9.2.5 MakeFSA

Application purpose

MakeFSA
MakeFSA constructs an FSA definition from the probed hidden unit activations from a recurrent Elman network simulated with tlearn. A cluster analysis program is required to extract the locations of the FSA's states within the hidden unit space. The resulting clusters are loaded into MakeFSA. Transition tables are generated for the hidden unit activation data, and used to construct a deterministic FSA.

Usage

```
*** MakeFSA -- write an FSA definition for a tlearn probe run
    Version 0.06 build 11:12:37 Oct  8 1999 [IRIX64 6.5 IP25 mips]
Usage: makefsa {k-means vectors} {pattern file} {probe file}
               {tags file} {FSA definition}
Where: {k-means vectors} -- mean vectors from k-means clustering
       {pattern file}    -- pattern definition of inputs
       {probe file}      -- output from tlearn probe
       {tags file}       -- tags representing input/output lines
                            ['-' for no translation]
       {FSA definition}  -- file to write FSA to ['-' for stdout]
```

Required input files:

- A codebook resulting from either ellipsoidal or spherical cluster analysis
- A pattern file representing the training sequence
- A .probe file generated by a probe run of tlearn
- An optional .reset file used by tlearn during learning

MakeFSA will generate an FSA definition file that can be simulated with Ingo Schellhammer's autosim FSA simulator.

Modes used

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

Source code

Makefile

```
TARGET=makefsa
VERSION=020
CFILES=makefsa.c   vector_utils.c   vector_read.c   tokenlst.c   tokscan.c   smdarray.c   cluster.c
corr_matrix.c gauss2.c 2darray.c
HFILES=stddefs.h   vector_utils.h   vector_read.h   tokenlst.h   tokscan.h   smdarray.h   cluster.h
corr_matrix.h gauss.h 2darray.h
OBJFILES=${TARGET}.o   vector_utils.o   vector_read.o   tokenlst.o   tokscan.o   smdarray.o   cluster.o
corr_matrix.o gauss2.o 2darray.o
DISTFILES=Makefile makefsa.help.make ${CFILES} ${HFILES} ${TARGET}
SYSVER=\"`uname -mprs`\"

CC=cc
CFLAGS=-apo -64 -DPLATFORM="${SYSVER}" -DUNIX
LFLAGS=-lm

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

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

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

dist: ${DISTFILES}
```

