9.2.10 o2clust

Application purpose

o2clust performs cluster analysis on a set of data. It uses an adaptive algorithm outlined in section 3.2 tailored to the second-order distance metric described in section 2.2.

Application notes

Although the correlation-matrix cluster representation implemented in the **corrmatrix** module is complete, the algorithm does not work at present. The problem seems to lie with forming clusters that have too few points, and therefore are unnaturally biased along an arbitrary axis. When forming clusters by adding points (as opposed to splitting larger clusters into progressively smaller clusters) the clusters grow from a few points to encompass (hopefully) a natural cluster. However, when a cluster contains one or two points, the correlation matrix is either singular or very close, and the resulting "shape" of the cluster is merely the axis through the two points. This severe skew persists until a greater number of points are used to form the matrix.

The preliminary solution was to form a "dummy" cluster around a cluster with few points. This dummy cluster would be used to calculate the correlation matrix until a certain threshold number of points existed within the cluster. The dummy cluster was spherical around the centroid of the real cluster. However, this did not solve the problem.

The second approach was to "steal" the nearest *n* points when creating a new cluster. This caused unnaturally shaped clusters.

Lipson and Siegelmann [1998] use the correlation matrix representation to form the receptive field of a neuron. The matrix is incrementally updated (via a "learning rate"). The receptive fields are initialised to an area before learning begins. No new "clusters" or neurons are generated. One approach with o2clust would be to likewise segment the data space to initialise the clusters. This would require some new method to create a new cluster, however.

Another possible method would be to incrementally adjust the correlation matrices via a learning rate so they did not "contain" the points but merely represented a "best guess" of a cluster. The data set could then be classified on the termination of the algorithm.

Usage

```
*** o2clust -- performs adaptive O2 clustering on a set of vectors
    Version 0.32.cp build 22:20:54 Oct 10 2000 [IRIX64 6.5 IP25]
Usage: o2clust {vector file} {cluster file} {classify file} {stop param}
Where: {vector file}   -- file to take vectors from
                          [specify '-' for stdin]
       {cluster file}  -- file to write clusters to
                          [specify '-' for stdout]
       {classify file} -- file to write classified vectors (from vector file) to
                          [specify '-' to skip this step]
       {stop param}    -- stop clustering when RMS error changes by less than
                          this factor.  i.e. 0.1 => 10%%, 0.25 => 25%%, etc.
```

Modules used

StdDefs, 2DArray, TokenLst, TokScan, Gauss, RunAvg, Vector_Utils, Vector_Read, CorrMatrix, Cluster

Source code

Makefile

```
TARGET=o2clust
VERSION=032cp
CFILES=${TARGET}.c cluster.c vector_utils.c vector_read.c corr_matrix.c tokscan.c tokenlst.c
gauss2.c 2darray.c runavg.c
HFILES=stddefs.h cluster.h vector_utils.h vector_read.h corr_matrix.h tokscan.h tokenlst.h gauss.h
runavg.h 2darray.h
OBJFILES=${TARGET}.o cluster.o vector_utils.o vector_read.o corr_matrix.o tokscan.o tokenlst.o
gauss2.o nrutil.o runavg.o 2darray.c
DISTFILES=Makefile o2clust.help.make ${CFILES} ${HFILES} ${TARGET}
SYSVER=\"`uname -mprs`\"

CC=cc
CFLAGS=-64 -pca -DPLATFORM="${SYSVER}" -DUNIX #-DDEBUG
LFLAGS=-64 -pca -lm -lmalloc

${TARGET}: ${OBJFILES}
   ${CC} -o ${TARGET} ${OBJFILES} ${LFLAGS}

help: ${TARGET}
   ${TARGET} --help

clean:
   rm -f *.o ${TARGET}

refresh: clean ${TARGET}

dist: ${DISTFILES}
   tar cvf ${TARGET}_v${VERSION}.tar ${DISTFILES}
   gzip -f ${TARGET}_v${VERSION}.tar

.c.o:
   ${CC} ${CFLAGS} -c $*.c
```

