07.01.2022

Laboratory of Artificial Intelligence

Politechnika Śląska

Data Clustering with HCM and FCM algorithms

Authors:
Artur Oleksiński
Wojciech Bieniek

Table of contents

1. Introduction p. 3
2. HCM p. 4
    - initialization
    - cluster centers
    - distances
    - stop condition
    - results
3. FCM p. 11
    - fuzzy factor
    - results
4. Conclusion p. 14
5. Source code
    - HCM p. 15
    - FCM p. 20

1. Introduction

Cluster analysis (clustering) is the field of statistical analysis focused on grouping unlabeled data with the use of statistical methods and **unsupervised learning** algorithms.
Algorithms create groups of similar data based on the difference analysis.
The main goal is to find patterns and correlations between samples in the datasets.

Centroid models such as Hard C-Means and Fuzzy C-Means represent each cluster(group) by a single mean vector. The main difference between the two is the fuzzy factor of the algorithm.

- **Hard clustering** algorithms assume that every sample can belong only to one group at a time.
- **Fuzzy clustering** algorithms, on the other hand, create the possibility for the samples to partially belong to several clusters simultaneously.

Both hard and fuzzy c-means models are defined by the same optimization problem.
The problem is stated as:

$$min_{(U,V)}\left\{J_m(U,V)\right\} = \sum_{i=1}^{c}\sum_{k=1}^{n} u_{ik}^{m} D_{ik}^{2}$$

$$U \in M_{hcm}, M_{fcm}\ for\ HCM\ and\ FCM$$

$$V = (v_1, v_2, ..., v_c) \in \Re^{cp};\ v_i \in \Re^{p} -\ the\ i^{th}\ point\ prototype$$

$$m \geq 1 -\ the\ degree\ of\ fuzzification(m = 1\ for\ HCM)$$

$$D_{ik}^{2} = ||x_k - v_i||_A^{2}$$

These algorithms are **susceptible to the outliers** in the dataset. Even if the sample is far away from the clusters, HCM and FCM algorithms force it to be in one of the groups. (Or partially in a couple of groups in case of FCM).

2. HCM

Hard C-Means is a centroid model for clustering unlabeled data.
Every sample in the given unlabeled dataset is analyzed and based on the mean distance clusters are created. Every sample belongs fully to one and only one cluster.
The distribution of samples along the cluster is managed by the partition table.

$$\begin{array}{ccccccccc}
[0, & \mathbf{1}, & 0, & 0, & \mathbf{1}, & 0, & 0, & 0] & \Omega1 \\
[\mathbf{1}, & 0, & 0, & \mathbf{1}, & 0, & 0, & 0, & 0] & \Omega2 \\
[0, & 0, & 0, & 0, & 0, & \mathbf{1}, & 0, & \mathbf{1}] & \Omega3 \\
[0, & 0, & \mathbf{1}, & 0, & 0, & 0, & \mathbf{1}, & 0] & \Omega4 \\
x1 & x2 & x3 & x4 & x5 & x6 & x7 & x8 & .
\end{array}$$

The exemplary partition table

$$\Omega - \ clusters$$
$$x - \ data\ samples$$

```
function c_means(data, number_of_clusters)
    J = []
    ϵ = 1e-20
    U  = zeros(Int, number_of_clusters, size(data,2))
    U  = init_random(U )
    U_prev = U
    V  = cluster_centers(U , data)
    d = euclidian_distances(data, V )
    append!(J, [criterion_function(U , d)])
    U  = update_partition_table(U , d)
    while frobenius_norm(U , U_prev) > ϵ
        V  = cluster_centers(U , data)
        d = euclidian_distances(data, V )
        U  = update_partition_table(U , d)
        append!(J, [criterion_function(U , d)])
    end
    V  = cluster_centers(U , data)

    return (U , V )
end
```

The initialization of the partition table is done randomly with respect to the three rules.

First, every sample is fully assigned to a cluster or is not assigned to it at all.
Second, every sample is assigned to one and only one cluster.
Third, every cluster is non-empty.

```julia
function init_random(partition_table)
    height = length(eachrow(partition_table))
    for (i, _) in enumerate(eachcol(partition_table))
        partition_table[rand(1:height), i] = 1
    end

    for (i, row) in enumerate(eachrow(partition_table))
        if sum(row) == 0
            rand_col = rand(1:size(partition_table, 2))
            partition_table[:, rand_col] .= 0
            partition_table[i, rand_col] = 1
        end
        i = 1
    end
    return partition_table
end
```

```julia
J = []
ϵ = 1e-20
U  = zeros(Int, number_of_clusters, size(data,2))
U  = init_random(U )
U_prev = U
```

The cluster centers(prototypes) are the mean values of the data samples for every cluster with respect to its row in the partition table.