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

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

MakeFSA.c

```
/* Makefsa -- Generates an FSA from either k-means or ellipsoidal cluster
 *             analysis and tlearn hidden units probe output
 *            Used to make a corresponding FSA for an elman network one step
 *             lookahead task
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 *         QUT MLRC LPG Semester 2 1999
 * Date: 4th October, 1999
 * Modified: 11th September, 2000
 * Version: 0.20
 *
 * Usage: makefsa {cluster type}        -- s (spherical) or e (elliptical)
 *                {cluster means input} -- whitespace-delimited, one vector per line
 *                {pattern file}        -- : (line #) (input) (output)
 *                {probe file}          -- output from tlearn -p run
 *                {tags file}           -- tags representing input / output lines
 *                                         (no translation occurs if '-' is specified)
 *                {resets file}         -- tlearn reset file
 *                                         Will reset FSA to zero state for each reset
 *                                         (no resets are used if '-' is specified)
 *                {FSA definiton file}  -- write FSA definition to this file
 *                                         (writes to stdout if '-' is specified)
 *
 * Modules used: vector_utils, vector_read, cluster, stddefs, smdarray
 *
 * Aknowledgements: Schellhammer, I., Diederich, J., Towsey, M., Brugman, C., "Knowledge Extraction
and Recurrent Neural Networks: An Analysis of an Elman Network trained on a Natural Language
Learning Task"
 */

/* -- Required modules -- */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <time.h>
#if defined(UNIX)
   #include <unistd.h>
#endif
#if defined(WIN32)
   #define  strcasecmp  _stricmp
#endif
#include <string.h>
#include "stddefs.h"
#include "vector_utils.h"
#include "vector_read.h"
#include "cluster.h"
#include "2darray.h"
#include "corr_matrix.h"
#include "smdarray.h"
#include "tokenlst.h"
#include "tokscan.h"


/* -- defines -- */

#define  APP_NAME       0
#define  HELP           1
#define  CLUSTER_TYPE   1
#define  CODEBOOK_FILE  2
#define  PATTERN_FILE   3
#define  PROBE_FILE     4
#define  TAGS_FILE      5
#define  RESET_FILE     6
#define  FSA_FILE       7
#define  NUMARGS        8

#define  VERSION        "0.20"
#define  HELP_FILE      "makefsa.help"
#define  MAXTAGLENGTH   20
#define  MAXCLASSES     100         /* Used for unigram array if */
                                    /* no tags file was supplied */
#define  MAX_CLUSTERS   200
```

```
#define  CODEBOOK_KMEANS      vector      **
#define  CODEBOOK_O2CLUST     corrMatrix  *

#define  Abort(x) AbortPrint((x), __LINE__, __FILE__);
#define  MAX(x, y)   ((x) > (y) ? (x) : (y))

/* -- kmeans typedefs -- */

typedef void  *codebook;             /* could be kmeans or o2clust */


/* -- 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 **meansFile, FILE **patternFile, FILE **probeFile,
         FILE **tagsFile, FILE **resetFile, FILE **fsaFile,
         bool *useTags, bool *useResets, bool *useO2);
void CloseFiles(FILE *cbFile, FILE *patternFile, FILE *probeFile, FILE *tagsFile, FILE *resetFile,
FILE *fsaFile);
void PrintBanner(char *argv[], FILE *fsaFile, bool useTags, bool useResets, bool useO2);
void ReadPattern(FILE *patternFile, int *predecessor, int *successor, bool *inFile);
int FindNearestNeighbour(vector *vect, codebook currentBook, bool useO2, int numSeeds);
bool CheckClusterDimensions(cluster *clust);
codebook ReadMeans(FILE *cbFile, bool useO2, int *numClusters);
void ReadTags(FILE *tagsFile, char **tagsArray[], int *numOutputs);
tokenList *ReadWholeFile(FILE *input);
void MakeFSADefHeader(FILE *fsaFile, int rescueState, int rescueOutput, int startState);
int FindStartState(codebook currentBook, bool useO2, int numClusters);
void ZeroUnigramArray(int array[], int size);
int FindMaxSlot(int array[], int size);
int InitResetFile(FILE *resetFile);
int ReadReset(FILE *resetFile);

/* -- Codebook functions -- */
/*codebook CodebookNew(int slots, bool useO2);*/
