PBImgAnalysis

P. Baillehache

March 12, 2019

Contents

Introduction

PBImgAnalysis is a C library providing structures and functions to perform various data analysis on images.

It implements the following algorithms:

- K-means clustering on the RGBA space of pixels in a user defined radius
- Intersection over Union (aka Jaccard index)

It uses the `PBErr`, `PBDataAnalaysis`, `GenBrush` libraries.

1 Interface

```
// ============ PBIMGANALYSIS.H ================

#ifndef PBIMGANALYSIS_H
#define PBIMGANALYSIS_H

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

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <execinfo.h>
#include <errno.h>
#include <string.h>
#include "pberr.h"
#include "pbdataanalysis.h"
#include "genbrush.h"
#include "genalg.h"
#include "neuranet.h"
#include "gdataset.h"

// ------------------ ImgKMeansClusters ----------------------

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

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

typedef struct ImgKMeansClusters {
  // Image on which the clustering is applied
  // Uses the GBSurfaceFinalPixels
  const GenBrush* _img;
  // Clusters result of the search
  KMeansClusters _kmeansClusters;
  // Size of the considered cell in the image around a given position
  // is equal to (_size * 2 + 1)
  int _size;
} ImgKMeansClusters;

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

// Create a new ImgKMeansClusters for the image 'img' and with seed 'seed'
// and type 'type' and a cell size equal to 2*'size'+1
ImgKMeansClusters ImgKMeansClustersCreateStatic(
  const GenBrush* const img, const KMeansClustersSeed seed,
  const int size);

// Free the memory used by a ImgKMeansClusters
void ImgKMeansClustersFreeStatic(ImgKMeansClusters* const that);

// Get the GenBrush of the ImgKMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
const GenBrush* IKMCImg(const ImgKMeansClusters* const that);

// Set the GenBrush of the ImgKMeansClusters 'that' to 'img'
#if BUILDMODE != 0
inline
#endif
void IKMCSetImg(ImgKMeansClusters* const that, const GenBrush* const img);
```

```
// Set the size of the cells of the ImgKMeansClusters 'that' to
// 2*'size'+1
#if BUILDMODE != 0
inline
#endif
void IKMCSetSizeCell(ImgKMeansClusters* const that, const int size);

// Get the number of cluster of the ImgKMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
int IKMCGetK(const ImgKMeansClusters* const that);

// Get the size of the cells of the ImgKMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
int IKMCGetSizeCell(const ImgKMeansClusters* const that);

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

// Search for the 'K' clusters in the image of the
// ImgKMeansClusters 'that'
void IKMCSearch(ImgKMeansClusters* const that, const int K);

// Print the ImgKMeansClusters 'that' on the stream 'stream'
void IKMCPrintln(const ImgKMeansClusters* const that,
  FILE* const stream);

// Get the index of the cluster at position 'pos' for the
// ImgKMeansClusters 'that'
int IKMCGetId(const ImgKMeansClusters* const that,
  const VecShort2D* const pos);

// Get the GBPixel equivalent to the cluster at position 'pos'
// for the ImgKMeansClusters 'that'
GBPixel IKMCGetPixel(const ImgKMeansClusters* const that,
  const VecShort2D* const pos);

// Convert the image of the ImageKMeansClusters 'that' to its clustered
// version
// IKMCSearch must have been called previously
void IKMCCluster(const ImgKMeansClusters* const that);

// Load the IKMC 'that' from the stream 'stream'
// There is no associated GenBrush object saved
// Return true upon success else false
bool IKMCLoad(ImgKMeansClusters* that, FILE* const stream);

// Save the IKMC 'that' to the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// There is no associated GenBrush object saved
// Return true upon success else false
bool IKMCSave(const ImgKMeansClusters* const that,
  FILE* const stream, const bool compact);
```

```
// Function which return the JSON encoding of 'that'
JSONNode* IKMCEncodeAsJSON(const ImgKMeansClusters* const that);

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

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

// ------------------ General functions ----------------------

// Return the Jaccard index (aka intersection over union) of the
// images 'that' and 'tho' for pixels of color 'rgba'
// 'that' and 'tho' must have same dimensions
float IntersectionOverUnion(const GenBrush* const that,
  const GenBrush* const tho, const GBPixel* const rgba);

// Return the similarity coefficient of the images 'that' and 'tho'
// (i.e. the sum of the distances of pixels at the same position
// over the whole image)
// Return a value in [0.0, 1.0], 1.0 means the two images are
// identical, 0.0 means they are binary black and white with each
// pixel in one image the opposite of the corresponding pixel in the
// other image.
// 'that' and 'tho' must have same dimensions
float GBSimilarityCoeff(const GenBrush* const that,
  const GenBrush* const tho);

// ------------------ ImgSegmentor ----------------------

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

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

typedef struct ImgSegmentor {
  // Tree of criterion
  GenTree _criteria;
  // Number of segmentation class
  int _nbClass;
  // Flag to apply or not the binarization on result of prediction
  // false by default
  bool _flagBinaryResult;
  // Threshold value for the binarization of result of prediction
  // If the result of prediction is above the threshold then
  // the result is considered equal to 1.0 else it is considered equal
  // to -1.0
  // 0.5 by default
  float _thresholdBinaryResult;
  // Nb of epoch for training, 1 by default
  unsigned int _nbEpoch;
  // Size pool for training
  // By default GENALG_NBENTITIES
  int _sizePool;
  // Nb elite for training
  // By default GENALG_NBELITES
  int _nbElite;
  // Threshold to stop the training once
  float _targetBestValue;
} ImgSegmentor;

typedef struct ImgSegmentorPerf {
```

```
  // Accuracy
  float _accuracy;
} ImgSegmentorPerf;

typedef struct ImgSegmentorTrainParam {
  // Nb of epochs
  int _nbEpoch;
} ImgSegmentorParam;

typedef enum ISCType {
  ISCType_RGB
} ISCType;

typedef struct ImgSegmentorCriterion {
  // Type of criteriion
  ISCType _type;
  // Nb of class
  int _nbClass;
} ImgSegmentorCriterion;

typedef struct ImgSegmentorCriterionRGB {
  // ImgSegmentorCriterion
  ImgSegmentorCriterion _criterion;
  // NeuraNet model
  NeuraNet* _nn;
} ImgSegmentorCriterionRGB;

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

// Create a new static ImgSegmentor with 'nbClass' output
ImgSegmentor ImgSegmentorCreateStatic(int nbClass);

// Free the memory used by the static ImgSegmentor 'that'
void ImgSegmentorFreeStatic(ImgSegmentor* that);

// Return the nb of criterion of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
long ISGetNbCriterion(const ImgSegmentor* const that);

// Add a new ImageSegmentorCriterionRGB to the ImgSegmentor 'that'
// under the node 'parent'
// If 'parent' is null it is inserted to the root of the ImgSegmentor
#if BUILDMODE != 0
inline
#endif
bool ISAddCriterionRGB(ImgSegmentor* const that,
  void* const parent);

// Return the nb of classes of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
int ISGetNbClass(const ImgSegmentor* const that);

// Return the flag controlling the binarization of the result of
// prediction of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
bool ISGetFlagBinaryResult(const ImgSegmentor* const that);
```

```
// Return the threshold controlling the binarization of the result of
// prediction of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
float ISGetThresholdBinaryResult(const ImgSegmentor* const that);

// Return the threshold controlling the stop of the training
#if BUILDMODE != 0
inline
#endif
float ISGetTargetBestValue(const ImgSegmentor* const that);

// Set the threshold controlling the stop of the training to 'val'
// Clip the value to [0.0, 1.0]
#if BUILDMODE != 0
inline
#endif
void ISSetTargetBestValue(ImgSegmentor* const that, const float val);

// Set the flag controlling the binarization of the result of
// prediction of the ImgSegmentor 'that' to 'flag'
#if BUILDMODE != 0
inline
#endif
void ISSetFlagBinaryResult(ImgSegmentor* const that,
  const bool flag);

