Lab 6

Albert Segarra Roca (S3255050), Carlos Humberto Paz Rodríguez (S3040577)

Group 16

Pattern Recognition

FMNS • RUG

October 26, 2016

1 Assignment 1

1.1

As we can see in the code provided in section 3.1 the implementation is straightforward. First, we initialize the prototypes with random points from the data, and we assign each point in data to a prototype by taking that with minimum distance to the point. Then, we start the loop by computing the average point among those of the same prototype, and we reassign all points to according to the new prototypes. This is looped until there is no difference between the previous assignment and the current one, meaning no point has been reassigned.

Note that we store and return a `prototypesHistory` which contains the list of prototype points for every iteration. This is later used to plot the arrows and intermediate positions of the prototypes.

The implementation for the plots can be found in section 3.2. For the data points and final prototypes it is straightforward and its results for $k = 2, 4, 8$ can be seen in Figures 1,2,3 respectively. Note that different colors indicate different clusters.

As we can see, for $K = 2$ the points are divided in two clear clusters, the one in the left and the one in the right. For $k = 4$ and $k = 6$, these two clusters are subdivided in 2 and 3 clusters respectively.

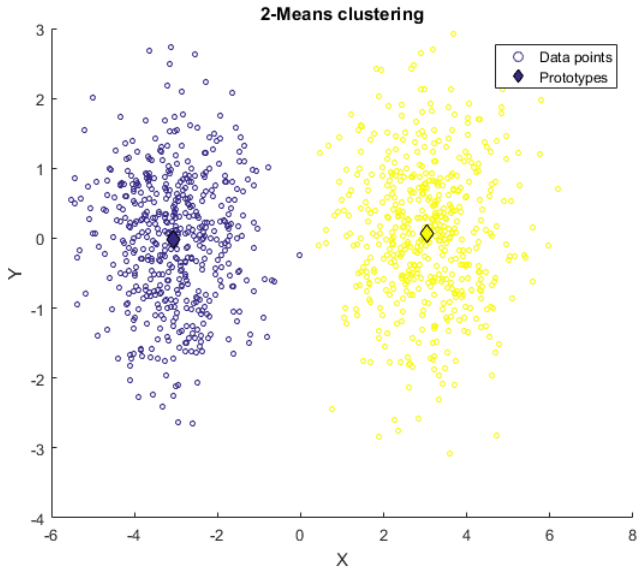

Figure 1: 2-means clustering data points and prototypes

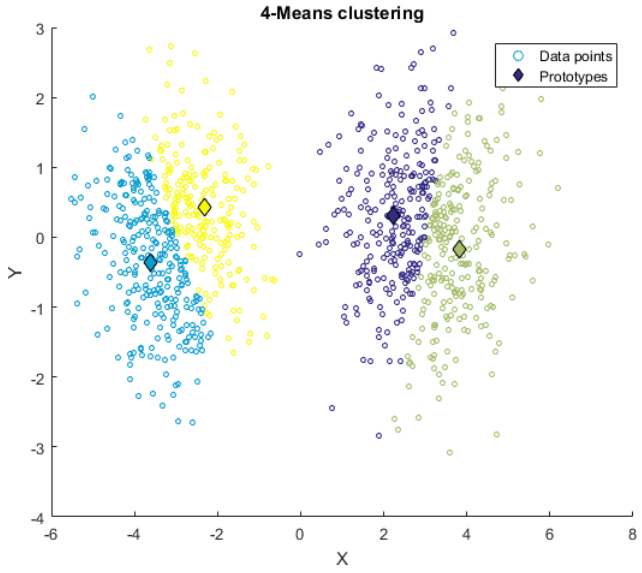

Figure 2: 4-means clustering data points and prototypes

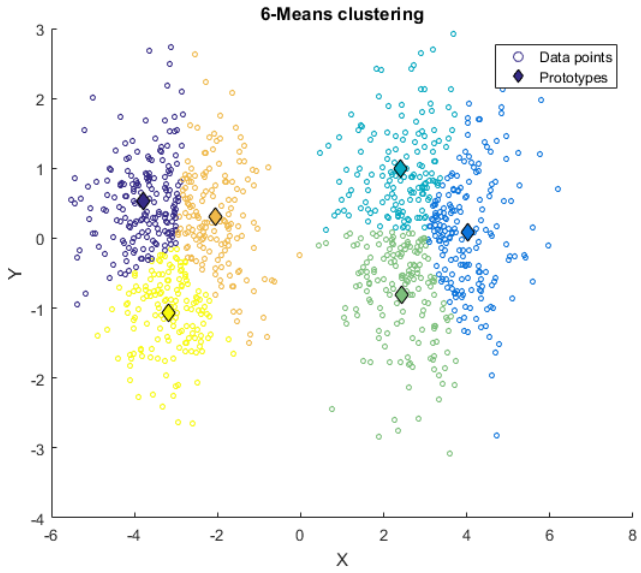

Figure 3: 6-means clustering data points and prototypes

In order to plot the prototype movements, we use the `prototypesHistory` returned by our `kmeans` function. We plot the initial, intermediate and final prototypes separately and with different markers to be able to differentiate them. For the intermediate prototypes, we loop through the `prototypesHistory` array and we print both a mark and an arrow from the previous prototype to the current prototype. We do so by using the `plotArrows` function, which plots an arrow from a prototype to another. Again, its results can be seen in Figures 4 ($k = 2$), 5 ($k = 4$) and 6 ($k = 6$).

We can see in this case how the prototypes quickly move to a much better prototype in the first step and then take smaller steps each time as the algorithm advances. We can also note in general that it takes more iterations for the algorithm to converge for bigger `K`.

