centroids[i].cluster == data[j].cluster) { /* if the data
point belongs to the cluster */
                counter++;
                x1_mean = x1_mean + data[j].cords->vector[0];
                x2_mean = x2_mean + data[j].cords->vector[1];
            }
        }
        x1_mean = x1_mean/ (double)counter;
        x2_mean = x2_mean/ (double)counter;
        if(centroids[i].cords->vector[0] != x1_mean || centroids[i].cords-
>vector[1] != x2_mean) { /* if the data point changes position */
            terminate++; /* increase the counter */
        }
        centroids[i].cords->vector[0] = x1_mean;
        centroids[i].cords->vector[1] = x2_mean;
    }
    if(terminate > 0) { /* if any centroid had its position changed */
        goto loop; /* loop */
    }
```

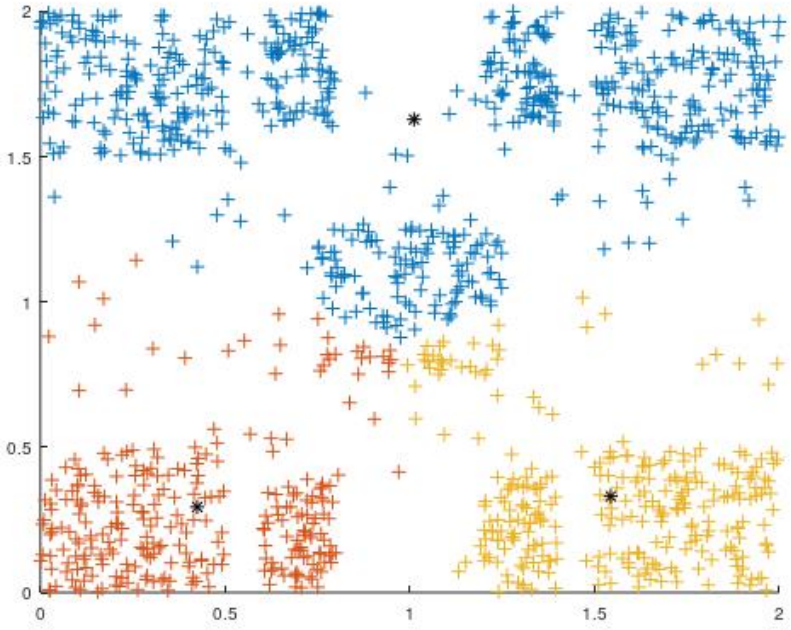
Τέλος, γίνεται υπολογισμός του σφάλματος ομαδοποίησης για κάθε ομάδα (cluster), καθώς και του συνολικού σφάλματος ομαδοποίησης (total clustering error) και εγγραφή των ομαδοποιημένων πλέον δεδομένων στο αρχείο test.txt και των centroid στο test2.txt.

```
    total_error = 0; /* calculate the clustering error for each cluster */
    /* and the total clustering error */
    for(i = 0; i < K; i++) {
        double temp = 0;
        for(j = 0; j < N; j++) {
            if(centroids[i].cluster == data[j].cluster) {
                temp += euclidean_distance(centroids[i].cords,
data[j].cords);
            }
        }