// Return the number of epoch for training the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
unsigned int ISGetNbEpoch(const ImgSegmentor* const that);

// Set the number of epoch for training the ImgSegmentor 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void ISSetNbEpoch(ImgSegmentor* const that, unsigned int nb);

// Return the size of the pool for training the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
int ISGetSizePool(const ImgSegmentor* const that);

// Set the size of the pool for training the ImgSegmentor 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void ISSetSizePool(ImgSegmentor* const that, int nb);

// Return the nb of elites for training the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
int ISGetNbElite(const ImgSegmentor* const that);

// Set the nb of elites for training the ImgSegmentor 'that' to 'nb'
#if BUILDMODE != 0
inline
```

```
#endif
void ISSetNbElite(ImgSegmentor* const that, int nb);

// Set the threshold controlling the binarization of the result of
// prediction of the ImgSegmentor 'that' to 'threshold'
#if BUILDMODE != 0
inline
#endif
void ISSetThresholdBinaryResult(ImgSegmentor* const that,
  const float threshold);

// Make a prediction on the GenBrush 'img' with the ImgSegmentor 'that'
// Return an array of pointer to GenBrush, one per output class, in
// greyscale, where the color of each pixel indicates the detection of
// the corresponding class at the given pixel, white equals no
// detection, black equals detection, 50% grey equals "don't know"
GenBrush** ISPredict(const ImgSegmentor* const that,
  const GenBrush* const img);

// Return the nb of criterion of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
const GenTree* ISCriteria(const ImgSegmentor* const that);

// Train the ImageSegmentor 'that' on the data set 'dataSet' using
// the data of the first category in 'dataSet'
// srandom must have been caled before calling ISTrain
void ISTrain(ImgSegmentor* const that,
  const GDataSetGenBrushPair* const dataset);

// Create a new static ImgSegmentorCriterion with 'nbClass' output
// and the type of criterion 'type'
ImgSegmentorCriterion ImgSegmentorCriterionCreateStatic(int nbClass,
  ISCType type);

// Free the memory used by the static ImgSegmentorCriterion 'that'
void ImgSegmentorCriterionFreeStatic(ImgSegmentorCriterion* that);

// Make the prediction on the 'input' values by calling the appropriate
// function according to the type of criterion
// 'input' 's format is width*height*3, values in [0.0, 1.0]
// Return values are width*height*nbClass, values in [-1.0, 1.0]
VecFloat* ISCPredict(const ImgSegmentorCriterion* const that,
  const VecFloat* input, const VecShort2D* const dim);

// Return the nb of class of the ImgSegmentorCriterion 'that'
#if BUILDMODE != 0
inline
#endif
int _ISCGetNbClass(const ImgSegmentorCriterion* const that);

// Return the number of int parameters for the criterion 'that'
long _ISCGetNbParamInt(const ImgSegmentorCriterion* const that);

// Return the number of float parameters for the criterion 'that'
long _ISCGetNbParamFloat(const ImgSegmentorCriterion* const that);

// Set the bounds of int parameters for training of the criterion 'that'
void _ISCSetBoundsAdnInt(const ImgSegmentorCriterion* const that,
  GenAlg* const ga, const long shift);
```

```
// Set the bounds of float parameters for training of the criterion 'that'
void _ISCSetBoundsAdnFloat(const ImgSegmentorCriterion* const that,
  GenAlg* const ga, const long shift);

// Set the values of int parameters for training of the criterion 'that'
void _ISCSetAdnInt(const ImgSegmentorCriterion* const that,
  const GenAlgAdn* const adn, const long shift);

// Set the values of float parameters for training of the criterion 'that'
void _ISCSetAdnFloat(const ImgSegmentorCriterion* const that,
  const GenAlgAdn* const adn, const long shift);

// Create a new ImgSegmentorCriterionRGB with 'nbClass' output
ImgSegmentorCriterionRGB* ImgSegmentorCriterionRGBCreate(int nbClass);

// Free the memory used by the ImgSegmentorCriterionRGB 'that'
void ImgSegmentorCriterionRGBFree(ImgSegmentorCriterionRGB** that);

// Make the prediction on the 'input' values with the
// ImgSegmentorCriterionRGB that
// 'input' 's format is width*height*3, values in [0.0, 1.0]
// Return values are width*height*nbClass, values in [-1.0, 1.0]
VecFloat* ISCRGBPredict(const ImgSegmentorCriterionRGB* const that,
  const VecFloat* input, const VecShort2D* const dim);

// Return the number of int parameters for the criterion 'that'
long ISCRGBGetNbParamInt(const ImgSegmentorCriterionRGB* const that);

// Return the number of float parameters for the criterion 'that'
long ISCRGBGetNbParamFloat(const ImgSegmentorCriterionRGB* const that);

// Set the bounds of int parameters for training of the criterion 'that'
void ISCRGBSetBoundsAdnInt(const ImgSegmentorCriterionRGB* const that,
  GenAlg* const ga, const long shift);

// Set the bounds of float parameters for training of the criterion 'that'
void ISCRGBSetBoundsAdnFloat(const ImgSegmentorCriterionRGB* const that,
  GenAlg* const ga, const long shift);

// Set the values of int parameters for training of the criterion 'that'
void ISCRGBSetAdnInt(const ImgSegmentorCriterionRGB* const that,
  const GenAlgAdn* const adn, const long shift);

// Set the values of float parameters for training of the criterion 'that'
void ISCRGBSetAdnFloat(const ImgSegmentorCriterionRGB* const that,
  const GenAlgAdn* const adn, const long shift);

// Return the NeuraNet of the ImgSegmentorCriterionRGB 'that'
#if BUILDMODE != 0
inline
#endif
const NeuraNet* ISCRGBNeuraNet(
  const ImgSegmentorCriterionRGB* const that);

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

#define ISCGetNbClass(That) _Generic(That, \
  ImgSegmentorCriterion*: _ISCGetNbClass, \
  const ImgSegmentorCriterion*: _ISCGetNbClass, \
  ImgSegmentorCriterionRGB*: _ISCGetNbClass, \
  const ImgSegmentorCriterionRGB*: _ISCGetNbClass, \
  default: PBErrInvalidPolymorphism) ((const ImgSegmentorCriterion*)That)
```

```
#define ISCGetNbParamInt(That) _Generic(That, \
  ImgSegmentorCriterion*: _ISCGetNbParamInt, \
  const ImgSegmentorCriterion*: _ISCGetNbParamInt, \
  ImgSegmentorCriterionRGB*: ISCRGBGetNbParamInt, \
  const ImgSegmentorCriterionRGB*: ISCRGBGetNbParamInt, \
  default: PBErrInvalidPolymorphism) ((const ImgSegmentorCriterion*)That)

#define ISCGetNbParamFloat(That) _Generic(That, \
  ImgSegmentorCriterion*: _ISCGetNbParamFloat, \
  const ImgSegmentorCriterion*: _ISCGetNbParamFloat, \
  ImgSegmentorCriterionRGB*: ISCRGBGetNbParamFloat, \
  const ImgSegmentorCriterionRGB*: ISCRGBGetNbParamFloat, \
  default: PBErrInvalidPolymorphism) ((const ImgSegmentorCriterion*)That)

#define ISCSetBoundsAdnInt(That, GenAlg, Shift) _Generic(That, \
  ImgSegmentorCriterion*: _ISCSetBoundsAdnInt, \
  const ImgSegmentorCriterion*: _ISCSetBoundsAdnInt, \
  ImgSegmentorCriterionRGB*: ISCRGBSetBoundsAdnInt, \
  const ImgSegmentorCriterionRGB*: ISCRGBSetBoundsAdnInt, \
  default: PBErrInvalidPolymorphism) ( \
    (const ImgSegmentorCriterion*)That, GenAlg, Shift)

#define ISCSetBoundsAdnFloat(That, GenAlg, Shift) _Generic(That, \
  ImgSegmentorCriterion*: _ISCSetBoundsAdnFloat, \
  const ImgSegmentorCriterion*: _ISCSetBoundsAdnFloat, \
  ImgSegmentorCriterionRGB*: ISCRGBSetBoundsAdnFloat, \
  const ImgSegmentorCriterionRGB*: ISCRGBSetBoundsAdnFloat, \
  default: PBErrInvalidPolymorphism) ( \
    (const ImgSegmentorCriterion*)That, GenAlg, Shift)

#define ISCSetAdnInt(That, Adn, Shift) _Generic(That, \
  ImgSegmentorCriterion*: _ISCSetAdnInt, \
  const ImgSegmentorCriterion*: _ISCSetAdnInt, \
  ImgSegmentorCriterionRGB*: ISCRGBSetAdnInt, \
  const ImgSegmentorCriterionRGB*: ISCRGBSetAdnInt, \
  default: PBErrInvalidPolymorphism) ( \
    (const ImgSegmentorCriterion*)That, Adn, Shift)

#define ISCSetAdnFloat(That, Adn, Shift) _Generic(That, \
  ImgSegmentorCriterion*: _ISCSetAdnFloat, \
  const ImgSegmentorCriterion*: _ISCSetAdnFloat, \
  ImgSegmentorCriterionRGB*: ISCRGBSetAdnFloat, \
  const ImgSegmentorCriterionRGB*: ISCRGBSetAdnFloat, \
  default: PBErrInvalidPolymorphism) ( \
    (const ImgSegmentorCriterion*)That, Adn, Shift)

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

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

#endif
```

