Help

```
/** @addtogroup CLUSTERING
* @{
*/

/**
 * @author Ricardo Rincón García
 *
 * @file clustering.c
 * @brief Library with 3 clustering methods : K-means, Recursive Bifurcation and
 *        Recursive Bifurcation of Reduced State Space.
 *
 * @date 17.08.15
 */
#include "
href../../common/math/clustering_h_src.pdfclustering.h"


/**
 * @brief Free memory allocated for a Model_kmeans structure.
 *
 * @param [in, out] mod Model_kmeans structure to free.
 *
 * @related Model_kmeans
 */
void freeModel_kmeans(Model_kmeans* mod)
{
    if(mod != NULL)
    {
        if(mod->centroids != NULL)
        {
            free(mod->centroids);
            mod->centroids = NULL;
        }
        if(mod->labels != NULL)
        {
            free(mod->labels);
            mod->labels = NULL;
        }
        if(mod->MsErr != NULL)
```

```
        {
            free(mod->MsErr);
            mod->MsErr = NULL;
        }
    }
}


/**
 * @brief Get the distance squared between two indicated points.
 *
 * @param [in] p1 Point 1.
 * @param [in] p2 Point 2.
 * @param [in] dim Points dimension.
 *
 * @return Distance squared between point 1 and point 2.
 */
static double getDistanceSq(double* p1, double* p2, int dim)
{
    int i;
    double distanceSq = 0.;
    assert(p1 != NULL);
    assert(p2 != NULL);

    for (i = 0; i < dim; i++)
        distanceSq += SQR(p2[i] - p1[i]);

    return distanceSq;
}


/**
 * @brief Assign to each point the nearest cluster.
 *
 * @param [in] X A vector of points (a matrix).
 * @param [in] n Number of points (number of rows of X).
 * @param [in] dim Points dimension (number of columns of X).
 * @param [in] k Number of clusters.
 * @param [in] centroids Cluster's centroids.
 * @param [out] assignedClustInd Index of the cluster which each point is assign
 * @param [out] argToMin The sum of squares within clusters; argument to minimiz
```

```
 *                        in kmeans.
 */
static void assignNearestCluster(double* X, int n, int dim, int k, double* centr
                                 int* assignedClustInd, double* argToMin)
{
  int i, j;
    int bestIndex;
    double closestDistance, currentDistance;

    assert(X != NULL);
    assert(centroids != NULL);
    assert(assignedClustInd != NULL);
    assert(argToMin != NULL);

    *argToMin = 0.;

    for (i = 0; i < n; i++)     // for each point
    {
        bestIndex = -1;
        closestDistance = INFINITY;

        for (j = 0; j < k; j++) // for each cluster
        {
            // distance squared between point and cluster centroid
            currentDistance = getDistanceSq(X + i*dim, centroids + j*dim, dim);

            if (currentDistance < closestDistance)
            {
                bestIndex = j+1; // clusters labels begin in 1
                closestDistance = currentDistance;
            }
        }
        assignedClustInd[i] = bestIndex;

        // sum distances squared of every point in the same cluster
        *argToMin += getDistanceSq(&X[i*dim],
                                   &centroids[(assignedClustInd[i]-1)*dim],
                                   dim);
    }

    return;
```

```
}


/**
 * @brief Get the new centroids of all clusters.
 *
 * @param [in] X A vector of points (a matrix).
 * @param [in] n Number of points (number of rows of X).
 * @param [in] dim Points dimension (number of columns of X).
 * @param [in] k Number of clusters.
 * @param [in] assignedClustInd Index of the cluster which each point belongs to
 * @param [out] newClustersCentroids The new centroids of all clusters.
 *
 * @warning If any cluster is empty.
