9.2.9 sclust

Application purpose

sclust performs adaptive spherical cluster analysis on a data set. The data can be of any dimensionality. sclust uses a modified adaptive Forgy's algorithm, and is deterministic (i.e. the analysis only needs to be performed once, and will always return the best result for the algorithm used). See section 1.0 for a description of cluster analysis and section 3.1 for a description of the adaptive algorithm.

Usage

```
*** sclust -- Performs adaptive clustering on a set of vectors
    Version 0.17 build build 17:01:17 Jun 30 2000
Usage: sclust {vector file} {means file} {av. dist. parm}
Where: {vector file}    -- file of vectors to analyse, one per line
                           ['-' for stdin]
       {means file}     -- filename to output the mean vectors to
                           ['-' for stdout]
       {av. dist. parm} -- see help file for information
                           (0.5 is a good value)
```

Modules used

StdDefs, TokenLst, TokScan, Vector_Utils, Vector_Read, Cluster

Source code

Makefile

```
TARGET=sclust
VERSION=017
CFILES=tokenlst.c tokscan.c vector_utils.c vector_read.c sclust.c cluster.c
HFILES=tokenlst.h tokscan.h vector_utils.h vector_read.h stddefs.h cluster.h
OBJFILES=tokenlst.o tokscan.o vector_utils.o vector_read.o sclust.o cluster.o
DISTFILES=Makefile sclust.help.make ${CFILES} ${HFILES} ${TARGET}

CC=cc
CFLAGS=-O2 -64 -apo
LFLAGS=-lm

${TARGET}: ${OBJFILES}
   ${CC} ${CFLAGS} -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
```

sclust.help.make

```
/* Help make file for sclust
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 * Date: 16th September, 1999
 * Modified:
 * Version: 0.02
 */

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

p("This program performs adaptive clustering (a modified Forgy'sAlgorithm)\n");
p("on a set of vectors.  This is a data compression technique also known as\n");
```

```
p("Vector Quantisation.\n");
p("\n");
p("Inputs are:\n");
p(" o A file of vectors, all of the same dimensionality, to perform clustering\n");
p("   on (the training set).\n");
p(" o A parameter that determines the distance ratio required for a pattern to\n");
p("   be used as the seed of a new cluster.\n");
p("\n");
p("Input formats:\n");
p("Vector files (training set, seeds):\n");
p(" o Vectors are input one per line\n");
p(" o Vector components are separated by whitespace (space or tab)\n");
p(" o All vectors in a file should be of the same dimensionality\n");
p("\n");
p("Output formats:\n");
p("Vector files (means output)\n");
p(" o Vectors are output one per line\n");
p(" o Vector components are tab-delimited\n");
p("\n");
p("Note that a '-' can be supplied on the command line to denote reading from\n");
p("stdin and writing to stdout.\n");
p("\n");
p("\n");

#undef p

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

sclust.c

```
/* sclust -- performs vector quantization on a set of vectors
 *           uses an adaptive modified Forgy's Algorithm to find
 *           the means of a set of clusters that represent the
 *           entire set
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 *         QUT MLRC LPG Semester 2 1999
 * Date: 15th September, 1999
 * Modified: 21st November, 2000
 * Version: 0.17
 *
 * Usage: sclust {input vectors file}  -- whitespace-delimited, one vector per line
 *                                        (accepts stdin if '-' is specified)
 *               {output means file}   -- tab-delimited, one mean vector per line
 *                                        (writes to stdout if '-' is specified)
 *               {av. dist. parameter} -- average distance a pattern must be from
 *                                        all clusters to create a new cluster
 *
 * Modules used: vector_utils, vector_read, cluster, stddefs, tokenlst, tokscan
 *
 * Aknowledgements: Towsey, M. 1998, "The Use of Neural Networks in the Automated Analysis
 *                   of the Electroencephalogram", Phd. Thesis, Univ. of QLD, Brisbane.
 *                  Sayood, K. 1996, "Introduction to Data Compression", Morgan Kaufmann
 *                   Publishers, San Francisco.
 *                  Das, S. & Moser, M. 'Dynamic On-Line Clustering and State Extraction:
 *                   An Approach to Symbolic Learning', "Neural Networks", vol. 11, no. 1, pp.53-64.
 *
 * Note: This code accompanies the report "To Build a Better Cluster"
 */