2 Code

2.1 pbimganalysis.c

```
// =========== PBIMGANALYSIS.C ================

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

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

// ------------------ ImgKMeansClusters ----------------------

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

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

// Get the input values for the pixel at position 'pos' according to
// the cell size of the ImgKMeansClusters 'that'
// The return is a VecFloat made of the sizeCell^2 pixels' value
// around pos ordered by (((r*256+g)*256+b)*256+a)
VecFloat* IKMCGetInputOverCell(const ImgKMeansClusters* const that,
  const VecShort2D* const pos);

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

// Create a new ImgKMeansClusters for the image 'img' and with seed 'seed'
// and type 'type' and a cell size equal to 2*'size'+1
ImgKMeansClusters ImgKMeansClustersCreateStatic(
  const GenBrush* const img, const KMeansClustersSeed seed,
  const int size) {
#if BUILDMODE == 0
  if (img == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'img' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (size < 0) {
    PBImgAnalysisErr->_type = PBErrTypeInvalidArg;
    sprintf(PBImgAnalysisErr->_msg, "'size' is invalid (%d>=0)", size);
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Declare the new ImgKMeansClusters
  ImgKMeansClusters that;
  // Set properties
  that._img = img;
  that._kmeansClusters = KMeansClustersCreateStatic(seed);
  that._size = size;
  // Return the new ImgKMeansClusters
  return that;
}

// Free the memory used by a ImgKMeansClusters
void ImgKMeansClustersFreeStatic(ImgKMeansClusters* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
```

```
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Reset the GenBrush associated to the IKMC
  that->_img = NULL;
  // Free the memory used by the KMeansClusters
  KMeansClustersFreeStatic((KMeansClusters*)IKMCKMeansClusters(that));
}

// Search for the 'K' clusters in the image of the
// ImgKMeansClusters 'that'
void IKMCSearch(ImgKMeansClusters* const that, const int K) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (K < 1) {
    PBImgAnalysisErr->_type = PBErrTypeInvalidArg;
    sprintf(PBImgAnalysisErr->_msg, "'K' is invalid (%d>0)", K);
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Create a set to memorize the input over cells
  GSetVecFloat inputOverCells = GSetVecFloatCreateStatic();
  // Get the dimension of the image
  VecShort2D dim = GBGetDim(IKMCImg(that));
  // Loop on pixels
  VecShort2D pos = VecShortCreateStatic2D();
  do {
    // Get the KMeansClusters input over the cell
    VecFloat* inputOverCell = IKMCGetInputOverCell(that, &pos);
    // Add it to the inputs for the search
    GSetAppend(&inputOverCells, inputOverCell);
  } while (VecStep(&pos, &dim));
  // Search the clusters
  KMeansClustersSearch((KMeansClusters*)IKMCKMeansClusters(that),
    &inputOverCells, K);
  // Free the memory used by the input
  while (GSetNbElem(&inputOverCells) > 0) {
    VecFloat* v = GSetPop(&inputOverCells);
    VecFree(&v);
  }
}

// Print the ImgKMeansClusters 'that' on the stream 'stream'
void IKMCPrintln(const ImgKMeansClusters* const that,
  FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (stream == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'stream' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Print the KMeansClusters of 'that'
```

```
  KMeansClustersPrintln(IKMCKMeansClusters(that), stream);
}

// Get the index of the cluster at position 'pos' for the
// ImgKMeansClusters 'that'
int IKMCGetId(const ImgKMeansClusters* const that,
  const VecShort2D* const pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (pos == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'pos' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Get the KMeansClusters input over the cell
  VecFloat* inputOverCell = IKMCGetInputOverCell(that, pos);
  // Get the index of the cluster for this pixel
  int id = KMeansClustersGetId(IKMCKMeansClusters(that), inputOverCell);
  // Free memory
  VecFree(&inputOverCell);
  // Return the id
  return id;
}

// Get the GBPixel equivalent to the cluster at position 'pos'
// for the ImgKMeansClusters 'that'
// This is the average pixel over the pixel in the cell of the cluster
GBPixel IKMCGetPixel(const ImgKMeansClusters* const that,
  const VecShort2D* const pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (pos == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'pos' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Declare the result pixel
  GBPixel pix;
  // Get the id of the cluster for the input pixel
  int id = IKMCGetId(that, pos);
  // Get the 'id'-th cluster's center
  const VecFloat* center =
    KMeansClustersCenter(IKMCKMeansClusters(that), id);
  // Declare a variable to calculate the average pixel
  VecFloat* avgPix = VecFloatCreate(4);
  // Calculate the average pixel
  for (int i = 0; i < VecGetDim(center); i += 4) {
    for (int j = 4; j--;) {
      VecSet(avgPix, j, VecGet(avgPix, j) + VecGet(center, i + j));
    }
  }
  VecScale(avgPix, 1.0 / round((float)VecGetDim(center) / 4.0));
```

```
  // Update the returned pixel values and ensure the converted value
  // from float to char is valid
  for (int i = 4; i--;) {
    float v = VecGet(avgPix, i);
    if (v < 0.0)
      v = 0.0;
    else if (v > 255.0)
      v = 255.0;
    pix._rgba[i] = (unsigned char)v;
  }
  // Free memory
  VecFree(&avgPix);
  // Return the result pixel
  return pix;
}

// Convert the image of the ImageKMeansClusters 'that' to its clustered
// version
// IKMCSearch must have been called previously
void IKMCCluster(const ImgKMeansClusters* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Get the dimension of the image
  VecShort2D dim = GBGetDim(IKMCImg(that));
  // Loop on pixels
  VecShort2D pos = VecShortCreateStatic2D();
  do {
    // Get the clustered pixel for this pixel
    GBPixel clustered = IKMCGetPixel(that, &pos);
    // Replace the original pixel
    GBSetFinalPixel((GenBrush*)IKMCImg(that), &pos, &clustered);
  } while (VecStep(&pos, &dim));
}

// Get the input values for the pixel at position 'pos' according to
// the cell size of the ImgKMeansClusters 'that'
// The return is a VecFloat made of the sizeCell^2 pixels' value
// around pos ordered by (((r*256+g)*256+b)*256+a)
VecFloat* IKMCGetInputOverCell(const ImgKMeansClusters* const that,
  const VecShort2D* const pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (pos == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'pos' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Create two vectors to loop on the cell
  VecShort2D from = VecShortCreateStatic2D();
  VecSet(&from, 0, -that->_size);
  VecSet(&from, 1, -that->_size);
  VecShort2D to = VecShortCreateStatic2D();
```

```
  VecSet(&to, 0, that->_size + 1);
  VecSet(&to, 1, that->_size + 1);
  // Get the pixel at the center of the cell, will be used as default
  // if the cell goes over the border of the image
  const GBPixel* defaultPixel = GBFinalPixel(IKMCImg(that), pos);
  // Declare a set to memorize the pixels in the cell
  GSet pixels = GSetCreateStatic();
  // Loop over the pixels of the cell
  VecShort2D posCell = from;
  VecShort2D posImg = VecShortCreateStatic2D();
  do {
    // If the position in the cell is inside the radius of the cell
    VecFloat2D posCellFloat = VecShortToFloat2D(&posCell);
    if ((int)round(VecNorm(&posCellFloat)) <= that->_size) {
      // Get the position in the image
      posImg = VecGetOp(pos, 1, &posCell, 1);
      // Get the pixel at this position
      const GBPixel* pix = GBFinalPixelSafe(IKMCImg(that), &posImg);
      if (pix == NULL)
        pix = defaultPixel;
      // Get the value to sort this pixel
      float valPix = 0.0;
      for (int iRgba = 4; iRgba--;)
        valPix += 256.0 * valPix + (float)(pix->_rgba[iRgba]);
      // Add the pixel to the set of pixels in the cell
      GSetAddSort(&pixels, pix, valPix);
    }
  } while (VecShiftStep(&posCell, &from, &to));
  // Declare the result vector
  VecFloat* res = VecFloatCreate(GSetNbElem(&pixels) * 4);
  // Loop over the sorted pixels of the cell
  int iPix = 0;
  while (GSetNbElem(&pixels)) {
    const GBPixel* pix = GSetDrop(&pixels);
    // Set the result value
    for (int i = 0; i < 4; ++i)
      VecSet(res, iPix * 4 + i, (float)(pix->_rgba[i]));
    ++iPix;
  }
  // Return the result
  return res;
}

// Load the IKMC 'that' from the stream 'stream'
// There is no associated GenBrush object saved
// Return true upon success else false
bool IKMCLoad(ImgKMeansClusters* that, FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (stream == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'stream' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#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 (!IKMCDecodeAsJSON(that, json)) {
    return false;
  }
  // Free the memory used by the JSON
  JSONFree(&json);
  // Return success code
  return true;
}

// Save the IKMC 'that' to the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// There is no associated GenBrush object saved
// Return true upon success else false
bool IKMCSave(const ImgKMeansClusters* const that,
  FILE* const stream, const bool compact) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (stream == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'stream' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Get the JSON encoding
  JSONNode* json = IKMCEncodeAsJSON(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* IKMCEncodeAsJSON(const ImgKMeansClusters* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Create the JSON structure
  JSONNode* json = JSONCreate();
  // Declare a buffer to convert value into string
  char val[100];
  // Encode the size
  sprintf(val, "%d", that->_size);
  JSONAddProp(json, "_size", val);
  // Encode the KMeansClusters
  JSONAddProp(json, "_clusters",
```

```
    KMeansClustersEncodeAsJSON(IKMCKMeansClusters(that)));
  // Return the created JSON
  return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool IKMCDecodeAsJSON(ImgKMeansClusters* that,
  const JSONNode* const json) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (json == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'json' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Free the memory eventually used by the IKMC
  ImgKMeansClustersFreeStatic(that);
  // Get the size from the JSON
  JSONNode* prop = JSONProperty(json, "_size");
  if (prop == NULL) {
    return false;
  }
  that->_size = atoi(JSONLabel(JSONValue(prop, 0)));
  if (that->_size < 0) {
    return false;
  }
  // Decode the KMeansClusters
  prop = JSONProperty(json, "_clusters");
  if (!KMeansClustersDecodeAsJSON(
    (KMeansClusters*)IKMCKMeansClusters(that), prop)) {
    return false;
  }
  // Return the success code
  return true;
}

// ------------------ ImgSegmentor ----------------------

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

// Create a new static ImgSegmentor with 'nbClass' output
ImgSegmentor ImgSegmentorCreateStatic(int nbClass) {
#if BUILDMODE == 0
  if (nbClass <= 0) {
    PBImgAnalysisErr->_type = PBErrTypeInvalidArg;
    sprintf(PBImgAnalysisErr->_msg, "'nbClass' is invalid (%d>0)",
      nbClass);
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Declare the new ImgSegmentor
  ImgSegmentor that;
  // Init properties
  that._nbClass = nbClass;
  that._criteria = GenTreeCreateStatic();
  that._flagBinaryResult = false;
  that._thresholdBinaryResult = 0.5;
```

```
  that._nbEpoch = 1;
  that._sizePool = GENALG_NBENTITIES;
  that._nbElite = GENALG_NBELITES;
  that._targetBestValue = 0.9999;
  // Return the new ImgSegmentor
  return that;
}

// Free the memory used by the static ImgSegmentor 'that'
void ImgSegmentorFreeStatic(ImgSegmentor* that) {
  if (that == NULL)
    return;
  GenTreeIterDepth iter = GenTreeIterDepthCreateStatic(ISCriteria(that));
  do {
    ImgSegmentorCriterion* criterion = GenTreeIterGetData(&iter);
    switch (criterion->_type) {
      case ISCType_RGB:
        ImgSegmentorCriterionRGBFree(
          (ImgSegmentorCriterionRGB**)&criterion);
        break;
      default:
        PBImgAnalysisErr->_type = PBErrTypeNotYetImplemented;
        sprintf(PBImgAnalysisErr->_msg,
          "Not yet implemented type of criterion");
        PBErrCatch(PBImgAnalysisErr);
        break;
    }
  } while (GenTreeIterStep(&iter));
  GenTreeIterFreeStatic(&iter);
  GenTreeFreeStatic((GenTree*)ISCriteria(that));
}

// Make a prediction on the GenBrush 'img' with the ImgSegmentor 'that'
// Return an array of pointer to GenBrush, one per output class, in
// greyscale, where the color of each pixel indicates the detection of
// the corresponding class at the given pixel, white equals no
