

[Studenti](#)[Ricerca](#)[Ateneo](#)[Servizi online](#)[Intranet](#)[Aulaweb](#)

[Dibris](#) ► [My courses](#) ► [Robotics Engineering \(Scuola Politecnica\)](#) ► [Anno Accademico 2013/14](#) ► [65862-1314](#) ► [Lab assignments](#) ► [Lab assignment 5: Unsupervised learning](#)
You are logged in as **Muhammad Farhan Ahmed (Logout)**

Navigation

[Dibris](#)[■ My home](#)[Dibris](#)[My profile](#)[Current course](#)[65862-1314](#)[Participants](#)[Badges](#)[General](#)[Activity dates and topics](#)[Notes and slides](#)[Lab assignments](#)[📄 Lab submission guidelines](#)[📁 Lab assignment 1: one-layer perceptrons](#)[📁 ...ent 2: Adaline and non-linearly separable problems](#)[📢 Feedback on course workload](#)

Lab assignment 5: Unsupervised learning

- Task 1: K-means clustering
- Task 2: Kohonen's Self-Organizing Map
- Task 3 (optional): Neural Gas

In this exercise, you will program methods for unsupervised learning. The assignment description is lengthy because there are several suggestions that should guide you while programming the algorithms.

As in the past multi-week assignment, it is advisable to concentrate on task 1 only, during the first week, and proceed to task 2 (and maybe 3) in the following weeks.

As usual, describe everything in a report.

Task 1: K-means clustering

Goal: Write a function implementing the k means clustering algorithm.

The function should receive two input arguments: a training set and a number of clusters, k . No target: it is an unsupervised method!

In Matlab the function should be defined as:


```
function [ y u ] = kmeans(x,k)
```


where y is a matrix of prototype vectors and u a membership matrix that represent the assignment of points to clusters (see below). If the input vectors are np and of size d , then y will be $k \times d$ and u will be $np \times k$

How to structure your function: The algorithm works by epoch, and its structure is the familiar one: initialize - loop - finalize.

The init phase is an initial value for y . It can be done:

- randomly (but it is advisable to stay inside the "cloud" of points of the training set)

 Lab assignment 3: Prototype-based classifiers

 Lab assignment 4: Training multi-layer networks

 Lab assignment 5: Unsupervised learning

Lab resources

General resources

My courses

Courses

Administration  

Course administration

My profile settings

- selecting k random points from the training set
- selecting k points from the training set according to some criterion
- selecting k suitable points that, because of prior knowledge, we know could be reasonable positions for the centroids
- ... (your idea here) ...

REMARK: The problem of selecting the value of k is a model order selection problem similar to choosing nh in multi-layer perceptrons, i.e., has many solutions and no one of them is theoretically optimal.

The loop consists of two alternate actions:

1. Starting from the current value of y , assign each point in the training set (row of x) to its nearest prototype (row of y) (see below for how to represent this assignment).
2. Starting from the current assignment, recompute each row of y as the mean of the points assigned to it (see below).

The stopping criterion is: no change happens. In other words, if either of the two above actions does not produce any change, then the other will not either, so it is useless to keep on looping, and we stop. (Of course, the usual "emergency" criterion of max.number of iterations should also be included.)

Finalization is empty, unless you want to print any final message.

How to structure your variables: A practical implementation can use a **membership** or **indicator vector** for each point. For a given point, its membership vector u has size equal to k . The point is assigned to prototype j (and therefore to cluster j) if $u(j) == 1$, and $u(k) == 0$ for all other k , $k \neq j$.

In Matlab, the memberships as a whole can better be represented as a $n \times k$ matrix u . Each row of u contains the membership vector for a point.

Example:

```
point 1: u(1,:) = [ 0 0 1 0 0 ] % point 1 is assigned to cluster 3
point 2: u(2,:) = [ 1 0 0 0 0 ] % point 2 is assigned to cluster 1
```

So the assignment phase consists in checking what is the nearest centroid to each point in the training set, and writing a 1 in the corresponding column of u , and zeros everywhere else.

Computation of the nearest prototype: In which column of u should I put a 1? You should already know how to do this. In Matlab, you can use the function `dist2.m` and the fact that the Matlab `min` function returns, in addition to the minimum, also its index.

Computation of the means: the j -th row of y is the mean of all points in the training set which have a 1 in the j -th position of the corresponding membership vector.

So the j -th column of u acts as a "filter", indicating all inputs that are assigned to prototype j . We can take the mean of the whole training set where each point is **multiplied by its membership**. In this way, points with zero membership will not contribute to the mean:

$$\mathbf{y}_j = \frac{1}{\sum_{l=1}^n u_{lj}} \sum_{l=1}^n u_{lj} \mathbf{x}_l$$

and you can sum all points, knowing that only those which are not multiplied by 0 will contribute to the summation.

In Matlab this mean can be computed very efficiently:

```
y = u' * x; % this is a matrix with k rows; each row is a sum of points
s = sum(u); % this is a vector with k entries; each entry is the number of points in a cluster
for j = 1:size(y,1)
    y(j,:)=y(j,:)./s(j); % this turns each sum into a mean
end
```

NOTE: Despite this extensive description, the resulting Matlab code should be quite short, about 30 lines plus the dist2 code (already given).

Test: run the k means program on the [Iris data](#) (reminder: discard last column, the target). Experiment with k from 2 to 5. Plot the resulting centroids over the data and see where they end up. How many clusters are there really? (Plot the data without class-related colors, and use two of the four inputs as you did in lab 1.)

Do some repeated experiments for each value of k to check how the solutions differ. Put some representative images in the report to support the description of your findings.

