# Feature extraction Techniques

## *On the go* documentation

For internal use only

*Rafael A. Calvo*
SEDAL

# Contents

# 1 Implementation of clasical techniques

⟨*pca.cc*⟩≡
  ⟨*Includes*⟩
  ⟨*svd*⟩
  ⟨*power method*⟩

## 1.1 SVD using Numerical recipes library

If the matrix $C = (x_1, x_2, ..., x_n)$ contains the **normalized** eigenvectors of a symmetric matrix $A$, then C is orthogonal.

$$A = ACC^T$$

we now substitute C,

$$A = A(x_1, x_2, ..., x_n)C^T$$

$$A = (Ax_1, Ax_2, ..., Ax_n)C^T$$

$$A = (\lambda_1 x_1, \lambda_2 x_2, ..., \lambda_n Ax_n)C^T$$

$$A = CDC^T$$

where D is a matrix with the eigenvalues on the diagonal.

The SVD routine has more general application since is for solving systems of equations that can be thought as m.n matrices, but we will coment it for this specific case.

In numerical recipes is made clear that matrices shuld be made tridiagonal and only then should the iterative methods for calculating eigenvalues should be applied.

⟨*Includes*⟩≡
```
include "main.h"
include "nr/nr.h"
include <stdlib.h>
include<Options.h>
```

Is usually more usefull to have the eigenvalues and eigenvectors sorted, so biggest eigenvalue (and corresponding eigenvector) are first.

$\langle svd \rangle \equiv$
```
static float **evalp;

int indexcmp(const void*a, const void*b) {
  float diff=(*evalp)[*(int*)a]-(*evalp)[*(int*)b];
  return (-diff<0) ? -1: (diff==0)? 0:1;
}
```

The main function for using SVD

We suppouse that `evect` contains the eigenvectors in rows (like in hebbians?).

$\langle svd \rangle + \equiv$
```

void PCA(const matrix  Cov, vector  eval, matrix  evect) {
  register row,col;
  float** FCov;
  float** Fevect;
  float* Feval;

  assert(Cov.filas()==Cov.columns());
  int m=Cov.filas();
```

allocation of free memory. In the matrices memory for rows is allocated in the next loop.

$\langle svd \rangle + \equiv$
```
  FCov = (float **)calloc(m+1, sizeof(float*));
  assert(FCov);
  Fevect = (float **)calloc(m+1, sizeof(float*));
  assert(Fevect);
  Feval = (float *)calloc(m+1, sizeof(float));
  assert(Feval);
```

Numerical Recipes uses indexes that start on 1, the `matrix` and `vector` objects should be transformed to `float **` and `float *`.

⟨*svd*⟩+≡
```
for(row=0;row<m;row++) {
  FCov[row+1] = (float *)calloc(m+1, sizeof(float));
  Fevect[row+1] =(float *)calloc(m+1, sizeof(float));
  Fevect[row+1] = evect[row].pointer()-1;
  for(col=0; col<m;col++) {
     FCov[row+1][col+1]=Cov[row][col];
  }
}
```

Given a matrix Cov[1..m][1..m] of the covariances (or correlations) of the data set, the svdcmp routine computes its singular value decomposition,

$$Cov = U.W.V^T$$

. The matrix U replaces Cov on output. The diagonal matrix of singular values W is output as a vector W[1..m]. The matrix V (not the traspose $V^T$) is output as v[1..m][1..m], so eigenvectors should be in rows.

so,

W is the vector with all the eigenvalues.
$V^T$ is the matrix with the eigenvectors in columns. Note that it comes out of `svdcmp` as V not its traspouse.

the call to the function in NR.

⟨*svd*⟩+≡
```
svdcmp(FCov, m, m, Feval, Fevect);
```

To do the sorting we use `qsort` of `stdlib.h`. It requires a global variable that containes the table (vector to be sorted) tt evalp. `indexcmp` is the comparisson function. qsort() is an implementation of the quick-sort algorithm. It sorts a table of data in place. The contents of the table are sorted in ascending order according to the user-supplied comparison function.

⟨*svd*⟩+≡
```
evalp =  Feval;

int* id=new int[m];
for(col=0;col<m;col++)
  id[col]=col+1;
qsort(id,m,sizeof(int),indexcmp);
```

id has the indexes of the sorted eigenvalues/vectors. Also svdcmp returns the eigenvectors in columns in what was FCov. I copy the eigenvalues/vectors sorted and in numerical recipes format to my eval vector and evect matrix.

⟨*svd*⟩+≡
```
    for(col=0;col<m;col++) {
      eval[col]=Feval[id[col]];
    }
    for(row=0;row<m;row++)
      for(col=0;col<m;col++) {
        evect[row][col]=Fevect[row+1][id[col]];
        // evect[col].pointer()=Fevect[id[col]]-1;
    }
   // for(col=0;col<m;col++)
   // evect[col].pointer()=Fevect[id[col]]-1;
  }
```

## 1.2 Power method

A simple standard procedure for computing approximate values of the eigenvalues of a $n \times n$ matrix $A = [a_{jk}]$ is the power method. A form of the technique was described in Hotellingś original paper on PCA [?] and an accelerated version was published by Hotelling in [?].

In this method we generate a random vector $x_0$ ($\neq 0$) with n components and the compute iteratively $x_1$, $x_2$,... $x_p$

$x_1 = Ax_0$,$x_2 = Ax_1$...., $x_p = Ax_{p-1}$

In [?] there is a demostration that makes things clear. The only difference is that x is normalised.

also [?] explains the method.

The code for the algorith is structured:

⟨*power method*⟩≡
  ⟨*Allocation of data structures*⟩
  ⟨*Create covariance matrix*⟩
  ⟨*For each dimension:*⟩
   ⟨*Initialise vector*⟩
   ⟨*normalise vector*⟩
   ⟨*Perform power method iterations*⟩
   ⟨*Deflate data*⟩
   ⟨*scores*⟩
  ⟨*End method*⟩

⟨*Allocation of data structures*⟩≡

```
void power method(matrix  data,matrix  scores, matrix  weights,vector  evals)
{
  uint out dim = evals.length();
  uint dim = data.columns();
  matrix C(dim,dim), P(out dim,dim);
  vector p(out dim);
  float w;

  cout << "\n..................PM...............\n";
  for(uint i=0;i<out dim;i++) {
    p[i]=0;
    for(uint j=0;j<dim;j++)
      P[i][j]=0;
  }
  for(uint i=0;i<dim;i++)
    for(uint j=0;j<dim;j++)
      C[i][j]=0;
```

⟨*Create covariance matrix*⟩≡

```
  C = Covar(data);
  cout << "\n..................PM...............\n";
```

⟨*For each dimension:*⟩≡

```
  for (uint d=0; d<out dim; d++) {
```

⟨*Initialise vector*⟩≡

```
    for (uint i=0; i<dim; i++)
      weights[d][i] = drand48()-.5;
```

⟨*normalise vector*⟩≡

```
    weights[d] /= weights[d].norma() ;
```

⟨*Perform power method iterations*⟩≡
```
      for (uint pass=0; pass<(uint)flag("pm it").value; pass++) {
        for(uint i=0;i< (uint) p.length();i++)  p[i] = 0;
        p =  C * weights[d];
        w = sqrt (p * p); //p.norma();
        if(!w) cout <<"\n Eval= 0 !!!";
        if (w) {
          p = p/w;
          evals[d]= w;
          weights[d]=p;
        }
       }
```

⟨*Deflate data*⟩≡
```
      for (uint x=0; x<dim; x++) {
        float sum = 0;
        for (uint y=0; y<dim; y++)
          sum += C[x][y] * weights[d][y];
        for (uint y=0; y<dim; y++)
          C[x][y] -= sum * weights[d][y];
      }
```

⟨*scores*⟩≡
```
      for (uint i=0; i<data.filas(); i++)
        for (uint x=0; x<out dim; x++)
          scores[i][x] = weights[x] * data[i];
```

⟨*End method*⟩≡
```
    }
  }
```

# 2 Implementation of Simple PCA

We explain here the implementation for the different variations the the SPCA. One single $C^{++}$ file will be generated including the different functions. Its structured:

⟨*spca.cc*⟩≡
```
        ⟨Includes⟩
        ⟨Iterative⟩
        ⟨Batch⟩
        ⟨Standard⟩
        ⟨Online⟩
        ⟨Standard L2⟩
```

⟨*Includes*⟩+≡
```
include "main.h"
include<Options.h>

extern char* encoding[256];
```

## 2.1 Iterative

⟨*Iterative*⟩≡
```
void SPCA iter(matrix  data,vector  classes,matrix  scores,matrix  weights,vector  evals)
{
```

⟨*Iterative*⟩+≡
```
        ⟨initialization⟩
        ⟨iterative loop⟩
        ⟨Normalization⟩
        ⟨Deflation⟩
    ⟨end⟩
```

## 2.2 Batch

⟨*Batch*⟩≡
```
void SPCA batch(matrix  data, vector  classes,matrix  scores, matrix  weights,vector  eval
{
```

⟨*Batch*⟩+≡
```
        ⟨initialization⟩
        ⟨batch loop⟩
        ⟨Normalization⟩
        ⟨Deflation⟩
    ⟨end⟩
```

## 2.3 Standard

The standard consists of 1 iterative and `sit` batch iterations.

