# PBDataAnalysis

P. Baillehache

July 25, 2019

## Contents



## Introduction

PBDataAnalysis is a C library providing structures and functions to perform various data analysis.

It implements the following algorithms:

- K-means clustering (random, Forgy and plusplus seeds)
- Principal Component Analysis

It uses the `PBErr`, `PBMath`, `PBJSon`, `GSet` libraries.

# 1 Definitions

## 1.1 K-means clustering

The goal of the K-means clustering algorithm is to find K Voronoi cells which clusters a data set in a way that for each cells the center of this cell is the nearest possible to the average value of the input data inside this cell.

The K-means algorithm is as follow, where 'seed' defines the way we initialise the algorithm: 'random', 'Forgy' or 'plusplus'.

As an example of use, code is provided to compute the target dimensions in the config file of YoloV3 given the target bounding boxes of the training data set, using K-means clustering.

```
if seed = random
  init the center of each cluster with a random value inside the bounds
  of the input data
else if seed = Forgy
  init the center of each cluster with one of the input data randomly
  choosen, one given input data can't be choosen twice
else if seed = pluspus
  choose one center uniformly at random from among the data points
  for each data point x, compute D(x), the distance between x and the
  nearest center that has already been chosen
  choose one new data point at random as a new center, using a weighted
  probability distribution where a point x is chosen with probability
  proportional to D(x)
  repeat the previous 2 steps until k centers have been chosen
loop until clusters' center have all converged
  init K empty sets S[]
  for each input data I
    ID = get the cluster containing I
    add I to S[ID]
  for each set S[I]
    AVG = calculate the average value of the input in S[I]
    if AVG is equal to the center of the I-th cluster
      the center of the I-th cluster has converged
    else
      set the center of the I-th cluster to AVG
```

## 1.2 Principal Component Analysis

PCA consists of projecting the data on a subset of the Eigen vectors of the covariance matrix for these data.

```
Projection on the 'n' first principal components of the 'nbSample'
samples (vectors of dimension 'dim') of the data set 'dataset'
A[i][j] <=> value of matrix A at the i-th column and j-th row
```

```
set := normalized and centered version of dataset
cov := covariance matrix of 'set' (matrix of dimensions 'dim' columns
  and 'dim' rows)
components := eigenVector of 'cov' ('dim' vectors of dimension 'dim')
A := create a matrix of dimensions 'dim' columns and 'n' rows
for i := 0 to 'dim'
  for j := 0 to 'n'
    A[i][j] := (j-th components)[i]
B := create a matrix of dimensions 'nbSample' columns and 'dim' rows
for i := 0 to 'nbSample'
  for j := 0 to 'dim'
    B[i][j] := (i-th sample)[j]
C := A * B (matrix of dimensions 'nbSample' columns and 'n' rows)

The projection of the i-th sample is the i-th column of C (vector of
dimension 'n')
```

# 2 Interface

```
// ============ PBDATAANALYSIS.H ================

#ifndef PBDATAANALYSIS_H
#define PBDATAANALYSIS_H

// ================= Include =================

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <execinfo.h>
#include <errno.h>
#include <string.h>
#include "pberr.h"
#include "pbmath.h"
#include "pbjson.h"
#include "gset.h"
#include "gdataset.h"

// ----------------- Principal component analysis ---------------

// ================= Define ==================

// ================= Data structure ===================

typedef struct PrincipalComponentAnalysis {
  // Principal components, stored from the most influent to the less one
  GSetVecFloat _components;
} PrincipalComponentAnalysis;

// ================ Functions declaration ====================

// Create a static PrincipalComponentAnalysis
PrincipalComponentAnalysis PrincipalComponentAnalysisCreateStatic();

// Free the memory used by the static PrincipalComponentAnalysis
void PrincipalComponentAnalysisFreeStatic(
  PrincipalComponentAnalysis* const that);
```

```
// Calculate the principal components for the 'dataset' and
// store the result into the PrincipalComponentAnalysis 'that'
void PCASearch(PrincipalComponentAnalysis* const that,
  const GDataSetVecFloat* const dataset);

// Get the 'dataset' converted through the first 'nb' components of the
// PrincipalComponentAnalysis 'that'
// Return a new data set, the dataset in arguments is not modified
// Return an empty dataset if the principal components have not yet been
// calculated (using the PCASearch function)
// Returned VecFloat have dimension 'nb'. Returned GSet has same nbsample
// as 'dataset'
// Dimension of VecFloat in 'dataset' must be equal to the size of
// components in 'that'
GDataSetVecFloat PCAConvert(const PrincipalComponentAnalysis* const that,
  const GDataSetVecFloat* const dataset, const int nb);

// Get the principal components of the PrincipalComponentAnalysis 'that'
#if BUILDMODE != 0
inline
#endif
const GSetVecFloat* PCAComponents(
  const PrincipalComponentAnalysis* const that);

// Get the number of principal components of the
// PrincipalComponentAnalysis 'that'
#if BUILDMODE != 0
inline
#endif
int PCAGetNbComponents(
  const PrincipalComponentAnalysis* const that);

// Print the principal components of the PrincipalComponentAnalysis 'that'
// on 'stream'
void PCAPrintln(const PrincipalComponentAnalysis* const that,
  FILE* const stream);

// ----------------- K-means clustering ---------------

// ================= Define ==================

// ================= Data structure ===================

typedef enum KMeansClustersSeed {
  KMeansClustersSeed_Random, KMeansClustersSeed_Forgy,
  KMeansClustersSeed_PlusPlus
} KMeansClustersSeed;
#define KMeansClustersSeed_Default KMeansClustersSeed_PlusPlus

typedef struct KMeansClusters {
  GSetVecFloat _centers;
  KMeansClustersSeed _seed;
} KMeansClusters;

// ================ Functions declaration ====================

// Create a KMeansClusters for a K-means clustering initialized
// using the 'seed' technique
KMeansClusters KMeansClustersCreateStatic(const KMeansClustersSeed seed);

// Free the memory used by a KMeansClusters
void KMeansClustersFreeStatic(KMeansClusters* const that);
```

```
// Create the set of 'K' clusters clustering the 'input' data according
// to the K-means algorithm
// 'K' must be inferior or equal to the number of input data
// srandom() must have been called before using this function
void KMeansClustersSearch(KMeansClusters* const that,
  const GSetVecFloat* const input, const int K);

// Get the set of clusters' center for the KMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
const GSetVecFloat* KMeansClustersCenters(
  const KMeansClusters* const that);

// Get the 'iCluster'-th cluster's center for the KMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* _KMeansClustersCenterFromId(
  const KMeansClusters* const that, const int iCluster);

// Get the seed of the KMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
KMeansClustersSeed KMeansClustersGetSeed(const KMeansClusters* const that);

// Set the seed of the KMeansClusters 'that' to 'seed'
#if BUILDMODE != 0
inline
#endif
void KMeansClustersSetSeed(KMeansClusters* const that,
  const KMeansClustersSeed seed);

// Get the center of the cluster including the 'input' data for the
// KMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* _KMeansClustersCenterFromPos(
  const KMeansClusters* const that, const VecFloat* input);

// Get the index of the cluster including the 'input' data for the
// KMeansClusters 'that'
int KMeansClustersGetId(const KMeansClusters* const that,
  const VecFloat* input);

// Get the seed of the KMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
int KMeansClustersGetK(const KMeansClusters* const that);

// Print the KMeansClusters 'that' on the stream 'stream'
void KMeansClustersPrintln(const KMeansClusters* const that,
  FILE* const stream);

// Load the KMeansClusters 'that' from the stream 'stream'
bool KMeansClustersLoad(KMeansClusters* that, FILE* const stream);

// Save the KMeansClusters 'that' to the stream 'stream'
```

```
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true upon success else false
bool KMeansClustersSave(const KMeansClusters* const that,
  FILE* const stream, const bool compact);

// Function which return the JSON encoding of 'that'
JSONNode* KMeansClustersEncodeAsJSON(const KMeansClusters* const that);

// Function which decode from JSON encoding 'json' to 'that'
bool KMeansClustersDecodeAsJSON(KMeansClusters* that,
  const JSONNode* const json);

// ================ Polymorphism ==================

#define KMeansClustersCenter(Cluster, Input) _Generic(Input, \
  VecFloat*: _KMeansClustersCenterFromPos, \
  const VecFloat*: _KMeansClustersCenterFromPos, \
  int: _KMeansClustersCenterFromId, \
  const int: _KMeansClustersCenterFromId, \
  default: PBErrInvalidPolymorphism)(Cluster, Input)

// ================ Inliner ====================

#if BUILDMODE != 0
#include "pbdataanalysis-inline.c"
#endif

#endif
```