$$v_i = \frac{\sum_{k=1}^{N} u_{ik} x_k}{\sum_{k=1}^{N} u_{ik}}$$

```
function cluster_centers(partition_table, data)

    size_of_data = size(data)
    v = zeros(size(partition_table, 1), size_of_data[1])

    for partition_table_row in 1:size(partition_table, 1)
        for data_row in 1:size_of_data[1]
            summed_nominator = 0
            summed_denominator = 0
            for entry_number in 1:size_of_data[2]
                summed_denominator += partition_table[partition_table_row,
entry_number]
                summed_nominator += data[data_row,
entry_number]*partition_table[partition_table_row, entry_number]
            end
            v[partition_table_row, data_row] =
summed_nominator/summed_denominator
        end
    end

    return rotr90(v)
end
```

```
V  = cluster_centers(U , data)
```

The distances are calculated as 2-norm Minkowski distances(Euclidean Distances).

$$||\vec{x} - \vec{v}||_2 = \left(\sum_{j=1}^{p} |x_j - v_j|^2\right)^{\frac{1}{2}}$$

The implementation below skips taking the root of the squared sum because it has no impact on the results.

```
function euclidian_distances(v1::Matrix{Int64}, v2::Matrix{Float64})
    v1 = [v1[i, :] for i in 1:size(v1,1)]
    v2 = [v2[i, :] for i in 1:size(v2,1)]
    s = size(v1,1)
    d = zeros((length(v1[1]),length(v2[2])))

    for j in 1:length(v2[2])
        for k in 1:length(v1[1])
            summed = 0
            for i in 1:s
                summed += (v1[i][k] - v2[i][j])^2
            end
            d[k, j] = summed
        end
    end

    return d'
end
```

```
d = euclidian_distances(data, V )
```

The stop condition used in our implementation is integrated with the main while loop. With every iteration, the Frobenius norm of the current and previous partition table is calculated and subtracted. If the result is smaller than an ε, the loop is terminated as it means that the algorithm is not improving and the results are not changing.

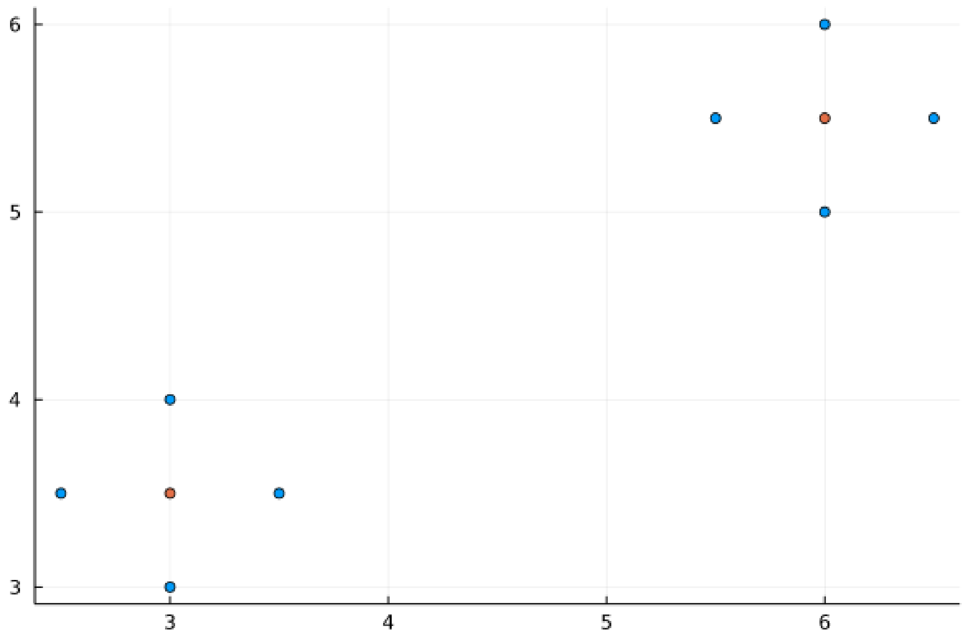
$$||A||_F \equiv \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$$

```julia
function frobenius_norm(m1::Matrix{Int64}, m2::Matrix{Int64})
    summed_first = 0
    summed_second = 0
    for i in 1:size(m1,2)
        for j in 1:size(m2,1)
            summed_first += m1[j,i]^2
            summed_second += m2[j,i]^2
        end
    end
    return abs(sqrt(summed_first) - sqrt(summed_second))
end
```

```julia
while frobenius_norm(U , U_prev) > ϵ
    V  = cluster_centers(U , data)
    d = euclidian_distances(data, V )
    U  = update_partition_table(U , d)
    append!(J, [criterion_function(U , d)])
end
```