codebook CodebookDestroy(codebook book, int slots, bool useO2);
codebook CodebookNewFromCluster(cluster *clust, bool useO2);
codebook CodebookMakeCopy(codebook book, int slots, bool useO2);
/*void WriteCodebook(FILE *output, codebook book, int slots, bool useO2);*/


/* -- main -- */

int main(int argc, char *argv[])
{
   FILE     *cbFile, *patternFile, *probeFile, *tagsFile, *resetFile, *fsaFile;
   codebook book;
   int      numClusters, numInputLines, numTeachExamples, numTransitions;
   bool     terminate, useO2;
   vector      *vect;
   int      predecessor, successor, thisState, lastState,
            startState, rescueOutput;
   bool     inPatFile, inProbeFile, useTags, useResets;
   smdarray fsaTransitions;
   char     **tagsArray;
   int      *unigramStats;


/* srand48((double) time(0) + (double) getpid()); /* Initiallise rand48 generator */

   ProcessCommandLine(  argc, argv,
            &cbFile, &patternFile, &probeFile,
            &tagsFile, &resetFile, &fsaFile,
            &useTags, &useResets, &useO2);

   PrintBanner(argv, fsaFile, useTags, useResets, useO2);

   if (useTags) {
      ReadTags(tagsFile, &tagsArray, &numInputLines);
      fprintf(stderr, "Read %d tags (%d input lines).\n", numInputLines, numInputLines);
      if (!(unigramStats = (int *) malloc(numInputLines * sizeof(int)))) {
         fprintf(stderr, "*** Could not allocate inputs unigram array.\n");
         fprintf(stderr, "    ([%d] classes)\n", numInputLines);
         Abort(8);
      }
      ZeroUnigramArray(unigramStats, numInputLines);
   } else {
      if (!(unigramStats = (int *) malloc(MAXCLASSES * sizeof(int)))) {
```

```
         fprintf(stderr, "*** Could not allocate inputs unigram array.\n");
         fprintf(stderr, "    ([%d] classes -- MAXCLASSES)\n", MAXCLASSES);
         Abort(9);
      }
      ZeroUnigramArray(unigramStats, MAXCLASSES);
   }

   if (useO2) {
      book = (CODEBOOK_O2CLUST) ReadMeans(cbFile, useO2, &numClusters);
   } else {
      book = (CODEBOOK_KMEANS) ReadMeans(cbFile, useO2, &numClusters);
   }
   if (numClusters == 0) {
      fprintf(stderr, "*** No clusters to convert!\n");
      Abort(5);
   }
   fprintf(stderr, "Read %d clusters.\n", numClusters);

   fsaTransitions = sArrayCreate();
   InitResetFile(resetFile);

   startState = FindStartState(book, useO2, numClusters);
   lastState = startState;       /* Begin at zero state */
   if (!useTags) numInputLines = 0; /* Count input lines if we weren't given it */
   numTeachExamples = 0;         /* Current training pattern number */
   inPatFile = inProbeFile = TRUE;
   terminate = FALSE;
   while (!terminate) {
      ReadPattern(patternFile, &predecessor, &successor, &inPatFile);
      unigramStats[predecessor]++;  /* Count input */
      if (useResets && ReadReset(resetFile) == numTeachExamples)
         lastState = startState;    /* Reset the FSA */
      if (!useTags) numInputLines = MAX(numInputLines, predecessor);
      if (!(vect = VectorReadLine(probeFile, &inProbeFile)))
         break;

      thisState = FindNearestNeighbour(vect, book, useO2, numClusters) + 1;
      if (!sArrayIncrement(fsaTransitions, 4, 1, lastState, predecessor, thisState, successor)) {
         fprintf(stderr, "*** Error manipulating smdArray.\n");
         fprintf(stderr,  "        Cell:  [%d][%d][%d][%d]\n",  lastState,  predecessor,  thisState,
successor);
         Abort(6);
      }
      if (lastState == 0) startState = thisState;  /* get start state */
      lastState = thisState;
      terminate = !(inPatFile && inProbeFile);
      numTeachExamples++;
      if (((numTeachExamples % 200) == 0) && (numTeachExamples > 1))
         fprintf(stderr, "Read %d inputs...\n", numTeachExamples);
   }

   fprintf(stderr, "Read %d inputs total.\n", numTeachExamples - 1);
   fprintf(stderr, "Read %d transitions total.\n", sArrayCountLeaves(fsaTransitions));

   rescueOutput = FindMaxSlot(unigramStats, numInputLines);
   MakeFSADefHeader(fsaFile, startState, 10, startState);

   numTransitions = 0;

   for (lastState = 1; lastState <= numClusters; lastState++) {
      for (predecessor = 0; predecessor < numInputLines; predecessor++) {