⟨*Standard*⟩≡
```
void SPCA std(matrix  data, vector  classes,matrix  scores, matrix  weights,vector  evals)
{
```

⟨*Standard*⟩+≡
```
        ⟨initialization⟩
        ⟨iterative loop⟩
        ⟨batch loop⟩
        ⟨Normalization⟩
        ⟨Deflation⟩
    ⟨end⟩
```

## 2.4 Online SPCA

⟨*Online*⟩≡

## 2.5 L2 SPCA

The L2 SPCA standard consists of 1 iterative and `sit` L2 batch iterations.

⟨*Standard L2*⟩≡
```
void SPCA 2(matrix  data, vector  classes,matrix  scores, matrix  weights,vector  evals)
{
```

⟨*Standard L2*⟩+≡
```
        ⟨initialization⟩
        ⟨iterative loop⟩
        ⟨L2 batch loop⟩
        ⟨Normalization⟩
        ⟨Deflation⟩
    ⟨end⟩
```

## 2.6 Common loops

⟨*initialization*⟩≡

```
int outputs=evals.length(), inputs=data.columns(), N=data.filas();
vector mean(inputs);


if (flag("Winit")) {
  char* wfile = flag("Winit").string;    // weights file
  ifstream fwp(wfile);
  if (!fwp) {
    cerr << "I can't open weights file\n";
    exit(1);
  }
  weights.carga(fwp);
}
for (int d=0; d<outputs; d++) {
  if (!flag("Winit"))  {
    for(int i=0;i<inputs;i++) {
      weights[d][i] = 0;
      mean[i] = 0;
    }
  }
  for(int i=0;i<inputs;i++) {
    mean[i] = 0;
  }
```

We repeat the following procedures for each of the desired outputs or number of pc.

We need a first approximation, usually used as *seed* for other iterative methods.

⟨*iterative loop*⟩≡

```
  // iterative
  for (int i=0; i<N; i++)
    if (weights[d] * data[i] >= 0.0)
      weights[d] += data[i];
```

The batch procedures follow. Many posibilities are posible, we first show a simple batch loop where only the sign of the correlation between the reduced data and the approximation that we have to the eigenvctor matrix, this is shown to maximize the L1 norm as defined in ??. The second, maximizes the L2 norm (see ??) and weights the data by the correlation. This method is in many ways equivalent to the Ojas Rule in Hebbian learning.

⟨*batch loop*⟩≡
```
for (uint it=0;it<(uint)flag("b its").value;it++)
  for (int i=0; i<N; i++)
    if (weights[d] * data[i] >= 0.0)
      mean += data[i];
  weights[d] = mean;
```

The second, maximizes the L2 norm (see ??) and weights the data by the correlation. This method is in many ways equivalent to the Ojas Rule in Hebbian learning.

⟨*L2 batch loop*⟩≡
```
for (uint it=0;it<(uint)flag("b its").value;it++)
  for (int i=0; i<N; i++)
    mean += (weights[d] * data[i]) * data[i];
  weights[d] = mean;
```

All eigenvectors have to be normalized.

⟨*Normalization*⟩≡
```
  weights[d] = weights[d] / weights[d].norma();
```

Now we will substract the projection over the just calculated PC, so the reduced data set will be in the space orthogonal to the PC. Over this data set I is that calculate the next PC. This important since the next PC will be calculated over the remaining variance only (error).

⟨*Deflation*⟩≡
```
  // next pc
  for (int i=0; i<N; i++) {
    scores[i][d] = weights[d] * data[i];
    data[i] -= scores[i][d] * weights[d];
  }
evals[d] = scores.separa col(d).variance() ;
}
```

$\langle \mathit{end} \rangle \equiv$

```
      if (flag("savesc"))   {
        char* name = flag("savesc").string;
        ofstream scores fp(name);
        if (!scores fp) {
          cerr << "I can't open file for saving scores\n";
          exit(1);
        }
        scores fp << scores;
      //  for (int patt=0; patt<N; patt++) {
      //scores fp << encoding[(int)classes[patt]] <<"\t" << scores[patt] << '\n';
      //  }
      }

   if (flag("save eval"))   {
        char* name = flag("save eval").string;
        ofstream eval fp(name);
        if (!eval fp) {
          cerr << "I can't open file for saving eigenvalues\n";
          exit(1);
        }
        eval fp << evals;
      }

  }
```

# 3 Implementation of Hebbian learning rules

⟨*hebb.cc*⟩≡
  ⟨*Includes*⟩
  ⟨*Global variables*⟩
  ⟨*Oja*⟩
  ⟨*Sanger*⟩
  ⟨*APEX*⟩

## 3.1 Common code

⟨*Includes*⟩+≡

```
include<Options.h>
include "main.h"
void dw oja(matrix  dw, matrix  weights, vector  data,
            vector  lrate, vector  var, vector  scores);
void dw sanger(matrix  dw, matrix  weights, vector  data,
               vector  lrate, vector  var, vector  scores);
void dw apex(matrix  dw,matrix  weights,vector  data,
             vector  lrate,matrix  lateralW, matrix  dlateralW,vector  var, vector  scores
```

⟨*Global variables*⟩≡

```
  bool super verbose;
  float lratei;
  int pass = 0;
  int lrate type;
  bool  Const Lrate=0, Sigma Lrate=0, Ev Lrate=0;
```

⟨*declarations*⟩≡

```
  bool reset;
  float nepochs, epocas = 0;
  int N=data.filas();    // number of data available
  int inputs  = data.columns();
  int outputs = weights.filas();
  matrix dw(outputs, inputs);         // weights variations
  vector lrate(outputs);              // un lrate para cada salida
```

⟨*more user interface*⟩≡
```
      cerr << "\nLrate?"; cin >> lratei;
      cout << "\n" << lratei;
      cerr << "\nnepochs?"; cin >> nepochs;
      cout << nepochs;
      cerr << "\nAll (1) or only result (0) [1/0]?"; cin >> super verbose;
      cerr << "\nRandom init matrix (1) File (0) [1/0]?"; cin >> reset;
      cerr << "\nLrate adaptation (0: no adapt, 1:sigma, 2:Exp. var) ";
           cin >> lrate type;

      if(lrate type==0) Const Lrate=1; else
      if(lrate type==1) Sigma Lrate=1; else
      if(lrate type==2) Ev Lrate=1; else
      cout <<"\nnon-existing lrate method.";

      if(reset) {
        weights = m drand48(outputs,inputs) - 0.5;
        for(int i=0;i<weights.filas();i++) {
          float length = weights[i].norma();
          weights[i] = weights[i]/length;//I think they only have to be small..
        }
        //cout <<"\nWeights: "<< weights;
        if (super verbose) {
          cout <<"\nInitial matrix after resetting:";
          //weights.imprime(); //save(weights);
        }
      }
      if (!reset) {
        static char archivo2[30];    // weights file
        cerr << "\nweights file: "; cin >> archivo2;
        ifstream fp2(archivo2);
        if (!fp2) {
          cerr << "I can't open weights file\n";
          exit(1);
        }
        weights.carga(fp2);
      }
      for (int z = 0; z < outputs; z++)
        lrate[z] = lratei;
```

### 3.2 Ojaś Rule

The networks described here are linear, with the ith output:

$$V_i = \sum_{j=1}^{N} W_{ij} d_j$$

$$\Delta W_{ij} = \eta V_i (d_j - \sum_{k=1}^{N} V_k W_{kj})$$

The anti Hebbian term `anti` is of dimension inputs.