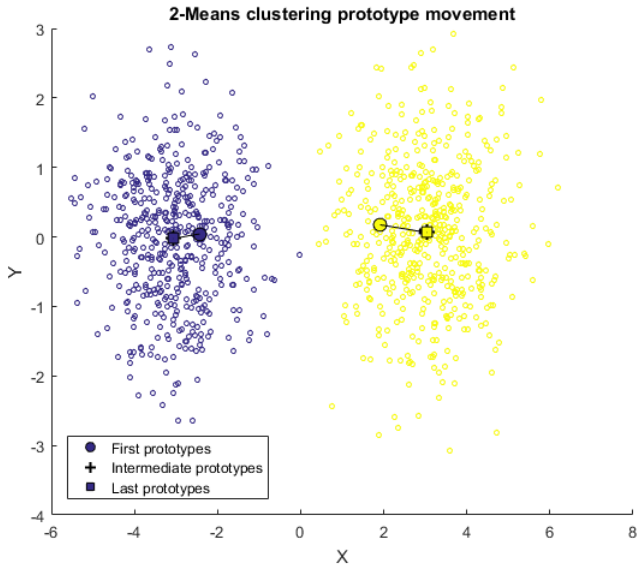

Figure 4: 2-means clustering prototype movement

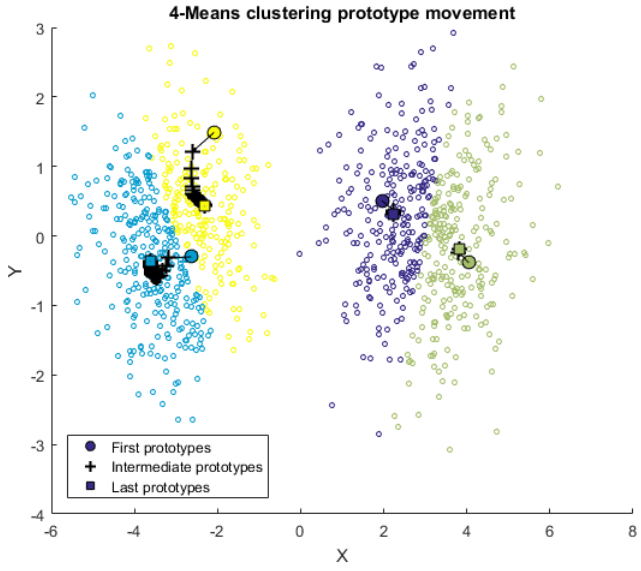

Figure 5: 4-means clustering prototype movement

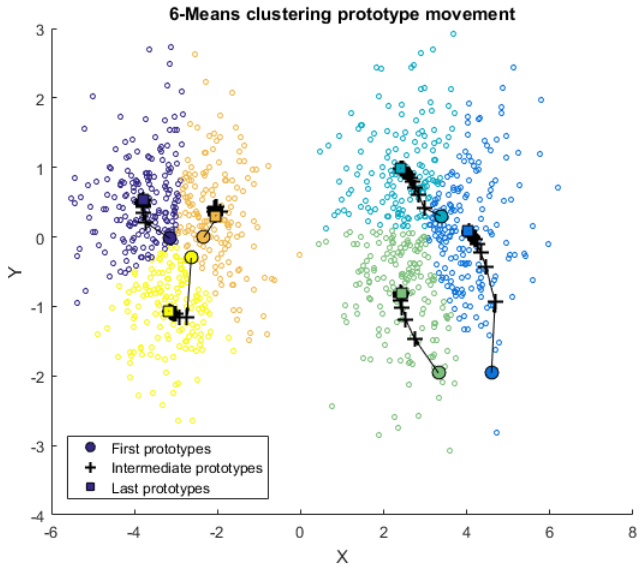

Figure 6: 6-means clustering prototype movement

1.2

In order to compute $J(k)$, the quantization error function, we need to compute half the sum of squares of euclidean distances between each point and its prototype. Then we compute $R(k)$ and $D(k)$ with the straightforward formulas indicated in the statement. As we can see in the code provided in section 3.3, we have chosen k_{max} to be 10, which we found to be a reasonably high value for the number of clusters such that higher values would not result in better clustering. In order to have more precise results, we compute each $J(k)$ as the average of the quantization error for 10 repetitions of the `k-means` algorithm, as the results can vary depending on the initialization of the prototypes.

a) As we can see in the plot of $D(k)$ (Figure 7) the function has a maximum for $k = 2$, and does roughly only decrease for higher values of k. We can conclude then that the optimal value for the number of clusters for this dataset is $k_{opt} = 2$. This makes sense if we look at figure 1, as there is a clear distinction between points belonging to the left cluster and the right cluster. Adding more clusters does not make many sense in this dataset, because it does only partition this two main clusters in smaller pieces.