To objectively check the progress of the algorithm, you can compute the (squared) distortion:

$$D = \sum_{l=1}^n \sum_{j=1}^k u_{lj} \|\mathbf{x}_l - \mathbf{y}_j\|^2$$

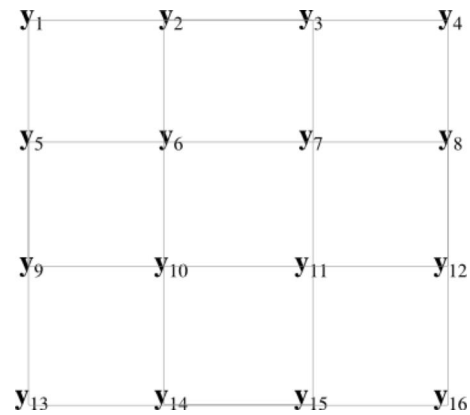
and print it at each epoch and at the end.

Task 2: Kohonen's Self-Organizing Map

Rationale: Experimenting with k means, you may have experienced the fact that repeated experiments result in different solutions. This means that the minimization procedure gets caught in local minima of the cost function, and you can see this by comparing the final distortions. Kohonen's Self-Organizing Map, despite being an attempt to build a biologically plausible model inspired by how our eye's retina works, in fact is an optimization method that introduces at least two remedies to his phenomenon.

- The first is learning by pattern. Updates occur at each pattern, and the learning rate η is decreased at each iteration.
- The second is involving not only the nearest prototype (as in k means), but, to a lesser extent, also the other prototypes. This introduces a correlation among them. To see intuitively why this improves training, consider the extreme case where all prototype behave as one and are updated exactly in the same way. This is a trivial optimization problem, because it is a one-prototype problem with a quadratic cost (a convex problem).

Idea of the algorithm: There is a vector y of k prototypes. For them, a fixed grid (a so-called topology) is defined: each prototype is considered connected to other prototypes as (for instance) in the following diagram:



This is a bi-dimensional, square grid, but other choices are possible. See below for how this particular grid can be implemented in software.

At each pattern, prototype number j is updated (REGARDLESS OF WHETHER IT IS THE WINNER OR NOT) with the following updating rule:

$$\mathbf{y}_j(t+1) = \mathbf{y}_j(t) + h \eta (\mathbf{x} - \mathbf{y}_j(t))$$

where:

- η depends only on the iteration - it will be larger (not too much!) at the beginning and progressively reducing toward the end; you already have experience with this parameter
- h depends on the iteration *and* on the distance of \mathbf{y}_j from the winner, computed as number of hops on the grid

Neighborhood: h is a *neighborhood* function. If \mathbf{y}_j is the winner, then $h = 1$ always.

If \mathbf{y}_j is not the winner, then it measures the distance of \mathbf{y}_j in terms of **hops needed to reach the winner by moving on the grid**, and decreases accordingly (with a suitable law). For instance, we can only consider the four neighbors at N - S - W - E:

- if \mathbf{y}_j is the winner, $h=1$
- if \mathbf{y}_j is one step away from the winner, $h = g$
- otherwise, $h = 0$

Or we can write h as an explicit function of the number of hops on the grid:

- $h = (1 - g \cdot \text{hops})$ if positive, or else 0 (piecewise linear)

or

- $h = \exp(-\text{hops}^2/g)$ (Gaussian)

In all cases, $0 < g < 1$. h decreases with distance (the number of hops on the grid) and g **decreases with the iteration** t . In other words, the neighborhood shrinks with time.

Computing the number of hops: on the square grid, the number of hops between two prototypes is the difference in column indexes plus the difference in row indexes. If the units are layed out by rows as in the figure (first row 1...4, second row 5...8, and so on), a function to convert prototype number (j) into row and col indexes (r, c) is:

```
function [row, col] = togrid(j,numcol)
    col = mod( ( j - 1 ) , numcol ) + 1;
    row = ceil ( j / numcol );
end
```

and a function to compute the distance (on the grid) between prototype j and prototype k is:

```
function d = griddist(j,k,numcol)
    [r1, c1] = togrid(j,numcol);
    [r2, c2] = togrid(k,numcol);
    d = abs(r1-r2)+abs(c1-c2);
end
```

(Note1: click on the function name to download the code.)

(Note2: Matlab already has a function analogous to "togrid", named ind2sub.)

Algorithm structure: The structure is the usual one for on-line training:

```
initialize
for iter = 1:maxiter
    shuffle training set (random permutation)
    for l=1:npatterns
        compute eta and g for this iteration
        x = pattern no. l
        compute distance of x from all prototypes y
        compute h for all prototypes
        compute update for all prototypes
        apply update
    end
    check stopping criterion
    compute and print distortion
end
(finalize)
```

Perform the same experiments as with k means (with k from 4 to 25 using squares: 4, 9, 16, 25).

Task 3 (optional): Neural Gas

Repeat Task 2 changing h as follows:


Instead of distance on a topology (grid), h is a function of the position (rank) of the current prototype when ordering all prototypes by increasing distance fro the winner.

That is, the nearest prototype (winner) has distance 0; the second nearest has distance 1; the third nearest has distance 2...

Make h decrease exponentially with the rank: $h = \exp(-\text{rank}/g)$, and g decrease with iterations.

UPLOAD BELOW YOUR CODE, DATA, AND REPORT

Submission status

Submission status	Submitted for grading
Grading status	Not graded
Due date	Monday, 16 December 2013, 11:55 PM
Time remaining	Assignment was submitted 5 hours 14 mins early
Last modified	Monday, 16 December 2013, 6:40 PM
File submissions	 Ahmed_LAB_5.rar