⟨*Oja*⟩≡
  ⟨*begin Oja*⟩
  ⟨*declarations*⟩
  ⟨*more user interface*⟩
  ⟨*Oja training*⟩
  ⟨*end*⟩
  ⟨*Oja Rule*⟩

⟨*begin Oja*⟩≡

```
void oja(matrix  data,matrix  scores,matrix  weights, vector  evals)
{
```

⟨*Oja training*⟩≡

```
  vector var(outputs);
  for(int i=0;i<outputs;i++) var[i] = 0.0001;
   pass=0;
  do {
    // for each pattern k
    for (int k = 0; k < N; k++) {
      dw oja(dw, weights, data[k], lrate, var, scores[k]);
      weights += dw; // update weights after each patt?
    }

// (27/3/98) the following is not necesary except for printing
// so set as option
 if(flag("Pevals")) {
   for (int i=0;i< outputs;i++)
     evals[i] = scores.separa col(i).variance();
   cout<<"\nEpochs: "<< epocas << "\nEV = " << evals;
 }
 // if(super verbose) show epoch(weights,data,evals,epocas,scores);
 } while (nepochs > epocas++);
```

⟨*end*⟩+≡
```
  }
```

⟨*Oja Rule*⟩≡
```
  void dw oja(matrix  dw, matrix  weights, vector  data,
              vector  lrate,vector  var, vector  V)
  {
    // generates dw por each pattern pat
    int outputs = weights.filas();
    int inputs = weights.columns();
    vector anti(inputs);
    assert(inputs >= outputs);
```

⟨*Oja Rule*⟩+≡
      ⟨*Learning rate adaptation*⟩

⟨*Oja Rule*⟩+≡
```
      V = weights * data;
    for (int i = 0; i < outputs; i++) {
      for (int j = 0; j < inputs; j++) {
        anti[j]=0;
        for (int l = 0; l < outputs; l++)
          anti[j] += V[l] * weights[l][j];
        dw[i][j] = lrate[i] * V[i] * (data[j] - anti[j]);
      }
    }
  }  // end dw oja
```

### 3.3 Sangerś Rule

Here we use Sangerś learning rule:

$$\Delta W_{ij} = \eta V_i (d_j - \sum_{k=1}^{i} V_k W_{kj})$$

The one output unit case reduces to Ojas rule described above. The main difference with Ojas rule is that weight vectors $W_k$ converge to the firsts principal components in order.

```
⟨Sanger⟩≡
  ⟨begin Sanger⟩
  ⟨declarations⟩
  ⟨more user interface⟩
  ⟨Sanger training⟩
  ⟨end⟩
  ⟨Sanger Rule⟩
```

```
⟨begin Sanger⟩≡
  void sanger(matrix  data,matrix  scores, matrix  weights, vector  evals)
  {
```

```
⟨Sanger training⟩≡
    vector var(outputs);
    for(int i=0;i<outputs;i++) var[i] = 0.0001;
    pass=0;
    do {
        // for each pattern k
        for (int k = 0; k < N; k++) {
          dw sanger(dw, weights, data[k], lrate, var, scores[k]);
          weights += dw; // update weights after each patt
        }
        if(flag("Pevals")) {
        for (int i=0;i< outputs;i++)
          evals[i] = scores.separa col(i).variance();
        cout<<"\nEpochs: "<< epocas << "\nEVal = " << evals;
        }
        //if(super verbose) show epoch(weights,data,evals,epocas,scores);
    } while (nepochs > epocas++);
```

```
⟨Sanger Rule⟩≡
  void dw sanger(matrix  dw, matrix  weights,
                 vector  data,vector  lrate, vector  var, vector  V)
  {
    // generates dw por each pattern k
    int outputs = weights.filas();
    int inputs  = weights.columns();
    vector anti(inputs);
    assert(inputs >= outputs);
```

```
⟨Sanger Rule⟩+≡
      ⟨Learning rate adaptation⟩
```

⟨*Sanger Rule*⟩+≡
```
    V = weights * data;
  for (int i = 0; i < outputs; i++) {
    for (int j = 0; j < inputs; j++) {
      anti[j] = 0;
      for (int l = 0; l <= i; l++)
        anti[j] += V[l] * weights[l][j];
      dw[i][j] = lrate[i] * V[i] * (data[j] - anti[j]);
    }
  }
} // end of dw sanger
```

### 3.4 APEX algorithm

Kung and Diamantaras (see the original in [?] or in their review book [?]) developed a new algorithm called Adaptive Principal components Extraction (APEX). Figure 1 shows the new architecture. The new feature is that it has lateral connections that will be updated by an anti-hebbian learning rule. The output $V_i$ is given by:

$$y = Wx - Cy$$

where the lateral connections are represented by $C$

$$C =$$

and the feedback signal vector is defined by the outputs of neurons $0, 1, ...j - 1$:

The update equations for the feedforward weights $W_{ij,k}$

$$W_{ij,k} = \beta_k(y_{ik}x_{jk} - y_{ik}^2 W_{ij,k}$$

and for the lateral connections:

$$W_{ij,k} = \beta_k(y_{ik}x_{jk} - y_{ik}^2 C_{ij,k}$$

where $\beta_k$ is the learning rate, assumed to be the same for both update equations.

⟨*APEX*⟩≡
  ⟨*begin APEX*⟩
  ⟨*declarations*⟩
  ⟨*APEX declarations and initialization*⟩
  ⟨*more user interface*⟩
  ⟨*APEX training*⟩
  ⟨*end*⟩
  ⟨*APEX Rule*⟩

⟨*APEX*⟩+≡

⟨*begin APEX*⟩≡
```
void apex(matrix  data,matrix  scores,matrix  weights, vector  evals)
{
```

⟨*APEX declarations and initialization*⟩≡
```

  matrix lateralW(outputs,outputs);   // rows: receptor
  matrix dlateralW(outputs,outputs);  // diff for each output
```

old version based on APEX algorithm: Adaptive Principal-component Extractor in "Neural Networks..." by S. Haykin, pg384

⟨*APEX declarations and initialization*⟩+≡
```
    float apexin;
    cout << "\nIninitial value of lateral weights: "<<flush;
  //cin >> apexin;

    apexin =0.0001;
      cout << "\napexin set: "<< apexin <<flush;
      for(int i=0;i<outputs;i++)
        for(int j=0;j<i;j++) {
          lateralW[i][j] = 0;
          lateralW[i][i] = 0;
          lateralW[j][i] = apexin;
          dlateralW[i][j]= 0;
          dlateralW[i][i]= 0;
          dlateralW[j][i]= 0;
        }
      lateralW[0][0] = 0;
      dlateralW[0][0]= 0;

      vector var(outputs);
      for(int i=0;i<outputs;i++) var[i] = 0.0001;
```

⟨*APEX training*⟩≡
```
      do {
        // for each pattern k
        for (int k = 0; k < N; k++) {
          dw apex(dw,weights,data[k],lrate,lateralW,dlateralW,var, scores[k]);
          weights += dw;
          lateralW += dlateralW;
          cout <<"\n passed k= "<<k;
        }
        if(flag("Pevals")) {
          for (int i=0;i< outputs;i++)
            evals[i] = scores.separa col(i).variance();
          cout<<"\nEpochs: "<< epocas << "\nEV = " << evals;
        }
        if(super verbose) show epoch(weights,data,evals,epocas,scores);
      } while (nepochs > epocas++);
```

⟨*APEX Rule*⟩≡
```
void dw apex(matrix  dw,matrix  weights,vector  data,vector  lrate,
             matrix  latW, matrix  dlatW,vector  var, vector  V)
{
  int outputs = weights.filas();
  int inputs  = weights.columns();
  vector anti(outputs);
  float lat;
```

⟨*APEX Rule*⟩+≡
  ⟨*Learning rate adaptation*⟩

⟨*APEX Rule*⟩+≡
```
 V = weights * data;
  for (int i=0;i<outputs;i++) {
    lat=0;
    for(int l=0;l<i;l++) {    // dlatW
      lat += V[l] * latW[i][l];
      V[l] -=  lat ;
      dlatW[l][i] = lrate[l] * V[l] * (V[i] - lat);
    }
    for (int j = 0; j < inputs; j++) {  // dw
      anti[j]=0;
      for(int l=0; l<=i;l++)
        anti[j] += V[l] * weights[l][j];
      dw[i][j]  = lrate[i] * V[i] * (data[j] - anti[j]);
    }
  }
} //end dw apex
```

### 3.5 Learning Rates

Each output has a learning rate. `lrate` is a vector with the corresponding values. $mathbf\beta = [\beta_1, \beta_2, \ldots, \beta_{outputs}]$

As choosing the appropiate learning rate is usually troublesome to say the least, we wrote some adaptation rules:

⟨*Learning rate adaptation*⟩≡
⟨*const lrate*⟩
⟨*sigma lrate*⟩
⟨*exp. var. lrate*⟩

1. **Constant learning rate**. All the learning rates are set at `lratei` as set by user.