 */
static void getNewClustersCentroids(double* X, int n, int dim, int k,
                                    int* assignedClustInd,
                                    double* newClustersCentroids)
{
    int i, j;
    int currentCluster;
    int* clustersCardinals = (int*) malloc(k*sizeof(int));
    if (clustersCardinals == NULL)
        return;


    assert(X != NULL);
    assert(assignedClustInd != NULL);
    assert(newClustersCentroids != NULL);


    // initialize cluster centroids coordinates sums to zero
    for (i = 0; i < k; i++) // for each cluster
    {
        clustersCardinals[i] = 0;

        for (j = 0; j < dim; j++)
            newClustersCentroids[i*dim + j] = 0.;
    }

    for (i = 0; i < n; i++) // for every point
```

```
    {
        // which cluster is it in?
        currentCluster = assignedClustInd[i] - 1; // clusters labels begin in 1

        // update count of members in that cluster
        clustersCardinals[currentCluster]++;

        // sum point coordinates for finding centroid
        for (j = 0; j < dim; j++)
            newClustersCentroids[currentCluster*dim + j] += X[i*dim + j];
    }

    // now divide each coordinate sum by number of members to find mean/centroid
    for (i = 0; i < k; i++) // for each cluster
    {
        if (clustersCardinals[i] == 0)
        {
            printf("WARNING: Cluster %d is empty! \ n", i+1);
            clustersCardinals[i] = 0;
        }
        else
            for (j = 0; j < dim; j++) // for each dimension
                newClustersCentroids[i*dim + j] /= clustersCardinals[i];
    }

    free(clustersCardinals);
    return;
}


/**
 * @brief Generate a random permutation of numbers between 0 and n-1.
 *
 * @param [in] n Length of the generate permutation.
 * @param [out] perm Permutation of numbers between 0 and n-1.
 * @param [in] rng Random generator.
 */
static void getPermutation(int n, int* perm, PnlRng* rng)
{
    int i, j, temp;
```

```
    assert(perm != NULL);
    assert(rng != NULL);

    // initialize the array {0, 1, ..., n-1}
    for (i = 0; i < n; i++)
        perm[i] = i;

    for (i = n-1; i >= 0; i--)
    {
        // generate a random number [0, n-1]
        j = (int) floor(pnl_rng_uni_ab(0.0, n-0.001, rng));

        // swap the last element with element at random index
        temp = perm[i];
        perm[i] = perm[j];
        perm[j] = temp;
    }

    return;
}


/**
 * @brief Inicialization of a Model_kmeans structure (for kmeans).
 *
 * @param [in] X A matrix (or vector if dim=1).
 * @param [in] n Number of points of X (number of rows).
 * @param [in] dim Points dimension (number of columns of X).
 * @param [in] k Number of clusters.
 * @param [out] mod Model_kmeans structure with initial algorithm information.
 * @param [in] centroids Initial cluster's centroids (optional, can be NULL).
 * @param [in] rng Random generator.
 *
 * @retval OK Inicialization done correctly.
 * @retval ERR An error occurred.
 *
 * @related Model_kmeans
 */
static int initializeModel_kmeans(double* X, int n, int dim, int k, Model_kmeans
                           double* centroids, PnlRng* rng)
{
```

```
int i, j;
int* aleatInd;
  assert(X != NULL);
  assert(mod != NULL);
  assert(rng != NULL);



  // initialisation of the Model_kmeans structure
  mod->centroids = (double*) malloc(k*dim*sizeof(double));
  if (mod->centroids == NULL)
      return ERR;
  if (centroids == NULL) // random inicialization of class centers
  {
      aleatInd = (int*) malloc(n*sizeof(int));
      if (aleatInd == NULL)
          return ERR;
      // take a random permutation of numbers between 0 and n-1
      getPermutation(n, aleatInd, rng);

      /* take the points of X (corresponding to k first numbers in permutation
         as rows) as centroids */
      for (i = 0; i < k; i++)
          for (j = 0; j < dim; j++)
              mod->centroids[i*dim + j] = X[aleatInd[i]*dim + j];

      free(aleatInd);
  }
  else
      memcpy(mod->centroids, centroids, k*dim*sizeof(double));

  mod->labels = (int*) malloc(k*sizeof(int));
  if (mod->labels == NULL)
      return ERR;
  for (i = 0; i < k; i++)
      mod->labels[i] = i+1;

  mod->t = 0;
  mod->MsErr = NULL;

  return OK;
```

```
}


/**
 * @brief Lloyd's algorithm to bundle the grid points using k-means clustering.