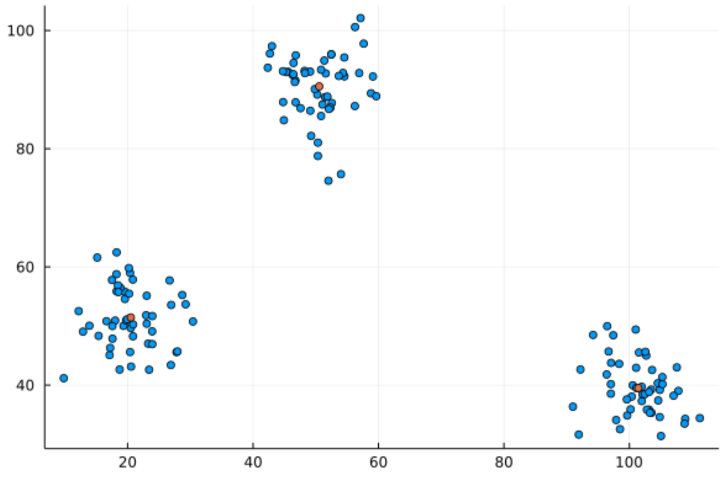
Results and overview

The hard c-means give sufficient results for clean data and appropriate clusters of samples. While using HCM on the data that follow the Gaussian distribution with visible boundaries one can expect satisfactory results.



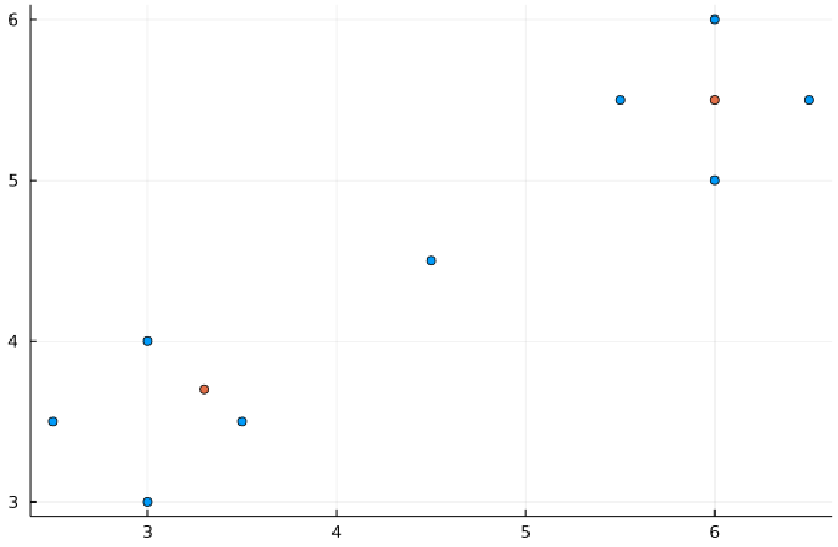
For the data generated by the given lines of code, the following clusters were created:

```
data = zeros(Float64, 2, 150)
for i in 1:50
    data[1, i] = rand(Normal(100, 5))
    data[2, i] = rand(Normal(40, 5))
end

for i in 50:100
    data[1, i] = rand(Normal(20, 4))
    data[2, i] = rand(Normal(50, 6))
end

for i in 100:150
    data[1, i] = rand(Normal(50, 5))
    data[2, i] = rand(Normal(90, 5))
end
```

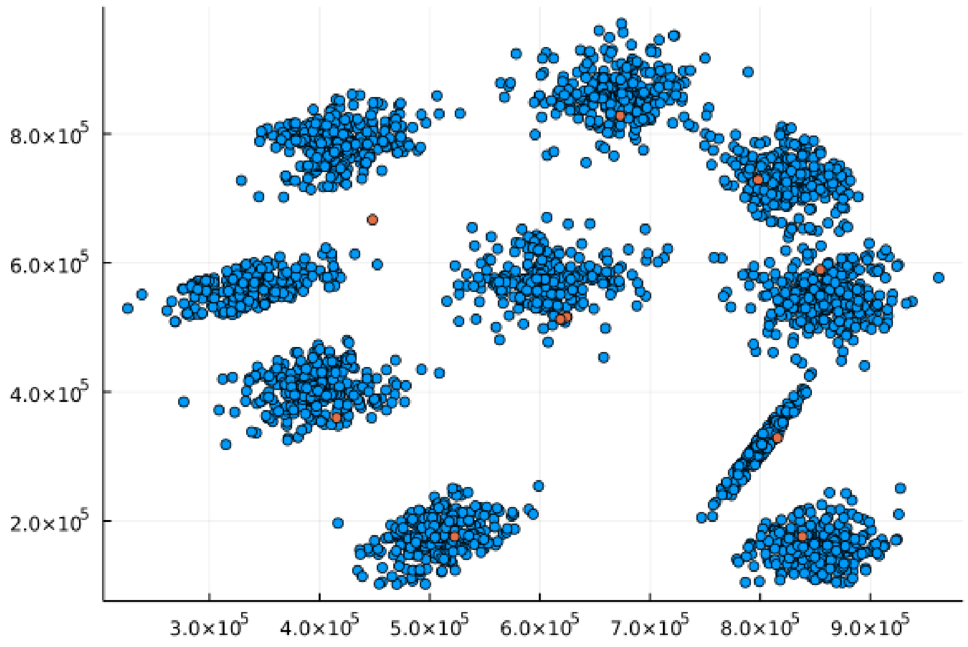


Outliers and Larger Datasets Handling

The HCM is not a very robust algorithm, the results are greatly influenced by the existence of the outliers. Taking as the example the point in equal distance between two clusters.
Hard c-means assigns this point to the first cluster in the partition table it was randomly introduced to. From the plot below, one can observe that the left-hand side cluster has the middle point in its group and the cluster center has shifted.