// detection, black equals detection, 50% grey equals "don't know"
GenBrush** ISPredict(const ImgSegmentor* const that,
  const GenBrush* const img) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (img == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'img' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Get the dimension of the input image
  VecShort2D dim = GBGetDim(img);
  // Calculate the area of the image
  long area = VecGet(&dim, 0) * VecGet(&dim, 1);
  // Create a temporary vector to convert the image into the input
  // of a criterion
  VecFloat* input = VecFloatCreate(area * 3);
  // Declare a vector to loop on position in the image
  VecShort2D pos = VecShortCreateStatic2D();
  // Convert the image's pixels into the input VecFloat
  do {
```

```
  GBPixel pix = GBGetFinalPixel(img, &pos);
  long iPos = GBPosIndex(&pos, &dim);
  for (int iRGB = 3; iRGB--;)
    VecSet(input, iPos * 3 + iRGB, (float)(pix._rgba[iRGB]) / 255.0);
} while (VecStep(&pos, &dim));
// Declare a set to memorize the temporary inputs while moving
// through the tree of criteria
GSet inputs = GSetCreateStatic();
// Add the initial input to the set
GSetAppend(&inputs, input);
// Create a set to memorize the prediction of each leaf criterion
GSet leafPred = GSetCreateStatic();
// Loop on criteria
GenTreeIterDepth iter = GenTreeIterDepthCreateStatic(ISCriteria(that));
do {
  // Get the criteria
  ImgSegmentorCriterion* criterion = GenTreeIterGetData(&iter);
  // Get the input on which to apply the criteria, this is the last
  // pushed input
  VecFloat* curInput = GSetTail(&inputs);
  // Do the prediction
  VecFloat* pred = ISCPredict(criterion, curInput, &dim);
  // If this criterion is a leaf in the tree of crieria
  if (GenTreeIsLeaf(GenTreeIterGetGenTree(&iter))) {
    // Add the result of the prediction to the set of final prediction
    GSetAppend(&leafPred, pred);
    // If the criterion is a last brother
    if (GenTreeIsLastBrother(GenTreeIterGetGenTree(&iter))) {
      // Drop and free the intermediate input
      (void)GSetDrop(&inputs);
      VecFree(&curInput);
      // In case the parent was the last brother it will be skipped
      // back by the GenTreeIterDepth and we need to drop its input
      // right away
      GenTree* parent = GenTreeParent(GenTreeIterGetGenTree(&iter));
      while (parent != NULL && GenTreeIsLastBrother(parent)) {
        curInput = GSetDrop(&inputs);
        VecFree(&curInput);
        parent = GenTreeParent(parent);
      }
    }
  // Else the criterion is a node in the tree
  } else {
    // Append the result of prediction to the intermediate input
    GSetAppend(&inputs, pred);
  }
} while(GenTreeIterStep(&iter));
GenTreeIterFreeStatic(&iter);
// Create temporary vectors to memorize the combined predictions
VecFloat* combPred = VecFloatCreate(area * ISGetNbClass(that));
VecFloat* finalPred = VecFloatCreate(area * ISGetNbClass(that));
// Combine the predictions over criteria
// The combination is the weighted average of prediction over criteria
// where the weight is the absolute value of the prediction
for (long i = area * (long)ISGetNbClass(that); i--;) {
  float sumWeight = 0.0;
  GSetIterForward iter = GSetIterForwardCreateStatic(&leafPred);
  do {
    VecFloat* pred = GSetIterGet(&iter);
    float v = VecGet(pred, i);
    VecSetAdd(combPred, i, v * fabs(v));
    sumWeight += fabs(v);
```

```
  } while (GSetIterStep(&iter));
  if (sumWeight > PBMATH_EPSILON)
    VecSet(combPred, i, VecGet(combPred, i) / sumWeight);
  else
    VecSet(combPred, i, 0.0);
}
// Combine the predictions over classes
// The combination is calculated as follow:
// finalPred(i) = (pred(i)*abs(combPred(i) - sum_{j!=i}
//   combPred(j)*abs(combPred(j)) / (sum_i abs(combPred(i))
VecSetNull(&pos);
do {
  for (long iClass = ISGetNbClass(that); iClass--;) {
    float sumWeight = 0.0;
    long iPos = GBPosIndex(&pos, &dim) * ISGetNbClass(that) + iClass;
    for (long jClass = ISGetNbClass(that); jClass--;) {
      long jPos = GBPosIndex(&pos, &dim) * ISGetNbClass(that) + jClass;
      float v = VecGet(combPred, jPos);
      if (iClass == jClass) {
        VecSetAdd(finalPred, iPos, v * fabs(v));
      } else {
        VecSetAdd(finalPred, iPos, -1.0 * v * fabs(v));
      }
      sumWeight += fabs(v);
    }
    if (sumWeight > PBMATH_EPSILON)
      VecSet(finalPred, iPos, VecGet(finalPred, iPos) / sumWeight);
    else
      VecSet(finalPred, iPos, 0.0);
  }
} while(VecStep(&pos, &dim));
// Allocate memory for the results
GenBrush** res = PBErrMalloc(PBImgAnalysisErr,
  sizeof(GenBrush*) * ISGetNbClass(that));
// Declare a variable to convert the prediction into pixel
GBPixel pix = GBColorWhite;
// Loop on classes
for (int iClass = ISGetNbClass(that); iClass--;) {
  // Create the result GenBrush
  res[iClass] = GBCreateImage(&dim);
  // Loop on position in the image
  VecSetNull(&pos);
  do {
    // Get the prediction value for this class and this position
    // and convert it to rgb value
    long iPos = GBPosIndex(&pos, &dim);
    float p = VecGet(finalPred, iPos * ISGetNbClass(that) + iClass);
    if (ISGetFlagBinaryResult(that)) {
      if (p > ISGetThresholdBinaryResult(that))
        p = 1.0;
      else
        p = -1.0;
    }
    unsigned char pChar = 255 -
      (unsigned char)round(255.0 * (p * 0.5 + 0.5));
    // Convert the prediction to a pixel
    pix._rgba[GBPixelRed] = pix._rgba[GBPixelGreen] =
      pix._rgba[GBPixelBlue] = pChar;
    // Set the pixel in the result image
    GBSetFinalPixel(res[iClass], &pos, &pix);
  } while (VecStep(&pos, &dim));
}
```

```
  // Free memory
  while (GSetNbElem(&leafPred) > 0) {
    VecFloat* pred = GSetPop(&leafPred);
    VecFree(&pred);
  }
  do {
    VecFloat* curInput = GSetDrop(&inputs);
    VecFree(&curInput);
  } while (GSetNbElem(&inputs) > 0);
  VecFree(&finalPred);
  VecFree(&combPred);
  // Return the result
  return res;
}

// Train the ImageSegmentor 'that' on the data set 'dataSet' using
// the data of the first category in 'dataSet'
// srandom must have been caled before calling ISTrain
void ISTrain(ImgSegmentor* const that,
  const GDataSetGenBrushPair* const dataset) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (dataset == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'dataset' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (ISGetNbClass(that) > GDSGetNbMask(dataset)) {
    PBImgAnalysisErr->_type = PBErrTypeInvalidData;
    sprintf(PBImgAnalysisErr->_msg,
      "Not enough masks in the dataset (%d<=%d)",
      ISGetNbClass(that), GDSGetNbMask(dataset));
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // If there is no criterion, nothing to do
  if (ISGetNbCriterion(that) == 0)
    return;
  // Memorize the current flag for binarization of results
  bool curFlagBinary = ISGetFlagBinaryResult(that);
  // Turn on the binarization
  ISSetFlagBinaryResult(that, true);
  // Create two vectors to memorize the number of int and float
  // parameters for each criterion
  VecLong* nbParamInt = VecLongCreate(ISGetNbCriterion(that));
  VecLong* nbParamFloat = VecLongCreate(ISGetNbCriterion(that));
  // Declare two variables to memorize the total number of int and
  // float parameters
  long nbTotalParamInt = 0;
  long nbTotalParamFloat = 0;
  // Decclare a varibale to memorize the color of the mask
  const GBPixel rgbaMask = GBColorBlack;
  // Get the number of int and float parameters for each criterion
  int iCrit = 0;
  GenTreeIterDepth iter = GenTreeIterDepthCreateStatic(ISCriteria(that));
  do {
    ImgSegmentorCriterion* crit = GenTreeIterGetData(&iter);
    long nb = ISCGetNbParamInt(crit);
```

```
  VecSet(nbParamInt, iCrit, nb);
  nbTotalParamInt += nb;
  nb = ISCGetNbParamFloat(crit);
  VecSet(nbParamFloat, iCrit, nb);
  nbTotalParamFloat += nb;
  ++iCrit;
} while (GenTreeIterStep(&iter));
// If there are parameters
if (nbTotalParamInt > 0 || nbTotalParamFloat > 0) {
  // Create the GenAlg to search parameters' value
  GenAlg* ga = GenAlgCreate(ISGetSizePool(that), ISGetNbElite(that),
    nbTotalParamFloat, nbTotalParamInt);
  // Loop on the criterion to initialise the parameters bound
  GenTreeIterReset(&iter);
  long shiftParamInt = 0;
  long shiftParamFloat = 0;
  do {
    ImgSegmentorCriterion* crit = GenTreeIterGetData(&iter);
    ISCSetBoundsAdnInt(crit, ga, shiftParamInt);
    shiftParamInt += ISCGetNbParamInt(crit);
    ISCSetBoundsAdnFloat(crit, ga, shiftParamFloat);
    shiftParamFloat += ISCGetNbParamFloat(crit);
  } while (GenTreeIterStep(&iter));
  // Initialise the GenAlg
  GAInit(ga);
  // Declare a variable to memorize the current best value
  float bestValue = 0.0;
  // Loop over epochs
  do {
    // Loop over the GenAlg entities
    for (int iEnt = 0; iEnt < GAGetNbAdns(ga); ++iEnt) {
      // If this entity is a new one
      if (GAAdnIsNew(GAAdn(ga, iEnt))) {
        // Loop on the criterion to set the criteria parameters with
        // this entity's adn
        GenTreeIterReset(&iter);
        shiftParamInt = 0;
        shiftParamFloat = 0;
        do {
          ImgSegmentorCriterion* crit = GenTreeIterGetData(&iter);
          ISCSetAdnInt(crit, GAAdn(ga, iEnt), shiftParamInt);
          shiftParamInt += ISCGetNbParamInt(crit);
          ISCSetAdnFloat(crit, GAAdn(ga, iEnt), shiftParamFloat);
          shiftParamFloat += ISCGetNbParamFloat(crit);
        } while (GenTreeIterStep(&iter));
        // Evaluate the ImgSegmentor for this entity's adn on the
        // dataset
        float value = 0.0;
        const int iCatTraining = 0;
        // Reset the iterator of the GDataSet
        GDSReset(dataset, iCatTraining);
        // Loop on the samples
        long iSample = 0;
        do {
          printf("Epoch %05ld/%05u ",
            GAGetCurEpoch(ga) + 1, ISGetNbEpoch(that));
          printf("Entity %03d/%03d ",
            iEnt + 1, GAGetNbAdns(ga));
          printf("Sample %05ld/%05ld ",
            iSample + 1, GDSGetSizeCat(dataset, iCatTraining));
          // Get the next sample
          GDSGenBrushPair* sample = GDSGetSample(dataset, iCatTraining);
```

```
          // Do the prediction on the sample
          GenBrush** pred = ISPredict(that, sample->_img);
          // Check the prediction against the masks
          float valMask = 0.0;
          for (int iMask = ISGetNbClass(that); iMask--;) {
            valMask += IntersectionOverUnion(
              sample->_mask[iMask], pred[iMask], &rgbaMask);
          }
          value += valMask / (float)GDSGetNbMask(dataset);
          // Free memory
          for (int iClass = ISGetNbClass(that); iClass--;)
            GBFree(pred + iClass);
          free(pred);
          GDSGenBrushPairFree(&sample);
          if (iSample + 1 < GDSGetSizeCat(dataset, iCatTraining))
            printf("\n");
          fflush(stdout);
          ++iSample;
        } while (GDSStepSample(dataset, iCatTraining));
        // Get the average value over all samples
        value /= (float)GDSGetSizeCat(dataset, iCatTraining);
        // Update the adn value of this entity
        GASetAdnValue(ga, GAAdn(ga, iEnt), value);
        // If the value is the best value
        if (value - bestValue > PBMATH_EPSILON) {
          bestValue = value;
        }
        printf("BestValue %f/%f\n", bestValue,
          ISGetTargetBestValue(that));
      }
    }
    // Step the GenAlg
    GAStep(ga);
  } while (GAGetCurEpoch(ga) < ISGetNbEpoch(that) &&
    bestValue < ISGetTargetBestValue(that));
  // Loop on the criterion to set the criteria to the best one
  GenTreeIterReset(&iter);
  shiftParamInt = 0;
  shiftParamFloat = 0;
  do {
    ImgSegmentorCriterion* crit = GenTreeIterGetData(&iter);
    ISCSetAdnInt(crit, GABestAdn(ga), shiftParamInt);
    shiftParamInt += ISCGetNbParamInt(crit);
    ISCSetAdnFloat(crit, GABestAdn(ga), shiftParamFloat);
    shiftParamFloat += ISCGetNbParamFloat(crit);
  } while (GenTreeIterStep(&iter));
  // Free memory
  GenAlgFree(&ga);
}
// Free memory
GenTreeIterFreeStatic(&iter);
VecFree(&nbParamInt);
VecFree(&nbParamFloat);
// Put back the flag for binarization in its original state
ISSetFlagBinaryResult(that, curFlagBinary);
}

// Create a new static ImgSegmentorCriterion with 'nbClass' output
// and the type of criteria 'type'
ImgSegmentorCriterion ImgSegmentorCriterionCreateStatic(int nbClass,
  ISCType type) {
#if BUILDMODE == 0
```

```
  if (nbClass <= 0) {
    PBImgAnalysisErr->_type = PBErrTypeInvalidArg;
    sprintf(PBImgAnalysisErr->_msg, "'nbClass' is invalid (%d>0)",
      nbClass);
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Declare the new ImgSegmentorCriterion
  ImgSegmentorCriterion that;
  // Set the properties
  that._nbClass = nbClass;
  that._type = type;
  // Return the new ImgSegmentorCriterion
  return that;
}