   ⟨*const lrate*⟩≡

```
if(Const Lrate){
  for (int i=0;i<outputs;i++)
    lrate[i] = lratei;  // constant lrate
}
```

2. **Inverse of the acumulated variance**. In vector var the square of of the outputs is acumulated and pass1 is incremented. Efectively for output `i` at pass (sample+sample(epoch-1)):

$$\beta_{i,pass1} = lrate_{initial} * \frac{1}{\sigma_{i,k}}$$

   ⟨*sigma lrate*⟩≡

```
if(Sigma Lrate) {
  ++pass;   // increase with every sample and epoch
  for (int i=0;i<outputs;i++) {
    var[i] += V[i] * V[i]; // acumulate with every sample and epoch
    lrate[i] = lratei * pass / var[i];  // acum. variance on outp. i
  }
}
```

3. **Inverse of the acumulated explained variance**

$$\beta_{i,pass1} = lrate_{initial} * \frac{1}{ev_{i,k}}$$

   ⟨*exp. var. lrate*⟩≡

```
if(Ev Lrate) {
  for (int i=0;i<outputs;i++) {
    var[i] += V[i] * V[i]; // acumulate with every sample and epoch
    lrate[i] = lratei * (++pass) / var.suma();  // acum var on all outp
  }
}
```

Figure 1: APEX neural network arquitecture

# 4 Simulation software

## 4.1 Main code main.cc

The main code is structured:

$\langle$*main.cc*$\rangle\equiv$
    $\langle$*includes*$\rangle$
    $\langle$*run method code*$\rangle$
    $\langle$*data management in Matt's format*$\rangle$
    $\langle$*main routine*$\rangle$
    $\langle$*definitions*$\rangle$
    $\langle$*General user interface*$\rangle$
    $\langle$*more definitions*$\rangle$
    $\langle$*begin bootstraping*$\rangle$
    $\langle$*run method*$\rangle$
    $\langle$*show and traspouse if needed*$\rangle$
    $\langle$*distance between methods*$\rangle$
    $\langle$*end bootstraping*$\rangle$
    $\langle$*end*$\rangle$

### 4.2 Header main.h

⟨*main.h*⟩≡
```
define NDEBUG
include <assert.h>
include <fstream.h>
include <stdlib.h>
include <math.h>
include <signal.h>
include <time.h>
include "vector.h"
include "matrix.h"
include "nr/nr.h"
include <sys/times.h>

//typedef float floatreal;

// i/o
extern void version(char *);
float trace(matrix  C);
vector e val(const matrix  , const matrix  , matrix  );
void show epoch(matrix  weights,matrix  data,vector  autovals,
              float epocas,matrix  scores);
void show bs txt(vector  evals);
void save bs result1(matrix  bs evals,float dtrace,ofstream  , uint o);
void save bs result2(matrix  bs evals,vector  bs trace,matrix  bs corr evect,
                     vector  bs corr,ofstream   bs out, uint o);
matrix m drand48(lint row, lint col);
void save(matrix  weights);
char* date ext(char* res f);
```

The learning rules implemented are: Sanger's (see 3.3):

⟨*main.h*⟩+≡
```
extern void sanger(matrix  data,matrix  scores,matrix  weights,vector  evals);
```
Oja's (see 3.2):

⟨*main.h*⟩+≡
```
extern void oja(matrix  data,matrix  scores,matrix  weights,vector  evals);
```
APEX (see 3.4):

⟨*main.h*⟩+≡
```
extern void apex(matrix  data,matrix  scores,matrix  weights,vector  evals);
```

Principal Component Analysis (PCA) using Singular Value Decomposition (SVD). The most common on the covariance matrix, but can also be used on the correlation matrix.

⟨*main.h*⟩+≡
```
extern void PCA(const matrix  ds, vector  eval, matrix  evect);
```

The power method is described in section ??

⟨*main.h*⟩+≡
```
extern void power method(matrix  data,matrix  scores, matrix  evect,vector  evals);
```

Work submitted to nips ref. ...

⟨*main.h*⟩+≡
```
extern void SPCA iter(matrix  data, vector  classes,matrix  scores,matrix  weights,vector
extern void SPCA batch(matrix  data, vector  classes,matrix  scores,matrix  weights,vecto
extern void SPCA std(matrix  data, vector  classes,matrix  scores,matrix  weights,vector
extern void SPCA 2(matrix  data, vector  classes,matrix  scores,matrix  weights,vector  ev
```

⟨*includes*⟩≡
```
include "main.h"
include <Options.h>
```

⟨*data management in Matt's format*⟩≡
```
typedef struct {
    uint label;
    float *vec;
} DataSetElt;
extern void read DS(char *filename);
extern uint VECTOR SIZE;
uint         Instances;
DataSetElt   *DataSet=0;
```

⟨*main routine*⟩≡
```
int main(int argc, char *argv[])
{
  options(argc, argv);
  version(argv[0]);
```

⟨*definitions*⟩≡

```
int cantdatos;    // number of data available
//indexes
lint inputs, outputs, outpcmp;
char* type;

read DS(flag("datafile").string);
inputs = VECTOR SIZE;

bool sangerT=0, ojaT=0, apexT=0, svdT=0, svdrT=0, spcaT=0, spca2T=0, pmT=0;
```

⟨*General user interface*⟩≡
```
if (!strcmp(flag("method").string, "Sanger"))
  {  sangerT = 1;}
else if (!strcmp(flag("method").string, "Oja"))
  {  ojaT = 1;}
else if (!strcmp(flag("method").string, "APEX"))
  {   apexT = 1;}
else if (!strcmp(flag("method").string, "SVD"))
  { svdT = 1;}
else if (!strcmp(flag("method").string, "SVD-R"))
  { svdrT = 1;}
else if (!strcmp(flag("method").string, "PM"))
  {  pmT = 1;}
else if (!strcmp(flag("method").string, "SPCA"))
  { spcaT = 1;}
else if (!strcmp(flag("method").string, "SPCA 2"))
  { spca2T = 1;}
else cout <<"\Error: Unidentified method\n";

if (flag("instances"))
  cantdatos = (int)flag("instances").value;
else
  cantdatos = Instances;

if (flag("factors")   !svdT   !svdrT )
  outputs = (lint)flag("factors").value;
else
  outputs = inputs;

 if (flag("factorsCmp")   !(flag("compare").string=="SPCA"))
  outpcmp = (lint)flag("factorsCmp").value;
 else
  outpcmp = inputs;

 cerr <<"\  outputs:"<<outputs<<"\toutcmp:"<<outpcmp <<"\n";

int verbose=0;
int savef=0;
if (flag("verbose"))  verbose = 1;
if (flag("savef"))    savef = 1;
```

⟨*more definitions*⟩≡

```
matrix weights(outputs, inputs); // eigvectors in rows
vector evals(outputs) ;          // eigenvalues
float dtrace = 0;
vector exp var(outputs);

matrix data(Instances,VECTOR SIZE);
vector classes(Instances);
matrix scores(Instances,outputs);

// output files
char* met x = ".method";
char* cmp x = ".compare";

ofstream bs method of(date ext(met x));  // file: date.method
if (! bs method of) {
  cerr << "I can't open file for saving bootstrap results\n";
  exit(1);
}

// open file   stream for writing results
ofstream bs corr of(date ext(cmp x));  // file: date.compare
if (! bs corr of) {
  cerr << "I can't open file for saving bootstrap correlation results\n";
  exit(1);
}
```