# 3 Code

## 3.1 pbdataanalysis.c

```
// ============ PBDATAANALYSIS.C ================

// ================ Include =================

#include "pbdataanalysis.h"
#if BUILDMODE == 0
#include "pbdataanalysis-inline.c"
#endif

// ----------------- Principal component analysis ---------------

// ================ Functions declaration ====================

// ================ Functions implementation ====================

// Create a static PrincipalComponentAnalysis
PrincipalComponentAnalysis PrincipalComponentAnalysisCreateStatic() {
  // Declare the PrincipalComponentAnalysis
  PrincipalComponentAnalysis that;
  // Init the properties
  that._components = GSetVecFloatCreateStatic();
  // Return the PrincipalComponentAnalysis
  return that;
}
```

```
// Free the memory used by the static PrincipalComponentAnalysis
void PrincipalComponentAnalysisFreeStatic(
  PrincipalComponentAnalysis* const that) {
  if (that == NULL)
    return;
  // Free memory
  while(GSetNbElem(PCAComponents(that)) > 0) {
    VecFloat* v = GSetPop(&(that->_components));
    VecFree(&v);
  }
}

// Calculate the principal components for the 'dataset' and
// store the result into the PrincipalComponentAnalysis 'that'
// http://setosa.io/ev/principal-component-analysis/
// https://www.dezyre.com/data-science-in-python-tutorial/principal-component-analysis-tutorial
void PCASearch(PrincipalComponentAnalysis* const that,
  const GDataSetVecFloat* const dataset) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (dataset == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'dataset' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Create a centered and normalized version of the dataset
  GDataSetVecFloat set = GDSClone(dataset);
  GDSNormalize(&set);
  GDSMeanCenter(&set);

  // Calculate the covariance matrix
  MatFloat* covariance = GDSGetCovarianceMatrix(&set);

  // Calculate the Eigen values and vectors of the covariance matrix
  that->_components = MatGetEigenValues(covariance);

  // Pop out the unused Eigen values
  VecFloat* v = GSetPop(&(that->_components));
  VecFree(&v);

  // Free memory
  MatFree(&covariance);
  GDataSetVecFloatFreeStatic(&set);
}

// Get the 'dataset' converted through the first 'nb' components of the
// PrincipalComponentAnalysis 'that'
// Return a new data set, the dataset in arguments is not modified
// Return an empty dataset if the principal components have not yet been
// calculated (using the PCASearch function)
// Returned VecFloat have dimension 'nb'. Returned GSet has same nbsample
// as 'dataset'
// Dimension of VecFloat in 'dataset' must be equal to the size of
// components in 'that'
GDataSetVecFloat PCAConvert(const PrincipalComponentAnalysis* const that,
  const GDataSetVecFloat* const dataset, const int nb) {
```

```
#if BUILDMODE == 0
  if (that == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (dataset == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'dataset' is null");
    PBErrCatch(GSetErr);
  }
  if (nb < 1) {
    PBDataAnalysisErr->_type = PBErrTypeInvalidArg;
    sprintf(PBDataAnalysisErr->_msg, "'nb' is invalid (%d>0)", nb);
    PBErrCatch(GSetErr);
  }
  if (VecGetDim(GSetGet(PCAComponents(that), 0)) !=
    VecGet(GDSSampleDim(dataset), 0)) {
    PBDataAnalysisErr->_type = PBErrTypeInvalidArg;
    sprintf(PBDataAnalysisErr->_msg,
      "Size of samples in dataset doesn't match size of component "
      "(%ld=%d)", VecGetDim(GSetGet(PCAComponents(that), 0)),
      VecGet(GDSSampleDim(dataset), 0));
    PBErrCatch(GSetErr);
  }
#endif

  // Create the matrix of 'nb' first transposed components
  VecShort2D dimFeatures = VecShortCreateStatic2D();
  VecSet(&dimFeatures, 0, VecGetDim(GSetGet(PCAComponents(that), 0)));
  VecSet(&dimFeatures, 1, nb);
  MatFloat* features = MatFloatCreate(&dimFeatures);
  VecShort2D pos = VecShortCreateStatic2D();
  do {
    MatSet(features, &pos, VecGet(GSetGet(PCAComponents(that),
      VecGet(&pos, 1)), VecGet(&pos, 0)));
  } while(VecStep(&pos, &dimFeatures));

  // Create the matrix of transposed vectors from the data set
  GDataSetVecFloat normDataset = GDSClone(dataset);
  GDSNormalize(&normDataset);
  GDSMeanCenter(&normDataset);
  VecShort2D dimData = VecShortCreateStatic2D();
  VecSet(&dimData, 0, GDSGetSize(dataset));
  VecSet(&dimData, 1, VecGet(&dimFeatures, 0));
  MatFloat* data = MatFloatCreate(&dimData);
  VecSetNull(&pos);
  do {
    MatSet(data, &pos,
      VecGet(GSetGet(GDSSamples(&normDataset), VecGet(&pos, 0)),
      VecGet(&pos, 1)));
  } while(VecStep(&pos, &dimData));

  // Multiply the matrices
  MatFloat* proj = MatGetProdMat(features, data);

  // Create the result data set from the resulting matrix
  GDataSetVecFloat res;
  GDataSet* ptrRes = (GDataSet*)&res;
  *ptrRes = GDataSetCreateStatic(GDataSetType_VecFloat);
  char* name = PBErrMalloc(PBDataAnalysisErr,
    strlen(GDSName(dataset)) + 100);
```

```
  sprintf(name, "%s - PCA%dD", GDSName(dataset), nb);
  ptrRes->_name = strdup(name);
  char *desc = PBErrMalloc(PBDataAnalysisErr,
    strlen(GDSDesc(dataset)) + 100);
  sprintf(desc, "%s\nProjection on the %d first principal components.",
    GDSDesc(dataset), nb);
  ptrRes->_desc = strdup(desc);
  ptrRes->_nbSample = GDSGetSize(dataset);
  for (int iSample = ptrRes->_nbSample; iSample--;)
    GSetAppend(&(ptrRes->_samples), VecFloatCreate(nb));
  VecSetNull(&pos);
  do {
    VecSet((VecFloat*)GSetGet(&(ptrRes->_samples), VecGet(&pos, 0)),
      VecGet(&pos, 1), MatGet(proj, &pos));
  } while(VecStep(&pos, MatDim(proj)));
  GDSResetCategories(ptrRes);

  // Free memory
  free(desc);
  free(name);
  MatFree(&features);
  MatFree(&data);
  MatFree(&proj);
  GDataSetVecFloatFreeStatic(&normDataset);

  // Return the result
  return res;
}

// Print the principal components of the PrincipalComponentAnalysis 'that'
// on 'stream'
void PCAPrintln(const PrincipalComponentAnalysis* const that,
  FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (stream == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'stream' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // If the components have been computed
  if (GSetNbElem(PCAComponents(that)) > 0) {
    // Create an iterator on the set of components
    // Iterate in reverse order to display the most important
    // components first
    GSetIterBackward iter =
      GSetIterBackwardCreateStatic(PCAComponents(that));
    do {
      // Get the component
      VecFloat* v = GSetIterGet(&iter);
      // Display the component
      VecPrint(v, stream); printf("\n");
    } while (GSetIterStep(&iter));
  }
}

// ----------------- K-means clustering ---------------
```

```
// ================= Define ==================

// ================ Functions declaration ====================

void KMeansClustersInitRandom(KMeansClusters* const that,
  const GSetVecFloat* const input);

void KMeansClustersInitForgy(KMeansClusters* const that,
  const GSetVecFloat* const input);

void KMeansClustersInitPlusPlus(KMeansClusters* const that,
  const GSetVecFloat* const input);

// ================ Functions implementation ====================

// Create a KMeansClusters for a K-means clustering initialized
// using the 'seed' technique
// srandom() must have been called before using this function
KMeansClusters KMeansClustersCreateStatic(const KMeansClustersSeed seed) {
  // Declare the KMeansClusters
  KMeansClusters clusters;
  // Init the properties
  clusters._seed = seed;