o2clust.help.make

```
/* Help make file for o2clust
 *
 * Author: Dylan Muir
 * Date: 3rd July, 2000
 * Modified:
 * Version: 0.15.cp
 */

#define p(x)   fprintf(helpFile, x);

p("\n");
p("Put your help info here\n");
p("\n");
p("\n");
p("blah blah blah\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");

#undef p

/* --- END of generic.help.make --- */
```

o2clust.c

```
/* o2clust -- performs order-2 (ellipsoid) clustering on a set of vectors
 *            Uses a modified version of the adaptive Forgy's algorithm
 *
 * Author: Dylan Muir(dr.muir@student.qut.edu.au)
 *         QUT MLRC Semester 2, November 1999
 * Date: 8th November, 1999
 * Modified: 10th October, 2000
 * Version: 0.32.cp (cross-platform Win32 and Unix)
 *
```

```
 * Usage: o2clust (vector file) (clusters file) (classified vectors file) (clustering parameter)
 *
 * Modules used: stddefs, vector_utils, vector_read, cluster, corr_matrix, 2darray, gauss2,
 *               runavg
 *
 *  Aknowledgements:  Lipson,  H.,  Siegelman,  H.T.,  "High  Order  Shape  Neurons  for  Data  Structure
Decomposition"
 *
 */

/* -- Compilation check -- */

#if !defined(UNIX) && !defined(WIN32)
   #error "Compilation for Unix or Win32 ONLY!"
#endif


/* -- Required modules -- */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#ifdef   UNIX
   #include <unistd.h>
#endif
#ifdef   WIN32
   #include <string.h>
   #define  srand48(x)  srand((unsigned int) (x))
   #define  strcasecmp  stricmp
   #define  PLATFORM    "Generic Win32 System"
#include <conio.h>
#include <ctype.h>
#endif
#include <math.h>
#include "stddefs.h"
#include "cluster.h"
#include "vector_utils.h"
#include "vector_read.h"
#include "corr_matrix.h"
#include "runavg.h"


/* -- defines -- */

#define  APP_NAME       0
#define  HELP           1
#define  VECTOR_FILE    1
#define  CLUSTER_FILE   2
#define  CLASSIFY_FILE  3
#define  CLUST_PARAM    4
#define  NUMARGS        5

#define  MAX_CLUSTERS         3000
#define  INIT_STOP_PARAM      0.001
#define  SAMPLES_TO_AVG       5

#define  VERSION        "0.32.cp"
#define  HELP_FILE      "o2clust.help"

#define  Abort(x) AbortPrint((x), __LINE__, __FILE__);


/* -- codebook stuff -- */

typedef  corrMatrix  *codebook;

codebook CodebookNew(int slots);
codebook CodebookDestroy(codebook book, int slots);
codebook CodebookNewFromCluster(cluster *clust);
codebook CodebookMakeCopy(codebook book, int slots);
void WriteCodebook(FILE *output, codebook book, int slots);
codebook InitialiseCodebook(int numClusters, cluster *vectorSet);
int FreeCluster(codebook book, int numSlots);
void WriteCodebook(FILE *output, codebook book, int slots);
double GetError(codebook book, int numClusters);

/* -- kmeans helper function prototypes -- */

void Usage(FILE *output, char *appName);
void Help(FILE *output, char *appName);
void AbortPrint(int code, int line, char *file);
void ProcessCommandLine(int argc, char *argv[],
```

```
        FILE **vectFile, FILE **clustFile, FILE **classFile,
        double *clustParam);
void CloseFiles(FILE *vectFile, FILE *clustFile, FILE *classFile);
void PrintBanner(char *argv[], FILE *vectFile, FILE *clustFile, FILE *classFile, double clustParam);
cluster *ReadTrainingSet(FILE *vectorFile, int *numTrainVectors);
vector **VectArrayNewFromCluster(cluster *clust);
int FindNearestNeighbour(vector *vect, codebook currentBook, int numClusters, bool useCorr, bool
ignore[]);
int FindWorstVector(vector **vectorSet, int numVectors, int clusterMembership[], codebook
currentBook, bool ignore[]);
int FindWorstVectorInCluster(vector **vectorSet, int numVectors, int clusterMembership[], codebook