 *
 * @param [in] X A matrix (or vector if dim=1) to cluster by rows.
 * @param [in] n Number of points to cluster (number of rows of X).
 * @param [in] dim Points dimension (number of columns of X).
 * @param [in] k Number of clusters.
 * @param [out] finalAssignedClust Vector indicating which cluster correspond to
 *                                 each row of X. Vector dimension -> n.
 * @param [out] mod Model_kmeans structure with run algorithm information.
 * @param [in] centroids Initial cluster's centroids (optional, can be NULL).
 * @param [in] maxIters Maximum number of iterations to run without convergence.
 * @param [in] rng Random generator.
 *
 * @retval OK Clustering done correctly.
 * @retval ERR An error occurred.
 *
 * @see initializeModel_kmeans
 * @see assignNearestCluster
 * @see getNewClustersCentroids
 */
int kmeans(double* X, int n, int dim, int k, int* finalAssignedClust, Model_kmea
           double* centroids, int maxIters, PnlRng* rng)
{
  double argToMin;
  int* prevAssignedClust;
  double prevArgToMin;
    assert(rng != NULL);

    if ( (X == NULL) || (finalAssignedClust == NULL) || (mod == NULL) )
        return ERR;

    // inicialization of model structure
    if (initializeModel_kmeans(X, n, dim, k, mod, centroids, rng) == ERR)
        return ERR;


    // if we have received centroids, we just classify points
```

```
if (centroids != NULL)
{
    assignNearestCluster(X, n, dim, k, mod->centroids, finalAssignedClust,
                         &argToMin);
    mod->t++;
    return OK;
}

prevAssignedClust = (int*) malloc(n*sizeof(int));
if (prevAssignedClust == NULL)
    return ERR;
prevArgToMin = INFINITY;

while (mod->t < maxIters)
{
    // classification - each point in nearest cluster
    assignNearestCluster(X, n, dim, k, mod->centroids, finalAssignedClust,
                         &argToMin);
    mod->t++;

    // see if we've failed to improve - currently solution worse than previo
    if (argToMin > prevArgToMin)
    {
        // restore old assignments
        memcpy(finalAssignedClust, prevAssignedClust, n*sizeof(int));
        // recalc centroids
        getNewClustersCentroids(X, n, dim, k, finalAssignedClust, mod->centr
        /*printf("\ nNegative progress made on iteration %d : (%.4f)\ n",
               mod->t, argToMin - prevArgToMin);*/
        mod->t--;
        break;
    }
    prevArgToMin = argToMin;

    if (mod->t != 1)
    {
        // finish if nothing changes between previous and current classifica
        if (memcmp(finalAssignedClust, prevAssignedClust, n*sizeof(int)) ==
        {
            //printf("\ nNo change made on iteration %d\ n", mod->t);
            break;
```

```
            }
        }

        // save this step as previous step for next iteration
        memcpy(prevAssignedClust, finalAssignedClust, n*sizeof(int));
        // update cluster centroids
        getNewClustersCentroids(X, n, dim, k, finalAssignedClust, mod->centroids
    }

    free(prevAssignedClust);
    return OK;
}


/**
 * @brief Get the number of appearances of integer i in vector vect.
 *
 * @param [in] vect Vector of integer numbers.
 * @param [in] len Vector's length.
 * @param [in] i Integer.
 *
 * @return Number of appearances of integer i in vector vect.
 *
 * @note It is used in recursiveBifurcation and recursiveBifurcationRSS to find
 *       the number of points clustered in cluster i.