Another problem appears in fuzzy datasets where the clusters of data overlap or cross each other. The HCM desperately tries to solve the clusters with mediocre results.



3. FCM

The Fuzzy c-means and hard c-means share many similarities.
They have the same criterion functions, use the same algorithms for calculating distances, and are very similar algorithmically.

```
function fuzzy_c_means(data, number_of_clusters, m)
    ϵ = 1e-26
    U = zeros(Float64, number_of_clusters, size(data,2))

    U = init_partition_matrix(U)

    V = cluster_centers(U, data, m)

    distances = euclidian_distances(V, data)
    U_prev = U
    U = update_partition_table(U, distances, m)

    while abs(frobenius_norm(U_prev, U)) > ϵ
        V = cluster_centers(U, data, m)
        distances = euclidian_distances(V, data)
        U_prev = U
        U = update_partition_table(U, distances, m)
    end
    display(U)
    scatter(data[1,:], data[2,:], legend=false)
    scatter!(V[1, :], V[2, :])
end
```

The main difference between the HCM and FCM is **the fuzzy factor**. This parameter creates soft boundaries between the clusters and allows us to partially assign one data sample to multiple clusters at the same time. An intuitive example of this approach would be clustering the apple based on color. The HCM would correctly separate individual apples into the RED and GREEN clusters. The moment algorithm would analyze the fuzzy apple and force it into a strictly green or red cluster. On the other hand, FCM would accept the special properties of the fuzzy apple and assign it to both clusters partially (for example 50/50).

Partition Table:
2×9 Matrix{Float64}

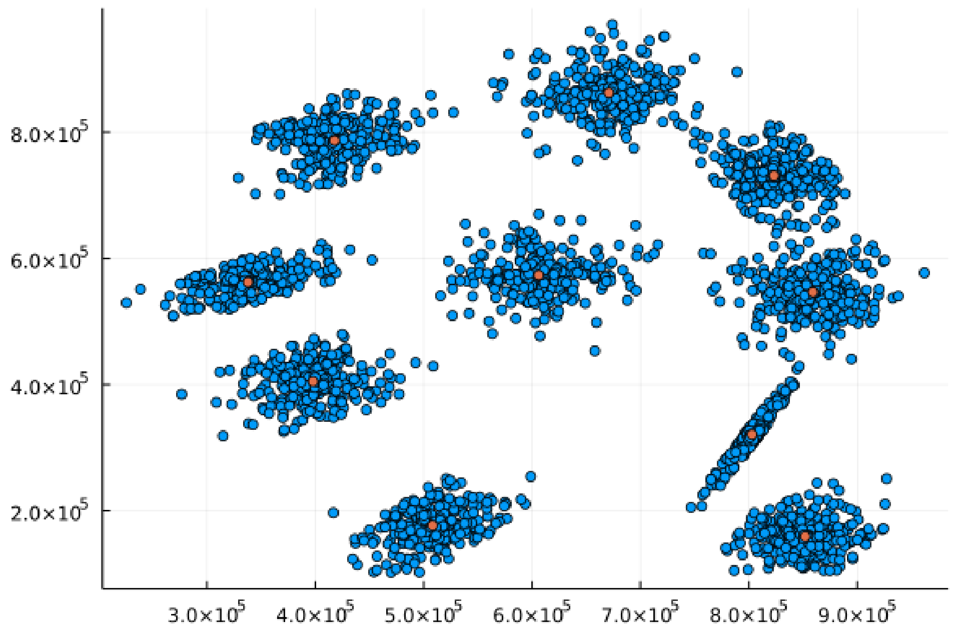
6.53523e-6	5.50471e-6	1.59481e-6	8.54356e-7	0.999999	0.999994	0.999998	0.999993	0.5
0.999993	0.999994	0.999998	0.999999	8.54356e-7	5.50471e-6	1.59481e-6	6.53523e-6	0.5