  clusters._centers = GSetVecFloatCreateStatic();
  // Return the KMeansClusters
  return clusters;
}

// Free the memory used by a PBDAKMeansClusters
void KMeansClustersFreeStatic(KMeansClusters* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
#endif
  // Free the memory used by the cluster centers
  while (GSetNbElem(&(that->_centers)) > 0) {
    VecFloat* v = GSetPop((GSetVecFloat*)(KMeansClustersCenters(that)));
    free(v);
  }
}

// Create the set of 'K' clusters clustering the 'input' data according
// to the K-means algorithm
// 'K' must be inferior or equal to the number of input data
// srandom() must have been called before using this function
void KMeansClustersSearch(KMeansClusters* const that,
  const GSetVecFloat* const input, const int K) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
  if (input == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'input' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
```

```
  if (K < 1) {
    PBDataAnalysisErr->_type = PBErrTypeInvalidArg;
    sprintf(PBDataAnalysisErr->_msg, "'K' is invalid (%d>=1)", K);
    PBErrCatch(PBDataAnalysisErr);
  }
  if (GSetNbElem(input) < K) {
    PBDataAnalysisErr->_type = PBErrTypeInvalidArg;
    sprintf(PBDataAnalysisErr->_msg, "'K' is invalid (%d<=%ld)", K,
      GSetNbElem(input));
    PBErrCatch(PBDataAnalysisErr);
  }
#endif
  // If there are alreay computed clusters,
  // free the memory used by the cluster centers
  while (GSetNbElem(&(that->_centers)) > 0) {
    VecFloat* v = GSetPop((GSetVecFloat*)(KMeansClustersCenters(that)));
    free(v);
  }
  // Allocate an array of sets used for computation
  GSetVecFloat* inputsByCluster =
    PBErrMalloc(PBDataAnalysisErr, sizeof(GSetVecFloat) * K);
  // Get the dimension of the input
  long dim = VecGetDim(GSetGet(input, 0));
  // Allocate memory for the clusters' center and sets used for
  // computation
  for (int iCenter = K; iCenter--;) {
    // The dimension of the means is the same as the one of the input
    // data
    VecFloat* v = VecFloatCreate(dim);
    GSetAppend((GSetVecFloat*)(KMeansClustersCenters(that)), v);
    inputsByCluster[iCenter] = GSetVecFloatCreateStatic();
  }
  // Initialise the means according to the seed
  switch(KMeansClustersGetSeed(that)) {
    case KMeansClustersSeed_Random:
      KMeansClustersInitRandom(that, input);
      break;
    case KMeansClustersSeed_Forgy:
      KMeansClustersInitForgy(that, input);
      break;
    case KMeansClustersSeed_PlusPlus:
      KMeansClustersInitPlusPlus(that, input);
      break;
    default:
      break;
  }
  // Create a vector used for computation
  VecFloat* w = VecFloatCreate(dim);
  // Loop until the clusters' center are aligned with their
  // real center according to input data
  float shift = 1.0;
  while (shift > PBMATH_EPSILON) {
    // Reset the shift
    shift = 0.0;
    // Loop on input data
    GSetIterForward iterInput = GSetIterForwardCreateStatic(input);
    do {
      // Get the input data
      VecFloat* v = GSetIterGet(&iterInput);
      // Get the id of the cluster containing this data
      int clusterId = KMeansClustersGetId(that, v);
      // Add the input to the corresponding set
```

```
      GSetAppend(inputsByCluster + clusterId, v);
    } while(GSetIterStep(&iterInput));
    // For each cluster
    for (int iCenter = K; iCenter--;) {
      // Reset the vector used for computation
      VecSetNull(w);
      // Memorize the number of input associated to this cluster
      int nbInput = GSetNbElem(inputsByCluster + iCenter);
      // If there are data in this cluster
      if (nbInput > 0) {
        // Loop on the input contained by this cluster
        GSetIterForward iterSet =
          GSetIterForwardCreateStatic(inputsByCluster + iCenter);
        do {
          // Get the input data
          VecFloat* v = GSetIterGet(&iterSet);
          // Sum it to the temporary vector
          VecOp(w, 1.0, v, 1.0);
        } while(GSetIterStep(&iterSet));
        // Get the center (average) of the input contained by this
        // cluster
        VecScale(w, 1.0 / (float)nbInput);
        // Update the shift
        shift += VecDist(KMeansClustersCenter(that, iCenter), w);
        // Update the cluster center with the center of the input
//VecPrint(KMeansClustersCenter(that, iCenter), stdout);printf(" ");
        VecCopy((VecFloat*)KMeansClustersCenter(that, iCenter), w);
        // Reset the sets of input data
        GSetFlush(inputsByCluster + iCenter);
      }
    }
//printf(" %f\n", shift);fflush(stdout);
  }
  // Free the memory used by the vector and sets used for computation
  free(inputsByCluster);
  VecFree(&w);
}

void KMeansClustersInitRandom(KMeansClusters* const that,
  const GSetVecFloat* const input) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
  if (input == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'input' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
#endif
  // Get the bounds of the input data
  GSetVecFloat bounds = GSetGetBounds(input);
  VecFloat* boundMin = GSetGet(&bounds, 0);
  VecFloat* boundMax = GSetGet(&bounds, 1);
  // Get the dimension of the data
  int dim = VecGetDim(GSetGet(input, 0));
  // For each cluster's center
  for (int iCenter = KMeansClustersGetK(that); iCenter--;) {
    // Get the iCenter-th cluster center
    VecFloat* center = (VecFloat*)KMeansClustersCenter(that, iCenter);
```

```
    // Initialize randomly the components of the iCenter-th cluster's
    // center
    for (int iDim = dim; iDim--;) {
      VecSet(center, iDim, VecGet(boundMin, iDim) +
        rnd() * (VecGet(boundMax, iDim) - VecGet(boundMin, iDim)));
    }
  }
  // Free memory
  GSetFlush(&bounds);
  VecFree(&boundMin);
  VecFree(&boundMax);
}

void KMeansClustersInitForgy(KMeansClusters* const that,
  const GSetVecFloat* const input) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
  if (input == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'input' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
#endif
  // Create a set to select the seeds
  GSetVecFloat forgyInp = GSetVecFloatCreateStatic();
  // Get the number of inputs
  long nbInput = GSetNbElem(input);
  // Declare a variable to pick one input randomly
  VecFloat* inp = NULL;
  // For each cluster's center
  for (int iCenter = KMeansClustersGetK(that); iCenter--;) {
    // Pick an input while avoiding picking twice the same
    // Supposes K is far less than the number of inputs
    do {
      inp = GSetGet(input, (long)round(rnd() * (nbInput - 1)));
    } while (GSetFirstElem(&forgyInp, inp) != NULL);
    // Set the center of the iCenter-th cluster to this input
    VecCopy((VecFloat*)KMeansClustersCenter(that, iCenter), inp);
  }
  // Empty the set used to select the seeds
  GSetFlush(&forgyInp);
}

void KMeansClustersInitPlusPlus(KMeansClusters* const that,
  const GSetVecFloat* const input) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
  if (input == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'input' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
  if (GSetNbElem(input) < KMeansClustersGetK(that)) {
    PBDataAnalysisErr->_type = PBErrTypeInvalidArg;
```

```
    sprintf(PBDataAnalysisErr->_msg, "not enough inputs (%ld>=%d)",
      GSetNbElem(input), KMeansClustersGetK(that));
    PBErrCatch(PBDataAnalysisErr);
  }
#endif
  // Create a copy of the set of inputs to select the seeds
  GSetVecFloat remainInp = GSetVecFloatCreateStatic();