// Free the memory used by the static ImgSegmentorCriterion 'that'
void ImgSegmentorCriterionFreeStatic(ImgSegmentorCriterion* that) {
  if (that == NULL)
    return;
  // Nothing to do
}

// Make the prediction on the 'input' values by calling the appropriate
// function according to the type of criteria
// 'input' 's format is width*height*3, values in [0.0, 1.0]
// Return values are width*height*nbClass, values in [-1.0, 1.0]
VecFloat* ISCPredict(const ImgSegmentorCriterion* const that,
  const VecFloat* input, const VecShort2D* const dim) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (input == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'input' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Declare a variable to memorize the result
  VecFloat* res = NULL;
  // Call the appropriate function based on the type
  switch(that->_type) {
    case ISCType_RGB:
      res = ISCRGBPredict((const ImgSegmentorCriterionRGB*)that,
        input, dim);
      break;
    default:
      break;
  }
  // Return the result
  return res;
}

// Return the number of int parameters for the criterion 'that'
long _ISCGetNbParamInt(const ImgSegmentorCriterion* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
```

```
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Declare a variable to memorize the result
  long res = 0;
  // Call the appropriate function based on the type
  switch(that->_type) {
    case ISCType_RGB:
      res = ISCRGBGetNbParamInt((const ImgSegmentorCriterionRGB*)that);
      break;
    default:
      break;
  }
  // Return the result
  return res;
}

// Return the number of float parameters for the criterion 'that'
long _ISCGetNbParamFloat(const ImgSegmentorCriterion* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Declare a variable to memorize the result
  long res = 0;
  // Call the appropriate function based on the type
  switch(that->_type) {
    case ISCType_RGB:
      res = ISCRGBGetNbParamFloat((const ImgSegmentorCriterionRGB*)that);
      break;
    default:
      break;
  }
  // Return the result
  return res;
}

// Create a new ImgSegmentorCriterionRGB with 'nbClass' output
ImgSegmentorCriterionRGB* ImgSegmentorCriterionRGBCreate(int nbClass) {
#if BUILDMODE == 0
  if (nbClass <= 0) {
    PBImgAnalysisErr->_type = PBErrTypeInvalidArg;
    sprintf(PBImgAnalysisErr->_msg, "'nbClass' is invalid (%d>0)",
      nbClass);
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Allocate memory for the new ImgSegmentorCriterionRGB
  ImgSegmentorCriterionRGB* that = PBErrMalloc(PBImgAnalysisErr,
    sizeof(ImgSegmentorCriterionRGB));
  // Create the parent ImgSegmentorCriterion
  that->_criterion = ImgSegmentorCriterionCreateStatic(nbClass,
    ISCType_RGB);
  // Create the NeuraNet
  const int nbInput = 3;
  const int nbHidden = fsquare(nbInput) * nbClass;
  VecLong* hidden = VecLongCreate(1);
  VecSet(hidden, 0, nbHidden);
  that->_nn = NeuraNetCreateFullyConnected(nbInput, nbClass, hidden);
```

```
  VecFree(&hidden);
  // Return the new ImgSegmentorCriterionRGB
  return that;
}

// Free the memory used by the ImgSegmentorCriterionRGB 'that'
void ImgSegmentorCriterionRGBFree(ImgSegmentorCriterionRGB** that) {
  if (that == NULL || *that == NULL)
    return;
  // Free memory
  ImgSegmentorCriterionFreeStatic((ImgSegmentorCriterion*)(*that));
  NeuraNetFree(&((*that)->_nn));
  free(*that);
}

// Make the prediction on the 'input' values with the
// ImgSegmentorCriterionRGB that
// 'input' 's format is width*height*3, values in [0.0, 1.0]
// Return values are width*height*nbClass, values in [-1.0, 1.0]
VecFloat* ISCRGBPredict(const ImgSegmentorCriterionRGB* const that,
  const VecFloat* input, const VecShort2D* const dim) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (input == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'input' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if ((VecGetDim(input) % 3) != 0) {
    PBImgAnalysisErr->_type = PBErrTypeInvalidArg;
    sprintf(PBImgAnalysisErr->_msg,
      "'input' 's dim is not multiple of 3 (%ld)", VecGetDim(input));
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Calculate the area of the input image
  long area = VecGet(dim, 0) * VecGet(dim, 1);
  // Allocate memory for the result
  VecFloat* res = VecFloatCreate(area * (long)ISCGetNbClass(that));
  // Declare variables to memorize the input/output of the NeuraNet
  VecFloat3D in = VecFloatCreateStatic3D();
  VecFloat* out = VecFloatCreate(ISCGetNbClass(that));
  // Apply the NeuraNet on inputs
  for (long iInput = area; iInput--;) {
    for (long i = 3; i--;)
      VecSet(&in, i, VecGet(input, iInput * 3L + i));
    NNEval(that->_nn, (VecFloat*)&in, out);
    for (long i = ISCGetNbClass(that); i--;)
      VecSet(res, iInput * (long)ISCGetNbClass(that) + i,
        VecGet(out, i));
  }
  // Free memory
  VecFree(&out);
  // Return the result
  return res;
}

// Return the number of int parameters for the criterion 'that'
```

```
long ISCRGBGetNbParamInt(const ImgSegmentorCriterionRGB* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  (void)that;
  return 0;
}

// Return the number of float parameters for the criterion 'that'
long ISCRGBGetNbParamFloat(const ImgSegmentorCriterionRGB* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  return NNGetGAAdnFloatLength(that->_nn);
}

// Set the bounds of int parameters for training of the criterion 'that'
void _ISCSetBoundsAdnInt(const ImgSegmentorCriterion* const that,
  GenAlg* const ga, const long shift) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (ga == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Call the appropriate function based on the type
  switch(that->_type) {
    case ISCType_RGB:
      ISCRGBSetBoundsAdnInt((const ImgSegmentorCriterionRGB*)that,
        ga, shift);
      break;
    default:
      break;
  }
}

// Set the bounds of float parameters for training of the criterion
// 'that'
void _ISCSetBoundsAdnFloat(const ImgSegmentorCriterion* const that,
  GenAlg* const ga, const long shift) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (ga == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
```

```
    sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Call the appropriate function based on the type
  switch(that->_type) {
    case ISCType_RGB:
      ISCRGBSetBoundsAdnFloat((const ImgSegmentorCriterionRGB*)that,
        ga, shift);
      break;
    default:
      break;
  }
}

// Set the bounds of int parameters for training of the criterion 'that'
void ISCRGBSetBoundsAdnInt(const ImgSegmentorCriterionRGB* const that,
  GenAlg* const ga, const long shift) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (ga == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Nothing to do
  (void)that;(void)ga;(void)shift;
}

// Set the bounds of float parameters for training of the criterion
// 'that'
void ISCRGBSetBoundsAdnFloat(const ImgSegmentorCriterionRGB* const that,
  GenAlg* const ga, const long shift) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (ga == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  VecFloat2D bounds = VecFloatCreateStatic2D();
  VecSet(&bounds, 0, -1.0);
  VecSet(&bounds, 1, 1.0);
  for (long iParam = ISCRGBGetNbParamFloat(that); iParam--;) {
    GASetBoundsAdnFloat(ga, iParam + shift, &bounds);
  }
}

// Set the values of int parameters for training of the criterion 'that'
void _ISCSetAdnInt(const ImgSegmentorCriterion* const that,
  const GenAlgAdn* const adn, const long shift) {
#if BUILDMODE == 0
```

```
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (adn == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Call the appropriate function based on the type
  switch(that->_type) {
    case ISCType_RGB:
      ISCRGBSetAdnInt((const ImgSegmentorCriterionRGB*)that,
        adn, shift);
      break;
    default:
      break;
  }
}

// Set the values of float parameters for training of the criterion
// 'that'
void _ISCSetAdnFloat(const ImgSegmentorCriterion* const that,
  const GenAlgAdn* const adn, const long shift) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (adn == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Call the appropriate function based on the type
  switch(that->_type) {
    case ISCType_RGB:
      ISCRGBSetAdnFloat((const ImgSegmentorCriterionRGB*)that,
        adn, shift);
      break;
    default:
      break;
  }
}

// Set the values of int parameters for training of the criterion 'that'
void ISCRGBSetAdnInt(const ImgSegmentorCriterionRGB* const that,
  const GenAlgAdn* const adn, const long shift) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (adn == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
    PBErrCatch(PBImgAnalysisErr);
```

```
  }
#endif
  // Nothing to do
  (void)that;(void)adn;(void)shift;
}

// Set the values of float parameters for training of the criterion
// 'that'
void ISCRGBSetAdnFloat(const ImgSegmentorCriterionRGB* const that,
  const GenAlgAdn* const adn, const long shift) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (adn == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  const VecFloat* adnF = GAAdnAdnF(adn);
  VecFloat* bases = VecFloatCreate(ISCRGBGetNbParamFloat(that));
  for (int i = ISCRGBGetNbParamFloat(that); i--;)
    VecSet(bases, i, VecGet(adnF, shift + i));
  NNSetBases((NeuraNet*)ISCRGBNeuraNet(that), bases);
  VecFree(&bases);
}

// ------------------ General functions ----------------------

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

// Return the Jaccard index (aka intersection over union) of the
// image 'that' and 'tho' for pixels of color 'rgba'
// 'that' and 'tho' must have same dimensions
float IntersectionOverUnion(const GenBrush* const that,
  const GenBrush* const tho, const GBPixel* const rgba) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (tho == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'tho' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (rgba == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'rgba' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (!VecIsEqual(GBDim(that), GBDim(tho))) {
    PBImgAnalysisErr->_type = PBErrTypeInvalidArg;
    sprintf(PBImgAnalysisErr->_msg,
      "'that' and 'tho' have different dimensions");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
```

```
  // Declare two variables to count the number of pixels in
  // intersection and union
  long nbUnion = 0;
  long nbInter = 0;
  // Declare a variable to loop through pixels
  VecShort2D pos = VecShortCreateStatic2D();
  // Loop through pixels
  do {
    // If the pixel is in the intersection
    if (GBPixelIsSame(GBFinalPixel(that, &pos), rgba) &&
      GBPixelIsSame(GBFinalPixel(tho, &pos), rgba)) {
      // Increment the number of pixels in intersection
      ++nbInter;
    }
    // If the pixel is in the union
    if (GBPixelIsSame(GBFinalPixel(that, &pos), rgba) ||
      GBPixelIsSame(GBFinalPixel(tho, &pos), rgba)) {
      // Increment the number of pixels in union
      ++nbUnion;
    }
  } while (VecStep(&pos, GBDim(that)));
  // Calculate the intersection over union
  float iou = (float)nbInter / (float)nbUnion;
  // Return the result
  return iou;
}

// Return the similarity coefficient of the images 'that' and 'tho'
// (i.e. the sum of the distances of pixels at the same position
// over the whole image)
// Return a value in [0.0, 1.0], 1.0 means the two images are
// identical, 0.0 means they are binary black and white with each
// pixel in one image the opposite of the corresponding pixel in the
// other image.
// 'that' and 'tho' must have same dimensions
float GBSimilarityCoeff(const GenBrush* const that,
  const GenBrush* const tho) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (tho == NULL) {
    PBImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'tho' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (!VecIsEqual(GBDim(that), GBDim(tho))) {
    PBImgAnalysisErr->_type = PBErrTypeInvalidArg;
    sprintf(PBImgAnalysisErr->_msg,
      "'that' and 'tho' have different dimensions");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Declare a variable to calculate the result
  float res = 0.0;
  // Declare a variable to loop through pixels
  VecShort2D pos = VecShortCreateStatic2D();
  // Loop through pixels
  do {
    const GBPixel* pixA = GBFinalPixel(that, &pos);
```

```
    const GBPixel* pixB = GBFinalPixel(tho, &pos);
    res += sqrt(
      fsquare((float)(pixA->_rgba[0]) - (float)(pixB->_rgba[0])) +
      fsquare((float)(pixA->_rgba[1]) - (float)(pixB->_rgba[1])) +
      fsquare((float)(pixA->_rgba[2]) - (float)(pixB->_rgba[2])) +
      fsquare((float)(pixA->_rgba[3]) - (float)(pixB->_rgba[3])));
  } while (VecStep(&pos, GBDim(that)));
  // Calculate the result
  res /= (float)GBArea(that) * 510.0;
  // Return the result
  return 1.0 - res;
}
```