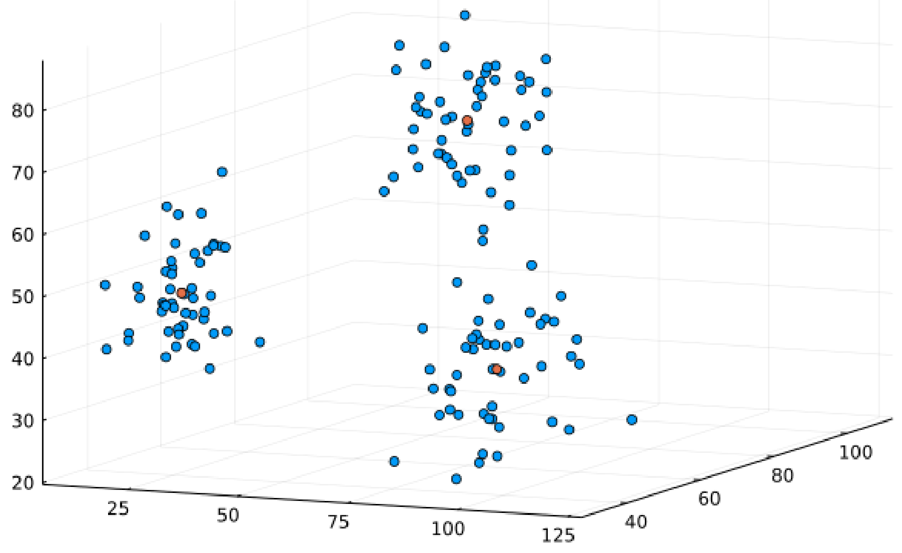
$$u_{ik} = \left[\sum_{j=1}^{c} \left(\frac{D_{ik}}{D_{jk}} \right)^{\frac{2}{m-1}} \right]^{-1}$$

```
function update_partition_table(partition_table, distances, m)
    partition_table = zeros(Float64, size(partition_table,1), size(partition_table,2))
    height = length(partition_table[:, 1])
    len = length(partition_table[1, :])
    for col in 1:len
        dist_sum = sum([x^(2/(1-m)) for x in distances[col, :]])
        if 1 in partition_table[:, col]
            set = findall(x->x==1, partition_table[:, col])
            for row in 1:height
                if row in set
                    partition_table[row, col] = 1
                else
                    partition_table[row, col] = 0
                end
            end
        else
            for row in 1:height
                partition_table[row, col] = (distances[col, row]^(2/(1-m)))/dist_sum
            end
        end

    end

    return partition_table
end
```

Results and overview

The FCM shows significant improvement in clustering for even more complicated datasets. Thanks to the fuzzy factor, the algorithm can assign the individual samples to multiple groups at the same time.

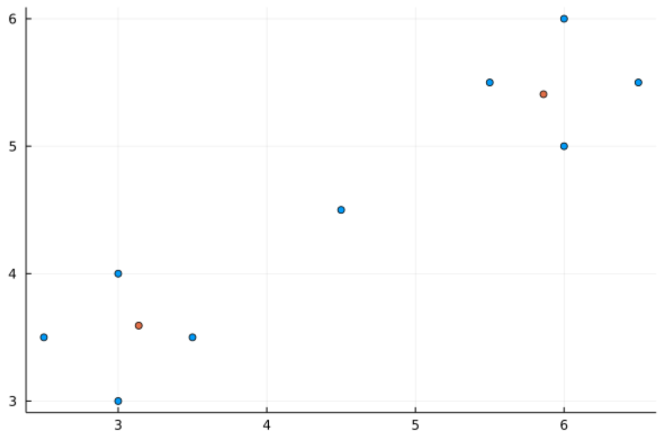




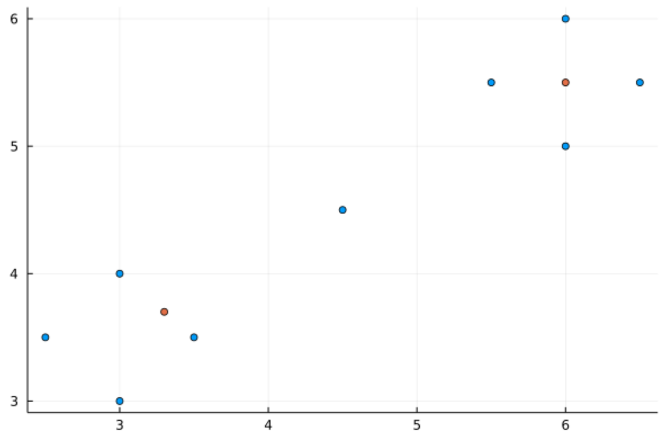
Handling outliers

The fuzzy c-means, even though works great with distributions and clean data sets, is not resistant to the outliers. In the comparison below one can see that the outlier in the middle affects the cluster centers evenly and more predictively than in the case of HCM.

Result of FCM clustering



Result of HCM clustering



Conclusion

Both of these centroid clustering algorithms do a great job at grouping unlabeled data. The Fuzzy C-Means show much better results in the case of bigger and more complicated datasets. Both FCM and HCM are susceptible to outliers, with FCM being more elegant in its approach to the problem.

To have more protection to the outliers, one can implement a more robust algorithm that uses more robust statistical methods like Fuzzy C-Medians(FCMED) or the MAD algorithm.