/* -- Required modules -- */

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include <time.h>
#if defined(UNIX)
   #include <unistd.h>
#endif
#if defined(WIN32)
   #include <string.h>
   #define  strcasecmp  _stricmp
#endif
#include "vector_utils.h"
#include "vector_read.h"
#include "cluster.h"
#include "stddefs.h"
```

```
/* -- kmeans defines -- */

#define  APP_NAME    0
#define  VECTOR_FILE 1
#define  HELP        1
#define  MEANS_FILE  2
#define  DIST_PARM   3
#define  NUMARGS     4

#define  VERSION  "0.17"

#define  MAX_CLUSTERS   5000

#define  Abort(x)    AbortPrint((x), __LINE__, __FILE__);
#define  SQR(x)      ((x) * (x))
#define  ABS(x)      ((x) < 0 ? (-(x)) : (x))

/* -- typedefs -- */

typedef  vector   **codebook;    /* an array of vector pointers */


/* -- helper function prototypes -- */

/* -- Function Usage
 * Purpose: Writes the program usage to 'output'
 */
void Usage(FILE *output, char *appName);

/* -- Function Help
 * Purpose: Writes the program help to 'output'
 */
void Help(FILE *output, char *appName);

/* -- Function AbortPrint
 * Purpose: Abnormal termination of the program.
 *          Writes the filename and line number, and terminates
 *          with error code 'code'
 */
void AbortPrint(int code, int line, char *file);

/* -- Function ProcessCommandLine
 * Inputs: 'argc', 'argv' : Command line arguments
 * Purpose: Retrieves the command line parameters
 * Outputs: 'vectorFile' - the file to read training patterns from
 *          'meansFile'  - the file to write the cluster centroids to
 *          'distParm'   - the distance parameter to use for clustering
 */
void  ProcessCommandLine(int  argc,  char  *argv[],  FILE  **vectorFile,  FILE  **meansFile,  float
*distParm);

/* -- Function CloseFiles
 * Pre: 'vectorFile' and 'meansFile' have been opened
 * Post: 'vectorFile' and 'meansFile' have been closed
 */
void CloseFiles(FILE *vectorFile, FILE *meansFile);

/* -- Function PrintBanner
 * Pre: all inputs have been previously initialised
 * Purpose: Prints a program benner describing the clustering to
 *          perform to stderr
 */
void PrintBanner(char *argv[], FILE *vectorFile, FILE *meansFile, float distParm);

/* -- Function ReadTrainingSet
 * Pre: 'vectorFile' has been opened for reading
 *      'numVectors' has been allocated
 * Post: a cluster was returned containing the training data
 *       && 'numVectors' contains the number of patterns read
 */
cluster *ReadTrainingSet(FILE *vectorFile, int *numVectors);

/* -- Function MakeClusters
 * Purpose: Creates 'numClusters' clusters
 *          Returns an array of clusters [0..numClusters-1]
 */
cluster **MakeClusters(int numClusters);

/* -- Function DestroyClusters
 * Purpose: Destroys a cluster array[0..numClusters-1]
 *          Returns NULL
 */
```

```
cluster **DestroyClusters(cluster **clusters, int numClusters);

/* -- Function FindNearestNeighbour
 * Pre: 'vect' is the data pattern to test
 *      'currentBook' is the current codebook of cluster centroids
 *      'numClusters' is the number of valid clusters in 'currentBook'
 * Post: Using the Euclidean distance, returns the entry in
 *       'currentBook' that is closest to 'vect'
 * Note: If there are two closest clusters, the function returns the first
 */
int FindNearestNeighbour(vector *vect, codebook currentBook, int numClusters);