currentBook, int clusterSlot, bool useCorr);
double CMatrixAvCorrWithinCluster(corrMatrix matrix);
bool UseCorrelation(codebook currentBook, int numClusters);
void ClassifyVectors(vector **vectorSet, int numTrainVectors, int clusterMembership[], codebook
currentBook, int lastCluster, bool useCorr, double stopParam, bool ignore[]);
double AverageCodebookCorrelation(vector *vect, codebook currentBook, int numClusters, double
*stddev);


/* -- main -- */

int main(int argc, char *argv[])
{
   FILE     *vectFile, *clustFile, *classFile;
   double   clustParam,
            lastError, thisError;
   int      lastCluster, numTrainVectors,
            *clusterMembership, slot;
   bool     firstRun, terminate;

   cluster  *vectorSetCluster;
   codebook currentBook;
   vector   **vectorSet;

   double   worstCorr,
            bestCorr,
            testCorr,
            bestDist,
            testDist,
            avCorr,
            stddevCorr;
   int      closest,
            oldClosest,
            nextClosest,
            worst,
            lastNewCluster,
            bestFromClust,
            bestToClust,
            fromClust,
            toClust,
            numClusters;

   bool     *ignore,
            firstPass,
            tracking,
            report;

   ProcessCommandLine(  argc, argv,
                        &vectFile, &clustFile, &classFile,
                        &clustParam);

   PrintBanner(argv, vectFile, clustFile, classFile, clustParam);
   vectorSetCluster = ReadTrainingSet(vectFile, &numTrainVectors);
   vectorSet = VectArrayNewFromCluster(vectorSetCluster);

   if (!(clusterMembership = (int *) malloc(numTrainVectors * sizeof(int)))) {
      fprintf(stderr, "*** Could not allocate cluster membership array.\n");
      Abort(24);
   }

   for (slot = 1; slot < numTrainVectors; slot++)
      clusterMembership[slot] = -1;

   clusterMembership[0] = 0;
   /* clusterMembership[1] = 1; */

   currentBook = InitialiseCodebook(numTrainVectors, vectorSetCluster);
   lastCluster = 1;

   if (!(ignore = malloc(numTrainVectors * sizeof(bool)))) {
      fprintf(stderr, "*** Could not allocate ignore array.\n");
```

```
      Abort(23);
   }

   for (slot = 0; slot < numTrainVectors; slot++)
      ignore[slot] = FALSE;

   lastError = GetError(currentBook, lastCluster + 1);
   printf("Initial (no classification) error: %f\n", lastError);

   if (numTrainVectors > 1000)
      report = TRUE;
   else
      report = FALSE;

   tracking = FALSE;
   terminate = FALSE;
   firstRun = FALSE;
   numClusters = 1;
   while (!terminate) {

      slot = 0;
      while (slot < numTrainVectors) {
         closest = FindNearestNeighbour(vectorSet[slot], currentBook, lastCluster + 1, TRUE, NULL);

         if (clusterMembership[slot] == -1) {     /* ie not clustered */
            avCorr  =  AverageCodebookCorrelation(vectorSet[slot],  currentBook,  lastCluster  +  1,
NULL);
            bestCorr = CMatrixCorrelationWithVect(currentBook[closest], vectorSet[slot]);
            //avCorr = CMatrixAvCorrWithinCluster(currentBook[closest]);
            stddevCorr = CMatrixStdDevCorrWithinCluster(currentBook[closest]);

            if (bestCorr > avCorr) {
            //if (fabs(bestCorr - avCorr) > (stddevCorr * clustParam)) {
               /* Make a new cluster with the current vector */
               numClusters++;
               closest = FreeCluster(currentBook, MAX_CLUSTERS);
               if (closest == -1) {
                  fprintf(stderr, "*** Exceeded compiler limit of clusters [%d].\n", MAX_CLUSTERS);
                  Abort(22);
               }