As an interesting improvement, in the next step after implementing the FCM, one could integrate it with the KNN classification algorithm to create an automated system of classification of new data to the unlabeled dataset.

HCM source code:

```
using Random
using Distributions
using CSV
using DataFrames
using Plots

# data = [randn((1,6)); randn((1,6))]
# data =  [2.5 3 3 3.5 5.5 6 6 6.5 ; 3.5 3 4 3.5 5.5 6 5 5.5]
# println(data)
# data = zeros(Float64, 2, 150)
# for i in 1:50
#     data[1, i] = rand(Normal(100, 5))
#     data[2, i] = rand(Normal(40, 5))
# end

# for i in 50:100
#     data[1, i] = rand(Normal(20, 4))
#     data[2, i] = rand(Normal(50, 6))
# end

# for i in 100:150
#     data[1, i] = rand(Normal(50, 5))
#     data[2, i] = rand(Normal(90, 5))
# end
dataset = CSV.read("data.csv", DataFrame)
data = rotr90(Matrix(dataset))
```

```
function cluster_centers(partition_table, data)

    size_of_data = size(data)
    v = zeros(size(partition_table, 1), size_of_data[1])

    for partition_table_row in 1:size(partition_table, 1)
        for data_row in 1:size_of_data[1]
            summed_nominator = 0
            summed_denominator = 0
            for entry_number in 1:size_of_data[2]
                summed_denominator += partition_table[partition_table_row, entry_number]
                summed_nominator += data[data_row,
entry_number]*partition_table[partition_table_row, entry_number]
            end
            v[partition_table_row, data_row] = summed_nominator/summed_denominator
        end
    end

    return rotr90(v)
end





function c_means(data, number_of_clusters)
    J = []
    ϵ = 1e-20
    U  = zeros(Int, number_of_clusters, size(data,2))
    U  = init_random(U )
    U_prev = U
    V  = cluster_centers(U , data)
    d = euclidian_distances(data, V )
    append!(J, [criterion_function(U , d)])
    U  = update_partition_table(U , d)
    while frobenius_norm(U , U_prev) > ϵ
        V  = cluster_centers(U , data)
        d = euclidian_distances(data, V )
        U_prev = U
        U  = update_partition_table(U , d)
        append!(J, [criterion_function(U , d)])
    end
    V  = cluster_centers(U , data)

    return (U , V )
end

function frobenius_norm(m1::Matrix{Int64}, m2::Matrix{Int64})
    summed_first = 0
    summed_second = 0
    for i in 1:size(m1,2)
        for j in 1:size(m2,1)
            summed_first += m1[j,i]^2
            summed_second += m2[j,i]^2
        end
    end
    return sqrt(summed_first) - sqrt(summed_second)
```

```julia
end


function euclidian_distances(v1::Matrix{Int64}, v2::Matrix{Float64})
    v1 = [v1[i, :] for i in 1:size(v1,1)]
    v2 = [v2[i, :] for i in 1:size(v2,1)]
    s = size(v1,1)
    d = zeros((length(v1[1]),length(v2[2])))

    for j in 1:length(v2[2])
        for k in 1:length(v1[1])
            summed = 0
            for i in 1:s
                summed += (v1[i][k] - v2[i][j])^2
            end
            d[ k, j] = sqrt(summed)
        end
    end

    return d'
end







function euclidian_distances(v1::Matrix, v2::Matrix)
    v1 = [v1[i, :] for i in 1:size(v1,1)]
    v2 = [v2[i, :] for i in 1:size(v2,1)]
    s = size(v1,1)
    d = zeros((length(v1[1]),length(v2[2])))

    for j in 1:length(v2[2])
        for k in 1:length(v1[1])
            summed = 0
            for i in 1:s
                summed += (v1[i][k] - v2[i][j])^2
