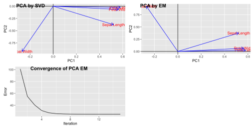
A Constrained EM Algorithm for PCA (from Ahn, J.-H. and J.-H. Oh, 2003)

```
1  upper<-function(A){A[lower.tri(A,diag=FALSE)]<-0;return(A)}
2  lower<-function(A){A[upper.tri(A,diag=FALSE)]<-0;return(A)}
3  PCA.EM<-function(X,q=2){
4    p<-ncol(X); n<-nrow(X)
5    W<-diag(p)[,1:q]; M<-X%*%W # Initialisation
6    Jold<-0; J<-1; iteration<-0; Error<-NULL
7    while ((abs(J - Jold)>1e-3)){
8      Jold<-sum((X-M%*%t(W))^2)
9      S <- solve(upper(t(W)%*%W)); M<- X%*%W%*%S # E-step
10     W<- (t(X)%*%M)%*%solve(lower(n*S+t(M)%*%M))# M-step
11     W<-apply(W,2,function(x){x/sqrt(sum(x^2))})#orthogonalisation
12     J<-sum((X-M%*%t(W))^2); Error[iteration<-iteration+1]<-J
13   }
14   return(list(W=data.frame(W),M=data.frame(M),Error=Error))}
```



Neural networks and unsupervised learning

Modeling of a neuron

The first modeling of the neuron was suggested in the 1940s by Mac Culloch and Pitts. It was a unit which according to several signals transmitted a binary response.

In general, a formal neuron has

- dendrites which receive the input signal and
- an axon which transmits the output signal

The input signal

$\boldsymbol{x}$ is a vector belonging most often to $\mathbb{R}^d$ or $\{0, 1\}^d$.

Dendrites

are characterized by a weight vector $\boldsymbol{w}$

The output

is a function of $\boldsymbol{x}$ and $\boldsymbol{w}$, which is the composition of an input function, $h(\boldsymbol{x}, \boldsymbol{w})$ and an output (or activation) function, $f(h)$

The neuron as a funtion

Most of the time the input function is a simple dot product:

$$h(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{x}^T \boldsymbol{w}$$

The activation functions are diverse but belong to large families (radial bases, sigmoid functions …).

A typical activation function is for example:

$$f(\boldsymbol{x}, \boldsymbol{w}) = \eta \cdot \frac{\exp\{(\boldsymbol{x}, \boldsymbol{w}\} - 1}{\exp\ (\boldsymbol{x}, \boldsymbol{w})\ + 1}.$$

The functions which have this appearance are said to be sigmoid

Neural networks

- Neurons can be connected to each other and then form a network.
- Learning consists of adjusting the free network parameters according to the desired goal, that is, to calculate the values of the weight vectors as a function of the inputs.

Two layers networks

Hebbian Learning (Hebb 1949)

Principle

An increase of synaptic strength between an input and an output neuron may be related to the firing rates of the input and output [Hebb, 1949].

Practical implementation

As a result, synaptic strengths will increase fastest between pairs of neurons whose responses are correlated, and the resulting increase in synaptic strength will lead to a further increase in the correlation.

$$\Delta \mathbf{w} \ = \ \eta \, y(\mathbf{x}) \mathbf{x},$$

or in scalar form with implicit n-dependence,

$$w_i(n+1) \ = \ w_i(n) + \eta \, y(\mathbf{x}) x_i$$

Increasing the correlation in this manner may lead to a useful pattern of synaptic strengths over a population of neurons.

Stochastic Gradient Descent (from Wikipedia)

Statistical estimation and machine learning consider the problem of minimizing an objective function that has the form of a sum:

$$Q(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} Q_i(\boldsymbol{w}),$$

where the parameter $\boldsymbol{w}$ that minimizes $Q(\boldsymbol{w})$ is to be estimated.

Each summand function Q_i is typically associated with the $\boldsymbol{i-th}$ observation in the data set (used for training).