         int   mpState = 0,      /* most probable state */
               mpOutput = 0,
               maxCount = 0,
               count;

         if (!sArrayDoesCellExist(fsaTransitions, 2, lastState, predecessor))
            continue;      /* this transition desn't exist */

         for (thisState = 0; thisState <= numClusters; thisState++) {
            if (sArrayDoesCellExist(fsaTransitions, 3, lastState, predecessor, thisState)) {
               if   ((count   =   sArrayBranchWeight(sArrayGetCell(fsaTransitions,   3,   lastState,
predecessor, thisState))) > maxCount) {
                  maxCount = count;
                  mpState = thisState;
               }
            }
         }

         maxCount = 0;

         for (successor = 0; successor < numInputLines; successor++) {
```

```
            if   ((count   =   sArrayAccess(fsaTransitions,   4,   lastState,   predecessor,   mpState,
successor)) > maxCount) {
               maxCount = count;
               mpOutput = successor;
            }
         }

         if (useTags)
            fprintf(fsaFile,    "%d,%d,%s,%s\n",    lastState,    mpState,    tagsArray[predecessor],
tagsArray[mpOutput]);
         else
            fprintf(fsaFile, "%d,%d,%d,%d\n", lastState, mpState, predecessor, mpOutput);

         numTransitions++;
      }
   }

   fprintf(stderr, "Pruned tree: %d transitions left.\n", numTransitions);

   CloseFiles(cbFile, patternFile, probeFile, tagsFile, resetFile, fsaFile);

   return 0;
}


/* -- helper functions -- */

void Usage(FILE *output, char *appName)
{
   fprintf(output, "\n*** MakeFSA -- write an FSA definition for a tlearn probe run\n");
   fprintf(output, "    Version %s build %s %s [%s]\n", VERSION, __TIME__, __DATE__, PLATFORM);
   fprintf(output,  "Usage:  %s  {cluster  type}  {codebook  file}  {pattern  file}  {probe  file}\n",
appName);
   fprintf(output, "               {tags file} {reset file} {FSA definition}\n");
   fprintf(output, "Where: {cluster type}    -- either (s)pherical or (e)lliptical\n");
   fprintf(output, "       {codebook file}   -- codebook from a clustering run\n");
   fprintf(output, "       {pattern file}    -- pattern definition of inputs\n");
   fprintf(output, "       {probe file}      -- output from tlearn probe\n");
   fprintf(output, "       {tags file}       -- tags representing input/output lines\n");
   fprintf(output, "                            ['-' for no translation]\n");
   fprintf(output, "       {reset file}      -- tlearn reset file.  Each reset will set the\n");
   fprintf(output, "                            FSA to the 'zero' state\n");
   fprintf(output, "                            ['-' for no resets]\n");
   fprintf(output, "       {FSA definition}  -- file to write FSA to ['-' for stdout]\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 **cbFile, FILE **patternFile, FILE **probeFile,
FILE **tagsFile, FILE **resetFile, FILE **fsaFile, bool *useTags, bool *useResets, bool *useO2)
{
   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 (strcasecmp(argv[CLUSTER_TYPE], "s") == 0) {
      *useO2 = FALSE;
   } else if (strcasecmp(argv[CLUSTER_TYPE], "e") == 0) {
```

```
      *useO2 = TRUE;
   } else {
      fprintf(stderr, "*** Cluster type must be either (s)pherical or (e)lliptical.\n");
      Abort(20);
   }

   if (!(*cbFile = fopen(argv[CODEBOOK_FILE], "rt"))) {
      fprintf(stderr,   "***   Could   not   open   cluster   means   file   [%s]   for   input.\n",
argv[CODEBOOK_FILE]);
      Abort(2);
   }

   if (!(*patternFile = fopen(argv[PATTERN_FILE], "rt"))) {
      fprintf(stderr, "*** Could not open pattern file [%s] for input.\n", argv[PATTERN_FILE]);
      Abort(3);
   }

   if (!(*probeFile = fopen(argv[PROBE_FILE], "rt"))) {
      fprintf(stderr, "*** Could not open probe file [%s] for input.\n", argv[PROBE_FILE]);
      Abort(4);
   }

   if (strcmp(argv[TAGS_FILE], "-") == 0) {
      *useTags = FALSE;
      *tagsFile = NULL;

   } else {
      *useTags = TRUE;
      if (!(*tagsFile = fopen(argv[TAGS_FILE], "rt"))) {
         fprintf(stderr, "*** Could not open tags file [%s] for input.\n", argv[TAGS_FILE]);
         Abort(5);
      }
   }

   if (strcmp(argv[RESET_FILE], "-") == 0) {