### 4.3 Bootstraping

⟨*begin bootstraping*⟩≡

```
int bs[Instances];   // this bs sample index table
int bs samples;      // total requested
int bs s;            // bs samples done

if (flag("bootstrap"))
  bs samples = (int)flag("bootstrap").value;
else
  bs samples = 1;

matrix bs evals(bs samples,outputs);    // to keep evals
vector bs corr(bs samples);   // corr between 2 methods
vector bs trace(bs samples);  // trace of each sample
matrix bs corr evect(bs samples,outputs); // corr between evects

for (bs s=0;bs s<bs samples;bs s++) {   // begins bootstraping

  if (!flag("bootstrap")) {
    for (uint in=0; in<Instances; in++) {
      classes[in] = (float) DataSet[in].label;
      for (uint d=0; d<VECTOR SIZE; d++)
        data[in][d] = DataSet[in].vec[d];
    }
  }

  if (flag("bootstrap")) {
    if(bs s==0) {  // is nice to have the first = original
      for (uint in=0; in<Instances; in++) {
        bs[in] = in;
        classes[in] = (float) DataSet[in].label;
        for (uint d=0; d<VECTOR SIZE; d++)
          data[in][d] = DataSet[in].vec[d];
      }
    }
    else {
      for (uint in=0; in<Instances; in++) {
        bs[in] = (int)(drand48()*Instances);
        // saving sample? bsfile << bs[in]<< " ";
        classes[in] = (float) DataSet[bs[in]].label;
        for (uint d=0; d<VECTOR SIZE; d++)
          data[in][d] = DataSet[bs[in]].vec[d];
      }
```

```
    }
  }
  vector mean;
  if (!flag("no mean corr")) {
    mean = data.mean correct();
  }
  if (flag("save mean"))   {
    char* name = flag("save mean").string;
    ofstream mean fp(name);
    if (!mean fp) {
      cerr << "I can't open file for saving means\n";
      exit(1);
    }
    mean fp << mean;
  }


  weights = m drand48(outputs,inputs) - 0.5;
  //    for(int i=0;i<weights.filas();i++) {
  //  float length = weights[i].norma();
  //  weights[i] = weights[i]/length;//I think they only have to be small..
  //}
```

⟨*run method*⟩≡

```
type = flag("method").string;
run method(data,classes,scores,weights,evals,dtrace,type);
```

some of the methods produce the eigenvectors in rows instead of columns, here I traspouse the matrix.

⟨*show and traspouse if needed*⟩≡

```
 exp var = evals * (100/dtrace);

if(savef)   save(weights);

bs evals[bs s] = evals;
bs trace[bs s] = dtrace;
if (!(flag("bootstrap")) || verbose)
  cout <<"\nmethod "<<flag("method").string<< evals;
if(verbose) show bs txt(evals);
```

A measure of similarity between two different methods is the correlation between its results. Note that Im using sample variance. For the same bootstrap sample, we produce another eigenvectors and eigenvalues with other method. We then calculate calculate the correlation between the solutions. We produce a mean and a variance for the bootstrap, the later been the standard error of the correlation..

Some methods change the data matrix to a scores matrix (e.g. SPCA) so `data` has to be read and mean corrrected again.

⟨*distance between methods*⟩≡
```
if (flag("compare")) { // begin compare
   vector evals 2(outpcmp);
   matrix weights 2(outpcmp,inputs);
   //matrix evect 2(inputs,outpcmp);
   vector ev(inputs), evo(inputs);
   char *cmp = flag("compare").string;

   int minouts;
   if (outputs<outpcmp) {minouts = outputs;} else { minouts = outpcmp; }

   // read data set again
   for (uint in=0; in<Instances; in++) {
     for (uint d=0; d<VECTOR SIZE; d++) {
       if(flag("bootstrap"))
         data[in][d] = DataSet[bs[in]].vec[d]; //bs[in] gen above
       else
         data[in][d] = DataSet[in].vec[d];
     }
   }

   // run 2nd method
   run method(data,classes,scores,weights 2,evals 2,dtrace,cmp);

   //    if (strcmp(flag("compare").string, "SVD"))
   //  evect 2 = (weights 2).T();// eigenvect set to columns

   // correlation on eigenvalues
   bs corr[bs s] = corr(evals, evals 2, minouts);
   // correlation on eigenvectors
   for(int o=0;o<outpcmp   o<outputs; o++) {
     ev  = weights[o];
     evo = weights[o];
     bs corr evect[bs s][o] = corr(ev,evo);
   }
```

```
      // stdout dump
      if(verbose)
        cout << "\ncorrelating " << flag("method").string
          << " with "<<flag("compare").string <<"...\n"
          <<"\nmethod eigenvalues"<<flag("method").string<< evals
          <<"\nmethod eigenvalues"<<flag("compare").string<< evals 2
          <<"\neigenvector for "<<flag("method").string<< weights
          <<"\neigenvector for "<<flag("compare").string<<weights 2;
   }   // end compare
```

⟨*end bootstraping*⟩≡

```
    }  // end bootstraping
    // method stats saved on file
    save bs result1(bs evals, dtrace,bs method of, 10);
    // usefull correlation results saved on file
      save bs result2(bs evals,bs trace,bs corr evect,bs corr,bs corr of,10);
```

⟨*end*⟩+≡

```
      return 0;
  }
```

⟨*run method code*⟩≡
```
float data trace(matrix  data) {
    float dtrace = 0;
    for (lint i=0;i<data.columns();i++)
      dtrace += data.separa col(i).variance();
    return dtrace;
}
float expvar(vector  evals, float dtrace)
{
  float ev = 0;
  ev = evals.suma();
  return (ev/dtrace)*100;
}
void run method(matrix  data, vector  classes,matrix  scores,matrix  weights,vector  evals
dtrace, char* type)
{
  struct tms start, stop;
  struct tms start alg, stop alg;
  lint inputs = data.columns();
  lint outputs = weights.filas();

  times( start);

 if (!strcmp(type, "SPCA"))  { // Standard L1 SPCA
   dtrace = data trace(data);
   times( start alg);
   SPCA std(data, classes, scores, weights, evals);
   times( stop alg);
   // scores are already returned in data but with extra columns,
   // if not timing, correct scores are produced but data is lost
 }
 else if (!strcmp(type, "SPCA 2")) { // Standard L2 SPCA
   dtrace = data trace(data);
   times( start alg);
   SPCA 2(data, classes, scores, weights, evals);
   times( stop alg);
   // scores are already returned in data but with extra columns,
   // if not timing, correct scores are produced but data is lost
 }
 else if (!strcmp(type, "Sanger"))    {  // Sanger's rule
    dtrace = data trace(data);
    times( start alg);
    sanger(data, scores, weights, evals);
    times( stop alg);
```

```
}
else if (!strcmp(type, "Oja"))    {  //regla de Oja
  dtrace = data trace(data);
  times( start alg);
  oja(data, scores, weights, evals);
  times( stop alg);
}
else if (!strcmp(type, "APEX"))    {
  dtrace = data trace(data);
  times( start alg);
  apex(data, scores, weights, evals);
  times( stop alg);
}
else if (!strcmp(type, "SVD"))    { // SVD on variance-covariance matrix
  if(outputs!=inputs) cout <<"Error: outputs!=inputs, wrong for PCA";
  times( start alg);
  matrix C(inputs, inputs);    // variance-covariance or correl matrix
  C=Covar(data);
  dtrace=trace(C);
  PCA(C, evals, weights);
  times( stop alg);
}
else if (!strcmp(type, "SVD-R"))   { // SVD on correlation matrix
  if(outputs!=inputs) cout <<"Error: outputs!=inputs, wrong for PCA";
  matrix C(inputs, inputs);    // variance-covariance or correl matrix
  C=Correl(data);
  dtrace=trace(C);
  times( start alg);
  PCA(C, evals, weights);
  times( stop alg);
}
else if (!strcmp(type, "PM"))    {  // power method
  dtrace = data trace(data);
  times( start alg);
  power method(data,scores,weights,evals);
  times( stop alg);
}
else cout <<"\Error: Unidentified method\n";

// end switch
times( stop);

if (flag("time"))
  fprintf(stdout, " Time:  gu  gs\n",
```

```
          (stop.tms utime - start.tms utime)*CLK TCK/10000.0,
          (stop.tms stime - start.tms stime)*CLK TCK/10000.0);

  if (flag("time alg"))
    fprintf(stdout, "Time for the algorithm:  gu  gs\n",
          (stop alg.tms utime - start alg.tms utime)*CLK TCK/10000.0,
          (stop alg.tms stime - start alg.tms stime)*CLK TCK/10000.0);

  if (flag("Ptype"))   cout << "\  Method: "<< type << '\n';
  if (flag("Pexpvar"))
    cout << "\  In "<<evals.length()
         <<" components: " <<expvar(evals,dtrace) <<"\ \n";
  if (flag("Pevals"))    cout << "\  "<<evals <<"("<<dtrace<<")" <<'\n';
  if (flag("Pweights"))  cout << weights << '\n';

  if(flag("Sweights"))   {
    ofstream weights str(flag("Sweights").string);
    if (! weights str) {
      cerr << "I can't open file for saving weights\n";
      exit(1);
    }
    weights.save(weights str);
  }
 }
```

Old version to calculate eigenvalues, I dont remember what I did.