  // For each remaining inputs
  GSetIterForward iter = GSetIterForwardCreateStatic(input);
  do {
    GSetAppend(&remainInp, (VecFloat*)GSetIterGet(&iter));
  } while (GSetIterStep(&iter));
  // Declare a variable to pick one input randomly
  VecFloat* inp = NULL;
  // Pick one input randomly for the first cluster's center
  GSetShuffle((GSet*)&remainInp);
  inp = GSetPop(&remainInp);
  // Set the center of the 1st cluster to this input
  VecCopy((VecFloat*)KMeansClustersCenter(that, 0), inp);
  // For each following cluster
  for (int iCenter = 1; iCenter < KMeansClustersGetK(that); ++iCenter) {
    // Declare a variable to memorize the sum of square of distances
    float sumDist = 0.0;
    // For each remaining inputs
    iter = GSetIterForwardCreateStatic(&remainInp);
    do {
      // Calculate the minimum distance from this input to the center
      // of already choosen cluster's center
      inp = GSetIterGet(&iter);
      float dist = VecDist(inp, KMeansClustersCenter(that, 0));
      for (int iChoosenCluster = 1; iChoosenCluster < iCenter;
        ++iChoosenCluster) {
        float d = VecDist(inp,
          KMeansClustersCenter(that, iChoosenCluster));
        if (d < dist)
          dist = d;
      }
      GSetElemSetSortVal((GSetElem*)GSetIterGetElem(&iter),
        fsquare(dist));
      sumDist += fsquare(dist);
    } while (GSetIterStep(&iter));
    // Sort the remaining inputs by their distance
    GSetSort(&remainInp);
    // Select randomly one input according to its squared distance
    float r = rnd() * sumDist;
    GSetIterBackward iterBack = GSetIterBackwardCreateStatic(&remainInp);
    float sum = GSetElemGetSortVal(GSetIterGetElem(&iterBack));
    while (r > sum && !GSetIterIsLast(&iter)) {
      GSetIterStep(&iterBack);
      sum += GSetElemGetSortVal(GSetIterGetElem(&iterBack));
    }
    inp = GSetIterGet(&iterBack);
    GSetIterRemoveElem(&iterBack);
    // Set the center of the 1st cluster to this input
    VecCopy((VecFloat*)KMeansClustersCenter(that, iCenter), inp);
  }
  // Empty the set used to select the seeds
  GSetFlush(&remainInp);
}

// Get the index of the cluster including the 'input' data for the
// KMeansClusters 'that'
```

```
int KMeansClustersGetId(const KMeansClusters* const that,
  const VecFloat* input) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
  if (input == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'input' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
#endif
  // Declare variables to memorize the id and distance to the input
  float dist = 0.0;
  int id = -1;
  int iCenter = 0;
  // Loop on the clusters' center
  GSetIterForward iter =
    GSetIterForwardCreateStatic(KMeansClustersCenters(that));
  do {
    // Get the center of the cluster
    VecFloat* center = GSetIterGet(&iter);
    // Calculate the distance to the input
    float d = VecDist(center, input);
    // If it's the first considered cluster or
    // if the distance is nearer
    if (id == -1 || dist > d) {
      id = iCenter;
      dist = d;
      // TODO: we can stop if the distance is less than half the
      // shortest distance between two clusters' center to make
      // this function faster
    }
    // Increment the center index
    ++iCenter;
  } while (GSetIterStep(&iter));
  // Return the id
  return id;
}

// Print the KMeansClusters 'that' on the stream 'stream'
void KMeansClustersPrintln(const KMeansClusters* const that,
  FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
  if (stream == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'stream' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
#endif
  // Loop on clusters' center
  GSetIterForward iter =
    GSetIterForwardCreateStatic(KMeansClustersCenters(that));
  do {
    // Get the cluster's center
```

```
    const VecFloat* v = GSetIterGet(&iter);
    // Print the cluster's center
    VecPrint(v, stream);
    printf("\n");
  } while (GSetIterStep(&iter));
}

// Load the KMeansClusters 'that' from the stream 'stream'
bool KMeansClustersLoad(KMeansClusters* that, FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
  if (stream == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'stream' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
#endif
  // Declare a json to load the encoded data
  JSONNode* json = JSONCreate();
  // Load the whole encoded data
  if (!JSONLoad(json, stream)) {
    return false;
  }
  // Decode the data from the JSON
  if (!KMeansClustersDecodeAsJSON(that, json)) {
    return false;
  }
  // Free the memory used by the JSON
  JSONFree(&json);
  // Return success code
  return true;
}

// Save the KMeansClusters 'that' to the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true upon success else false
bool KMeansClustersSave(const KMeansClusters* const that,
  FILE* const stream, const bool compact) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
  if (stream == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'stream' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
#endif
  // Get the JSON encoding
  JSONNode* json = KMeansClustersEncodeAsJSON(that);
  // Save the JSON
  if (!JSONSave(json, stream, compact)) {
    return false;
  }
  // Free memory
```

```
  JSONFree(&json);
  // Return success code
  return true;
}

// Function which return the JSON encoding of 'that'
JSONNode* KMeansClustersEncodeAsJSON(const KMeansClusters* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
#endif
  // Create the JSON structure
  JSONNode* json = JSONCreate();
  // Declare a buffer to convert value into string
  char val[100];
  // Encode the seed
  sprintf(val, "%u", that->_seed);
  JSONAddProp(json, "_seed", val);
  // Encode the centers
  JSONArrayStruct setCenters = JSONArrayStructCreateStatic();
  // If there are clusters
  if (KMeansClustersGetK(that) > 0) {
    // For each cluster
    GSetIterForward iter =
      GSetIterForwardCreateStatic(KMeansClustersCenters(that));
    do {
      VecFloat* center = GSetIterGet(&iter);
      JSONArrayStructAdd(&setCenters, VecEncodeAsJSON(center));
    } while (GSetIterStep(&iter));
    JSONAddProp(json, "_centers", &setCenters);
  }
  // Free memory
  JSONArrayStructFlush(&setCenters);
  // Return the created JSON
  return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool KMeansClustersDecodeAsJSON(KMeansClusters* that,
  const JSONNode* const json) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
  if (json == NULL) {
    PBDataAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBDataAnalysisErr->_msg, "'json' is null");
    PBErrCatch(PBDataAnalysisErr);
  }
#endif
  // Free the memory eventually used by the clusters
  KMeansClustersFreeStatic(that);
  // Get the seed from the JSON
  JSONNode* prop = JSONProperty(json, "_seed");
  if (prop == NULL) {
    return false;
  }
```

```
  int seed = atoi(JSONLblVal(prop));
  if (seed < 0 || seed > KMeansClustersSeed_PlusPlus) {
    return false;
  }
  that->_seed = (KMeansClustersSeed)seed;
  // Decode the centers
  prop = JSONProperty(json, "_centers");
  if (prop == NULL) {
    return false;
  }
  // For each cluster
  for (int iCluster = 0; iCluster < JSONGetNbValue(prop); ++iCluster) {
    // Decode the center of the cluster
    JSONNode* center = JSONValue(prop, iCluster);
    VecFloat* v = NULL;
    if (!VecDecodeAsJSON(&v, center))
      return false;
    GSetAppend((GSet*)KMeansClustersCenters(that), v);
  }
  // Return the success code
  return true;
}
```