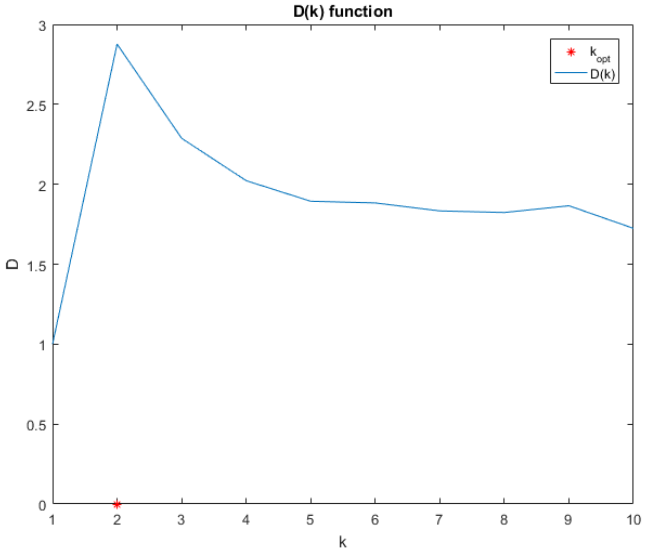

Figure 7: $D(k)$ function with k_{opt}

b) Also when we plot $J(k)$ and $R(k)$ (Figure 8) we see how k_{opt} is achieved when both functions have the biggest distance and $J(k)$ is below $R(k)$, thus $D(k) = \frac{R(k)}{J(k)}$ is optimized because $R(k)$ has the highest value as compared to $J(k)$.

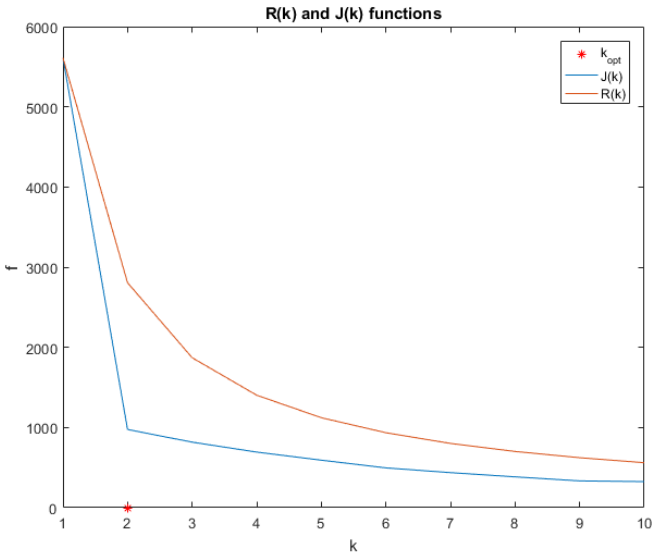

Figure 8: $R(k)$ and $J(k)$ functions with k_{opt}

1.3

a) In order to implement the `k-means++` initialization, we use the algorithm described in the statement. The only interesting non-trivial step is how to pick a data point as a new prototype with probability proportional to the distance to its closest prototype. As seen in function `selectPrototypeDx` in section 3.1 (lines 14-32), to accomplish that we first pick a random number R in the range $(0, sum(D))$. This random number is then used to pick a point X in the cumulative function D_c of D such that this $D_c(X) \geq R$. Because each point X takes a span of the range of the random number which is proportional to $D(X)$, the point has a probability to be selected which is proportional to its span, thus proportional to $D(X)$.

b) To compare the two algorithms, we run them both 20 times and compute the mean and standard deviations of their quantization errors. The implementation can be found in section 3.4.