            end
            d[ k, j] = summed
        end
    end

    return d'
end

function euclidian_distances(v1::Vector, v2)
    s = size(v1,1)
    d = zeros((length(v1[1]), s))

    for j in 1:length(v2[1])
        for k in 1:length(v1[1])
            summed = 0
            for i in 1:s
                summed += (v1[i][k] - v2[i][j])^2
            end
            d[k, j] = summed
        end
    end
    display(d)
    return d'
end

function update_partition_table(partition_table, distances)
    partition_table = zeros(Int, size(partition_table,1), size(partition_table,2))
    for i in 1:size(partition_table, 2)
        for k in 1:size(partition_table, 1)
            data = distances[:, i]
            if minimum(data) == distances[k, i]
                partition_table[k, i] = 1
                break
            end
        end
    end

    return partition_table
end









function criterion_function(partition_table, distances)
    J = 0
    for column in 1:size(partition_table,2)
        for row in 1:size(partition_table,1)
            J += partition_table[row, column]*distances[row, column]
        end
    end
    return J
end

function init_random(partition_table)
    height = length(eachrow(partition_table))
    for (i, _) in enumerate(eachcol(partition_table))
        partition_table[rand(1:height), i] = 1
    end

    for (i, row) in enumerate(eachrow(partition_table))
        if sum(row) == 0
            rand_col = rand(1:size(partition_table, 2))
            partition_table[:, rand_col] .= 0
            partition_table[i, rand_col] = 1
        end
        i = 1
    end
    return partition_table
end

function show_partition_matrix(partition_matrix)
    for (i,y) in enumerate(eachrow(partition_matrix))
        println(y, " Ω", i)
    end
    print(join([join((" x", i)) for i in 1:length(eachcol(partition_matrix))]), "\n")
```

```
end

function euclidian_distances_test()
    v1 = [2.5 3 3 3.5 5.5 6 6 6.5;3.5 3 4 3.5 5.5 6 5 5.5]
    v2 = [4.25 4.75;4.25 4.75]
    result = euclidian_distances(v1, v2)
    display(result)
end

function test_cluster_centers()
    v1 = [2.5 3 3 3.5 5.5 6 6 6.5;3.5 3 4 3.5 5.5 6 5 5.5]
    pt = [0 1 1 0 1 0 1 0 ; 1 0 0 1 0 0 0 0 ; 0 0 0 0 0 1 0 1]
    result = cluster_centers(pt, v1)
    display(result)
end

function test_criterion_function()
    J = [Inf]
    data = [2.5 3 3 3.5 5.5 6 6 6.5;3.5 3 4 3.5 5.5 6 5 5.5]
    U  = [0 1 1 0 1 0 1 0 ; 1 0 0 1 0 0 0 0 ; 0 0 0 0 0 1 0 1]
    show_partition_matrix(U )
    V  = cluster_centers(U , data)
    display(V )
    d = euclidian_distances(data, V )
    display(d)
    append!(J, [criterion_function(U , d)])
    println(J)
end
function test_table_update()
    data = [1 3 8 9 13 14 18 19; 0 1.5 8.5 9 13.5 14 18.5 19.3]
    U  = [0 1 0 0 1 0 0 0 ; 1 0 0 1 0 0 0 0 ; 0 0 0 0 0 1 0 1 ; 0 0 1 0 0 0 1 0]
    show_partition_matrix(U )
    V  = cluster_centers(U , data)
    d = euclidian_distances(data, V )
    update_partition_table(U , d)
    scatter(data[1,:], data[2,:], legend=false)
    scatter!(V [:, 1], V [:, 2])
end

# euclidian_distances_test()
# test_cluster_centers()
# test_criterion_function()
# test_table_update()

part_table, centers = c_means(data,13)
# display(centers)

scatter(data[1,:], data[2,:], legend=false)
scatter!(centers[1, :], centers[2, :])
```