# 4 Makefile

```
# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILD_MODE?=1

all: pbmake_wget main

# Automatic installation of the repository PBMake in the parent folder
pbmake_wget:
if [ ! -d ../PBMake ]; then wget https://github.com/BayashiPascal/PBMake/archive/master.zip; unzip master.zip; rm -f

# Makefile definitions
MAKEFILE_INC=../PBMake/Makefile.inc
include $(MAKEFILE_INC)

# Rules to make the executable
repo=pbdataanalysis
$($(repo)_EXENAME): \
$($(repo)_EXENAME).o \
$($(repo)_EXE_DEP) \
$($(repo)_DEP)
$(COMPILER) `echo "$($(repo)_EXE_DEP) $($(repo)_EXENAME).o" | tr ' ' '\n' | sort -u` $(LINK_ARG) $($(repo)_LINK_ARG)

$($(repo)_EXENAME).o: \
$($(repo)_DIR)/$($(repo)_EXENAME).c \
$($(repo)_INC_H_EXE) \
$($(repo)_EXE_DEP)
$(COMPILER) $(BUILD_ARG) $($(repo)_BUILD_ARG) `echo "$($(repo)_INC_DIR)" | tr ' ' '\n' | sort -u` -c $($(repo)_DIR)/$
```

# 5 Unit tests

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <math.h>
#include "pbdataanalysis.h"

void UnitTestKMean() {
  srandom(3);
  GSetVecFloat input = GSetVecFloatCreateStatic();
  int K = 3;
  float mean = 0.0;
  float sigma = 3.0;
  Gauss gauss = GaussCreateStatic(mean, sigma);
  int nbData = 100;
  VecFloat2D tgtMean[3];
  FILE* csvFile = fopen("./kmeancluster.csv", "w");
  for (int iInput = 0; iInput < K; ++iInput) {
    tgtMean[iInput] = VecFloatCreateStatic2D();
    VecSet(tgtMean + iInput, 0, rnd() * 50.0);
    VecSet(tgtMean + iInput, 1, rnd() * 50.0);
    printf("Target #%d: ", iInput);
    VecPrint(tgtMean + iInput, stdout);
    printf("\n");
    fprintf(csvFile, "%f %f\n",
      VecGet(tgtMean + iInput, 0),
      VecGet(tgtMean + iInput, 1));
  }
  fprintf(csvFile, "\n");
  for (int iData = 0; iData < nbData; ++iData) {
    for (int iInput = 0; iInput < K; ++iInput) {
      VecFloat* vFloat = VecFloatCreate(2);
      GSetAppend(&input, vFloat);
      VecSet(vFloat, 0, VecGet(tgtMean + iInput, 0) +
        GaussRnd(&gauss));
      VecSet(vFloat, 1, VecGet(tgtMean + iInput, 1) +
        GaussRnd(&gauss));
      fprintf(csvFile, "%f %f ",
        VecGet(vFloat, 0), VecGet(vFloat, 1));
    }
    fprintf(csvFile, "\n");
  }
  fprintf(csvFile, "\n");
  printf("--- Seed: random\n");
  KMeansClustersSeed seed = KMeansClustersSeed_Random;
  KMeansClusters clusters = KMeansClustersCreateStatic(seed);
  KMeansClustersSearch(&clusters, &input, K);
  if (GSetNbElem(KMeansClustersCenters(&clusters)) != K) {
    PBDataAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBDataAnalysisErr->_msg, "_GetKMeansClusterFloat NOK");
    PBErrCatch(PBDataAnalysisErr);
  }
  for (int iCenter = 0; iCenter < K; ++iCenter) {
    const VecFloat* vFloat = KMeansClustersCenter(&clusters, iCenter);
    printf("Cluster #%d: ", iCenter);
    VecPrint(vFloat, stdout);
    printf("\n");
    fprintf(csvFile, "%f %f\n",
      VecGet(vFloat, 0), VecGet(vFloat, 1));
```

```
}
KMeansClustersFreeStatic(&clusters);
printf("--- Seed: forgy\n");
seed = KMeansClustersSeed_Forgy;
clusters = KMeansClustersCreateStatic(seed);
KMeansClustersSearch(&clusters, &input, K);
if (GSetNbElem(KMeansClustersCenters(&clusters)) != K) {
  PBDataAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBDataAnalysisErr->_msg, "_GetKMeansClusterFloat NOK");
  PBErrCatch(PBDataAnalysisErr);
}
for (int iCenter = 0; iCenter < K; ++iCenter) {
  const VecFloat* vFloat = KMeansClustersCenter(&clusters, iCenter);
  printf("Cluster #%d: ", iCenter);
  VecPrint(vFloat, stdout);
  printf("\n");
  fprintf(csvFile, "%f %f\n",
    VecGet(vFloat, 0), VecGet(vFloat, 1));
}
KMeansClustersFreeStatic(&clusters);
printf("--- Seed: plusplus\n");
seed = KMeansClustersSeed_PlusPlus;
clusters = KMeansClustersCreateStatic(seed);
KMeansClustersSearch(&clusters, &input, K);
if (GSetNbElem(KMeansClustersCenters(&clusters)) != K) {
  PBDataAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBDataAnalysisErr->_msg, "_GetKMeansClusterFloat NOK");
  PBErrCatch(PBDataAnalysisErr);
}
for (int iCenter = 0; iCenter < K; ++iCenter) {
  const VecFloat* vFloat = KMeansClustersCenter(&clusters, iCenter);
  printf("Cluster #%d: ", iCenter);
  VecPrint(vFloat, stdout);
  printf("\n");
  fprintf(csvFile, "%f %f\n",
    VecGet(vFloat, 0), VecGet(vFloat, 1));
}
FILE* fd = fopen("./kmeancluster.txt", "w");
if (!KMeansClustersSave(&clusters, fd, false)) {
  PBDataAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBDataAnalysisErr->_msg, "KMeansClustersSave NOK");
  PBErrCatch(PBDataAnalysisErr);
}
fclose(fd);
KMeansClusters loadClusters =
  KMeansClustersCreateStatic(KMeansClustersSeed_Default);
fd = fopen("./kmeancluster.txt", "r");
if (!KMeansClustersLoad(&loadClusters, fd)) {
  PBDataAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBDataAnalysisErr->_msg, "KMeansClustersLoad NOK");
  PBErrCatch(PBDataAnalysisErr);
}
fclose(fd);
if (clusters._seed != loadClusters._seed ||
  GSetNbElem(KMeansClustersCenters(&clusters)) !=
    GSetNbElem(KMeansClustersCenters(&loadClusters))) {
  PBDataAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBDataAnalysisErr->_msg, "KMeansClustersLoad NOK");
  PBErrCatch(PBDataAnalysisErr);
}
GSetIterForward iter =
  GSetIterForwardCreateStatic(KMeansClustersCenters(&clusters));
```