3 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=pbimganalysis
$($(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)/$
```

4 Unit tests

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

void UnitTestImgKMeansClusters() {
  srandom(1);
```

```
  for (int size = 0; size < 6; ++size) {
    for (int K = 2; K <= 6; ++K) {
      char* fileName = "./ImgKMeansClustersTest/imgkmeanscluster.tga";
      GenBrush* img = GBCreateFromFile(fileName);
      ImgKMeansClusters clusters = ImgKMeansClustersCreateStatic(
        img, KMeansClustersSeed_Forgy, size);
      IKMCSearch(&clusters, K);

      FILE* fd = fopen("./imgkmeanscluster.txt", "w");
      if (!IKMCSave(&clusters, fd, false)) {
        PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBImgAnalysisErr->_msg, "IKMCSave NOK");
        PBErrCatch(PBImgAnalysisErr);
      }
      fclose(fd);
      fd = fopen("./imgkmeanscluster.txt", "r");
      if (!IKMCLoad(&clusters, fd)) {
        PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBImgAnalysisErr->_msg, "IKMCLoad NOK");
        PBErrCatch(PBImgAnalysisErr);
      }
      IKMCSetImg(&clusters, img);
      fclose(fd);

      printf("%s size K=%d cell=%d:\n",
        fileName, K, IKMCGetSizeCell(&clusters));
      IKMCPrintln(&clusters, stdout);
      IKMCCluster(&clusters);
      char fileNameOut[50] = {'\0'};
      sprintf(fileNameOut,
        "./ImgKMeansClustersTest/imgkmeanscluster%02d-%02d.tga", K, size);
      GBSetFileName(img, fileNameOut);
      GBRender(img);
      GBFree(&img);
      ImgKMeansClustersFreeStatic(&clusters);
    }
  }
  printf("UnitTestImgKMeansClusters OK\n");
}

void UnitTestIntersectionOverUnion() {
  char* fileNameA = "./iou1.tga";
  GenBrush* imgA = GBCreateFromFile(fileNameA);
  char* fileNameB = "./iou2.tga";
  GenBrush* imgB = GBCreateFromFile(fileNameB);
  GBPixel rgba = GBColorBlack;
  float iou = IntersectionOverUnion(imgA, imgB, &rgba);
  if (!ISEQUALF(iou, 6.0 / 10.0)) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "IntersectionOverUnion failed");
    PBErrCatch(PBImgAnalysisErr);
  }
  GBFree(&imgA);
  GBFree(&imgB);
  printf("UnitTestIntersectionOverUnion OK\n");
}

void UnitTestGBSimilarityCoefficient() {
  char* fileNameA = "./iou1.tga";
  GenBrush* imgA = GBCreateFromFile(fileNameA);
  char* fileNameB = "./iou2.tga";
  GenBrush* imgB = GBCreateFromFile(fileNameB);
```

```
  float sim = GBSimilarityCoeff(imgA, imgA);
  if (!ISEQUALF(sim, 1.0)) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "GBSimilarityCoefficient failed");
    PBErrCatch(PBImgAnalysisErr);
  }
  sim = GBSimilarityCoeff(imgA, imgB);
  if (!ISEQUALF(sim, 0.965359)) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "GBSimilarityCoefficient failed");
    PBErrCatch(PBImgAnalysisErr);
  }
  GBFree(&imgA);
  GBFree(&imgB);
  printf("UnitTestIntersectionOverUnion OK\n");
}

void UnitTestImgSegmentorRGB() {
  int nbClass = 2;
  ImgSegmentorCriterionRGB* criterion =
    ImgSegmentorCriterionRGBCreate(nbClass);
  if (ISCGetNbClass(criterion) != nbClass) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg,
      "ImgSegmentorCriterionRGBCreate failed");
    PBErrCatch(PBImgAnalysisErr);
  }
  int imgArea = 4;
  VecFloat* input = VecFloatCreate(imgArea * 3);
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 2);
  VecSet(&dim, 1, 2);
  VecFloat* output = ISCRGBPredict(criterion, input, &dim);
  if (VecGetDim(output) != imgArea * nbClass) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISCRGBPredict failed");
    PBErrCatch(PBImgAnalysisErr);
  }
  VecFree(&input);
  VecFree(&output);
  ImgSegmentorCriterionRGBFree(&criterion);
  printf("UnitTestImgSegmentorRGB OK\n");
}

void UnitTestImgSegmentorCreateFree() {
  int nbClass = 2;
  ImgSegmentor segmentor = ImgSegmentorCreateStatic(nbClass);
  if (segmentor._nbClass != nbClass ||
    segmentor._flagBinaryResult != false ||
    segmentor._nbEpoch != 1 ||
    !ISEQUALF(segmentor._thresholdBinaryResult, 0.5) ||
    !ISEQUALF(segmentor._targetBestValue, 0.9999)) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ImgSegmentorCreateStatic failed");
    PBErrCatch(PBImgAnalysisErr);
  }
  ImgSegmentorFreeStatic(&segmentor);
  printf("UnitTestImgSegmentorCreateFree OK\n");
}

void UnitTestImgSegmentorAddCriterionGetSet() {
  int nbClass = 2;
```

```
ImgSegmentor segmentor = ImgSegmentorCreateStatic(nbClass);
if (ISCriteria(&segmentor) != &(segmentor._criteria)) {
  PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBImgAnalysisErr->_msg, "ISCriteria failed");
  PBErrCatch(PBImgAnalysisErr);
}
if (ISGetNbClass(&segmentor) != nbClass) {
  PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBImgAnalysisErr->_msg, "ISGetNbClass failed");
  PBErrCatch(PBImgAnalysisErr);
}
if (ISGetNbCriterion(&segmentor) != 0) {
  PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBImgAnalysisErr->_msg, "ISGetNbCriterion failed");
  PBErrCatch(PBImgAnalysisErr);
}
if (!ISAddCriterionRGB(&segmentor, NULL) ||
  GenTreeGetSize(ISCriteria(&segmentor)) != 1 ||
  ((ImgSegmentorCriterion*)GSetGet(&(segmentor._criteria._subtrees),
    0))->_type != ISCType_RGB) {
  PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBImgAnalysisErr->_msg, "ISAddCriterion failed");
  PBErrCatch(PBImgAnalysisErr);
}
if (ISGetNbCriterion(&segmentor) != 1) {
  PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBImgAnalysisErr->_msg, "ISGetNbCriterion failed");
  PBErrCatch(PBImgAnalysisErr);
}
ISSetFlagBinaryResult(&segmentor, true);
if (segmentor._flagBinaryResult != true) {
  PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBImgAnalysisErr->_msg, "ISSetFlagBinaryResult failed");
  PBErrCatch(PBImgAnalysisErr);
}
if (ISGetFlagBinaryResult(&segmentor) != true) {
  PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBImgAnalysisErr->_msg, "ISGetFlagBinaryResult failed");
  PBErrCatch(PBImgAnalysisErr);
}
ISSetThresholdBinaryResult(&segmentor, 1.0);
if (!ISEQUALF(segmentor._thresholdBinaryResult, 1.0)) {
  PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBImgAnalysisErr->_msg, "ISSetThrehsoldBinaryResult failed");
  PBErrCatch(PBImgAnalysisErr);
}
if (!ISEQUALF(ISGetThresholdBinaryResult(&segmentor), 1.0)) {
  PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBImgAnalysisErr->_msg, "ISGetThresholdBinaryResult failed");
  PBErrCatch(PBImgAnalysisErr);
}
if (ISGetSizePool(&segmentor) != GENALG_NBENTITIES) {
  PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBImgAnalysisErr->_msg, "ISGetSizePool failed");
  PBErrCatch(PBImgAnalysisErr);
}
ISSetSizePool(&segmentor, GENALG_NBENTITIES + 100);
if (ISGetSizePool(&segmentor) != GENALG_NBENTITIES + 100) {
  PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBImgAnalysisErr->_msg, "ISSetSizePool failed");
  PBErrCatch(PBImgAnalysisErr);
}
```

```
  if (ISGetNbElite(&segmentor) != GENALG_NBELITES) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISGetNbElite failed");
    PBErrCatch(PBImgAnalysisErr);
  }
  ISSetNbElite(&segmentor, GENALG_NBELITES + 10);
  if (ISGetNbElite(&segmentor) != GENALG_NBELITES + 10) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISSetNbElite failed");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (!ISEQUALF(ISGetTargetBestValue(&segmentor), 0.9999)) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISGetTargetBestValue failed");
    PBErrCatch(PBImgAnalysisErr);
  }
  ISSetTargetBestValue(&segmentor, 0.5);
  if (!ISEQUALF(ISGetTargetBestValue(&segmentor), 0.5)) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISSetTargetBestValue failed");
    PBErrCatch(PBImgAnalysisErr);
  }
  ImgSegmentorFreeStatic(&segmentor);
  printf("UnitTestImgSegmentorAddCriterionGetSet OK\n");
}

void UnitTestImgSegmentorPredict() {
  int nbClass = 2;
  ImgSegmentor segmentor = ImgSegmentorCreateStatic(nbClass);
  ISAddCriterionRGB(&segmentor, NULL);
  char* fileNameIn = "ISPredict-in.tga";
  char fileNameOut[20];
  GenBrush* img = GBCreateFromFile(fileNameIn);
  GenBrush** res = ISPredict(&segmentor, img);
  for (int iClass = nbClass; iClass--;) {
    sprintf(fileNameOut, "ISPredict-out%02d.tga", iClass);
    GBSetFileName(res[iClass], fileNameOut);
    GBRender(res[iClass]);
  }
  ImgSegmentorFreeStatic(&segmentor);
  for (int iClass = nbClass; iClass--;)
    GBFree(res + iClass);
  free(res);
  GBFree(&img);
  printf("UnitTestImgSegmentorPredict OK\n");
}

void UnitTestImgSegmentorTrain() {
  srandom(0);
  int nbClass = 2;
  ImgSegmentor segmentor = ImgSegmentorCreateStatic(nbClass);
  ISAddCriterionRGB(&segmentor, NULL);
  char* cfgFilePath = PBFSJoinPath(
    ".", "UnitTestImgSegmentorTrain", "dataset.json");
  GDataSetGenBrushPair dataSet =
    GDataSetGenBrushPairCreateStatic(cfgFilePath);
  ISSetSizePool(&segmentor, 20);
  ISSetNbElite(&segmentor, 5);
  ISSetNbEpoch(&segmentor, 50);
  //ISSetSizePool(&segmentor, 2);
  //ISSetNbElite(&segmentor, 2);
  //ISSetNbEpoch(&segmentor, 2);
```

```
  ISSetTargetBestValue(&segmentor, 0.9);
  ISTrain(&segmentor, &dataSet);
  char* imgFilePath = PBFSJoinPath(
    ".", "UnitTestImgSegmentorTrain", "img000.tga");
  GenBrush* img = GBCreateFromFile(imgFilePath);
  ISSetFlagBinaryResult(&segmentor, true);
  GenBrush** pred = ISPredict(&segmentor, img);
  for (int iClass = nbClass; iClass--;) {
    char outPath[100];
    sprintf(outPath, "pred000-%03d.tga", iClass);
    char* predFilePath = PBFSJoinPath(
      ".", "UnitTestImgSegmentorTrain", outPath);
    GBSetFileName(pred[iClass], predFilePath);
    GBRender(pred[iClass]);
    GBFree(pred + iClass);
    free(predFilePath);
  }
  free(pred);
  GBFree(&img);
  free(cfgFilePath);
  free(imgFilePath);
  GDataSetGenBrushPairFreeStatic(&dataSet);
  ImgSegmentorFreeStatic(&segmentor);
  printf("UnitTestImgSegmentorTrain OK\n");
}

void UnitTestImgSegmentor() {
  UnitTestImgSegmentorCreateFree();
  UnitTestImgSegmentorAddCriterionGetSet();
  UnitTestImgSegmentorPredict();
  UnitTestImgSegmentorTrain();
  printf("UnitTestImgSegmentor OK\n");
}

void UnitTestAll() {
  //UnitTestImgKMeansClusters();
  UnitTestIntersectionOverUnion();
  UnitTestGBSimilarityCoefficient();
  UnitTestImgSegmentorRGB();
  UnitTestImgSegmentor();
}

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

5 Unit tests output