/* -- Function FindFurthestVector
 * Pre: 'vectorSet' is the array of all training vectors
 *      'numTrainVectors' is the number of entries in 'vectorSet'
 *      'clusterMembership' is the array associating training vectors with clusters
 *      'currentBook' is the current codebook
 * Post: Using Euclidean distance, returns the index into 'vectorSet' that is
 *       furthest away from its classified cluster
 */
int FindFurthestVector(codebook vectorSet, int numTrainVectors, int clusterMembership[], codebook
currentBook);

/* -- Function GetError
 * Purpose: Calculates the classification error of the current codebook
 */
double GetError(cluster **clusters, codebook book, int numClusters);

/* -- Function ClusterRMSDistance
 * Purpose: Computes the mean square average distance for the patterns
 *          within a cluster
 */
double ClusterRMSDistance(cluster *clust, vector *vect);

/* -- Function InitialiseCodebook
 * Purpose: Sets up the inital codebook for the modified Forgy's
 *          algorithm.  See the text for details
 */
codebook InitialiseCodebook(int numClusters, cluster *vectorSet, int numSlots);

/* -- Function CodebookAverageDistance
 * Purpose: Computes the average distance for a vectors to all other clusters
 */
double CodebookAverageDistance(vector *vect, codebook book, int numClusters);

/* -- Function CodebookCentroid
 * Purpose: Computes the centroid (Euclidean mean) of a cluster
 *          Returns the centroid as a vector
 */
vector *CodebookCentroid(codebook book, int numSlots);

/* -- Function FreeCluster
 * Purpose: Finds the first free cluster slot in the cluster array
 */
int FreeCluster(codebook book, int numSlots);


/* -- Codebook functions -- */

/* -- Function CodebookNew
 * Purpose: Creates a new codebook with 'slots' entries
 */
codebook CodebookNew(int slots);

/* -- Function CodebookDestroy
 * Purpose: Destroys a codebook with 'slots' entries
 */
codebook CodebookDestroy(codebook book, int slots);

/* -- Function CodebookNewFromCluster
 * Pre:'clust' is an allocated cluster
 * Post: A codebook was created containing an entry for each
 *       vector in 'clust'
 */
codebook CodebookNewFromCluster(cluster *clust);

/* -- Function CodebookMakeCopy
 * Purpose: Creates a complete copy of a codebook
 */
codebook CodebookMakeCopy(codebook book, int slots);

/* -- Function WriteCodebook
```

```
 * Purpose: Writes the vectors in 'book' to 'output'
 */
void WriteCodebook(FILE *output, codebook book, int slots);


/* -- main -- */

int main(int argc, char *argv[])
{
   FILE     *vectorFile,   /* The file to read the training patterns from */
            *meansFile;    /* The file to write the cluster centroids to */
   int      lastCluster;   /* The last valid entry in the cluster array */
   bool     terminate;     /* flag indicating the completion of clustering */
   float    distParm;      /* param.dist, as described in the text */
   double   lastError,     /* Previous calssification error */
            thisError;     /* Current classification error */

   codebook currentBook,   /* The current set of centroids */
            vectorSet;     /* Set of all training vectors */

   int   slot,                /* An index into various arrays */
         oldClosest,          /* Previsous closest cluster for a vector */
         numTrainVectors,     /* How many vectors read from traing file */
         *clusterMembership;  /* What cluster does this vector belong to? */

   cluster  **clusters,          /* Array of current clusters */
            *vectorSetCluster;   /* Contains the entire training set */

   double   avDist,        /* The average distance from all vectors to their closest cluster */
            closestDist;   /* The distance to the closest cluster */
   int      closest;       /* The index of the closest cluster in 'clusters' */


   /* Read in the program parameters */
   ProcessCommandLine(  argc, argv,
                        &vectorFile, &meansFile, &distParm);

   /* Display the function to perform */
   PrintBanner(argv, vectorFile, meansFile, distParm);

   /* Input training set */
   vectorSetCluster = ReadTrainingSet(vectorFile, &numTrainVectors);
   vectorSet = CodebookNewFromCluster(vectorSetCluster);


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

   /* Initialise codebook */
   {
      int   furthestVect;     /* Index of furthest vector */