Sum-minimization problems arise:

- in least squares and in maximum-likelihood estimation,
- empirical risk minimization.

When used to minimize the above function, a standard (or “batch”) gradient descent method would perform the following iterations:

$$\boldsymbol{w} := \boldsymbol{w} - \eta \nabla Q(\boldsymbol{w}) = \boldsymbol{w} - \frac{\eta}{n}\sum_{i=1}^{n} \nabla Q_i(\boldsymbol{w}),$$

where η is a step size (sometimes called the learning rate in machine learning).

Iterative method

Fluctuations in the total objective function as gradient steps with respect to mini-batches are taken.

In stochastic (or "on-line") gradient descent, the true gradient is approximated by a gradient at a single example:

$$\boldsymbol{w} := \boldsymbol{w} - \eta \nabla Q_i(\boldsymbol{w}).$$

- As the algorithm sweeps through the training set, it performs the above update for each training example. - Several passes can be made over the training set until the algorithm converges. - If this is done, the data can be shuffled for each pass to prevent cycles. -

Stochastic Gradient in pseudocode

1. Choose an initial vector of parameters $\boldsymbol{w}$ and learning rate η
2. Repeat until an approximate minimum is obtained:
    a. Randomly shuffle examples in the training set.
    b. For $i = 1, 2, \cdots, n$ do: $\boldsymbol{w} := \boldsymbol{w} - \eta \nabla Q_i(\boldsymbol{w})$.

Adaptative learning rate

A distinction exists between constant gain algorithms,

$$\eta_n \geq 0, \quad lim_{n \to \infty} \eta_n = \eta > 0$$

and decreasing gain algorithms,

$$\sum_{n=0}^{\infty} \eta_n = \infty, \ \ \sum_{n=0}^{\infty} \eta_n^2 < \infty.$$

The first are dedicated to the estimation of parameters changing slowly over time and the second to the estimation of stable parameters.

K-means and Winner take all

is a computational principle applied in computational models of neural networks by which neurons in a layer compete with each other for activation.

The simplest form of competitive learning modifies only the weight vector of “the best” neuron at every stage of learning.

In fact, with each presentation of a input (a vector of the training set), two steps are performed:

1. choose the best neuron, i.e. the one that shows the most important output
2. modify the weight vector of this neuron.

When the activation function is increasing (which is not true for the functions with radial basis), the winning neuron is the one that produces the greatest value of function entry.

If we consider a dot product as an input function, the weight vector of the winner, i^*, checks:

$$\forall i,\ (\boldsymbol{w}_{i^*} \cdot \boldsymbol{x}) \geq (\boldsymbol{w}_i \cdot \boldsymbol{x}).$$

And if *the weight vectors are normalized*, the winner is the neuron that has the weight vector, closest to the input $\boldsymbol{x}$, in the sense of the Euclidean distance.

The coordinates of the winner's weight vector are updated using a rule of the type following :

$$\boldsymbol{w}_{i*}(t+1) = \boldsymbol{w}_{i^*}(t) + \eta(t) \cdot (\boldsymbol{x} - \boldsymbol{w}_{i^*}(t)),\ \eta(t) \leq 1$$

where $\eta(t)$ is the training step at iteration t.

Stochastic Gradient for Kmeans (online Kmeans)

The criterion to be optimized can be written as

$$Q(\boldsymbol{w}_1, \cdots, \boldsymbol{w}_K) = \frac{1}{2n} \sum\nolimits_i \sum_k I_{(k = \arg min_\ell \|\boldsymbol{x}_i - \boldsymbol{w}_\ell\|^2)} \|\boldsymbol{x}_i - \boldsymbol{w}$$

1. Choose an initial vector of parameters $\boldsymbol{w}$ and learning rate η
2. Repeat until an approximate minimum is obtained:
   a. Randomly shuffle examples in the training set.
   b. For $i = 1, 2, \cdots, n$ do:
      1. For $k = 1, 2, \cdots, K$
      - $\nabla Q_i(\boldsymbol{w}_k) = -I_{(k = \arg min_\ell \|\boldsymbol{x}_i - \boldsymbol{w}_\ell\|^2)} (\boldsymbol{x}_i - \boldsymbol{w}_k)$
      - $\boldsymbol{w}_k := \boldsymbol{w}_k - \eta \nabla Q_i(\boldsymbol{w}_k)$.