```
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=2 cell=1:
<190.271,188.622,189.519,255.874>
<57.922,71.614,92.852,255.544>
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=3 cell=1:
<197.903,195.060,194.940,255.852>
<46.857,55.700,72.989,255.384>
<129.141,141.318,156.154,255.440>
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=4 cell=1:
```

```
<49.314,59.658,46.134,255.156>
<156.342,159.087,163.036,255.568>
<56.903,76.562,152.418,255.000>
<201.616,198.516,198.111,255.828>
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=5 cell=1:
<42.357,54.043,156.886,255.000>
<47.936,59.604,46.270,255.149>
<119.585,133.399,145.312,255.076>
<177.630,176.173,177.662,255.664>
<206.329,203.216,202.496,255.772>
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=6 cell=1:
<210.086,207.070,206.155,255.687>
<188.060,185.241,185.757,255.701>
<90.991,116.830,139.485,255.000>
<46.868,57.760,44.244,255.109>
<37.108,37.526,155.019,255.000>
<153.019,156.372,160.882,255.265>
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=2 cell=3:
<196.476,194.722,195.635,255.874,194.379,192.612,193.523,255.874,192.848,191.093,192.012,255.874,191.376,189.680,190
<70.561,84.186,107.671,255.546,66.415,80.028,103.270,255.546,63.722,77.315,100.200,255.546,60.385,74.097,95.695,255.5
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=3 cell=3:
<142.267,153.344,167.998,255.445,138.511,149.808,164.556,255.445,135.723,147.234,162.003,255.445,132.033,143.971,158
<203.108,200.359,200.281,255.851,201.244,198.444,198.356,255.851,199.906,197.080,196.987,255.851,198.732,195.894,195
<58.046,67.067,87.513,255.383,54.053,62.941,83.006,255.383,51.554,60.343,79.943,255.383,48.753,57.583,75.451,255.383
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=4 cell=3:
<166.321,168.400,172.509,255.561,163.232,165.432,169.521,255.561,160.871,163.233,167.332,255.561,158.048,160.752,164
<70.162,89.174,164.837,255.000,65.785,84.909,161.050,255.000,63.053,82.167,158.272,255.000,59.666,79.014,154.477,255
<59.902,70.791,61.053,255.151,55.989,66.727,56.274,255.151,53.496,64.141,53.107,255.151,50.658,61.260,48.356,255.151
<206.331,203.363,203.001,255.829,204.541,201.518,201.148,255.829,203.264,200.202,199.827,255.829,202.160,199.067,198
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=5 cell=3:
<210.420,207.434,206.768,255.779,208.729,205.682,205.001,255.779,207.507,204.426,203.748,255.779,206.477,203.349,202
<50.209,62.203,167.101,255.000,46.743,58.512,163.959,255.000,44.835,56.378,161.714,255.000,43.170,54.670,158.697,255
<59.032,70.967,60.922,255.146,55.106,66.935,56.215,255.146,52.602,64.363,53.094,255.146,49.762,61.509,48.493,255.146
<183.893,182.408,184.051,255.650,181.493,180.014,181.653,255.650,179.745,178.301,179.954,255.650,178.076,176.728,178
<134.728,147.284,160.113,255.070,130.219,143.112,155.970,255.070,126.814,140.019,152.823,255.070,121.763,135.616,148
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=6 cell=3:
<162.431,165.080,170.039,255.241,159.193,161.986,166.967,255.241,156.704,159.677,164.713,255.241,153.771,157.139,162
<213.719,210.815,209.984,255.706,212.075,209.117,208.270,255.706,210.884,207.889,207.039,255.706,209.881,206.837,205
<107.498,132.241,155.378,255.000,102.122,127.262,150.460,255.000,98.306,123.685,146.780,255.000,92.555,118.440,141.3
<192.867,190.218,190.870,255.688,190.799,188.111,188.753,255.688,189.313,186.613,187.259,255.688,188.002,185.310,185
<56.646,67.997,57.533,255.094,52.988,64.197,53.064,255.094,50.691,61.814,50.148,255.094,48.147,59.221,45.884,255.094
<44.721,45.847,165.918,255.000,41.315,42.113,162.715,255.000,39.486,40.058,160.372,255.000,37.977,38.626,156.902,255
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=2 cell=5:
<199.430,197.609,198.528,255.874,197.521,195.689,196.598,255.874,196.238,194.395,195.310,255.874,195.197,193.361,194
<78.934,92.056,117.044,255.548,75.278,88.404,113.260,255.548,72.596,85.783,110.397,255.548,69.885,83.286,106.888,255
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=3 cell=5:
<149.809,159.522,173.907,255.455,146.579,156.494,170.976,255.455,144.233,154.351,168.862,255.455,141.879,152.364,166
<205.448,202.802,202.692,255.851,203.748,201.041,200.928,255.851,202.615,199.870,199.765,255.851,201.748,198.969,198
<64.703,73.812,96.796,255.376,61.085,70.068,92.790,255.376,58.466,67.395,89.768,255.376,55.886,64.901,85.866,255.376
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=4 cell=5:
<66.243,77.243,71.372,255.136,62.701,73.569,67.060,255.136,60.104,70.905,63.742,255.136,57.409,68.284,59.158,255.136
<171.552,172.993,177.353,255.555,168.834,170.341,174.682,255.555,166.914,168.507,172.824,255.555,165.089,166.896,171
<79.445,97.444,170.074,255.000,75.572,93.734,166.962,255.000,72.750,91.111,164.705,255.000,70.150,88.822,162.500,255
<208.325,205.503,205.109,255.830,206.690,203.810,203.413,255.830,205.603,202.684,202.297,255.830,204.781,201.817,201
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=5 cell=5:
<54.882,67.031,171.555,255.000,51.444,63.486,168.824,255.000,49.247,61.151,166.998,255.000,47.545,59.373,165.387,255
<65.832,77.871,71.395,255.135,62.236,74.186,67.096,255.135,59.606,71.515,63.810,255.135,56.881,68.903,59.281,255.135
<186.952,185.472,187.222,255.642,184.764,183.256,184.995,255.642,183.269,181.759,183.506,255.642,182.056,180.570,182
<212.159,209.285,208.596,255.782,210.618,207.695,206.991,255.782,209.598,206.636,205.930,255.782,208.832,205.822,205
<144.112,154.792,167.690,255.088,140.445,151.386,164.343,255.088,137.642,148.913,161.823,255.088,134.585,146.458,159
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=6 cell=5:
<122.059,128.593,115.779,255.000,116.812,123.342,109.980,255.000,112.450,119.063,105.202,255.000,107.648,114.591,99.7
```