⟨*stat.cc*⟩≡

```
include "vector.h"
include "matrix.h"
include "main.h"

vector eigen W(const matrix  data,const matrix  weights) {
  // Basado en calculos de Rafael.
  int N=data.filas();            // N = casos o patrones
  int outputs=weights.filas();
  matrix salidas(N,data.columns());
  float suma;
  vector autoval(outputs);
  for(int r=0;r<outputs;r++) {
    suma=0;
    for(int k=0;k<N;k++) {
      salidas[k] = weights.T() * data[k];
      // original: salidas[k] = weights * data[k];
      suma+=data[k][r]*salidas[k][r];
    }
    //          cout << "\nautoval["<<r<<"]="<<suma<<"/"<<(N*weights[r][r]);
    autoval[r]=suma/(N*weights[r][r]);
    //          autoval[r]=(salidas.separa col(r)).variance();
  }
return autoval;
}
```

# 5 Tool functions

## 5.1 Eigenvalue calculation

New version for calculation eigenvalues, basically I calculate the variance of the data in the new representation, the variance in each variable is the eigenvalue. The explained variance of each component i is $\sigma_i = variance(y_i)$

The weights $W_{ij}$ where i are outputs and j are inputs would be the traspouse of the eigenvectors.

$\langle stat.cc \rangle{+}{\equiv}$

```
vector e val(const matrix  data,const matrix  weights, matrix  scores)
{
  int outputs = weights.filas();
  vector eval(outputs);

  for (int i=0; i<outputs; i++)
        for (int j=0; j<data.filas(); j++) {
          scores[j][i] = data[j] * weights[i];
        }
  for (int i=0;i<weights.filas();i++)
    eval[i] = scores.separa col(i).variance();
  return eval;
}
```

## 5.2 Statistical tools

Some statistical functions are needed that are to specific to be in the `matrix` or `vector` libraries.

⟨*stat.cc*⟩+≡
```
float trace(matrix  Cov) {
  float trace=0;
  assert(Cov.filas()==Cov.columns());
  for(int i=0;i<Cov.columns();i++)
    trace+= Cov[i][i];
  return trace;
}
void wsmall(matrix  weights)
{
  for (int i = 0; i < weights.filas(); i++)
    if (weights[i].norma2() > 0.25)
      cout << "\nInitial weights are not small enough." <<
        "problems in the output unit: " << i;
}
```

### 5.3 I/O functions

Some functions are needed for printing and saving, that are to specific to be in the `matrix` or `vector` libraries.

⟨*ioh.cc*⟩≡

```
include "vector.h"
include "matrix.h"
include "main.h"

void imprime con suma transp(matrix  weights) {
  cout << "\nMatriz de pesos traspuesta (pa que entre)\n";
  int i;
  for(i=0;i<weights.filas();i++)  cout <<"\tFactor "<<i;
  cout<<'\n';
  for(i=0;i<weights.columns();i++) {
    cout << "Input: "<<i<<": ";
    for(int j=0;j<weights.filas();j++)  {
      cout<<weights[j][i]<<"\t";
    }
    cout << "\n";
  }
  vector col(weights.filas());

  for(i=0;i<weights.filas();i++) {
    cout<<weights[i]*weights[i]<<'\t';
  }
  cout << "\nComunalities:\n";
  for(i=0;i<weights.columns();i++) {
    col = weights.separa col(i);
    float suma=col.norma2();
    cout<<"Input " << i <<": " <<suma <<"\n";
  }
}

void show epoch(matrix  weights,matrix  data,vector  evals,float epocas,matrix  scores)
{
  int outputs = weights.filas();
  cout << "\n------  EPOCHS: " << epocas;
  evals = e val(data,weights,scores);
  //evals = eigen W(data,weights);
  vector var exp(outputs);
  cout << "\neigenvalues:"<<evals;
}
```

```
char* date ext(char* res f)
{
time t t= time(0);
  char*p= ctime( t);
  char name[25];
  strncpy(name,p,3);
  strncpy(name+3,p+4,1);
  strncpy(name+4,p+11,2);
  strncpy(name+6,p+14,2);
  strncpy(name+8,res f,5);
  return name;
}
void save(matrix   weights) {
  time t t= time(0);
  char*p= ctime( t);
  char name[15];
  strncpy(name,p,3);
  strncpy(name+3,p+4,1);
  strncpy(name+4,p+11,2);
  strncpy(name+6,p+14,2);
  strncpy(name+8,".wts",5);
  ofstream ov(name);
  if (!ov) {
      cerr << "I can't open file for saving \n";
      exit(1);
  }
  weights.save(ov);
}
// void show txt(vector  evals, float dtrace, matrix  data, matrix  evect,
//           vector  exp var)
// {
//   cout<<"\nsum of evals:"<<evals.suma()<<" || original trace= "<< dtrace;
//   cout << "\n--------\nEigenvalues:" << evals;
//   evals = e val(data,evect.T(),scores);
//   cout << "\nEigenvalues (as variance of scores):" << evals;
//   cout << "\nexp. variances ( ):" << exp var;
//   cout<<"\nEigenvectors: \n";
//   evect.imprime();   // eigenvectors must be in columns

//   uint outputs = evect.columns();
//   vector tot evar(outputs);
//   tot evar[0] = exp var[0];

//   for(uint i=1;i<outputs;i++)
```

```
//     tot evar[i] = tot evar[i-1] + exp var[i];
//   cout << "\nexp/ variances acum.:" << tot evar;
//   cout<<"\n---";

// }

void show bs txt(vector  evals)
{
  cout << "\n--------\nEigenvalues:" << evals;
}
```

What do we expect to obtain from bootstrap?

If only one method is been studied: mean and standard deviation of eigenvectors (taking care of sign) and eigvalues.

If two method are been compared: Each mean and standard dev. as above. The correlation of the means above. The mean bootstraped correlation and its standard error.