   (a) `k-means` **quantization error:**
   Mean: **8.3610**
   Standard deviation: **0.6889**

   (b) `k-means++` **quantization error:**
   Mean: **6.7081**
   Standard deviation: **0.4615**

   Now as we can see, the `k-means++` algorithm clearly has a lower mean quantization error than the `k-means` algorithm, although the difference is not very big. This is because k-means++ uses a better initialization for the prototypes, which is actually very important for the algorithm because it tends to get stuck in local optima.

c) Now, to compute the desired p-value and null hyphotesis test, we use Matlab's `ttest2` function, for a two-sample t-student test, without the assumption of equal variances (Welch's t-test). The resulting p-value is with the null hypothesis that the two data vectors are from populations with equal mean.

   p-value: 2.5151e-10

   As we can see, this value is very low, which indicates that the probability that they have equal means is very low. Also, with the first return value of `ttest2`, which returns 1 in our case, we can reject the null hyphotesis (equal means) with a significance of 5% that and thus with that significance we can say that the means are different. Because `k-means++` quantization error mean is lower we can conclude that `k-means++` outperforms `k-means` algorithm.

2 Assignment 2

2.1

The full implementation of the batch neural gas algorithm can be found in section 3.5. The only non-trivial step is explained in the comments, which is the computation of the ranking for each prototype, for which sorting is used to compute the number of prototypes with smaller distance to the point.

2.2

In order to make all the plots, we used the function provided in section 3.6.