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

      currentBook = InitialiseCodebook(MAX_CLUSTERS, vectorSetCluster, numTrainVectors);
      furthestVect = FindFurthestVector(vectorSet, numTrainVectors, clusterMembership, currentBook);
      currentBook[1] = VectorMakeCopy(vectorSet[furthestVect]);

      /* Initialise clusters array */
      clusters = MakeClusters(MAX_CLUSTERS);
      clusters[0] = ClusterMakeCopy(vectorSetCluster);
      clusters[0] = ClusterRemoveVector(clusters[0], vectorSet[furthestVect]);
      clusters[1] = ClusterAddVector(clusters[1], VectorMakeCopy(vectorSet[furthestVect]));

      lastCluster = 1;
   }

   /* Get the initial training error */
   lastError = GetError(clusters, currentBook, lastCluster + 1);

   terminate = FALSE;
   while (!terminate) {
      slot = 0;
      while (slot < numTrainVectors) {
         avDist = CodebookAverageDistance(vectorSet[slot], currentBook, lastCluster + 1);
         closest = FindNearestNeighbour(vectorSet[slot], currentBook, lastCluster + 1);
         closestDist = VectorEuclideanDist(vectorSet[slot], currentBook[closest]);
```

```
         if (closestDist > (avDist * distParm)) {
            /* Make a new cluster with the current vector */
            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;
         }

         oldClosest = clusterMembership[slot];

         /* Update the previous cluster's centroid after removing the vector */
         if (oldClosest != -1) {
            clusters[oldClosest] = ClusterRemoveVector(clusters[oldClosest], vectorSet[slot]);
            if (currentBook[oldClosest] != NULL) VectorDeallocate(currentBook[oldClosest]);
            currentBook[oldClosest] = ClusterCentroid(clusters[oldClosest]);
         }

         /* Update the current cluster's centroid after adding the vector */
         clusters[closest] = ClusterAddVector(clusters[closest], VectorMakeCopy(vectorSet[slot]));
         clusterMembership[slot] = closest;

         if (currentBook[closest] != NULL) VectorDeallocate(currentBook[closest]);
         currentBook[closest] = ClusterCentroid(clusters[closest]);

         slot++;     /* Next training vector */
      }

      /* Get current training error */
      thisError = GetError(clusters, currentBook, lastCluster + 1);
      if ((lastError >= thisError) && (lastError - thisError < 0.0001))
         terminate = TRUE;

      fprintf(stderr, "RMS error: %f (%d clusters)\n", thisError, lastCluster + 1);
      lastError = thisError;
   }

   /* Output final clusters */
   WriteCodebook(meansFile, currentBook, MAX_CLUSTERS);

   /* Clean up and exit */
   clusters = DestroyClusters(clusters, MAX_CLUSTERS);

   CloseFiles(vectorFile, meansFile);

   return 0;
}


/* -- helper functions -- */

void Usage(FILE *output, char *appName)
{
   fprintf(output, "\n*** sclust -- Performs adaptive clustering on a set of vectors\n");
   fprintf(output, "    Version %s build %s %s\n", VERSION, __TIME__, __DATE__);
   fprintf(output, "Usage: %s {vector file} {means file} {av. dist. parm}\n", appName);
   fprintf(output, "Where: {vector file}    -- file of vectors to analyse, one per line\n");
   fprintf(output, "                           ['-' for stdin]\n");
   fprintf(output, "       {means file}     -- filename to output the mean vectors to\n");
   fprintf(output, "                           ['-' for stdout]\n");
   fprintf(output, "       {av. dist. parm} -- see help file for information\n");
   fprintf(output, "                           (0.5 is a good value)\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  **vectorFile,  FILE  **meansFile,  float
*distParm)
{
   if ((argc > 1) && (strcasecmp(argv[HELP], "--help") == 0)) {
      /* Write help to a file */
      FILE  *helpFile;
```

```
      if (!(helpFile = fopen("sclust.help", "wt"))) {
         fprintf(stderr, "*** Could not create help file [sclust.help].\n");
         Abort(15);
      }