 */
static int getNumberAppearances_i(int* vect, int len, int i)
{
   int j;
    int n = 0;

    assert(vect != NULL);

    for (j=0; j<len; j++)
        if (vect[j] == i)
            n++;

    return n;
}
```

```
/**
 * @brief Get the bit at a specified position of the binary notation of a number
 *
 * @param [in] num Unsigned number.
 * @param [in] pos Position.
 *
 * @return 0 or 1 corresponding to the bit at specified position of number in bi
 *
 * @note Only numbers without sign.
 */
static unsigned int bitGet(unsigned int num, unsigned int pos)
{
    unsigned int bit;
    unsigned int uno = 1;
    uno = uno << pos;

    if ((num & uno) == 0)
        bit = 0;
    else
        bit = 1;

    return bit;
}


/**
 * @brief Macro to use A as a matrix.
 */
#define A(i,j) A[i*Ncols + j]
/**
 * @brief Macro to use Ac as a matrix.
 */
#define Ac(i,j) Ac[i*Ncols + j]
/**
 * @brief Macro to use B as a matrix.
 */
#define B(i,j) B[i*Npsbt + j]
/**
 * @brief Macro to use Baux as a 3D vector.
 */
#define Baux(i,j,k) Baux[i*Ncols + j + k*(Nrows*Ncols)]
```

```
/**
 * @brief Recursive Bifurcation method.
 *
 * The number of partitions/bundles, after p iterations, is equal to (2^d)^p.
 *
 * @param [in] S A matrix to cluster by rows.
 * @param [in] p Number of recursive iterations to do.
 * @param [out] S_Clusterized Vector indicating which cluster correspond to each
 *                            row of S. Vector dimension -> Nrows.
 *
 * @retval OK Clustering done correctly.
 * @retval ERR An error occurred.
 */
int recursiveBifurcation(const PnlMat* S, int p, int* S_Clusterized)
{
   int Nrows;
   int Ncols;
int* A;
int* Ac;
int Npsbt;
    int* Baux;
    int* B;
int i, j, k,l;
PnlVect* mu;
int* S_ClusterizedAux;
        int vNrows, vNrows2;
        PnlMat* v ;
        int* S_ClusterizedAux2;
        int* S_ClusterizedAux2Aux;


 if ( (S == NULL) || (S_Clusterized == NULL) )
        return ERR;
   Nrows = S->m;
   Ncols = S->n;

    if (p <= 0) // Number of iterations must be strictly positive
        return ERR;
    else if (p == 1)
```

```
{

    // Step 1 : Compute the mean of the given set of grid points, along each
    //          dimension
    mu = pnl_vect_create(S->n);
    pnl_mat_sum_vect(mu, S, 'r');
    pnl_vect_div_scalar(mu, (double) (S->m));

    // Step 2 : The grid points are bundled separately along each dimension.
    A = (int*) malloc((S->m)*(S->n)*sizeof(int));
    if (A == NULL)
    {
        pnl_vect_free(&mu);
        return ERR;
    }
    Ac = (int*) malloc((S->m)*(S->n)*sizeof(int));
    if (Ac == NULL)
    {
        pnl_vect_free(&mu);
        free(A);
        return ERR;
    }
    for (i = 0; i < S->n; i++)
    {
        for (j = 0; j < S->m; j++)
        {
            if (MGET(S, j, i) > GET(mu, i))
            {
                A(j,i) = 1;
                Ac(j,i) = 0;
            }
            else
            {
                A(j,i) = 0;
                Ac(j,i) = 1;
            }
        }
    }
    pnl_vect_free(&mu);

    // Step 3 : The 2^(S->n) unique non-overlapping bundles are then obtaine
```

```
//          using the following intersections of these sets.