        total_error += temp;
        printf("Clustering error for c = %d is c_error = %lf\n", i+1, temp);
    }
    return total_error;

void log_results() {
    int i,j;
    /* then write the data points and their assigned cluster to a file  for
testing */
    fp = fopen("test.txt", "w");
    for(i = 0; i < N; i++) {
        fprintf(fp, "%lf,%lf,%d\n", data[i].cords->vector[0], data[i].cords-
>vector[1], data[i].cluster);
    }
    fclose(fp);
    /* also write the cluster coordinates and id's */
    fp = fopen("test2.txt", "w");
    for(i = 0; i < K; i++) {
        fprintf(fp, "%lf,%lf,%d\n", centroids[i].cords->vector[0],
centroids[i].cords->vector[1], centroids[i].cluster);
    }
}
```

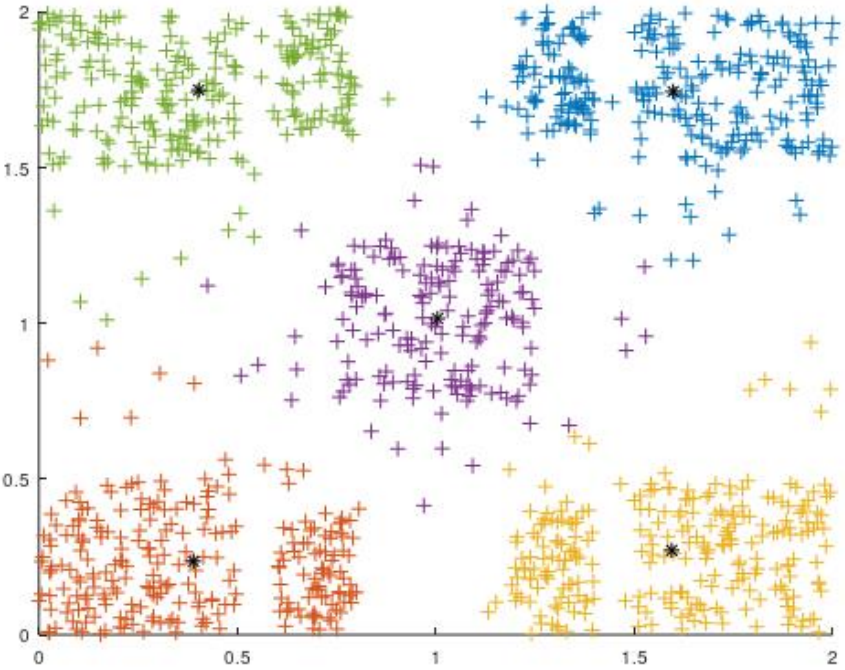
Αφού εκτελέσαμε το πρόγραμμα 20 φορές για κάθε τιμή του M, κρατήσαμε τη λύση με το μικρότερο total clustering error. Έτσι έχουμε:

→ M = 3



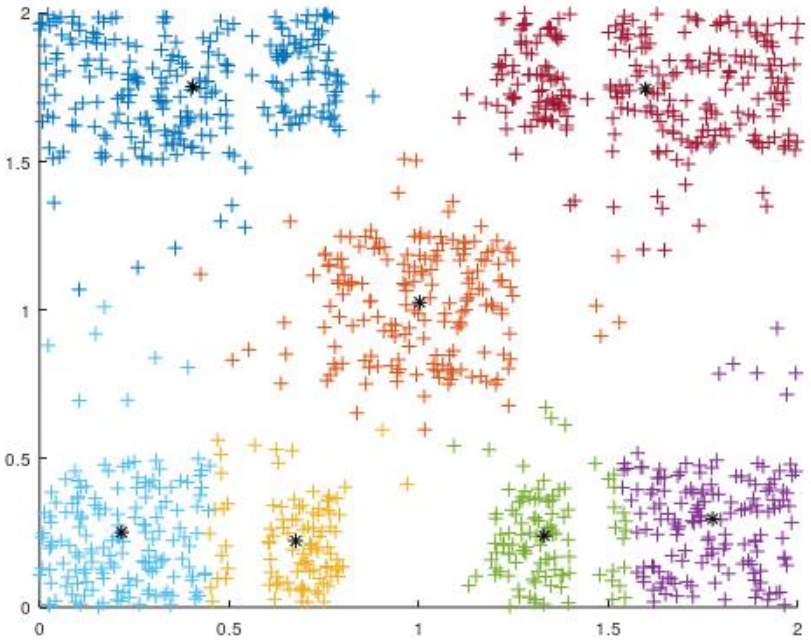
Total clustering error c-total = 571.161726

→ M = 5



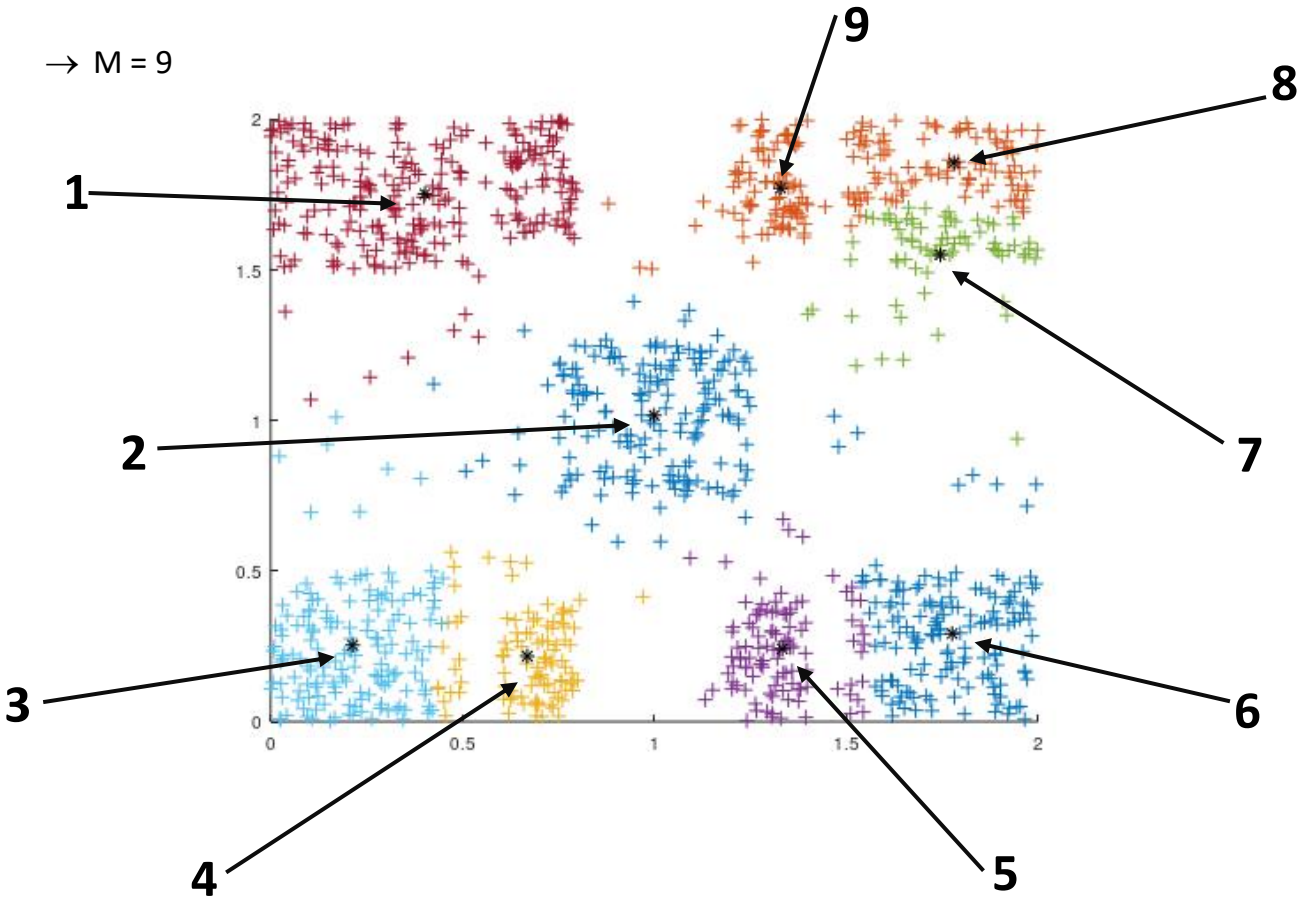
Total clustering error c-total = 318.733427

→ M = 7



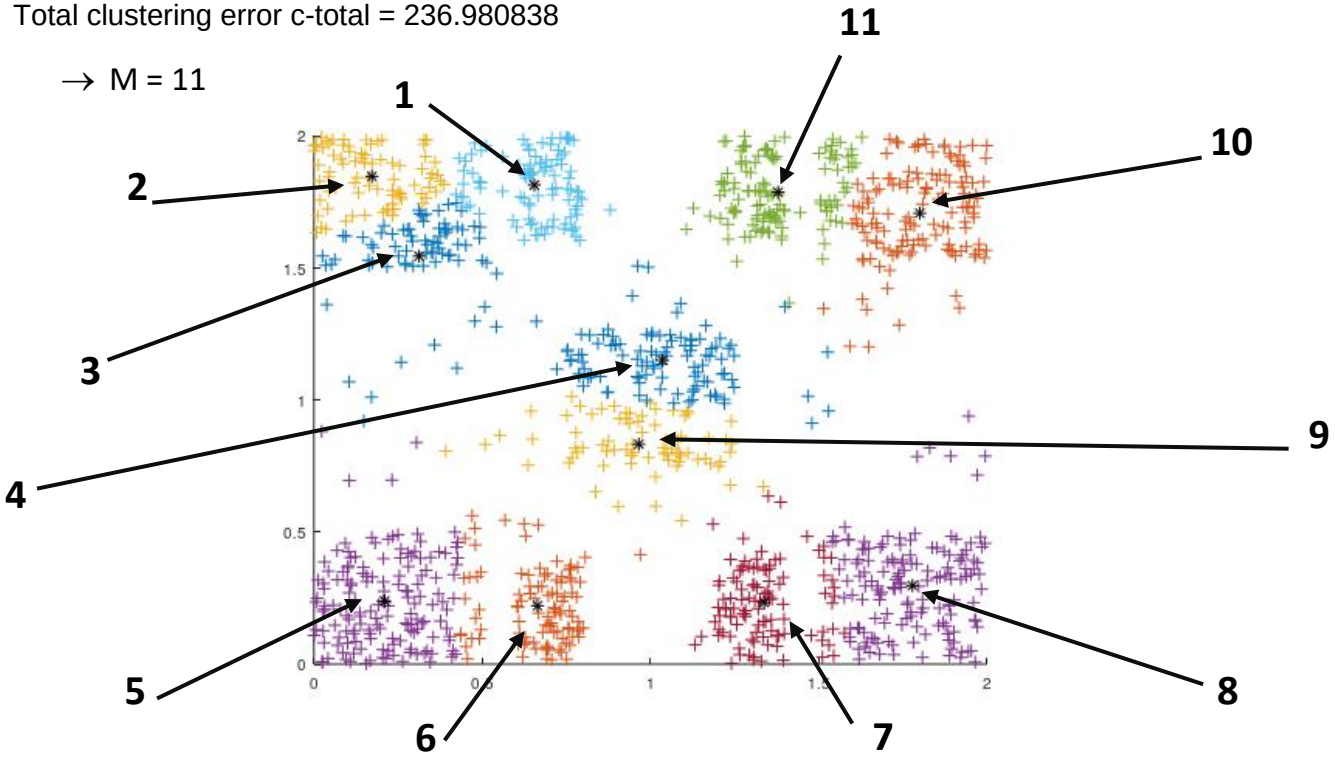
Total clustering error c-total = 268.181389

→ M = 9



Total clustering error c-total = 236.980838

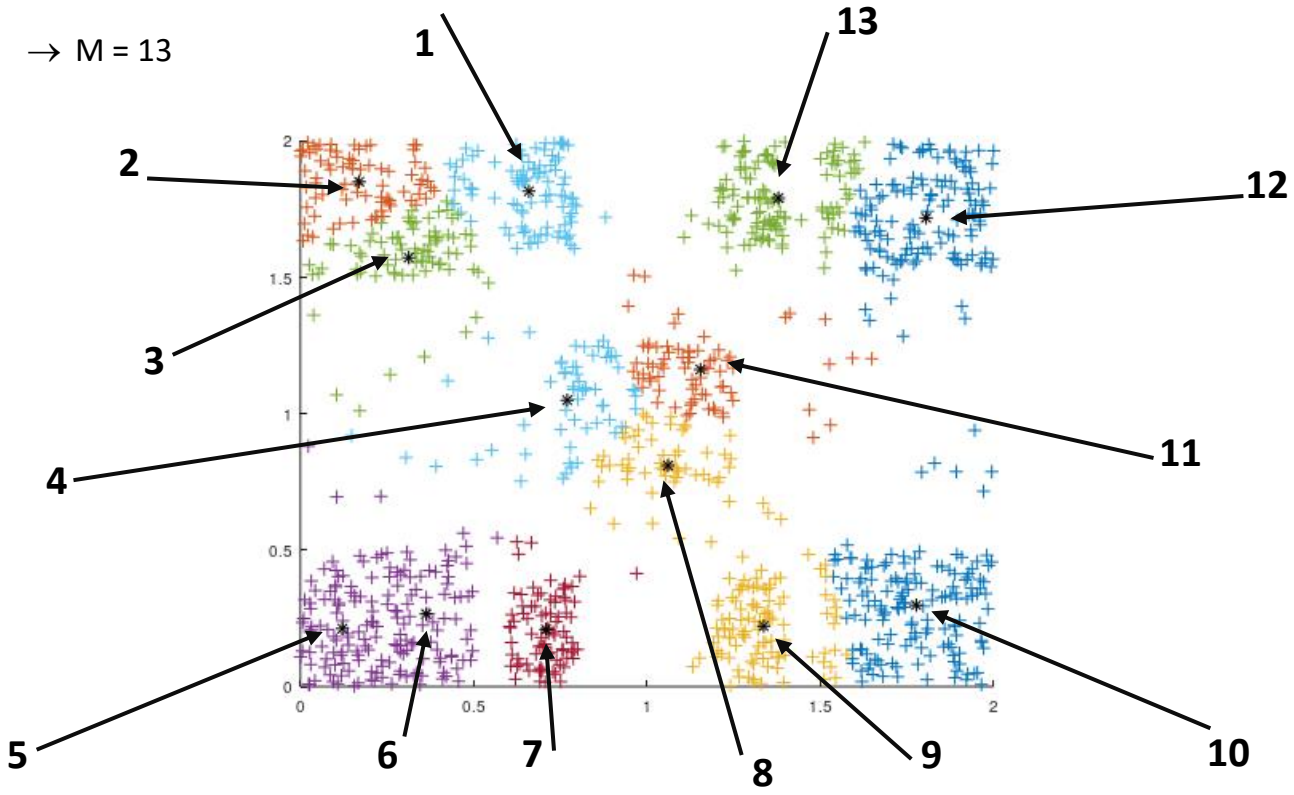
→ M = 11



Total clustering error c-total = 204.525960

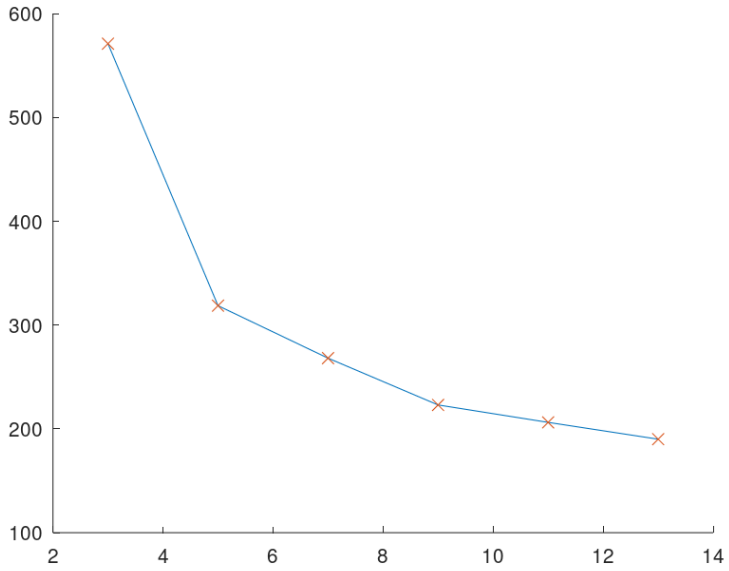
*Παρατήρηση: Οι ομάδες 3 και 4 είναι διαφορετικές μεταξύ τους.

→ M = 13



Total clustering error c-total = 191.473544
**Παρατήρηση: Οι ομάδες 5 και 6, όπως και οι 8 και 9 είναι διαφορετικές μεταξύ τους.

Βάσει των παραπάνω αποτελεσμάτων, το σφάλμα ομαδοποίησης μεταβάλλεται ως εξής:



Συμπεραίνουμε πως από το σφάλμα ομαδοποίησης δε μπορούμε να εκτιμήσουμε τον πραγματικό αριθμό των ομάδων, ο οποίος στο πρόβλημά μας είναι 9. Στο σχήμα έχουμε το φαινόμενο του «γονάτου» να συμβαίνει για M = 9, ενώ μετά το σφάλμα ομαδοποίησης μειώνεται λίγο για τις υπόλοιπες τιμές του M. Το βέλτιστο (ελάχιστο) σφάλμα ομαδοποίησης μειώνεται (ακόμα και λίγο) όσο μεγαλώνει το M. Άρα, δεν είναι ιδανικό από αυτό να αποφασίσουμε τον σωστό ακριβή αριθμό ομάδων που απαιτείται για την επίλυση του προβλήματος.