               if (closest > lastCluster)
                  lastCluster = closest;
            }

         } else {
            if (closest == clusterMembership[slot]) {      /* Not going to move */
               ignore[closest] = TRUE;
               nextClosest  =  FindNearestNeighbour(vectorSet[slot],  currentBook,  lastCluster  +  1,
TRUE, ignore);
               if (nextClosest != -1) {
                  testCorr = CMatrixNormedCorrWithVect(currentBook[nextClosest], vectorSet[slot]);
                  testDist                    =                    VectorEuclideanDist(vectorSet[slot],
CMatrixCentroid(currentBook[nextClosest]));
               } else {
                  testCorr = 9000.0;
                  testDist = 9000.0;
               }
               ignore[closest] = FALSE;

               bestCorr = CMatrixCorrelationWithVect(currentBook[closest], vectorSet[slot]);
               bestDist                    =                    VectorEuclideanDist(vectorSet[slot],
CMatrixCentroid(currentBook[closest]));

               //avCorr = AverageCodebookCorrelation(vectorSet[slot], currentBook, lastCluster + 1,
&stddevCorr);
               avCorr = CMatrixAvCorrWithinCluster(currentBook[closest]);
               stddevCorr = CMatrixStdDevCorrWithinCluster(currentBook[closest]);

               if ((currentBook[closest] -> num_vectors > 10) && ((bestCorr - avCorr) > (stddevCorr
* clustParam))) {
                  if    ((testCorr    -    CMatrixAvCorrWithinCluster(currentBook[nextClosest]))    >
(CMatrixStdDevCorrWithinCluster(currentBook[nextClosest]) * clustParam)) {
                     /* Make a new cluster with the current vector */
                     numClusters++;
                     closest = FreeCluster(currentBook, MAX_CLUSTERS);
                     if (closest == -1) {
                        fprintf(stderr,   "***   Exceeded   compiler   limit   of   clusters   [%d].\n",
MAX_CLUSTERS);
                        Abort(22);
                     }
                     if (closest > lastCluster)
                        lastCluster = closest;
```

```
                } else {
                   closest = nextClosest;
                }

             }
          }
       }

       oldClosest = clusterMembership[slot];

       if (oldClosest != closest) {
          /* Update the previous cluster's centroid after removing the vector */
          if (oldClosest != -1) {
             CMatrixDeleteVector(currentBook[oldClosest], vectorSet[slot]);
             if (currentBook[oldClosest] -> num_vectors == 0) {
                currentBook[oldClosest] = CMatrixDestroy(currentBook[oldClosest]);
                numClusters--;
             }
          }
             /* Update the current cluster's centroid after adding the vector */
          if (currentBook[closest] == NULL) {
             currentBook[closest] = CMatrixCreate(vectorSet[slot] -> dimensions);
          }
          CMatrixAddVector(currentBook[closest], VectorMakeCopy(vectorSet[slot]));
          clusterMembership[slot] = closest;
       }

       slot++;     /* Next training vector */
       if (report && (slot % (int) (numTrainVectors / 10) == 0))
          printf("|");

    }

    thisError = GetError(currentBook, lastCluster + 1);

    if (!firstRun && fabs(thisError - lastError) < INIT_STOP_PARAM) //(fabs(lastError - thisError)
/ lastError) > clustParam) /* Percentage change */
          terminate = TRUE;

    fprintf(stderr, "RMS error: %f (%5.2f%% change) [%d clusters]\n", thisError, (fabs(lastError -
thisError) / lastError) * 100.0, numClusters);
    lastError = thisError;