      Help(helpFile, argv[APP_NAME]);
      printf("Created sclust.help\n");
      exit(0);
   }

   if (argc < NUMARGS) {
      /* Not enough arguments */
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Invalid number of arguments.\n");
      Abort(1);
   }

   if (argc > NUMARGS) {
      /* Too many arguments */
      fprintf(stderr, "--- Warning: extra arguments ignored.\n");
   }

   if (strcmp(argv[VECTOR_FILE], "-") == 0) {
      /* Read training vectors from console */
      *vectorFile = stdin;
   }

   else if (!(*vectorFile = fopen(argv[VECTOR_FILE], "rt"))) {
      /* Could not open file to read training vectors from */
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Could not open vector file [%s] for reading.\n", argv[VECTOR_FILE]);
      Abort(2);
   }

   if (strcmp(argv[MEANS_FILE], "-") == 0) {
      /* Write training vectors to console */
      *meansFile = stdout;
   }

   else if (!(*meansFile = fopen(argv[MEANS_FILE], "wt"))) {
      /* Could not open file to write clusters to */
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Could not open means file [%s] for writing.\n", argv[MEANS_FILE]);
      Abort(3);
   }

   if (!isdigit(*argv[DIST_PARM])) {
      /* Could not translate distance parameter */
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Error converting distance parameter from [%s].\n", argv[DIST_PARM]);
      Abort(5);
   }

   *distParm = (float) atof(argv[DIST_PARM]);
   if ((*distParm > 2.0) || (*distParm < 0.0)) {
      /* Invalid distance parameter */
      fprintf(stderr, "*** Distance parameter [%2f] must be between 0.0 and 1.0.\n");
      Abort(26);
   }
}

void CloseFiles(FILE *vectorFile, FILE *meansFile)
{
   if (vectorFile != stdin) fclose(vectorFile);
   if (meansFile != stdout) fclose(meansFile);
}


void PrintBanner(char *argv[], FILE *vectorFile, FILE *meansFile, float distParm)
{
   fprintf(stderr, "- sclust ver %s\n", VERSION);
   fprintf(stderr, " o Vectors: << [");
   vectorFile == stdin ?
      fprintf(stderr, "- stdin]\n") :
      fprintf(stderr, "%s]\n", argv[VECTOR_FILE]);

   fprintf(stderr, " o Means: >> [");
   meansFile == stdout ?
      fprintf(stderr, "- stdout]\n") :
      fprintf(stderr, "%s]\n", argv[MEANS_FILE]);

   fprintf(stderr, " o distance parameter: [%f]\n", distParm);
```

```
}


codebook CodebookNew(int slots)
{
   codebook newBook;

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

   return newBook;
}

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) {
      /* Copy each vector into the new codebook */
      newBook[slot] = VectorMakeCopy(cIndex -> vect);
      cIndex = cIndex -> next;
      slot++;
   }

   return newBook;
}

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

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

   free(book);
   return NULL;
}

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

   fprintf(stderr, "Reading training vectors...\n");

   /* Read in vectors as a cluster */
   vectorSet = ClusterReadFromFile(vectorFile);
   if (ClusterIsEmpty(vectorSet)) {
      fprintf(stderr, "*** Empty training set.\n");
      Abort(10);
   }

   /* Get the cluster size */
   *numVectors = ClusterSize(vectorSet);
   fprintf(stderr, "Read %d vectors.\n", *numVectors);

   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);

   return vectorSet;
}

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) {
      /* Copy each vector into the new codebook */
      newBook[slot] = VectorMakeCopy(book[slot]);
      slot++;
   }

   return newBook;
}

cluster **MakeClusters(int numSeeds)
{
   cluster  **newClusterArray;
   int      slot;       /* index into cluster array */

   if (!(newClusterArray = (cluster **) malloc(numSeeds * sizeof(cluster *)))) {
      fprintf(stderr, "*** Unable to allocate cluster space.\n");
      Abort(14);
   }

   slot = 0;
   while (slot < numSeeds) {
      /* Allocate each cluster */
      newClusterArray[slot] = ClusterNew();
      slot++;
   }