```
  GSetIterForward iterLoad =
    GSetIterForwardCreateStatic(KMeansClustersCenters(&loadClusters));
  do {
    VecFloat* u = GSetIterGet(&iter);
    VecFloat* v = GSetIterGet(&iterLoad);
    if (VecIsEqual(u, v) == false) {
      PBDataAnalysisErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBDataAnalysisErr->_msg, "KMeansClustersLoad NOK");
      PBErrCatch(PBDataAnalysisErr);
    }
  } while (GSetIterStep(&iter) && GSetIterStep(&iterLoad));
  KMeansClustersFreeStatic(&loadClusters);
  KMeansClustersFreeStatic(&clusters);
  fclose(csvFile);
  while (GSetNbElem(&input) > 0) {
    VecFloat* v = GSetPop(&input);
    VecFree(&v);
  }
  printf("UnitTestKMean OK\n");
}

void UnitTestPCA() {
  char* datasetPath = "./unitTestPCA.json";
  GDataSetVecFloat dataset =
    GDataSetVecFloatCreateStaticFromFile(datasetPath);
  PrincipalComponentAnalysis pca =
    PrincipalComponentAnalysisCreateStatic();
  PCASearch(&pca, &dataset);
  printf("Components:\n");
  PCAPrintln(&pca, stdout);
  GDataSetVecFloat convDataSet1D = PCAConvert(&pca, &dataset, 1);
  printf("Projection on 1st component:\n");
  GSetIterForward iter =
    GSetIterForwardCreateStatic(GDSSamples(&convDataSet1D));
  do {
    VecFloat* sample = GSetIterGet(&iter);
    VecPrint(sample, stdout); printf("\n");
  } while (GSetIterStep(&iter));
  GDataSetVecFloat convDataSet2D = PCAConvert(&pca, &dataset, 2);
  printf("Projection on 2nd component:\n");
  iter = GSetIterForwardCreateStatic(GDSSamples(&convDataSet2D));
  do {
    VecFloat* sample = GSetIterGet(&iter);
    VecPrint(sample, stdout); printf("\n");
  } while (GSetIterStep(&iter));
  GDataSetVecFloatFreeStatic(&convDataSet1D);
  GDataSetVecFloatFreeStatic(&convDataSet2D);
  PrincipalComponentAnalysisFreeStatic(&pca);
  GDataSetVecFloatFreeStatic(&dataset);
  printf("UnitTestPCA OK\n");
}

void UnitTestAll() {
  UnitTestKMean();
  UnitTestPCA();
}

int main(void) {
  UnitTestAll();
  return 0;
}
```

# 6 Unit tests output

```
Target #0: <28.069,11.249>
Target #1: <19.655,22.197>
Target #2: <14.252,7.239>
--- Seed: random
Cluster #0: <19.957,21.695>
Cluster #1: <13.881,7.343>
Cluster #2: <28.442,11.917>
--- Seed: forgy
Cluster #0: <28.442,11.917>
Cluster #1: <19.957,21.695>
Cluster #2: <13.881,7.343>
--- Seed: plusplus
Cluster #0: <28.442,11.917>
Cluster #1: <13.881,7.343>
Cluster #2: <19.957,21.695>
UnitTestKMean OK
Components:
<0.016,-0.279,-0.035,0.315,-0.010,0.174,-0.209,-0.125,-0.006,-0.006,-0.037,0.088,0.033,-0.000,0.584,-0.011,-0.617>
<0.010,0.435,-0.297,-0.142,-0.285,-0.246,0.452,0.201,0.035,0.009,-0.034,-0.019,-0.144,0.246,0.474,0.046,-0.072>
<0.008,0.099,0.158,-0.102,0.084,-0.753,-0.053,-0.002,-0.034,-0.018,0.042,0.082,0.057,-0.262,-0.221,-0.036,-0.496>
<-0.071,0.439,-0.037,-0.303,-0.321,0.510,-0.074,-0.026,-0.001,0.023,0.159,0.088,0.044,-0.324,-0.226,0.036,-0.382>
<0.065,-0.070,-0.157,0.016,-0.042,-0.078,0.069,-0.067,0.012,-0.018,-0.083,0.003,-0.084,-0.851,0.322,0.043,0.324>
<0.033,0.706,0.202,0.308,0.305,-0.012,-0.421,-0.050,-0.030,-0.025,-0.159,-0.068,0.029,0.008,0.188,-0.035,0.152>
<-0.007,-0.068,0.563,0.127,-0.714,-0.112,-0.177,0.190,-0.050,-0.082,0.007,0.104,0.061,0.022,0.105,-0.055,0.184>
<0.034,0.119,-0.071,0.354,-0.214,-0.117,0.185,-0.733,0.066,0.013,0.391,0.058,0.099,0.070,-0.088,0.180,0.092>
<-0.103,0.028,0.561,-0.273,0.279,0.141,0.492,-0.211,-0.083,-0.065,-0.110,0.102,0.356,-0.030,0.233,-0.006,-0.013>
<0.137,-0.069,-0.088,-0.657,-0.062,-0.133,-0.457,-0.365,-0.116,0.111,0.091,-0.067,0.077,0.160,0.273,-0.086,0.145>
<0.321,-0.006,0.048,0.028,-0.212,0.033,0.120,-0.247,0.087,-0.024,-0.510,-0.660,0.015,-0.021,-0.146,-0.151,-0.154>
<0.535,-0.006,0.363,-0.050,0.141,0.074,0.076,0.084,0.141,0.377,0.316,-0.073,-0.471,-0.020,0.062,0.211,-0.058>
<-0.293,0.006,0.131,-0.077,0.001,0.019,0.030,-0.317,0.160,-0.154,-0.205,0.254,-0.698,0.036,-0.034,-0.378,-0.013>
<0.007,0.006,0.035,0.053,0.077,0.041,0.086,0.076,-0.475,-0.399,0.508,-0.388,-0.136,-0.051,0.070,-0.394,-0.023>
<-0.517,-0.039,0.132,-0.090,0.042,-0.049,-0.103,0.057,0.565,-0.071,0.251,-0.489,-0.008,-0.036,0.127,0.208,-0.014>
<-0.107,0.029,-0.033,0.115,-0.017,-0.008,0.081,0.051,0.181,0.631,0.150,-0.018,0.219,-0.055,0.028,-0.676,0.036>
<0.453,0.021,-0.062,-0.054,0.054,0.027,-0.013,0.076,0.582,-0.496,0.136,0.217,0.208,0.017,0.015,-0.290,0.027>
Projection on 1st component:
<0.059>
<-0.179>
<0.043>
<0.077>
Projection on 2nd component:
<0.059,0.008>
<-0.179,0.013>
<0.043,-0.104>
<0.077,0.083>
UnitTestPCA OK
```

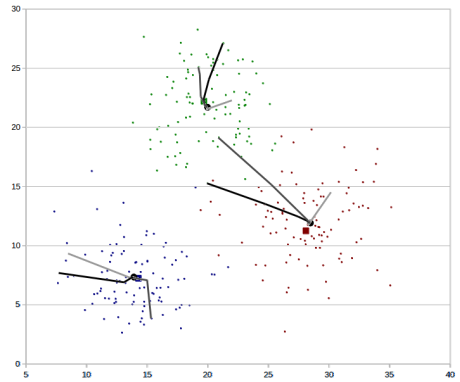
kmeancluster.csv:

```
28.069008 11.249166
19.654589 22.196920
14.252063 7.239053

28.633823 12.292907 19.235598 18.823493 12.592678 3.948273
26.052284 19.234270 18.743282 24.418528 10.458279 5.078588
26.130377 12.850240 22.535856 21.905999 10.845601 13.083977
26.455097 8.574337 18.511803 20.898257 14.739879 4.883637
28.733614 10.599995 23.665422 25.562562 14.357016 6.407752
28.550997 10.792174 15.249710 18.156440 20.529245 7.550764
```

```
29.843285 10.751228 20.759161 25.732025 9.862355 9.238699
29.386349 10.359577 22.475607 19.885625 14.946790 11.217808
28.227690 11.757756 22.867130 25.741074 14.902649 10.882385
28.538338 19.813602 18.626719 26.148878 11.498481 5.557252
23.947056 8.374684 20.750635 24.407253 13.187488 6.842857
28.845385 11.675118 22.477848 25.658518 16.712572 6.494166
28.240320 6.265765 18.270994 16.920400 12.141680 9.390920
24.723717 8.320883 22.146622 22.992454 11.246556 7.748080
26.633812 6.446450 13.803344 20.392096 18.239250 9.086375
30.953035 9.323501 21.829857 21.131970 16.259020 7.211295
31.116953 12.879289 23.409569 22.789968 14.732009 6.460821
20.959503 12.602968 17.141291 23.852451 14.999218 8.711921
27.956499 13.609019 18.733019 22.014161 15.527146 10.999743
31.872864 8.943518 18.358654 24.653771 8.360085 10.208816
26.097677 16.264061 24.531662 23.721465 12.468403 6.878687
26.388678 11.445029 20.422182 25.463302 14.714397 3.323504
24.513933 7.059963 18.509027 22.554037 18.467571 4.957978
29.366932 10.861627 17.426935 22.157186 14.995366 8.656789
31.595938 14.901086 19.876196 26.168687 12.279941 6.107559
27.047731 18.721298 17.749092 27.143915 17.660381 4.754192
24.668179 13.485799 21.459522 22.727382 13.219142 7.327538
30.785263 15.390684 23.087008 21.896501 11.119055 6.156090
25.330399 12.283074 18.255594 22.551985 12.758935 11.746021
24.778790 12.417658 15.977521 20.013983 17.043211 8.389001
31.640587 12.987788 17.524673 20.441366 14.605970 4.453372
29.095758 11.579761 22.153564 18.537164 14.559906 8.431289
32.779095 15.358914 20.257149 25.212011 13.068298 10.726462
35.030907 6.639762 21.238081 25.346833 16.072897 6.441304
26.896448 16.172876 14.700457 27.644926 8.433315 7.357404
32.441208 13.260548 18.330910 24.867006 14.426729 3.567520
19.391371 12.997996 15.806515 19.831161 15.215398 5.320875
28.895748 9.814055 20.318626 23.252077 7.718215 7.693003
27.068031 10.696910 16.536890 22.733471 15.396071 5.993648
32.208397 16.384440 25.096729 21.963248 11.924290 10.051946
26.491650 6.049949 20.415474 22.120099 10.869869 5.957185
26.589045 10.089059 18.382231 22.843460 14.501829 10.078506
26.485561 12.192287 19.841579 19.588705 13.052887 9.544923
24.927652 12.940019 17.671413 26.235970 15.761093 6.421089
31.238827 18.318321 19.240604 25.089768 11.164451 6.369447
28.939318 12.142779 18.180834 20.816853 12.375098 5.505887
27.630131 10.561348 22.605457 19.463206 14.375712 7.359557
27.850735 13.993543 15.209973 21.950428 14.728844 5.261156
30.143599 11.337689 15.328620 22.779089 8.273642 8.736053
30.769220 12.207006 19.299946 24.585787 17.362402 4.612987
32.244652 10.282433 17.320114 19.385283 12.901281 2.647302
24.538588 11.602779 17.704199 17.823857 18.038988 7.197000
32.762169 13.371628 17.284149 17.552123 8.941116 7.436787
22.792772 10.254848 22.606411 20.504032 16.266470 4.143282
29.497095 8.332836 20.680696 21.482531 7.318890 12.883151
25.626740 11.106086 21.263109 27.102179 13.899856 5.798704
29.959078 5.556306 17.444815 18.728331 13.491052 3.413284
29.565657 11.146928 16.102915 18.778041 12.468431 7.059418
20.874537 9.183593 20.529369 20.739050 11.255786 9.527601
28.047850 10.108036 23.392248 19.218472 16.328957 9.925617
28.980659 13.423824 25.531322 18.621785 21.656321 8.177464
23.947527 13.464624 20.417976 25.766448 13.998024 8.555393
20.401991 15.498606 20.430428 24.843374 12.270667 5.141373
28.698526 13.782323 23.226360 18.814351 18.963131 14.917116
24.172115 14.940063 20.003445 25.918981 12.409090 5.246104
28.780060 12.445168 17.376379 16.831505 10.603637 5.899348
29.091532 14.746472 17.044893 23.312667 17.813711 4.991866
29.301987 14.153702 18.020756 25.616368 11.686535 7.868994
```

```
31.022701 8.606833 22.743259 17.513430 15.340607 3.845795
29.180979 11.557543 23.135773 22.946237 16.151800 5.273566
27.930099 14.438169 22.556080 24.529539 12.015292 9.173822
33.846203 16.903858 18.179155 16.632879 12.875151 9.295219
25.154898 11.033173 23.341743 19.887143 16.519030 10.240063
35.106670 13.242561 21.428787 21.090490 12.723807 7.010818
25.209135 12.850850 25.295937 18.056234 13.486193 7.795741
29.438828 15.266409 23.976778 24.548757 9.845288 4.550106
29.725252 6.946030 18.197956 24.123977 15.505687 5.925257
27.486713 8.837636 20.891312 19.136850 13.277709 6.045727
29.519789 14.146199 23.007465 22.318643 13.021230 13.619497
26.245506 13.107678 21.434750 21.703260 13.752280 5.025379
33.955357 7.920842 19.684978 21.106571 7.609594 6.834551
27.990343 10.415890 16.665651 17.502065 14.444242 7.779034
26.135830 12.700714 22.301006 19.147244 17.841988 9.468630
25.933100 15.111071 22.559328 21.635068 12.447936 10.128994
27.279961 15.185239 15.790156 16.344281 15.927351 5.365943
25.760311 13.632949 20.413557 18.830910 11.671001 7.179330
31.429111 14.423276 19.958885 15.272198 17.757336 3.014698
32.629345 10.570016 18.707993 22.012554 14.535404 3.854448
28.192547 8.295099 16.633928 24.252713 16.008110 5.620173
26.773861 9.202417 23.548534 18.603051 13.867304 5.248979
29.160664 10.902171 20.796232 18.358614 11.421997 3.795157
30.813576 8.899913 23.060558 15.621968 10.409031 16.299699
20.219097 13.715732 20.710800 25.666638 17.539631 7.110938
33.206272 13.169051 21.665442 26.510429 16.426327 8.984314
33.969456 18.170853 23.012854 21.830002 15.478644 7.657965
29.226646 9.814377 22.302011 19.372134 11.912398 8.626128
26.332895 2.736907 19.141897 28.252583 17.338530 8.767639
31.999931 13.560342 18.478439 22.093272 20.308399 7.575285
24.406387 14.606756 15.232427 18.946994 14.076594 8.607274
33.677944 15.398574 16.708639 20.123031 11.821822 5.521711