```
<45.095,60.064,58.809,255.000,42.358,57.240,55.337,255.000,40.462,55.316,52.733,255.000,38.509,53.467,48.536,255.000
<189.671,187.725,189.036,255.656,187.571,185.589,186.895,255.656,186.150,184.154,185.472,255.656,185.020,183.034,184
<149.449,158.712,171.734,255.106,146.100,155.596,168.734,255.106,143.683,153.455,166.634,255.106,141.178,151.450,164
<213.027,210.185,209.453,255.766,211.498,208.611,207.863,255.766,210.489,207.561,206.810,255.766,209.732,206.750,206
<55.830,68.313,172.068,255.000,52.365,64.761,169.346,255.000,50.157,62.443,167.528,255.000,48.476,60.700,165.946,255
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=2 cell=7:
<201.644,199.780,200.696,255.874,199.768,197.907,198.810,255.874,198.567,196.687,197.604,255.874,197.648,195.749,196
<86.669,98.945,125.369,255.550,83.005,95.305,121.615,255.550,80.512,92.902,119.042,255.550,77.818,90.498,115.689,255
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=3 cell=7:
<70.399,79.334,104.925,255.367,66.730,75.600,100.916,255.367,64.299,73.130,98.145,255.367,61.599,70.615,94.264,255.3
<155.792,164.280,178.510,255.462,152.674,161.287,175.641,255.462,150.461,159.258,173.734,255.462,148.466,157.547,172
<207.120,204.555,204.411,255.851,205.434,202.831,202.683,255.851,204.380,201.735,201.589,255.851,203.590,200.898,200
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=4 cell=7:
<209.684,206.975,206.561,255.832,208.068,205.329,204.906,255.832,207.059,204.276,203.858,255.832,206.308,203.476,203
<175.623,176.540,181.128,255.547,172.974,173.908,178.490,255.547,171.167,172.146,176.765,255.547,169.635,170.721,175
<71.277,82.075,80.625,255.117,67.689,78.426,76.264,255.117,65.335,75.982,73.210,255.117,62.578,73.415,68.529,255.117
<88.550,105.210,174.067,255.000,84.657,101.467,171.016,255.000,81.922,99.052,168.997,255.000,79.268,96.808,167.102,2
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=5 cell=7:
<151.748,160.666,173.471,255.108,148.281,157.342,170.255,255.108,145.696,155.055,168.064,255.108,143.191,152.995,165
<213.373,210.577,209.861,255.785,211.852,209.036,208.300,255.785,210.911,208.055,207.310,255.785,210.211,207.312,206
<58.906,70.983,174.882,255.000,55.259,67.294,172.178,255.000,53.005,65.076,170.492,255.000,51.009,63.063,169.071,255
<189.199,187.717,189.638,255.635,187.027,185.516,187.421,255.635,185.613,184.072,186.000,255.635,184.516,182.977,184
<71.437,83.356,80.848,255.121,67.788,79.680,76.489,255.121,65.388,77.217,73.438,255.121,62.557,74.635,68.788,255.121
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=6 cell=7:
<192.077,190.127,191.461,255.650,190.003,188.017,189.334,255.650,188.666,186.644,187.982,255.650,187.649,185.621,186
<214.305,211.538,210.774,255.767,212.805,210.023,209.239,255.767,211.875,209.052,208.260,255.767,211.185,208.318,207
<50.330,65.733,68.878,255.000,47.258,62.676,65.124,255.000,45.328,60.684,62.470,255.000,43.098,58.644,57.877,255.000
<60.290,72.657,175.417,255.000,56.614,68.957,172.705,255.000,54.370,66.754,171.013,255.000,52.305,64.728,169.581,255
<156.385,164.236,177.140,255.125,153.175,161.150,174.186,255.125,150.828,159.029,172.235,255.125,148.754,157.250,170
<134.576,138.757,129.035,255.000,129.895,134.038,123.835,255.000,126.493,130.748,120.143,255.000,122.356,127.155,115
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=2 cell=9:
<204.186,202.308,203.226,255.874,202.340,200.463,201.361,255.874,201.189,199.274,200.215,255.874,200.360,198.417,199
<96.589,107.605,135.559,255.553,92.894,103.918,131.888,255.553,90.445,101.585,129.408,255.553,88.456,99.737,127.377,2
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=3 cell=9:
<77.469,85.914,114.780,255.351,73.747,82.101,110.815,255.351,71.312,79.677,108.105,255.351,69.283,77.740,105.846,255
<162.943,170.032,184.018,255.475,159.865,167.055,181.196,255.475,157.775,165.061,179.378,255.475,156.261,163.636,177
<209.078,206.613,206.405,255.851,207.409,204.911,204.693,255.851,206.393,203.858,203.655,255.851,205.648,203.079,202
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=4 cell=9:
<147.502,159.327,180.313,255.156,143.960,155.951,177.220,255.156,141.478,153.684,175.164,255.156,139.524,151.939,173
<189.994,188.873,191.417,255.619,187.746,186.586,189.115,255.619,186.277,185.067,187.668,255.619,185.234,184.026,186
<214.201,211.493,210.752,255.798,212.677,209.962,209.221,255.798,211.770,209.017,208.267,255.798,211.097,208.316,207
<70.930,79.658,109.649,255.284,67.233,75.870,105.680,255.284,64.904,73.528,102.992,255.284,62.997,71.674,100.758,255
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=5 cell=9:
<78.186,89.657,92.373,255.095,74.471,85.934,87.993,255.095,72.069,83.496,84.985,255.095,70.093,81.631,82.424,255.095
<160.507,167.461,180.162,255.139,157.249,164.245,177.074,255.139,154.969,162.121,175.029,255.139,153.198,160.502,173
<64.240,76.008,178.717,255.000,60.254,72.022,176.070,255.000,57.763,69.684,174.396,255.000,55.733,67.730,173.180,255
<214.799,212.092,211.320,255.788,213.303,210.584,209.803,255.788,212.408,209.652,208.854,255.788,211.744,208.954,208
<191.942,190.517,192.646,255.628,189.761,188.307,190.420,255.628,188.342,186.842,189.029,255.628,187.347,185.840,187
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=6 cell=9:
<195.210,193.269,194.511,255.643,193.119,191.142,192.381,255.643,191.810,189.786,191.066,255.643,190.875,188.839,190
<150.052,151.042,144.757,255.000,145.961,146.810,140.246,255.000,143.087,144.071,137.112,255.000,140.490,141.710,134
<66.323,78.661,179.429,255.000,62.320,74.644,176.744,255.000,59.838,72.335,175.041,255.000,57.765,70.332,173.817,255
<164.190,170.640,183.533,255.154,161.152,167.652,180.630,255.154,158.990,165.557,178.758,255.154,157.487,164.113,177
<215.807,213.117,212.300,255.767,214.345,211.653,210.820,255.767,213.467,210.737,209.886,255.767,212.815,210.051,209
<57.408,72.775,81.710,255.000,54.024,69.481,77.725,255.000,51.910,67.324,74.978,255.000,50.251,65.783,72.640,255.000
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=2 cell=11:
<103.291,113.311,142.588,255.556,99.645,109.599,138.936,255.556,97.172,107.278,136.491,255.556,95.249,105.496,134.58
<205.673,203.810,204.744,255.873,203.826,201.964,202.859,255.873,202.704,200.792,201.730,255.873,201.894,199.948,200
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=3 cell=11:
<82.710,90.576,122.267,255.343,78.897,86.649,118.287,255.343,76.480,84.258,115.593,255.343,74.579,82.451,113.457,255
<167.524,173.683,187.365,255.487,164.558,170.759,184.552,255.487,162.513,168.805,182.774,255.487,161.017,167.396,181
<210.315,207.923,207.651,255.850,208.647,206.214,205.928,255.850,207.643,205.181,204.904,255.850,206.908,204.402,204
```

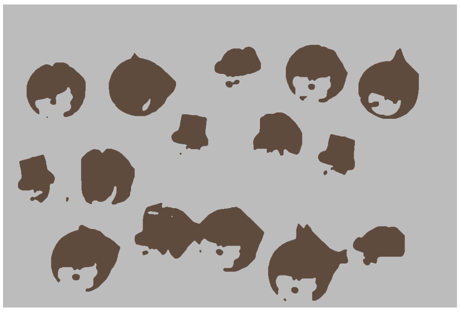
```
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=4 cell=11:
<75.708,83.925,117.235,255.274,71.888,79.979,113.237,255.274,69.513,77.609,110.540,255.274,67.708,75.866,108.409,255
<215.116,212.477,211.685,255.798,213.593,210.939,210.156,255.798,212.699,210.013,209.220,255.798,212.041,209.323,208
<154.117,164.413,184.340,255.180,150.868,161.176,181.285,255.180,148.426,158.990,179.299,255.180,146.483,157.259,177
<192.084,190.945,193.594,255.616,189.762,188.651,191.258,255.616,188.356,187.153,189.827,255.616,187.343,186.122,188
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=5 cell=11:
<83.249,94.097,101.231,255.079,79.441,90.245,96.803,255.079,77.043,87.870,93.792,255.079,75.162,86.073,91.375,255.079
<68.328,79.505,181.051,255.000,64.124,75.368,178.421,255.000,61.559,72.973,176.732,255.000,59.570,71.153,175.533,255
<165.729,171.586,184.133,255.161,162.804,168.527,181.095,255.161,160.560,166.425,179.120,255.161,158.830,164.833,177
<193.738,192.354,194.603,255.624,191.464,190.109,192.329,255.624,190.082,188.651,190.943,255.624,189.102,187.653,189
<215.627,212.985,212.171,255.789,214.132,211.474,210.659,255.789,213.254,210.560,209.729,255.789,212.604,209.874,209
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=6 cell=11:
<196.962,195.091,196.314,255.637,194.823,192.955,194.140,255.637,193.509,191.580,192.825,255.637,192.583,190.634,191
<216.604,213.962,213.103,255.769,215.146,212.500,211.632,255.769,214.297,211.606,210.719,255.769,213.659,210.932,210
<70.341,82.140,181.645,255.000,66.122,78.006,178.974,255.000,63.550,75.585,177.268,255.000,61.513,73.710,176.053,255
<168.407,174.211,187.354,255.172,165.483,171.246,184.447,255.172,163.417,169.211,182.599,255.172,161.927,167.772,181
<62.493,77.283,91.390,255.000,58.906,73.770,87.228,255.000,56.700,71.607,84.403,255.000,55.019,70.037,82.114,255.000
<158.205,157.647,153.292,255.000,154.523,153.677,149.119,255.000,151.883,151.184,146.267,255.000,149.633,149.100,143
UnitTestImgKMeansClusters OK
UnitTestIntersectionOverUnion OK
```

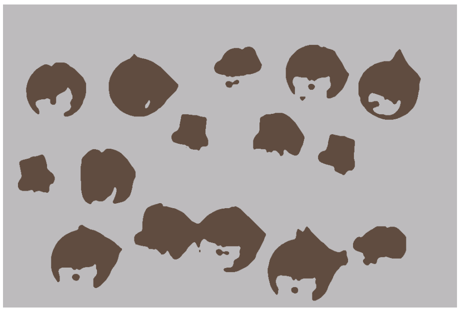
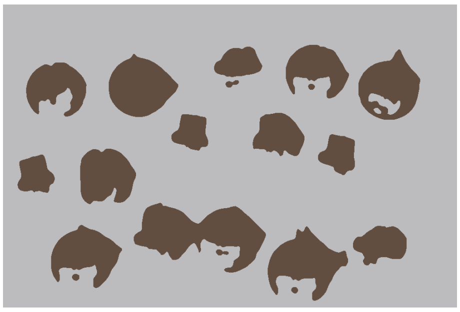
5.1 K-Means clustering on RGBA space

imgkmeanscluster.tga:

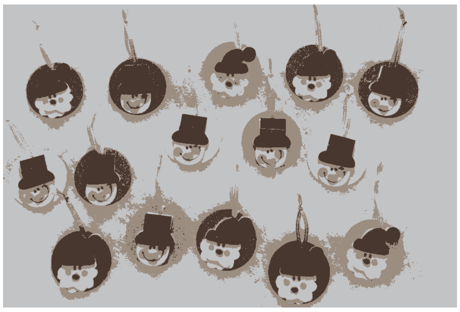
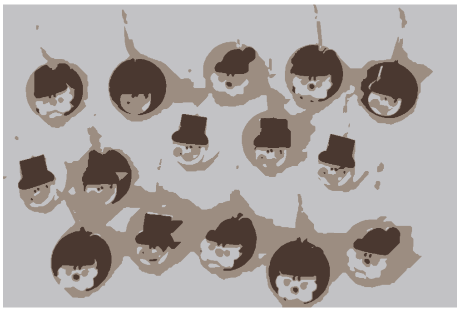
clustering for K equals 2 to 6 and radius equals 0 to 5:
K=2:

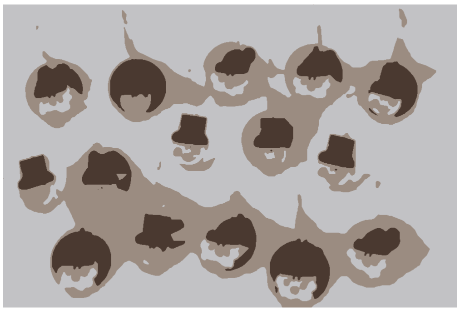


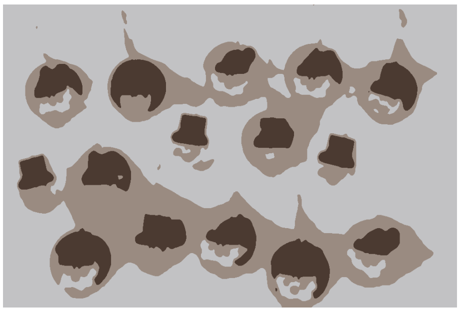
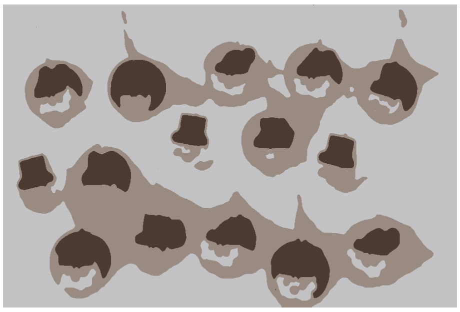


K=3:

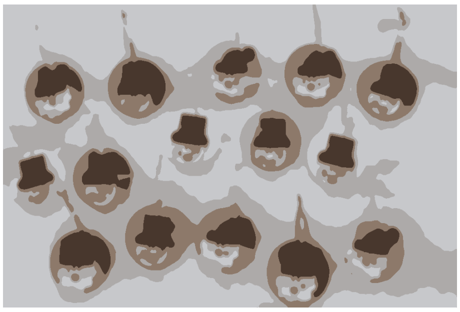






K=4:



K=5:




K=6:

imgkmeanscluster.txt:


```
{
  "_size":"5",
  "_clusters":{
    "_seed":"1",
    "_centers":[
      {
        "_dim":"388",
```

```
      "_val":["196.962387","195.091156","196.313553","255.637268","194.823151","192.954971","194.140289","255.63726
    },
    {
      "_dim":"388",
      "_val":["216.603745","213.961731","213.103195","255.768936","215.145920","212.499786","211.631653","255.7689
    },
    {
      "_dim":"388",
      "_val":["70.341324","82.139534","181.645172","255.000000","66.121559","78.005623","178.973862","255.000000","
    },
    {
      "_dim":"388",
      "_val":["168.406982","174.211227","187.353622","255.172226","165.482742","171.246460","184.447128","255.1722
    },
    {
      "_dim":"388",
      "_val":["62.493431","77.282890","91.389877","255.000000","58.905697","73.769623","87.228317","255.000000","5
    },
    {
      "_dim":"388",
      "_val":["158.205353","157.647125","153.291733","255.000000","154.523422","153.677002","149.119064","255.0000
    }
  ]
 }
}
```