    firstRun = FALSE;
 }

 //ClassifyVectors(vectorSet, numTrainVectors, clusterMembership, currentBook, lastCluster, TRUE,
INIT_STOP_PARAM, NULL);

 {
    int numFinalClusters = 0;

    printf("clusters: ");
    for (slot = 0; slot <= lastCluster; slot++) {
       if (currentBook[slot]) {
          printf("%d           ",           currentBook[slot]           ->           num_vectors,
CMatrixStdDevCorrWithinCluster(currentBook[slot])/CMatrixAvCorrWithinCluster(currentBook[slot]));
          numFinalClusters++;
       }
    }
    printf("\n");
    printf("%d clusters (final), RMS err: %f\n", numFinalClusters, thisError);
 }

 /* Get rid of small clusters */

 terminate = FALSE;
 {
    double sum;

    sum = 0.0;
    for (slot = 0; slot <= lastCluster; slot++) {
       if (currentBook[slot] != NULL) {
          sum += (double) currentBook[slot] -> num_vectors;
       }
    }
    sum /= (double) numClusters;

    printf("Culling < %d\n", (int) (sum));

    while (!terminate) {
       terminate = TRUE;
       for (slot = 0; slot <= lastCluster; slot++) {
```

```
            if (currentBook[slot] != NULL) {
               if (currentBook[slot] -> num_vectors < (int) (sum))
                  ignore[slot] = TRUE;
            } else
               ignore[slot] = FALSE;
         }
         ClassifyVectors(vectorSet,  numTrainVectors,  clusterMembership,  currentBook,  lastCluster,
TRUE, INIT_STOP_PARAM, ignore);

         for (slot = 0; slot <= lastCluster; slot++) {
            if (currentBook[slot] != NULL) {
               if (currentBook[slot] -> num_vectors <  sum)
                  ;//terminate = FALSE;           /* Need another pass to cull */
            }
         }

      }
   }


   WriteCodebook(clustFile, currentBook, numTrainVectors);

   {
      int numFinalClusters = 0;

      fprintf(stderr, "clusters: ");
      for (slot = 0; slot <= lastCluster; slot++) {
         if (currentBook[slot]) {
            fprintf(stderr, "%d ", currentBook[slot] -> num_vectors);
            numFinalClusters++;
         }
      }
      fprintf(stderr, "\n");
      fprintf(stderr, "%d clusters (final), RMS err: %f\n", numFinalClusters, thisError);
   }


   /* If we are to write out the classifications, do so here */

   if (classFile != NULL) {
      printf("Writing classification.\n");
      for (slot = 0; slot < numTrainVectors; slot++) {
         fprintf(classFile, "%d\t", clusterMembership[slot]);
         VectorWrite(classFile, vectorSet[slot]);
      }
   }


   /* We need to destroy everything here! */

   CodebookDestroy(currentBook, lastCluster + 1);

   CloseFiles(vectFile, clustFile, classFile);

   return 0;
}


/* -- helper functions -- */

void Usage(FILE *output, char *appName)
{
   fprintf(output, "\n*** o2clust -- performs adaptive O2 clustering on a set of vectors\n");
   fprintf(output, "    Version %s build %s %s [%s]\n", VERSION, __TIME__, __DATE__, PLATFORM);
   fprintf(output,  "Usage:  %s  {vector  file}  {cluster  file}  {classify  file}  {stop  param}\n",
appName);
   fprintf(output, "Where: {vector file}   -- file to take vectors from\n");
   fprintf(output, "                          [specify '-' for stdin]\n");
   fprintf(output, "       {cluster file}  -- file to write clusters to\n");
   fprintf(output, "                          [specify '-' for stdout]\n");
   fprintf(output, "        {classify file} -- file to write classified vectors (from vector file)
to\n");
   fprintf(output, "                          [specify '-' to skip this step]\n");
   fprintf(output,  "         {stop param}      -- stop clustering when RMS error changes by less