Npsbt = (int) pow(2., (double) S->n);
Baux = (int*) malloc((S->m)*(S->n)*Npsbt*sizeof(int));
if (Baux == NULL)
{
    free(A);
    free(Ac);
    return ERR;
}
B = (int*) malloc((S->m)*Npsbt*sizeof(int));
if (B == NULL)
{
    free(A);
    free(Ac);
    free(Baux);
    return ERR;
}
// Truth table of S->n elements -> 2^(S->n) posibilities
for (i = 0; i < Npsbt; i++)
{
    for (j = 0; j < S->n; j++)
    {
        if (bitGet((unsigned int) i, (unsigned int) j) == 0)
            for (k = 0; k < S->m; k++)
                Baux(k, j, i) = A(k,j);
        else
            for (k = 0; k < S->m; k++)
                Baux(k, j, i) = Ac(k,j);
    }
    // For each posibility, we do an and by columns, we obtain (B) a
    // matrix which indicates in which cluster is each path.
    // Each row (path) only has one entry 1 if there isn't an error
    for (j = 0; j < S->m; j++)
    {
        B(j,i) = 1;
        for (k = 0; k < S->n; k++)
            B(j,i) = B(j,i) && Baux(j,k,i);
    }
}

for (j = 0; j < S->m; j++)
```

```
            for (k = 0; k < Npsbt; k++)
                if (B(j,k) == 1)
                    S_Clusterized[j] = k+1;

        free(B);
        free(Baux);
        free(A);
        free(Ac);
    }
    else
    {
        // Step 4 : Bundles B_{t_m-1}(1), ... can be split further, in the next
        //          iteration, again following the steps above.
        S_ClusterizedAux = (int*) malloc((S->m)*sizeof(int));
        if (S_ClusterizedAux == NULL)
            return ERR;
        if (recursiveBifurcation(S, p-1, S_ClusterizedAux) == ERR)
        {
            free(S_ClusterizedAux);
            return ERR;
        }


        v = pnl_mat_new();
        S_ClusterizedAux2 = NULL;
        S_ClusterizedAux2Aux = NULL;
        for (i = 1; i <= (int) pow(2., ((double) (S->n))*(p-1.)); i++)
        {
            vNrows = getNumberAppearances_i(S_ClusterizedAux, S->m, i);
            pnl_mat_resize(v, vNrows, S->n);
            vNrows2 = 0;
            for (k = 0; k < S->m; k++)
            {
                if (S_ClusterizedAux[k] == i)
                {
                    for (l = 0; l < S->n; l++)
                        MLET(v, vNrows2, l) = MGET(S, k, l);
                    vNrows2++;
                }
            }
```

```
            S_ClusterizedAux2Aux = (int*) realloc(S_ClusterizedAux2,
                                                  vNrows*sizeof(int));
            if (S_ClusterizedAux2Aux == NULL)
                abort();
            S_ClusterizedAux2 = S_ClusterizedAux2Aux;
            if (recursiveBifurcation(v, 1, S_ClusterizedAux2) == ERR)
            {
                pnl_mat_free(&v);
                free(S_ClusterizedAux2);
                free(S_ClusterizedAux);
                printf("There is an empty cluster\ n\ n");
                return ERR;
            }

            for (k = 0; k < vNrows; k++)
            {
                S_ClusterizedAux2[k] = S_ClusterizedAux2[k] +
                                       (i-1)*((int) pow(2., (double) S->n));
            }

            vNrows2 = 0;
            for (k = 0; k < S->m; k++)
            {
                if (S_ClusterizedAux[k] == i)
                {
                    S_Clusterized[k] = S_ClusterizedAux2[vNrows2];
                    vNrows2++;
                }
            }
        }

        pnl_mat_free(&v);
        free(S_ClusterizedAux2);
        S_ClusterizedAux2 = NULL;
        free(S_ClusterizedAux);
        S_ClusterizedAux = NULL;
    }

    return OK;
}
```

```
/**
 * @brief Recursive Bifurcation of Reduced State Space method.
 *
 * The number of partitions/bundles, after p iterations, is equal to 2^p.
 *
 * @param [in] S A matrix to cluster by rows.
 * @param [in] p Number of recursive iterations to do.