Kmeans implementation

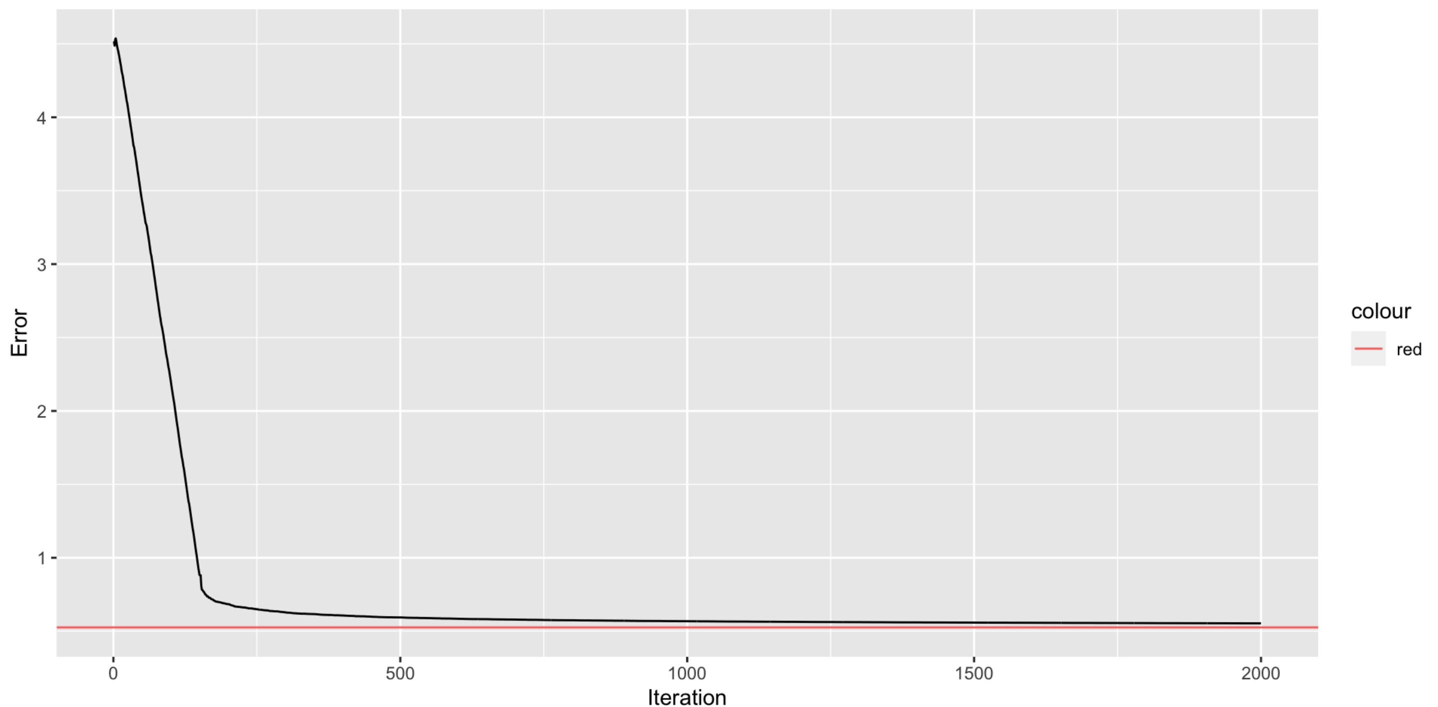
```
1  kmeans.winner.take.all<-function(X,K=2,max.iteration=2000){
2    p<-ncol(X);  n<-nrow(X);shuffling<-sample(1:n,n)
3    X<-X[shuffling,];  W<-X[sample(1:n,K),]
4    Q<-rep(0,max.iteration); cluster<-rep(0,n)
5    distances<-rep(sum(diag(var(X)))*(n-1)/n,n)
6    for (i in 1:max.iteration){
7      x<-cbind(X[(i-1)%%n + 1,])
8      distances.x.to.W<-sum(x^2)-2*as.matrix(x)%*%t(W)+ colSums(t(W^2))
9      winner.index<-which.min(distances.x.to.W)
10     W[winner.index,]<-W[winner.index,] + 1/i*(x-W[winner.index,])
11     cluster[(i-1)%%n + 1]<-winner.index
12     distances[(i-1)%%n + 1]<-distances.x.to.W[winner.index]
13     Q[i]<-mean(distances) }
14   return(list(W=W,Q=Q,cluster=cluster[order(shuffling)]))
15 }
```