The FCM source code

```julia
using Distributions
using Random
using CSV
using DataFrames
using Plots

data =  [2.5 3 3 3.5 5.5 6 6 6.5 4.5; 3.5 3 4 3.5 5.5 6 5 5.5 4.5]

# data = zeros(Float64, 3, 150)
# for i in 1:50
#     data[1, i] = rand(Normal(100, 10))
#     data[2, i] = rand(Normal(40, 5))
#     data[3, i] = rand(Normal(40, 10))
# end

# for i in 50:100
#     data[1, i] = rand(Normal(20, 4))
#     data[2, i] = rand(Normal(50, 6))
#     data[3, i] = rand(Normal(50, 10))
# end

# for i in 100:150
#     data[1, i] = rand(Normal(50, 5))
#     data[2, i] = rand(Normal(90, 10))
#     data[3, i] = rand(Normal(70, 8))
# end

function show_partition_matrix(partition_matrix)
    for (i,y) in enumerate(eachrow(partition_matrix))
```

```julia
        println(y, " Ω", i)
    end
    print(join([join((" x", i)) for i in 1:length(eachcol(partition_matrix))]), "\n")
end

function fuzzy_c_means(data, number_of_clusters, m)
    ϵ = 1e-26
    U = zeros(Float64, number_of_clusters, size(data,2))

    U = init_partition_matrix(U)

    V = cluster_centers(U, data, m)

    distances = euclidian_distances(V, data)
    U_prev = U
    U = update_partition_table(U, distances, m)

    while abs(frobenius_norm(U_prev, U)) > ϵ
        V = cluster_centers(U, data, m)
        distances = euclidian_distances(V, data)
        U_prev = U
        U = update_partition_table(U, distances, m)
    end
    display(U)
    scatter(data[1,:], data[2,:], legend=false)
    scatter!(V[1, :], V[2, :])
end

function init_partition_matrix(partition_table)
    height = length(partition_table[:, 1])
    len = length(partition_table[1, :])

    for col in 1:len
        for row in 1:rand(1:height)
            partition_table[rand(1:height), col] += 1
        end
    end

    for row in 1:height
        if sum(partition_table[row, :]) == 0
            partition_table[row, rand(1:height)] += 1
        end
    end

    for i in 1:len
        col_sum = sum(partition_table[:, i])
        partition_table[:, i] = partition_table[:, i] ./ col_sum
    end

    return partition_table
end

function init_random_partition_matrix(partition_matrix)
    for (k, member) in enumerate(eachcol(partition_matrix))
        resources = 1
        for i in 1:length(member)-1
            value = rand(Uniform(0, resources))
            resources -= value
            partition_matrix[i, k] = value
        end
        partition_matrix[end, k] = resources
    end

    return partition_matrix
end

function cluster_centers(partition_table, data)  #  prototype
    size_of_data = size(data)
    v = zeros(size(partition_table, 1), size_of_data[1])

    for partition_table_row in 1:size(partition_table, 1)
        for data_row in 1:size_of_data[1]
            summed_nominator = 0
            summed_denominator = 0
            for entry_number in 1:size_of_data[2]
                summed_denominator += partition_table[partition_table_row, entry_number]
                summed_nominator += data[data_row,
entry_number]*partition_table[partition_table_row, entry_number]
            end
            v[partition_table_row, data_row] = summed_nominator/summed_denominator
        end
    end

    return rotr90(v)
end



function cluster_centers(partition_table, data, m)
    size_of_data = size(data)
    v = zeros(size(partition_table, 1), size_of_data[1])

    for partition_table_row in 1:size(partition_table, 1)
        for data_row in 1:size_of_data[1]
            summed_nominator = 0
            summed_denominator = 0
            for entry_number in 1:size_of_data[2]
                fuzzied_pt = (partition_table[partition_table_row, entry_number]^m)
                summed_denominator += fuzzied_pt
                summed_nominator += data[data_row, entry_number]*fuzzied_pt
            end
            v[partition_table_row, data_row] = summed_nominator/summed_denominator
        end
    end
    return rotr90(v)
end

function frobenius_norm(m1::Matrix{Float64}, m2::Matrix{Float64})
    summed_first = 0
    summed_second = 0
    for i in 1:size(m1,2)
        for j in 1:size(m2,1)
            summed_first += m1[j,i]^2
            summed_second += m2[j,i]^2
        end
    end
    return sqrt(summed_first) - sqrt(summed_second)
end

function euclidian_distances(v1::Matrix{Float64}, v2)
```

```julia
    v1 = [v1[i, :] for i in 1:size(v1,1)]
    v2 = [v2[i, :] for i in 1:size(v2,1)]
    s = size(v1,1)
    d = zeros((length(v1[1]),length(v2[2])))

    for j in 1:length(v2[2])
        for k in 1:length(v1[1])
            summed = 0
            for i in 1:s
                summed += (v1[i][k] - v2[i][j])^2
            end
            d[ k, j] = sqrt(summed)
        end
    end

    return d'
end
```

```julia
function euclidian_distances_sqrd(v1::Matrix{Float64}, v2::Matrix{Float64})
    v1 = [v1[i, :] for i in 1:size(v1,1)]
    v2 = [v2[i, :] for i in 1:size(v2,1)]
    s = size(v1,1)
    d = zeros((length(v1[1]),length(v2[2])))

    for j in 1:length(v2[2])
        for k in 1:length(v1[1])
            summed = 0
            for i in 1:s
                summed += (v1[i][k] - v2[i][j])^2
            end
            d[ k, j] = summed
        end
    end

    return d'
end

function update_partition_table(partition_table, distances, m)
    partition_table = zeros(Float64, size(partition_table,1), size(partition_table,2))
    height = length(partition_table[:, 1])
    len = length(partition_table[1, :])

    for col in 1:len
        dist_sum = sum([x^(2/(1-m)) for x in distances[col, :]])
        if 1 in partition_table[:, col]
            set = findall(x->x==1, partition_table[:, col])
            for row in 1:height
                if row in set
                    partition_table[row, col] = 1
                else
```

```julia
                    partition_table[row, col] = 0
                end
            end
        else
            for row in 1:height
                partition_table[row, col] = (distances[col, row]^(2/(1-m)))/dist_sum
            end
        end

    end

    return partition_table
end

function criterion_function(partition_table, distances)
    J = 0
    for column in 1:size(partition_table,2)
        for row in 1:size(partition_table,1)
            J += partition_table[row, column]*distances[row, column]
        end
    end
    return J
end




function init_random(partition_table)
    height = length(eachrow(partition_table))
    for (i, _) in enumerate(eachcol(partition_table))
        partition_table[rand(1:height), i] = 1
    end

    for (i, row) in enumerate(eachrow(partition_table))
        if sum(row) == 0
            rand_col = rand(1:size(partition_table, 2))
            partition_table[:, rand_col] .= 0
            partition_table[i, rand_col] = 1
        end
        i = 1
    end
    return partition_table
end

fuzzy_c_means(data, 2, 1.3)
```