      *useResets = FALSE;
      *resetFile = NULL;

   } else {
      *useResets = TRUE;
      if (!(*resetFile = fopen(argv[RESET_FILE], "rt"))) {
         fprintf(stderr, "*** Could not open reset file [%s] for input.\n", argv[TAGS_FILE]);
         Abort(6);
      }
   }

   if (strcmp(argv[FSA_FILE], "-") == 0)
      *fsaFile = stdout;

   else {
      if (!(*fsaFile = fopen(argv[FSA_FILE], "wt"))) {
         fprintf(stderr,   "***   Could   not   open   FSA   definition   file   [%s]   for   output.\n",
argv[FSA_FILE]);
         Abort(7);
      }
   }
}

void CloseFiles(FILE *cbFile, FILE *patternFile, FILE *probeFile, FILE *tagsFile, FILE *resetFile,
FILE *fsaFile)
{
   if (cbFile != NULL) fclose(cbFile);
   if (patternFile != NULL) fclose(patternFile);
   if (probeFile != NULL) fclose(probeFile);
   if (tagsFile != NULL) fclose(tagsFile);
   if (resetFile != NULL) fclose(resetFile);
   if (fsaFile != stdout) fclose(fsaFile);
}


void PrintBanner(char *argv[], FILE *fsaFile, bool useTags, bool useResets, bool useO2)
{
   fprintf(stderr, "- MakeFSA ver %s\n", VERSION);
   fprintf(stderr, " o Clustering type: ");
   useO2 ?
      fprintf(stderr, "ellipsoidal\n") :
      fprintf(stderr, "spherical\n");
   fprintf(stderr, " o Clusters: << [%s]\n", argv[CODEBOOK_FILE]);
   fprintf(stderr, " o Input patterns: << [%s]\n", argv[PATTERN_FILE]);
   fprintf(stderr, " o Hidden units probe: << [%s]\n", argv[PROBE_FILE]);
   fprintf(stderr, " o I/O line tags definitions: << [");
   useTags ?
```

```
      fprintf(stderr, "%s]\n", argv[TAGS_FILE]) :
      fprintf(stderr, "not translated]\n");
   fprintf(stderr, " o Resets: << [");
   useResets ?
      fprintf(stderr, "taken from %s]\n", argv[RESET_FILE]) :
      fprintf(stderr, "not used]\n");
   fprintf(stderr, " o FSA definition: >> [");
   fsaFile == stdout ?
      fprintf(stderr, "- stdout]\n") :
      fprintf(stderr, "%s]\n", argv[FSA_FILE]);
}

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

   Usage(helpFile, appName);

#include "makefsa.help.make"
}


void ReadPattern(FILE *patternFile, int *predecessor, int *successor, bool *inFile)
{
   int     patternNum;

   fscanf(patternFile, "%d %d %d", &patternNum, predecessor, successor);

   if (feof(patternFile))
      *inFile = FALSE;
}

codebook CodebookNew(int slots, bool useO2)
{
   codebook newBook;

   if (useO2) {
      if (!(newBook = (CODEBOOK_O2CLUST) malloc(slots * sizeof(corrMatrix))))
         return NULL;
   } else {
      if (!(newBook = (vector **) malloc(slots * sizeof(vector *))))
         return NULL;
   }

   return newBook;
}

bool CheckClusterDimensions(cluster *clust)
{
   int     dimensions;             /* dimension to check against */
   cluster *cIndex;                /* index into cluster */

   if (clust == NULL) return TRUE; /* empty cluster, so we pass */

   dimensions = clust -> vect -> dimensions;
   cIndex = clust -> next;         /* the first one passes by definition */

   while (cIndex != NULL) {
      if (cIndex -> vect -> dimensions != dimensions)
         return FALSE;

         cIndex = cIndex -> next;
   }

   return TRUE;
}

codebook CodebookNewFromCluster(cluster *clust, bool useO2)
{
   codebook        newBook;               /* new codebook */
   cluster         *cIndex;               /* index into cluster */
   int             slot;                  /* slot index into codebook */

   if (!(newBook = CodebookNew(ClusterSize(clust), useO2)))
      return NULL;

   slot = 0;
   cIndex = clust;
   while (cIndex != NULL) {
      ((CODEBOOK_KMEANS) newBook)[slot] = VectorMakeCopy(cIndex -> vect);
```

```
      cIndex = cIndex -> next;
      slot++;
   }

   return newBook;
}

codebook CodebookDestroy(codebook book, bool useO2, int slots)
{
   int     slot;

   slot = 0;
   while (slot < slots) {
      if(useO2) {
         CMatrixDestroy(((CODEBOOK_O2CLUST) book)[slot]);
      } else {
         VectorDeallocate(((CODEBOOK_KMEANS) book)[slot]);
      }

      slot++;
   }

   free(book);
   return NULL;
}

int FindNearestNeighbour(vector *vect, codebook currentBook, bool useO2, int numSeeds)
{
   int      slot;           /* index into codebook */