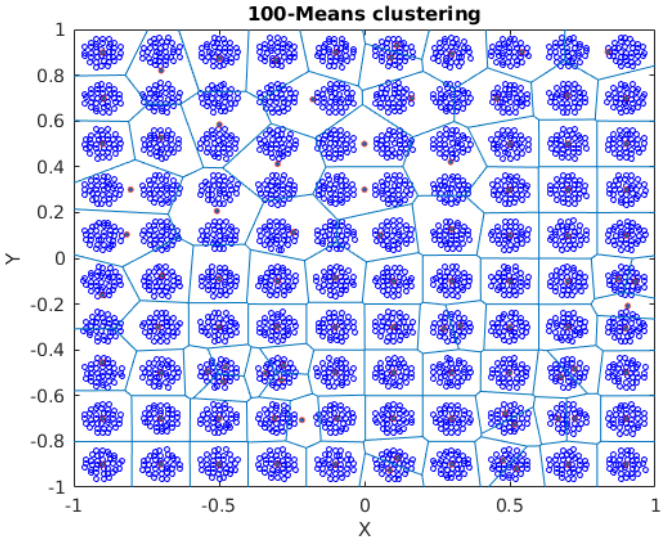

Figure 9: 100-means clustering for checkerboard data

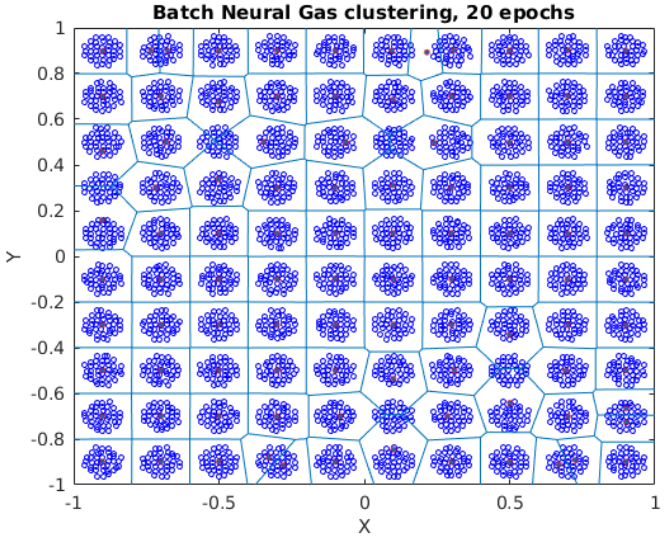

Figure 10: Batch Neural Gas clustering with 20 epochs for checkerboard data

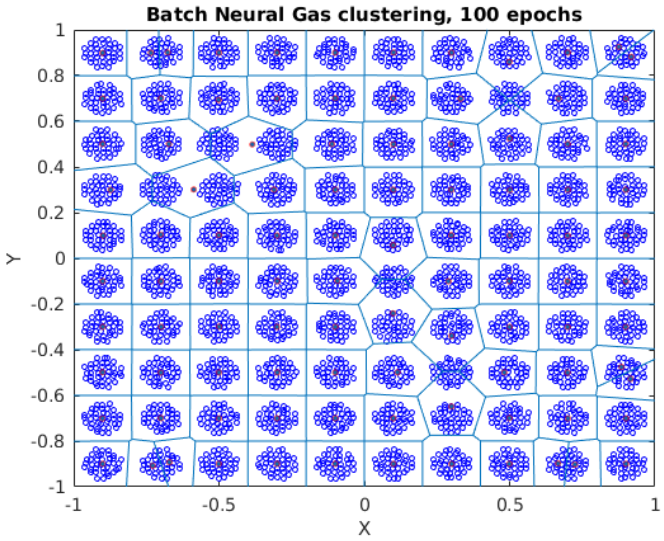

Figure 11: Batch Neural Gas clustering with 100 epochs for checkerboard data

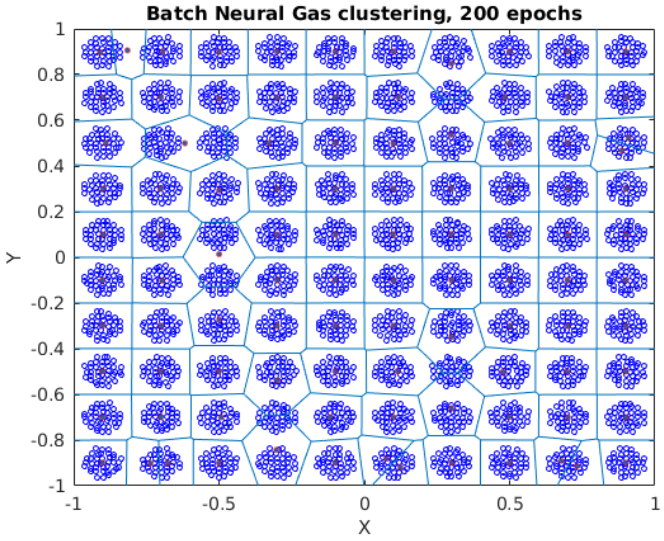

Figure 12: Batch Neural Gas clustering with 200 epochs for checkerboard data

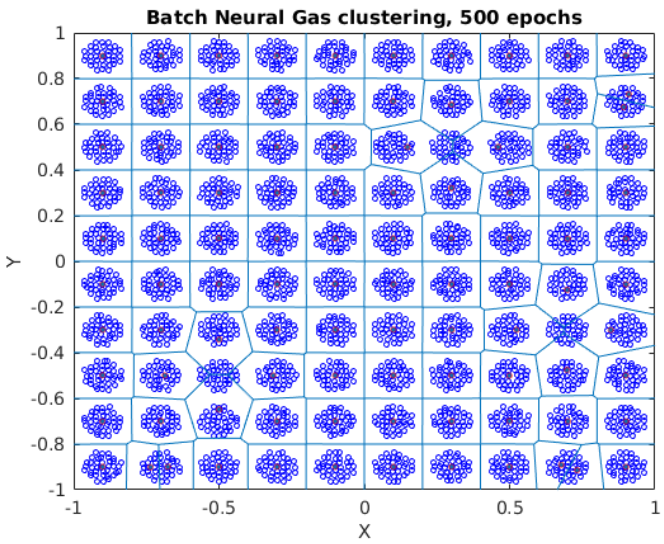

Figure 13: Batch Neural Gas clustering with 500 epochs for checkerboard data

2.3

As we can see by comparing figures 9 and 10, even with only 20 epochs the batch neural gas algorithm already outputs an apparently better clustering than the k-means algorithm. The results get even better as we increase the number of epochs for the `BNG` algorithm, as we can see by comparing figures 10, 11, 12 and 13, although it doesn't improve very significantly.

The difference between both algorithms is that `k-means` tries to minimize directly the quantization error, whereas `BNG` minimizes a cost function instead with a gradient descent.

The improvement seen in the `BNG` algorithm as compared to the `k-means` algorithm is because the latter usually gets stuck in local optima, and it is difficult for it to find the global optimum. Also, it is much more sensitive to the initialization of the prototypes, as compared to the `BNG` algorithm, and also it is very sensitive to outliers and the variation of size and density of the clusters, which makes it not adequate for some data sets. Both algorithms share the disadvantage that we need to give it the number of desired clusters, which is sometimes difficult to guess beforehand.