than\n");
   fprintf(output,  "                               this factor.  i.e. 0.1 => 10%%, 0.25 => 25%%,
etc.\n");
   fprintf(output, "For detailed info, type %s --help\n", appName);
   fprintf(output, "\n");
}

void AbortPrint(int code, int line, char *file)
```

```
{
   fprintf(stderr, "Aborting [%s | %d]...\n", file, line);
   exit(code);
}

void ProcessCommandLine(int argc, char *argv[], FILE **vectFile, FILE **clustFile, FILE **classFile,
double *clustParam)
{
   if ((argc > 1) && (strcasecmp(argv[HELP], "--help") == 0)) {
      FILE  *helpFile;

      if (!(helpFile = fopen(HELP_FILE, "wt"))) {
         fprintf(stderr, "*** Could not create help file [%s].\n", HELP_FILE);
         Abort(1);
      }

      Help(helpFile, argv[APP_NAME]);
      printf("Created %s\n", HELP_FILE);
      exit(0);
   }

   if (argc < NUMARGS) {
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Invalid number of arguments.\n");
      Abort(2);
   }

   if (argc > NUMARGS)
      fprintf(stderr, "--- Warning: extra arguments ignored.\n");

   if (strcmp(argv[VECTOR_FILE], "-") == 0)
      *vectFile = stdin;
   else if (!(*vectFile = fopen(argv[VECTOR_FILE], "rt"))) {
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Could not open vector file [%s] for reading.\n", argv[VECTOR_FILE]);
      Abort(3);
   }

   if (strcmp(argv[CLUSTER_FILE], "-") == 0)
      *clustFile = stdout;
   else if (!(*clustFile = fopen(argv[CLUSTER_FILE], "wb"))) {
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Could not open cluster file [%s] for writing.\n", argv[CLUSTER_FILE]);
      Abort(4);
   }

   if (strcmp(argv[CLASSIFY_FILE], "-") == 0)
      *classFile = NULL;
   else if (!(*classFile = fopen(argv[CLASSIFY_FILE], "wb"))) {
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr,   "***   Could   not   open   classification   file   [%s]   for   writing.\n",
argv[CLASSIFY_FILE]);
      Abort(4);
   }

   if (!isdigit(*argv[CLUST_PARAM])) {
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr,   "***   Error   converting   RMS   error   stop   parameter   from   [%s].\n",
argv[CLUST_PARAM]);
      Abort(5);
   }

   *clustParam = atof(argv[CLUST_PARAM]);
}

void CloseFiles(FILE *vectFile, FILE *clustFile, FILE *classFile)
{
   if (vectFile != stdin) fclose(vectFile);
   if (clustFile != stdout) fclose(clustFile);
   if (classFile != NULL) fclose(classFile);
}


void PrintBanner(char *argv[], FILE *vectFile, FILE *clustFile, FILE *classFile, double clustParam)
{
   fprintf(stderr, "- o2-clustering ver %s\n", VERSION);
   fprintf(stderr, " o Vector Input: << [");
   vectFile == stdin ?
      fprintf(stderr, "- stdin]\n") :
      fprintf(stderr, "%s]\n", argv[VECTOR_FILE]);
   fprintf(stderr, " o Clusters output: >> [");
   clustFile == stdout ?
```

```
      fprintf(stderr, "- stdout]\n") :
      fprintf(stderr, "%s]\n", argv[CLUSTER_FILE]);
   fprintf(stderr, " o Classification output: ");
   classFile == NULL ?
      fprintf(stderr, "skipped\n") :
      fprintf(stderr, ">> [%s]\n", argv[CLASSIFY_FILE]);
   fprintf(stderr, " o RMS error stop parameter: [%5.3f] (%3.2f%%)\n", clustParam, clustParam *
100.0);
}

void Help(FILE *helpFile, char *appName)
{
   fprintf(helpFile, "-------------------------------------\n");
   fprintf(helpFile, "   Help file for progname ver %s\n", VERSION);
   fprintf(helpFile, "-------------------------------------\n\n");

   Usage(helpFile, appName);

#include "o2clust.help.make"
}


/* -- Codebok stuff -- */

codebook CodebookNew(int slots)
{
   codebook newBook;

   if (!(newBook = (codebook) malloc(slots * sizeof(corrMatrix))))