19.956741 21.694563
13.880673 7.343436
28.442358 11.917129
28.442358 11.917129
19.956741 21.694563
13.880673 7.343436
28.442358 11.917129
13.880673 7.343436
19.956741 21.694563
```

random seed in grey, Forgy seed in black, plusplus in dark grey:




kmeancluster.txt:

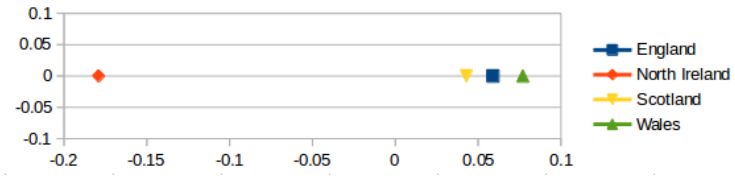

```
{
  "_seed":"2",
  "_centers":[
    {
      "_dim":"2",
      "_val":["28.442358","11.917129"]
    },
    {
      "_dim":"2",
      "_val":["13.880673","7.343436"]
    },
    {
      "_dim":"2",
      "_val":["19.956741","21.694563"]
    }
  ]
}
```


unitTestPCA.json:

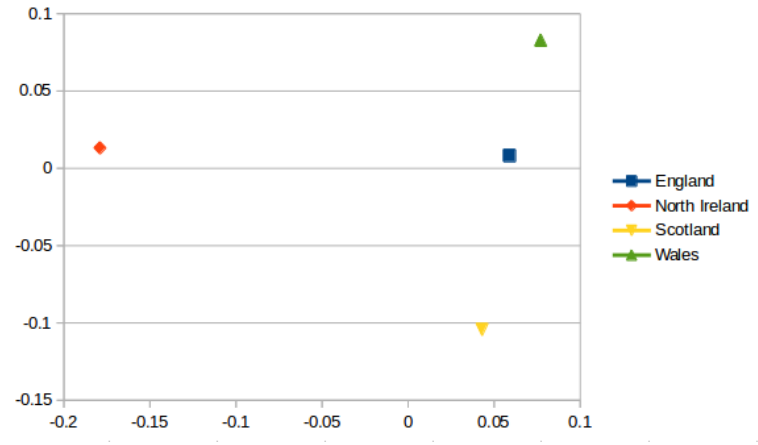

```
{
  "dataSet": "unitTestPCA",
  "dataSetType": "0",
  "desc": "Average consumption of 17 types of food in grams per person per week for every country in the UK: England
  "dim": {
    "_dim":"1",
    "_val":["17"]
  },
  "nbSample": "4",
  "samples": [
    {
      "_dim":"17",
      "_val":["375.0","57.0","245.0","1472.0","105.0","54.0","193.0","147.0","1102.0","720.0","253.0","685.0","488.0"
    },
    {
      "_dim":"17",
      "_val":["135.0","47.0","267.0","1494.0","66.0","41.0","209.0","93.0","674.0","1033.0","143.0","586.0","355.0","
    },
    {
```

```
      "_dim":"17",
      "_val":["458.0","53.0","242.0","1462.0","103.0","62.0","184.0","122.0","957.0","566.0","171.0","750.0","418.0"
    },
    {
      "_dim":"17",
      "_val":["475.0","73.0","227.0","1582.0","103.0","64.0","235.0","160.0","1137.0","874.0","265.0","803.0","570.0
    }
  ]
}
```


PCA1D:




PCA2D:




# 7 YoloBoxes

main.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <math.h>
#include "pbdataanalysis.h"

float* GetYoloBoxes(int K, int nbData, float* data) {

  // Init the random generator
  srandom(time(NULL));

  // Create the set of input data
  GSetVecFloat input = GSetVecFloatCreateStatic();
  for (int iData = 0; iData < nbData; ++iData) {
    VecFloat* vFloat = VecFloatCreate(2);
    GSetAppend(&input, vFloat);
    for (int iInput = 0; iInput < 2; ++iInput) {
      VecSet(vFloat, iInput, data[iData * 2 + iInput]);
    }
  }

  // Create the KMeansClusters
  KMeansClustersSeed seed = KMeansClustersSeed_PlusPlus;
  KMeansClusters clusters = KMeansClustersCreateStatic(seed);

  // Search the K-Means
  KMeansClustersSearch(&clusters, &input, K);

  // Store the found K-Means
  float* ret = malloc(sizeof(float) * K * 2);
  if (data == NULL) {
    printf("Failed to allocate memory\n");
    exit(1);
  }
  for (int iCenter = 0; iCenter < K; ++iCenter) {
    const VecFloat* vFloat = KMeansClustersCenter(&clusters, iCenter);
    ret[iCenter * 2] = VecGet(vFloat, 0);
    ret[iCenter * 2 + 1] = VecGet(vFloat, 1);
  }

  // Free memory
  KMeansClustersFreeStatic(&clusters);
  while (GSetNbElem(&input) > 0) {
    VecFloat* v = GSetPop(&input);
    VecFree(&v);
  }

  // Return the result
  return ret;
}

// Arguments:
// main <width image> <height image> <K, equals to 'num' in the
// yolo config file> <input file>
// where input file has the following format: first line is the
// number of boxes in input, following lines are relative width
// and relative height separated by a space of each box, one per line

int main(int argc, char** argv) {

  // Check the number of arguments
```

```
  if (argc != 5) {
    printf("Invalid number of arguments (%d)\n", argc);
    exit(1);
  }

  // Get the image dimension
  int width = atoi(argv[1]);
  int height = atoi(argv[2]);

  // Get the number of clusters
  int K = atoi(argv[3]);

  // Get the data file name
  char* filename = argv[4];
  FILE* fp = fopen(filename, "r");

  // Read the number of data
  int nbData;
  if (fscanf(fp, "%d\n", &nbData) == EOF) {
    printf("Failed to read the data\n");
    exit(1);
  }

  // Get the data
  float* data = malloc(sizeof(float) * (nbData * 2));
  if (data == NULL) {
    printf("Failed to allocate memory\n");
    exit(1);
  }
  for (int iData = 0; iData < nbData; ++iData) {
    if (fscanf(fp, "%f %f\n", data + iData * 2, data + iData * 2 + 1) == EOF) {
      printf("Failed to read the data\n");
      exit(1);
    }
  }

  // Close the input file
  fclose(fp);

  // Search the clusters
  float* ret = GetYoloBoxes(K, nbData, data);

  // Display the results
  for (int i = 0; i < K; ++i) {
    int w = (int)round(ret[2 * i] * (float)width);
    int h = (int)round(ret[2 * i + 1] * (float)height);
    printf("%d,%d ", w, h);
  }
  printf("\n");

  // Free memory
  free(data);
  free(ret);

  return 0;
}
```

example of use and output:

```
cat Data/*.txt > targets.txt; \
```

```
wc -l targets.txt | awk '{print $1}' > input.txt; \
cat targets.txt | awk '{printf "%f %f\n", $4, $5}' >> input.txt
main 250 250 9 input.txt
89,87 37,37 69,70 31,31 87,78 56,57 41,41 80,82 46,47
```