So, generally, the `BNG` algorithm gives better results, although `k-means` algorithm works quite well for gaussian-shaped clusters, but still the `BNG` algorithm is much more computationally expensive than the `k-means` algorithm, which is generally quite fast.

3 Appendix

3.1 `k-means` function implementation (with `k-means++`)

```
function [classes, prototypesHistory] = kmeans(data, k, is_kmeanspp, initialize_file)
    if nargin < 4
       initialize_file = false;
       if nargin < 3
           is_kmeanspp = false;
       end
    end

    function classes = reassign(data, prototypes)
        % Compute distances between every point and prototype
        distances = pdist2(data, prototypes);
        % For every point, get the index of the prototype with min distance
        [~, classes] = min(distances, [], 2);
    end

    function prototype = selectPrototypeDx(Dsq)
        % Selects a prototype x with probability proportional to
        % D(x, closestPrototype(x))
        sumDistances = sum(Dsq);
        randX = rand*sumDistances; % Get random in range (0, sumDistances)
        i = 1;
        accSum = Dsq(i);

        while randX > accSum
            i = i + 1;
            accSum = accSum + Dsq(i);
        end

        % Pick prototype in the position X where the cumulative sum is
        % greater than the random. Obviously elements with greater D(x)
        % will span a bigger portion of the range of the random number,
        % and thus the probability of selecting them will be higher
        prototype = data(i, :);
    end

n = size(data, 1);

prototypesHistory = [];

if is_kmeanspp
    % k-means++ prototype initialization
    prototypes = zeros(k, 2);
    prototypes(1, :) = data(randi(n), :);
    diff = data - repmat(prototypes(1, :), n, 1);
    % Not using pdist^2 cause it would be redundant and inneficient
    Dsq = diff(:, 1).^2 + diff(:, 2).^2;

    for prototypeIndex = 2:k-1
        prototypes(prototypeIndex, :) = selectPrototypeDx(Dsq);
        classes = reassign(data, prototypes(1:prototypeIndex, :));
        diff = prototypes(classes, :) - data;
        Dsq = diff(:, 1).^2 + diff(:, 2).^2;
    end

    if k > 1
        prototypes(k, :) = selectPrototypeDx(Dsq);
    end
elseif initialize_file
    sbrace = @(x,y)(x{y});
    fromfile = @(x)(sbrace(struct2cell(load(x)),1));
    prototypes=fromfile('clusterCentroids.mat');
```

```
else
    % Compute initial prototypes selecting k random points from data
    prototypes = data(randi(n, k, 1), :);

end

% Assign points to the prototype with minimum distance
classes = reassign(data, prototypes);

someReassigned = true;
it = 1;
while someReassigned
  prototypesHistory(:, :, it) = prototypes;
  it = it + 1;

  prototypes = zeros(k, 2); % Assume 2D data
  for i = 1:n % Compute sum vector of all vectors belonging to a specific prototype
      prototypes(classes(i), :) = prototypes(classes(i), :) + data(i, :);
  end

  % Compute mean vector dividing the sum vector by the number of points in
  % the prototype
  prototypes = prototypes ./ repmat(hist(classes, 1:k)', 1, 2);

  auxClasses = classes;
  classes = reassign(data, prototypes);
  someReassigned = ~isequal(auxClasses, classes);
end
prototypesHistory(:, :, it) = prototypes;

end
```