      return NULL;

   return newBook;
}

codebook CodebookDestroy(codebook book, int slots)
{
   int     slot;

   slot = 0;
   while (slot < slots) {
      CMatrixDestroy(book[slot]);
      slot++;
   }

   free(book);
   return NULL;
}

codebook CodebookNewFromCluster(cluster *clust)
{
   codebook newBook;               /* new codebook */
   cluster  *cIndex;               /* index into cluster */
   int      slot;                  /* slot index into codebook */

   if (!(newBook = CodebookNew(ClusterSize(clust))))
      return NULL;

   slot = 0;
   cIndex = clust;
   while (cIndex != NULL) {
      newBook[slot] = CMatrixCreate(cIndex -> vect -> dimensions);
      CMatrixAddVector(newBook[slot], cIndex -> vect);
      cIndex = cIndex -> next;
      slot++;
   }

   return newBook;
}

codebook CodebookMakeCopy(codebook book, int slots)
{
   codebook newBook;
   int      slot;                  /* index into codebook slots */

   if (!(newBook = CodebookNew(slots)))
      return NULL;

   slot = 0;
   while (slot < slots) {
      newBook[slot] = CMatrixMakeCopy(book[slot]);
      slot++;
   }
```

```
   return newBook;
}


codebook InitialiseCodebook(int numClusters, cluster *vectorSet)
{
   codebook newBook;
   int      slot,
            numDimensions;
   cluster  *cIndex;

   if (!(newBook = CodebookNew(numClusters))) {
      fprintf(stderr, "*** Could not initialise codebok.\n");
      Abort(20);
   }

   newBook[0] = CMatrixCreate(vectorSet -> vect -> dimensions);
   CMatrixAddVector(newBook[0], vectorSet -> vect);

   for (slot = 1; slot < numClusters; slot++) {
      newBook[slot] = NULL;
   }
   return newBook;
}


cluster *ReadTrainingSet(FILE *vectorFile, int *numTrainVectors)
{
   cluster *vectorSet;

   fprintf(stderr, "Reading training vectors...\n");
   vectorSet = ClusterReadFromFile(vectorFile);
   if (ClusterIsEmpty(vectorSet)) {
      fprintf(stderr, "*** Empty training set.\n");
      Abort(10);
   }

   fprintf(stderr, "Read %d vectors.\n", ClusterSize(vectorSet));

   if  (!ClusterCheckDimensionality(vectorSet))  {            /*  make  sure  they're  all  the  same
dimensionality */
      fprintf(stderr, "*** The training vectors must all be the same dimensionality.\n");
      Abort(7);
   }
   fprintf(stderr, "Training dimension: %d\n", vectorSet -> vect -> dimensions);

   *numTrainVectors = ClusterSize(vectorSet);
   return vectorSet;
}


int  FindNearestNeighbour(vector  *vect,  codebook  currentBook,  int  numClusters,  bool  useCorr,  bool
ignore[])
{
   int      slot;          /* index into codebook */
   double   minCorr,       /* current maximum correlation (lowest absolute val) */
            corr;          /* current correlation */
   int      minSlot;       /* current slot of maximum correlation */

   minSlot = -1;
   slot = 0;
   while (slot < numClusters) {
      if (currentBook[slot] && ((ignore == NULL) || (!ignore[slot]))) {
         if (useCorr)
            corr = CMatrixNormedCorrWithVect(currentBook[slot], vect);
         else
            corr = VectorEuclideanDist(CMatrixCentroid(currentBook[slot]), vect);

         if (((minSlot == -1) || (corr < minCorr))) {
            minCorr = corr;
            minSlot = slot;
         }
      }
      slot++;
   }
   return minSlot;
}


int FreeCluster(codebook book, int numSlots)
{
```

```
   int   slot;

   for (slot = 0; slot < numSlots; slot++)
      if (book[slot] == NULL)
         return slot;

   return -1;
}

void WriteCodebook(FILE *output, codebook book, int slots)
{
   int     slot;

   for (slot = 0; slot < slots; slot++) {
      if (book[slot] != NULL) {