One line kmeans example with Fisher iris

```
1 data(iris)
2 X<-iris[,1:4]
3 set.seed(1)
4 kmeans.winner.take.all(X,3)->res
5 table(res$cluster,iris$Species)
```

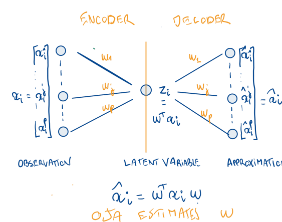
```
   setosa versicolor virginica
1      50          0         0
2       0         45        11
3       0          5        39
```

One line kmeans example with Fisher iris



PCA and Oja's rule

Consider a linear neuron with output $z = \boldsymbol{w}^T \boldsymbol{x}$ that returns a linear combination of its inputs $\boldsymbol{x}$ using presynaptic weights $\boldsymbol{w}$.



PCA according OJA

Oja's rule defines the change in presynaptic weights $\boldsymbol{w}$ given the output response $\boldsymbol{y}$ of a neuron to its inputs $\boldsymbol{x}$ to be

$$\boldsymbol{w} := \boldsymbol{w} - \eta z(\boldsymbol{x} - z\boldsymbol{w})$$

Stochastic Gradient PCA and Oja's rule

The criterion to be optimized can be written as

$$Q(\boldsymbol{w}) = \frac{1}{2n}\sum_i \|\boldsymbol{x}_i - \hat{\boldsymbol{x}}_i\|^2 = \frac{1}{2n}\sum_i \|\boldsymbol{x}_i - y_i\boldsymbol{w}\|^2 = \frac{1}{2n}$$

where $\|\boldsymbol{w}\|^2 = 1$.

1. Choose an initial vector of parameters $\boldsymbol{w}$ and learning rate η
2. Repeat until an approximate minimum is obtained:
   a. Randomly shuffle examples in the training set.
   b. For $i = 1, 2, \cdots, n$ do:
      - $\nabla Q_i(\boldsymbol{w}) = y_i(\boldsymbol{x}_i - y_i\boldsymbol{w})$
      - $\boldsymbol{w} := \boldsymbol{w} - \eta\nabla Q_i(\boldsymbol{w})$.

Oja's rule implementation

```
1  Oja.rule<-function(X,max.iteration=10000,eta=0.001){
2    p<-ncol(X);  n<-nrow(X)
3    w<-rbind(rep(1,p)); w<-w/(sqrt(sum(w^2)))
4    Q<-rep(0,max.iteration)
5    for (i in 1:max.iteration){
6      Q[i]<- 1/(2*n) *sum((X - X%*%t(w)%*%w)^2)
7      x<-X[(i-1)%%n + 1,]
8      y<-sum(w*x)
9      w<-w + eta*y*(x-w*y)}
10   return(list(w=w,Q=Q))
11 }
```


Oja's rule example with Fisher iris