3.2 Plotting functions for exercise 1.1

```
function plotArrows(previousPrototypes, prototypes)
  for i = 1:k
    plot_arrow(previousPrototypes(i, 1), previousPrototypes(i, 2), prototypes(i, 1),
        prototypes(i, 2));
  end
end
```

```
function kmeansplotter(K)
data = load('kmeans1');
data = data.kmeans1;

[classes, prototypesHistory] = kmeans(data, K);
iterations = size(prototypesHistory, 3);

% Plot 1: points and prototypes
figure(1);
scatter(data(:, 1), data(:, 2), 10, classes, 'DisplayName', 'Data points');
hold on;
lastPrototypes = prototypesHistory(:, :, iterations);
scatter(lastPrototypes(:, 1), lastPrototypes(:, 2), ...
        70, [1:K]', 'filled', 'd', 'MarkerEdgeColor',[0 0 0], 'DisplayName', 'Prototypes');
xlabel('X');
ylabel('Y');
tt = sprintf('%d-Means clustering', K);
title(tt);
legend('show');
```

```
hold off;

% Plot 2: prototype movement
figure(2);
scatter(data(:, 1), data(:, 2), 10, classes);
hold on;
firstPrototypes = prototypesHistory(:, :, 1);
firstScatter = scatter(firstPrototypes(:, 1), firstPrototypes(:, 2), ...
        70, [1:K]', 'filled', 'o', 'MarkerEdgeColor',[0 0 0]);

for i = 2:iterations-1
    previousPrototypes = prototypesHistory(:, :, i - 1);
    prototypes = prototypesHistory(:, :, i);

    intermediateScatter = scatter(prototypes(:, 1), prototypes(:, 2), ...
                    70, '+', 'MarkerEdgeColor', [0 0 0], 'LineWidth', 1.5);
    plotArrows(previousPrototypes, prototypes);
end

lastScatter = scatter(lastPrototypes(:, 1), lastPrototypes(:, 2), ...
        70, [1:K]', 'filled', 's', 'MarkerEdgeColor', [0 0 0]);
plotArrows(prototypesHistory(:, :, iterations - 1), lastPrototypes);

xlabel('X');
ylabel('Y');
tt = sprintf('%d-Means clustering prototype movement', K);
title(tt);

if K == 1
    legend([firstScatter lastScatter], ...
            'First prototypes', 'Last prototypes', 'Location', 'southwest');
else
    legend([firstScatter intermediateScatter lastScatter], ...
            'First prototypes', 'Intermediate prototypes', 'Last prototypes', 'Location', '
               southwest');
end
hold off;
```

3.3 Function for quantization error analysis and plotting

```
function qe = quantization_error(data, classes, prototypes)
    % I do not use pdist because it would be redundant and inefficient to
    % square a sqrt
    qe = sum((data(:, 1) - prototypes(classes, 1)).^2 + ...
                            (data(:, 2) - prototypes(classes, 2)).^2)/2;
end
```

```
function kmeansquantization()
REPETITIONS = 30;

data = load('kmeans1');
data = data.kmeans1;
n = size(data, 1);

k_max = min(10, n);

J = zeros(k_max, 1);
for k = 1:k_max
    for time = 1:REPETITIONS
        [classes, prototypesHistory] = kmeans(data, k);
```

```
        lastPrototypes = prototypesHistory(:, :, size(prototypesHistory, 3));
        J(k) = J(k) + quantization_error(data, classes, lastPrototypes);
    end
end

J = J ./ REPETITIONS;

R = J(1).*(1./[1:k]'); % J(1)*k^(-2/d), d=2

D = R./J;

[~, k_opt] = max(D);

% Plot 1, D function with k_opt
figure(1);
d_plot = plot(D);
hold on;
k_opt_plot = plot(k_opt, 0, 'r*');
title('D(k) function');
xlabel('k');
ylabel('D');
xlim([1 10]);
legend([k_opt_plot d_plot], 'k_{opt}', 'D(k)');
hold off;

% Plot 2, R and J function with k_opt
figure(2);
j_plot = plot(J);
hold on;
r_plot = plot(R);
k_opt_plot = plot(k_opt, 0, 'r*');
title('R(k) and J(k) functions');
xlabel('k');
ylabel('f');
xlim([1 10]);
legend([k_opt_plot j_plot r_plot], 'k_{opt}', 'J(k)', 'R(k)');
hold off;
end
```

3.4 Script to compare `k-means` with `k-means++`

```
function qe = quantization_error(data, classes, prototypes)
    % I do not use pdist because it would be redundant and inefficient to
    % square a sqrt
    qe = sum((data(:, 1) - prototypes(classes, 1)).^2 + ...
                            (data(:, 2) - prototypes(classes, 2)).^2)/2;
end
```

```
K = 100;
RUNS = 20;

load checkerboard.mat

data = checkerboard;

J20 = zeros(RUNS, 1);
J20pp = zeros(RUNS, 1);
for i = 1:RUNS
    [classes, prototypesHistory] = kmeans(data, K, false);
    lastPrototypes = prototypesHistory(:, :, size(prototypesHistory, 3));
```

```
    J20(i) = quantization_error(data, classes, lastPrototypes);

    [classes, prototypesHistory] = kmeans(data, K, true);
    lastPrototypes = prototypesHistory(:, :, size(prototypesHistory, 3));
    J20pp(i) = quantization_error(data, classes, lastPrototypes);
end

% Exercise b)
J20mean = mean(J20);
J20std = std(J20);

J20ppmean = mean(J20pp);
J20ppstd = std(J20pp);

% Exercise c)
[~, p] = ttest2(J20, J20pp, 'Vartype', 'unequal');

disp('Quantization error k-means:');
disp('Mean:');
disp(J20mean);
disp('Std:');
disp(J20std);

disp('Quantization error k-means++:');
disp('Mean:');
disp(J20ppmean);
disp('Std:');
disp(J20ppstd);

disp('p-value');
disp(p);
```