         CMatrixWrite(output, book[slot]);
      }
   }
}


vector **VectArrayNewFromCluster(cluster *clust)
{
   vector   **newBook;              /* new codebook */
   cluster  *cIndex;                /* index into cluster */
   int      slot;                   /* slot index into codebook */

   if (!(newBook = (vector **) malloc(ClusterSize(clust) * sizeof(vector *))))
      return NULL;

   slot = 0;
   cIndex = clust;
   while (cIndex != NULL) {
      newBook[slot] = VectorMakeCopy(cIndex -> vect);
      cIndex = cIndex -> next;
      slot++;
   }

   return newBook;
}

double GetError(codebook book, int numClusters)
{
   double  bookError;
   int     slot;

   bookError = 0.0;
   slot = 0;
   while (slot < numClusters) {
      if (book[slot]) bookError += CMatrixRMSCorrelation(book[slot]);
      slot++;
   }

   bookError /= (double) numClusters;
   return sqrt(bookError);
}

double  AverageCodebookCorrelation(vector  *vect,  codebook  currentBook,  int  numClusters,  double
*stddev)
{
   double   corr, sum, sumSqr;
   int      slot, numReal;

   sum = 0.0;
   sumSqr = 0.0;
   numReal = 0;
   for (slot = 0; slot < numClusters; slot++) {
      if (currentBook[slot] != NULL) {
         corr = CMatrixNormedCorrWithVect(currentBook[slot], vect);
         sum += corr;
         sumSqr += corr * corr;
         numReal++;
      }
   }

   if (stddev != NULL) {
      *stddev = sqrt(((double) numReal * sumSqr - (sum * sum)) / (double) (numReal * (numReal -
1)));
   }
   return (sum / (double) numReal);
}
```

```
void ClassifyVectors(vector **vectorSet, int numTrainVectors, int clusterMembership[], codebook
currentBook, int lastCluster, bool useCorr, double stopParam, bool ignore[])
{
   bool     classify;         /* Do we continue classifying? */
   runAvg   *avg;             /* Running average of RMS error */
   int      slot,             /* Vector to look at */
            closest,          /* Closest cluster to  current vector */
            oldClosest,       /* Cluster vector was last in */
            tries;            /* Number of times we've tried to classify them all */
   double   closestCorr,      /* Distance to neasrest cluster */
            RMSerror,         /* current RMS error */
            lastError;

   lastError = GetError(currentBook, lastCluster + 1);
   avg = InitRunningAverage(SAMPLES_TO_AVG);
   tries = 0;
   classify = TRUE;
   while (classify) {
      slot = 0;
      while (slot < numTrainVectors) {
         closest = FindNearestNeighbour(vectorSet[slot], currentBook, lastCluster + 1, useCorr,
ignore);
         closestCorr = VectorEuclideanDist(currentBook[closest] -> centroid, vectorSet[slot]);
         oldClosest = clusterMembership[slot];

         if (oldClosest != closest) {
            if (oldClosest != -1) {
               CMatrixDeleteVector(currentBook[oldClosest], vectorSet[slot]);
               if (CMatrixIsEmpty(currentBook[oldClosest])) {
                  CMatrixDestroy(currentBook[oldClosest]);
                  currentBook[oldClosest] = NULL;
               }
            }

            if (!currentBook[closest])
               currentBook[closest] = CMatrixCreate(vectorSet[slot] -> dimensions);

            CMatrixAddVector(currentBook[closest], vectorSet[slot]);
            clusterMembership[slot] = closest;
         }
         slot++;
      }

      RMSerror = GetError(currentBook, lastCluster + 1);
      if (fabs(RMSerror - lastError) < stopParam)
         classify = FALSE;

      tries++;
      if (tries % 10 == 0) {
         printf(".");
         stopParam *= 2;
      }
   }
}


/* --- END of o2clust.c --- */
```