 * @param [out] S_Clusterized Vector indicating which cluster correspond to each
 *                            row of S. Vector dimension -> Nrows.
 * @param [in] mappingF_ptr Function pointer to the function to reduce the state
 *
 * @retval OK Clustering done correctly.
 * @retval ERR An error occurred.
 */
int recursiveBifurcationRSS(const PnlMat* S, int p, int* S_Clusterized,
                            void (*mappingF_ptr) (const PnlMat*, PnlVect*))
{
  PnlVect* h;
  double mu;
  int i;
  int* S_ClusterizedAux;
   int k, l;
  int vNrows, vNrows2;
        PnlMat* v;
        int* S_ClusterizedAux2;
        int* S_ClusterizedAux2Aux;

    if ( (S == NULL) || (S_Clusterized == NULL) )
        return ERR;
    if (mappingF_ptr == NULL) // No mapping function has been selected
        return ERR;

    if (p <= 0) // Number of iterations must be strictly positive
        return ERR;
    else if (p == 1)
    {
        // mappingF_ptr : R^d --> R function to apply to S to reduce the state
        //                space, mapping
        h = pnl_vect_create(S->m);
        mappingF_ptr(S, h);
```

```
    // Step 1 : Compute the mean of the given set of points
    mu = pnl_vect_sum(h) / ((double) S->m);


    // Step 2 : The vector points are bundled
    for (i = 0; i < S->m; i++)
    {
        if (GET(h, i) > mu)
            S_Clusterized[i] = 1;
        else
            S_Clusterized[i] = 2;
    }

    pnl_vect_free(&h);
}
else
{
    // Step 4 : Bundles B_{t_m-1}(1), ... can be split further, in the next
    //          iteration, again following the steps above.
    S_ClusterizedAux = (int*) malloc((S->m)*sizeof(int));
    if (S_ClusterizedAux == NULL)
        return ERR;
    if (recursiveBifurcationRSS(S, p-1, S_ClusterizedAux, mappingF_ptr) == E
    {
        free(S_ClusterizedAux);
        return ERR;
    }


    v = pnl_mat_new();
   S_ClusterizedAux2 = NULL;
   S_ClusterizedAux2Aux = NULL;
    for (i = 1; i <= (int) pow(2., p-1.); i++)
    {
        vNrows = getNumberAppearances_i(S_ClusterizedAux, S->m, i);
        pnl_mat_resize(v, vNrows, S->n);
        vNrows2 = 0;
        for (k = 0; k < S->m; k++)
        {
            if (S_ClusterizedAux[k] == i)
```

```
            {
                for (l = 0; l < S->n; l++)
                    MLET(v, vNrows2, l) = MGET(S, k, l);
                vNrows2++;
            }
        }

        S_ClusterizedAux2Aux = (int*) realloc(S_ClusterizedAux2,
                                              vNrows*sizeof(int));
        if (S_ClusterizedAux2Aux == NULL)
            abort();
        S_ClusterizedAux2 = S_ClusterizedAux2Aux;
        if (recursiveBifurcationRSS(v, 1, S_ClusterizedAux2, mappingF_ptr) =
        {
            pnl_mat_free(&v);
            free(S_ClusterizedAux2);
            free(S_ClusterizedAux);
            printf("There is an empty cluster\ n\ n");
            return ERR;
        }

        for (k = 0; k < vNrows; k++)
            S_ClusterizedAux2[k] = S_ClusterizedAux2[k] + (i-1)*2;

        vNrows2 = 0;
        for (k = 0; k < S->m; k++)
        {
            if (S_ClusterizedAux[k] == i)
            {
                S_Clusterized[k] = S_ClusterizedAux2[vNrows2];
                vNrows2++;
            }
        }
    }

    pnl_mat_free(&v);
    free(S_ClusterizedAux2);
    S_ClusterizedAux2 = NULL;
    free(S_ClusterizedAux);
    S_ClusterizedAux = NULL;
}
```

```
    return OK;
}

/** @}*/
```