⟨*ioh.cc*⟩+≡

```
include<Options.h>

// Code for comparing 2 methods

void save bs result2(matrix  bs evals,vector  bs trace, matrix  bs corr evect, vector  bs
bs out, uint o)
{
  bs out << "\  latex code generated by hebb"
         << "\n\ Comparison results"
         <<"\n\ using "<<(int)flag("bs s").value <<" bootstrap samples"
         << "\n\ On class "<< flag("class").value
         << "\n\ pm it="<<flag("pm it").value
         << " b its= " << flag("b its").value
         << "\n\ using "<< bs evals.columns()<<" outputs"
         <<"\n\ Method: "<< flag("method").string
           << " Compared with: " << flag("compare").string <<"\n";

  uint outputs = bs evals.columns();
  matrix bs summary(2,outputs); // mean and desvio
  matrix bs sum corr(2,outputs);

  for(int i=0;i<bs corr evect.filas();i++)
     bs corr evect[i]= bs corr evect[i].abs val();

  // this line to print all corr on 1st eigvect
//  vector tmp(100);
//  tmp = bs corr evect.separa col(0);
//  cout << tmp;

  for (uint col=0;col<outputs ;col++) {
    bs sum corr[0][col] = bs corr evect.separa col(col).media();
    bs sum corr[1][col] = bs corr evect.separa col(col).desvio();
  }
  if(flag("latex"))
    matrix2latex(bs sum corr, bs out,o);
```

```
  float dtrace = 1;
  dtrace = bs trace.media();

  float corr=0;
  corr = bs sum corr[0].suma()/outputs;

  bs out << "\nCorrelation between eigenvectors (As vector)"
          << " of "<< flag("method").string << " and "
          << flag("compare").string <<": "
          << bs sum corr[0]
          << " \n+/- (1 std dev)\n "
          << bs sum corr[1];

  vector bs corrW(outputs);
  for(uint i=0;i<outputs;i++)
    bs corrW[i] = 0;
  for(uint i=0;i<outputs;i++)
    for(int j=0;j<bs corr evect.filas();j++)
      bs corrW[i]+=(bs corr evect[j][i]*bs evals[j][i])/bs evals[j].suma();

  bs out << "\nThe global correlation between eigenvectors"
          << "(all means added, divided by  outputs): "
          << corr  <<"\n"
          << "\nWeighted (by evals/evals.suma() ):"<< bs corrW
          << "\nMean of weighted:"<<bs corrW.media();


  bs out  << "\n\n"
           << "\nCorrel. between eigenvalues corr(pca,other)= "
           << bs corr.media() <<"+- "
           << bs corr.desvio()<<"\n";
}

void save bs result1(matrix  bs evals, float trace,ofstream  bs out, uint o)
{

  bs out << "\  latex code generated by hebb"
         << "\n\ Results for method: "<< flag("method").string
           <<"\n\ using "<<flag("bs s").value <<"bootstrap samples"
           << "\n\ On class "<< flag("class").value
             << "\n\ using "<<bs evals.columns()<<" outputs";

  if (flag("latex")) {
```

```
    bs out << "\n\\documentstyle[epsfig]{article}\n\\begin{document}"
           << "\n\\section{Result of methods}";
  }
  uint outputs = bs evals.columns();
  matrix bs summary(2,outputs); // mean and desvio

  for (uint col=0;col<outputs ;col++) {
    bs summary[0][col] = bs evals.separa col(col).media();
    bs summary[1][col] = bs evals.separa col(col).desvio();
  }

  if(flag("latex"))
    matrix2latex(bs summary, bs out,o);
  bs out <<"\nTrace: "<<trace;
  bs out<<"\nEigenvalues Bootstraped mean (row 1) and 1 stand. dev. (row 2):\n";
  bs summary.save(bs out);
}
```

## 5.4 Graphic Tools

Mathew has implemeted two tools for managing results. Making postscript files, looking at plots of statistics... They are in `db2ps` and `boxplot`. I document here the info files and write some examples.

⟨*db2ps.info*⟩≡

```
db2ps [options] [-- items [-blank] [-nl] ...]
           Convert a file in Matts DB format to PostScript.

invert
invert      Invert the images ( 0s s).
down=upsidedown
down        Show the images upsidedown ( 0s s).
box
box         Draw a box around the images ( 0s s).
ts=time-series
time-series Display the data as time-series plots ( 0s s).
l=label
l, label    Put labels under figures (fontsize s  s).
prolog
prolog      PostScript prolog file s.
i=input
i, input    Input filename s.
o=output
o, output   Output filename s.
w=width
w, width    Width of each image in the grid w s.
h=height
h, height   Height of each image in the grid h s.
wr=wrap
wr, wrap    Number of images per line wr s.
px=padx
px, padx    Padding between images padx s.
py=pady
py, pady    Padding between lines pady s.

-blank      Insert a blank rectangle between images.
-nl         Insert a new line between images.
```

⟨*saving in DB*⟩≡

```
float V[VEC SIZE];  //must be normilzed to 0,1
  uchar buf[VEC SIZE];
  for(uint i=o;i<VEC SIZE;i++)
    int t=V[i]*256;
    if (t<0)
      buf[i]=0;
    else if (t>255)
      buf[i]=255;
    else
      buf[i] = (uchar) t;

FILE *fout;
fwrite(buf,sizeof(uchar),VRC SIZE,fout);
```

@

# 6 User Manual

⟨*hebb.rc*⟩≡

```
method=SPCA
-compare=SVD
datafile=/home/feat/data/escuela/escuela.mp
-Winit
-instances
-no mean corr
-factors
-factorsCmp
-verbose
-class=8
-savef=0
savesc=scores.pc
save eval=exper.eval
save mean=exper.mean
-bootstrap=0
-time
-b its=1
-pm it=10
bs s=1
-time alg
-latex
-Ptype
-Pevals
-Pexpvar
-Pweights
-Sweights=Wend.wts
```

⟨*hebb.opt*⟩≡

```
00hebb:
00Principal component analysis techniques.

00INPUT
Input filename and attributes.
01D=datafile
Input dataset filename.
11i=instances
Number of instances to be used from dataset.
11c=class
Class of points.

00METHOD
01m=method
Sanger, Oja, APEX, SPCA, SPCA 2, SVD, PM
01cmp=compare
Sanger, Oja, APEX, SPCA, SPCA 2, SVD, PM
01f=factors
Maximum number of principal components to determine.
01fc=factorsCmp
Maximum number of principal components in compare method.
11bs s=bootstrap
Botstrap samples
00I/O
01res=R=resources
Resources filename.

01b its=batch
SPCA batch iterations

01pm it=powerm
Power method iterations

01s=savef
Save files
11v=verbose
Level of verbosity.
11t=time
Measure of CPU time
11l=latex
Latex output summary
01Pt=Ptype
Print method name
```

```
01Pe=Pexpvar
Print total explained variance
01Pv=Pevals
Print eigenvalues and trace
```

Old version to calculate eigenvalues, I dont remember what I did.

⟨*stat.cc*⟩+≡

```
include "vector.h"
include "matrix.h"
include "main.h"

vector eigen W(const matrix  data,const matrix  weights) {
  // Basado en calculos de Rafael.
  int N=data.filas();           // N = casos o patrones
  int outputs=weights.filas();
  matrix salidas(N,data.columns());
  float suma;
  vector autoval(outputs);
  for(int r=0;r<outputs;r++) {
    suma=0;
    for(int k=0;k<N;k++) {
      salidas[k] = weights.T() * data[k];
      // original: salidas[k] = weights * data[k];
      suma+=data[k][r]*salidas[k][r];
    }
    //          cout << "\nautoval["<<r<<"]="<<suma<<"/"<<(N*weights[r][r]);
    autoval[r]=suma/(N*weights[r][r]);
    //          autoval[r]=(salidas.separa col(r)).variance();
  }
return autoval;
}
```

# 7 Tool functions

## 7.1 Eigenvalue calculation

New version for calculation eigenvalues, basically I calculate the variance of the data in the new representation, the variance in each variable is the eigenvalue. The explained variance of each component i is $\sigma_i = variance(y_i)$

The weights $W_{ij}$ where i are outputs and j are inputs would be the traspouse of the eigenvectors.

$\langle stat.cc \rangle{+}\equiv$

```
vector e val(const matrix  data,const matrix  weights, matrix  scores)
{
  int outputs = weights.filas();
  vector eval(outputs);

  for (int i=0; i<outputs; i++)
        for (int j=0; j<data.filas(); j++) {
          scores[j][i] = data[j] * weights[i];
        }
  for (int i=0;i<weights.filas();i++)
    eval[i] = scores.separa col(i).variance();
  return eval;
}
```

## 7.2 Statistical tools

Some statistical functions are needed that are to specific to be in the `matrix` or `vector` libraries.

⟨*stat.cc*⟩+≡
```
float trace(matrix  Cov) {
  float trace=0;
  assert(Cov.filas()==Cov.columns());
  for(int i=0;i<Cov.columns();i++)
    trace+= Cov[i][i];
  return trace;
}
void wsmall(matrix  weights)
{
  for (int i = 0; i < weights.filas(); i++)
    if (weights[i].norma2() > 0.25)
      cout << "\nInitial weights are not small enough." <<
        "problems in the output unit: " << i;
}
```

## 7.3 I/O functions

Some functions are needed for printing and saving, that are to specific to be in the `matrix` or `vector` libraries.