   double   minDist,        /* current minimum distance */
            dist;           /* current distance */
   double   minSlot;        /* current slot of minimum distance*/

   minSlot = 0;
   if (useO2) {
      minDist = CMatrixCorrelationWithVect(((CODEBOOK_O2CLUST) currentBook)[0], vect);
   } else {
      minDist = VectorEuclideanDist(vect, ((CODEBOOK_KMEANS) currentBook)[0]);
   }

   slot = 1;
   while (slot < numSeeds) {
      if (useO2) {
         dist = CMatrixCorrelationWithVect(((CODEBOOK_O2CLUST) currentBook)[slot], vect);
      } else {
         dist = VectorEuclideanDist(vect, ((CODEBOOK_KMEANS) currentBook)[slot]);
      }

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

   return (int) minSlot;
}


codebook ReadMeans(FILE *cbFile, bool useO2, int *numClusters)
{
   corrMatrix  *book;
   cluster  *clust;

   if (useO2) {
      int   slot;

      if (!(book = (CODEBOOK_O2CLUST) malloc(MAX_CLUSTERS * sizeof(corrMatrix))))
         return NULL;

      for (slot = 0; slot < MAX_CLUSTERS; slot++)
         book[slot] = NULL;

      slot = 0;
      while (!feof(cbFile)) {
         book[slot] = CMatrixRead(cbFile);
         if (book[slot] != NULL)
            slot++;
      }
      *numClusters = slot;
      return book;
```

```
   } else {
      clust = ClusterNew();
      clust = ClusterReadFromFile(cbFile);

      *numClusters = ClusterSize(clust);
      return CodebookNewFromCluster(clust, useO2);
   }
}

tokenList *ReadWholeFile(FILE *input)
{
   tokenList   *list, *temp;
   bool        inFile;

   inFile = TRUE;
   list = ReadLineTokens(input, &inFile);          /* Get the first line */

   if (list == NULL)
      return list;

   while (inFile) {
      temp = ReadLineTokens(input, &inFile);  /* Get the next line */
      list = ConcatenateTokenList(list, temp);
   }

   return list;
}

void ReadTags(FILE *tagsFile, char **tagsArray[], int *numOutputs)
{
   tokenList   *allTags, *index;
   int         numTags;

   allTags = ReadWholeFile(tagsFile);

   if (allTags == NULL) {
      fprintf(stderr, "*** Error reading tags file.\n");
      Abort(7);
   }

   numTags = 0;
   index = allTags;
   while (index != NULL) {
      index = index -> NEXT;
      numTags++;
   }

   *numOutputs = numTags;

   if (!Allocate2DArray(tagsArray, 1, numTags, MAXTAGLENGTH)) {
      fprintf(stderr, "*** Could not allocate tag array.\n");
      Abort(8);
   }

   index = allTags;
   numTags = 0;
   while (index != NULL) {
      strcpy((*tagsArray)[numTags], index -> token);
      index = index -> NEXT;
      numTags++;
   }
   DestroyTokenList(allTags);
}

void MakeFSADefHeader(FILE *fsaFile, int rescueState, int rescueOutput, int startState)
{
   fprintf(fsaFile, "Finite State Automaton Simulator\n");
   fprintf(fsaFile, "Rescue State = %d\n", rescueState);
   fprintf(fsaFile, "Rescue Output = %d\n", rescueOutput);
   fprintf(fsaFile, "Start State = %d\n", startState);
   fprintf(fsaFile, "Transitions:\n");
   fprintf(fsaFile, "(FromState,ToState,Input,Output)\n");
}

int FindStartState(codebook currentBook, bool useO2, int numClusters)
{
   vector   *zero;
   int   startState;

   if (useO2) {
      zero = VectorZero(((CODEBOOK_O2CLUST) currentBook)[0] -> dimensions);
   } else {
```

```
      zero = VectorZero(((CODEBOOK_KMEANS) currentBook)[0] -> dimensions);
   }

   startState  = FindNearestNeighbour(zero, currentBook, useO2, numClusters);

   zero = VectorDeallocate(zero);
   return startState;
}

void ZeroUnigramArray(int array[], int size)
{
   int   count;

   for (count = 0; count < size; count++)
      array[count] = 0;
}

int FindMaxSlot(int array[], int size)
{
   int   maxSlot = 0,
         maxValue = 0,
         count;

   for (count = 0; count < size; count++) {
      if (array[count] > maxValue) {
         maxValue = array[count];
         maxSlot = count;
      }
   }

   return maxSlot;
}

int InitResetFile(FILE *resetFile)
{
   int   numResets;

   fscanf(resetFile, "%d\n", &numResets);
   return numResets;
}


int ReadReset(FILE *resetFile)
{
   int   reset;

   fscanf(resetFile, "%d\n", &reset);
   return reset;
}


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

makefsa.help.make