   return newClusterArray;
}

cluster **DestroyClusters(cluster **clusters, int numSeeds)
{
   int   slot;

   slot = 0;
   while (slot < numSeeds) {
      clusters[slot] = ClusterDestroy(clusters[slot]);
      slot++;
   }

   free (clusters);

   return NULL;
}


int FindNearestNeighbour(vector *vect, codebook currentBook, int numClusters)
{
   int      slot,    /* index into codebook */
            minSlot; /* current slot of minimum distance */
   double   minDist, /* current minimum distance */
            dist;    /* current distance */

   minSlot = 0;
   minDist = VectorEuclideanDist(vect, currentBook[0]);
   slot = 1;
   while (slot < numClusters) {
      if (currentBook[slot] != NULL) {
         dist = VectorEuclideanDist(vect, currentBook[slot]);

         if (dist < minDist) {
            minDist = dist;
            minSlot = slot;
         }
      }
      slot++;
   }

   return minSlot;
}

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

   bookError = 0.0;
   slot = 0;
   while (slot < numClusters) {
      bookError += ClusterRMSDistance(clusters[slot], book[slot]);
      slot++;
```

```
   }

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

double ClusterRMSDistance(cluster *clust, vector *vect)
{
   double   errSum;
   cluster  *cIndex;

   if (ClusterIsEmpty(clust))
      return 0.0;

   errSum = 0.0;
   cIndex = clust;
   while (cIndex != NULL) {
      errSum += SQR(VectorEuclideanDist(vect, cIndex -> vect));
      cIndex = cIndex -> next;
   }

   errSum /= (double) ClusterSize(clust);
   return sqrt(errSum);
}

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

   for (slot = 0; slot < slots; slot++) {
      if (book[slot] != NULL) {
         VectorWrite(output, book[slot]);
         fprintf(output, "\n");
      }
   }
}

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

   Usage(helpFile, appName);

#include "sclust.help.make"
}

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

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

   if (!(newBook[0] = ClusterCentroid(vectorSet))) {
      fprintf(stderr, "*** Could not create initial centroid vector.\n");
      Abort(21);
   }

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

   return newBook;
}

vector *CodebookCentroid(codebook book, int numSlots)
{
   vector   *vectSum;      /* Current sum of vectors in codebook */
   int      slot, entries;

   if (numSlots == 0)
      return NULL;

   if (!(vectSum = VectorZero(book[0] -> dimensions)))
      return NULL;

   entries = 0;
   slot = 0;
```

```
   while (slot < numSlots) {
      if (book[slot] != NULL) {
         VectorSum(vectSum, book[slot]);
         entries++;
      }
      slot++;
   }

   /* Find the average by dividing by the number of vectors */
   VectorDivideScalar(vectSum, (double) entries);
   return vectSum;
}

int FreeCluster(codebook book, int numSlots)
{
   int   slot;

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

   return -1;
}


double CodebookAverageDistance(vector *vect, codebook book, int numClusters)
{
   double  distance;    /* Cumulative distance */
   int     slot;        /* Index into codebook */

   distance = 0.0;
   slot = 0;
   while (slot < numClusters) {
      if (book[slot] == NULL) {
         numClusters--;
         slot++;
         continue;
      }
      distance += VectorEuclideanDist(vect, book[slot]);
      slot++;
   }

   return distance / (double) numClusters;
}


int FindFurthestVector(codebook vectorSet, int numTrainVectors, int clusterMembership[], codebook
currentBook)
{
   double   distance,      /* Current test distance */
            furthestDist;  /* Current furthest distance */
   int      slot,          /* index into 'vectorSet' */
            furthestSlot;  /* Current furthest vector */

   slot = 0;
   furthestSlot = -1;
   furthestDist = 0.0;

   while (slot < numTrainVectors) {
      distance = VectorEuclideanDist(vectorSet[slot], currentBook[clusterMembership[slot]]);

      if (distance > furthestDist || furthestSlot == -1) {
         furthestDist = distance;
         furthestSlot = slot;
      }

      slot++;
   }

   return furthestSlot;
}

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