⟨*ioh.cc*⟩+≡

```
include "vector.h"
include "matrix.h"
include "main.h"

void imprime con suma transp(matrix  weights) {
  cout << "\nMatriz de pesos traspuesta (pa que entre)\n";
  int i;
  for(i=0;i<weights.filas();i++)  cout <<"\tFactor "<<i;
  cout<<'\n';
  for(i=0;i<weights.columns();i++) {
    cout << "Input: "<<i<<": ";
    for(int j=0;j<weights.filas();j++)  {
      cout<<weights[j][i]<<"\t";
    }
    cout << "\n";
  }
  vector col(weights.filas());

  for(i=0;i<weights.filas();i++) {
    cout<<weights[i]*weights[i]<<'\t';
  }
  cout << "\nComunalities:\n";
  for(i=0;i<weights.columns();i++) {
    col = weights.separa col(i);
    float suma=col.norma2();
    cout<<"Input " << i <<": " <<suma <<"\n";
  }
}

void show epoch(matrix  weights,matrix  data,vector  evals,float epocas,matrix  scores)
{
  int outputs = weights.filas();
  cout << "\n------  EPOCHS: " << epocas;
  evals = e val(data,weights,scores);
  //evals = eigen W(data,weights);
  vector var exp(outputs);
  cout << "\neigenvalues:"<<evals;
}
```

```
char* date ext(char* res f)
{
time t t= time(0);
  char*p= ctime( t);
  char name[25];
  strncpy(name,p,3);
  strncpy(name+3,p+4,1);
  strncpy(name+4,p+11,2);
  strncpy(name+6,p+14,2);
  strncpy(name+8,res f,5);
  return name;
}
void save(matrix   weights) {
  time t t= time(0);
  char*p= ctime( t);
  char name[15];
  strncpy(name,p,3);
  strncpy(name+3,p+4,1);
  strncpy(name+4,p+11,2);
  strncpy(name+6,p+14,2);
  strncpy(name+8,".wts",5);
  ofstream ov(name);
  if (!ov) {
      cerr << "I can't open file for saving \n";
      exit(1);
  }
  weights.save(ov);
}
// void show txt(vector  evals, float dtrace, matrix  data, matrix  evect,
//           vector  exp var)
// {
//   cout<<"\nsum of evals:"<<evals.suma()<<" || original trace= "<< dtrace;
//   cout << "\n--------\nEigenvalues:" << evals;
//   evals = e val(data,evect.T(),scores);
//   cout << "\nEigenvalues (as variance of scores):" << evals;
//   cout << "\nexp. variances ( ):" << exp var;
//   cout<<"\nEigenvectors: \n";
//   evect.imprime();   // eigenvectors must be in columns

//   uint outputs = evect.columns();
//   vector tot evar(outputs);
//   tot evar[0] = exp var[0];

//   for(uint i=1;i<outputs;i++)
```

```
//     tot evar[i] = tot evar[i-1] + exp var[i];
//   cout << "\nexp/ variances acum.:" << tot evar;
//   cout<<"\n---";

// }

void show bs txt(vector  evals)
{
  cout << "\n--------\nEigenvalues:" << evals;
}
```

What do we expect to obtain from bootstrap?

If only one method is been studied: mean and standard deviation of eigenvectors (taking care of sign) and eigvalues.

If two method are been compared: Each mean and standard dev. as above. The correlation of the means above. The mean bootstraped correlation and its standard error.

⟨*ioh.cc*⟩+≡

```
include<Options.h>

// Code for comparing 2 methods

void save bs result2(matrix  bs evals,vector  bs trace, matrix  bs corr evect, vector  bs
bs out, uint o)
{
  bs out << "\  latex code generated by hebb"
         << "\n\ Comparison results"
         <<"\n\ using "<<(int)flag("bs s").value <<" bootstrap samples"
         << "\n\ On class "<< flag("class").value
         << "\n\ pm it="<<flag("pm it").value
         << " b its= " << flag("b its").value
         << "\n\ using "<< bs evals.columns()<<" outputs"
         <<"\n\ Method: "<< flag("method").string
           << " Compared with: " << flag("compare").string <<"\n";

  uint outputs = bs evals.columns();
  matrix bs summary(2,outputs); // mean and desvio
  matrix bs sum corr(2,outputs);

  for(int i=0;i<bs corr evect.filas();i++)
     bs corr evect[i]= bs corr evect[i].abs val();

  // this line to print all corr on 1st eigvect
//  vector tmp(100);
//  tmp = bs corr evect.separa col(0);
//  cout << tmp;

  for (uint col=0;col<outputs ;col++) {
    bs sum corr[0][col] = bs corr evect.separa col(col).media();
    bs sum corr[1][col] = bs corr evect.separa col(col).desvio();
  }
  if(flag("latex"))
    matrix2latex(bs sum corr, bs out,o);
```

```
  float dtrace = 1;
  dtrace = bs trace.media();

  float corr=0;
  corr = bs sum corr[0].suma()/outputs;

  bs out << "\nCorrelation between eigenvectors (As vector)"
          << " of "<< flag("method").string << " and "
          << flag("compare").string <<": "
          << bs sum corr[0]
          << " \n+/- (1 std dev)\n "
          << bs sum corr[1];

  vector bs corrW(outputs);
  for(uint i=0;i<outputs;i++)
    bs corrW[i] = 0;
  for(uint i=0;i<outputs;i++)
    for(int j=0;j<bs corr evect.filas();j++)
      bs corrW[i]+=(bs corr evect[j][i]*bs evals[j][i])/bs evals[j].suma();

  bs out << "\nThe global correlation between eigenvectors"
          << "(all means added, divided by  outputs): "
          << corr  <<"\n"
          << "\nWeighted (by evals/evals.suma() ):"<< bs corrW
          << "\nMean of weighted:"<<bs corrW.media();


  bs out  << "\n\n"
           << "\nCorrel. between eigenvalues corr(pca,other)= "
           << bs corr.media() <<"+- "
           << bs corr.desvio()<<"\n";
}

void save bs result1(matrix  bs evals, float trace,ofstream  bs out, uint o)
{

  bs out << "\  latex code generated by hebb"
         << "\n\ Results for method: "<< flag("method").string
           <<"\n\ using "<<flag("bs s").value <<"bootstrap samples"
           << "\n\ On class "<< flag("class").value
             << "\n\ using "<<bs evals.columns()<<" outputs";

  if (flag("latex")) {
```

```
    bs out << "\n\\documentstyle[epsfig]{article}\n\\begin{document}"
           << "\n\\section{Result of methods}";
  }
  uint outputs = bs evals.columns();
  matrix bs summary(2,outputs); // mean and desvio

  for (uint col=0;col<outputs ;col++) {
    bs summary[0][col] = bs evals.separa col(col).media();
    bs summary[1][col] = bs evals.separa col(col).desvio();
  }

  if(flag("latex"))
    matrix2latex(bs summary, bs out,o);
  bs out <<"\nTrace: "<<trace;
  bs out<<"\nEigenvalues Bootstraped mean (row 1) and 1 stand. dev. (row 2):\n";
  bs summary.save(bs out);
}
```

## 7.4 Graphic Tools

Mathew has implemeted two tools for managing results. Making postscript files, looking at plots of statistics... They are in `db2ps` and `boxplot`. I document here the info files and write some examples.

⟨*db2ps.info*⟩+≡

```
db2ps [options] [-- items [-blank] [-nl] ...]
           Convert a file in Matts DB format to PostScript.

invert
invert      Invert the images ( 0s s).
down=upsidedown
down        Show the images upsidedown ( 0s s).
box
box         Draw a box around the images ( 0s s).
ts=time-series
time-series Display the data as time-series plots ( 0s s).
l=label
l, label    Put labels under figures (fontsize s  s).
prolog
prolog      PostScript prolog file s.
i=input
i, input    Input filename s.
o=output
o, output   Output filename s.
w=width
w, width    Width of each image in the grid w s.
h=height
h, height   Height of each image in the grid h s.
wr=wrap
wr, wrap    Number of images per line wr s.
px=padx
px, padx    Padding between images padx s.
py=pady
py, pady    Padding between lines pady s.

-blank      Insert a blank rectangle between images.
-nl         Insert a new line between images.
```

⟨*saving in DB*⟩+≡

```
float V[VEC SIZE];  //must be normilzed to 0,1
  uchar buf[VEC SIZE];
  for(uint i=o;i<VEC SIZE;i++)
    int t=V[i]*256;
    if (t<0)
      buf[i]=0;
    else if (t>255)
      buf[i]=255;
    else
      buf[i] = (uchar) t;

FILE *fout;
fwrite(buf,sizeof(uchar),VRC SIZE,fout);
```