3.5 Implementation of the Batch Neural Gas algorithm

```
function [prototypes] = batchNG(Data, n, epochs, xdim, ydim)

% Batch Neural Gas
%   Data contains data,
%   n is the number of clusters,
%   epoch the number of iterations,
%   xdim and ydim are the dimensions to be plotted, default xdim=1,ydim=2

error(nargchk(3, 5, nargin));  % check the number of input arguments
if (nargin<4)
  xdim=1; ydim=2;   % default plot values
end;

[dlen,dim] = size(Data);

%prototypes =  % small initial values
%% or
sbrace = @(x,y)(x{y});
fromfile = @(x)(sbrace(struct2cell(load(x)),1));
prototypes=fromfile('clusterCentroids.mat');

lambda0 = n/2; %initial neighborhood value
% lambda
lambda = lambda0 * (0.01/lambda0).^([0:(epochs-1)]/epochs);
% note: the lecture slides refer to this parameter as sigma^2
%       instead of lambda
```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Action

for i=1:epochs,
  D_prototypes = zeros(n,dim);   % difference for vectors is initially zero
  D_prototypes_av = zeros(n,1);       % the same holds for the quotients

  for j=1:dlen  % consider all points at once for the batch update
    % sample vector
    x = Data(j,:); % sample vector
    X = x(ones(n,1),:);  % we'll need this

    % neighborhood ranking

    diffV = X - prototypes;
    % We don't need the exact value of the distance, just a magnitude,
    % so the distance squared is enough and more efficient to compute
    distancesSq = diffV(:, 1).^2 + diffV(:, 2).^2;
    [~, sortedPrototypeIndices] = sort(distancesSq);

    % Ranking(i) is the index of the prototype in the sorted
    % array
    % Note that this is the number of prototypes with a distance
    % smaller to the current data point because its sorted increasingly
    ranking(sortedPrototypeIndices) = 1:n;


    % DISTANCE!!!
    % 1-BMU, 2-BMU, etc. (hint:sort)
    %find ranking,h,H

    % accumulate update
    aux = exp(-ranking/lambda(i))';
    D_prototypes = D_prototypes + repmat(aux, 1, 2).*X;
    D_prototypes_av = D_prototypes_av + aux;
  end

  prototypes = D_prototypes./repmat(D_prototypes_av, 1, 2);

  % track
  if 1
    fprintf(1,'%d / %d \r',i,epochs);
    hold off
    plot(Data(:,xdim),Data(:,ydim),'bo','markersize',3)
    hold on
    plot(prototypes(:,xdim),prototypes(:,ydim),'r.','markersize',10,'linewidth',3)
    % write code to plot decision boundaries
    voronoi(prototypes(:, 1), prototypes(:, 2));
    %pause
    %or
    drawnow
  end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

3.6 Plotter to compare `kmeans` and `batch neural gas`

```
function kmeansbatchplotter()
    function plotBatch(epochs)
```

```
3           batchNG(data, K, epochs);
4           xlabel('X');
5           ylabel('Y');
6           tt = sprintf('Batch Neural Gas clustering, %d epochs', epochs);
7           title(tt);
8       end
9
10  K = 100;
11
12  data = load('checkerboard');
13  data = data.checkerboard;
14
15  % Plot for k-means
16  figure(1);
17  [~, prototypesHistory] = kmeans(data, K, false, true);
18  iterations = size(prototypesHistory, 3);
19  lastPrototypes = prototypesHistory(:, :, iterations);
20  plot(data(:,1),data(:,2),'bo','markersize',3)
21  hold on
22  plot(lastPrototypes(:,1),lastPrototypes(:,2),'r.','markersize',10,'linewidth',3)
23  voronoi(lastPrototypes(:, 1), lastPrototypes(:, 2));
24  xlabel('X');
25  ylabel('Y');
26  tt = sprintf('%d-Means clustering', K);
27  title(tt);
28  hold off;
29  pause;
30
31  % Plot for batch neural 20
32  figure(2);
33  plotBatch(20);
34  pause;
35
36  % Plot for batch neural 100
37  figure(3);
38  plotBatch(100);
39  pause;
40
41  % Plot for batch neural 200
42  figure(4);
43  plotBatch(200);
44  pause;
45
46  % Plot for batch neural 500
47  figure(5);
48  plotBatch(500);
49
50  end
```