```
/* Help make file for MakeFSA
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au
 * Date: 4th October, 1999
 * Modified: 13th September, 2000
 * Version: 0.02
 */

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

p("\n");
p("This program creates a deterministic FSA from a set of state definitions\n");
p("(clustering codebook) and a set of training vectors.  This is used to obtain an\n");
p("FSA from a tlearn hidden units probe run and an AVQ (Euclidean distance or correlation\n");
p("distance) analysis of the hidden unit states.\n");
p("\n");
p("MakeFSA will generate a state for each cluster in the AVQ analysis.  The\n");
p("training vectors will be read and a transition table generated from state to\n");
p("state given each input.  Non-deterministic transitions are culled by taking\n");
p("the most probable state moved to for each input, as well as the most probable\n");
p("output for each transition.\n");
p("\n");
p("The FSA definition is augmented by a start state and a rescue state, as well as\n");
p("a rescue output.  These are defined as:\n");
p(" o Start state: the state corresponding to the cluster containing the zero\n");
p("   vector.\n");
```

```
p(" o Rescue state: at this stage, this is the same as the start state.\n");
p(" o Rescue output: the most common output for all inputs (i.e. a noun in\n");
p("   language data).\n");
p("\n");
p("Inputs are:\n");
p(" o The type of clustering that was performed (spherical or elliptical).\n");
p(" o A file corresponding to the clusters identified by the AVQ analysis.\n");
p(" o A pattern file with a list of input/output pairs for the training data.\n");
p(" o A file of vectors containing the training data (hidden unit activations)\n");
p(" o An optional file specifing names ('tags') to give to each of the\n");
p("   intput / output lines (tags file).\n");
p("\n");
p("\n");
p("Input formats:\n");
p("Vector files(k-means clusters, training set):\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 the file should be of the same dimensionality.\n");
p("\n");
p("Codebook file:\n");
p(" o For k-means clusters, see above.\n");
p(" o For o2clust clusters, the file format is determined by CMatrixWrite in the");
p("   corr_matrix.c source file.\n");
p("\n");
p("Pattern file:\n");
p(" o each line follows the same pattern: input# intput output.\n");
p(" o input# starts from zero and counts up to the number of inputs.\n");
p(" o input and output represent the input and output lines that should be\n");
p("   activated.\n");
p("\n");
p("Tags file:\n");
p(" o Tags are separated by whitespace (space, tab or <cr>).\n");
p(" o If this file is given, there MUST be at least as many tags as input / output\n");
p("   lines.  The program's behavious is undefined otherwise.\n");
p("\n");
p("\n");
p("Output formats:\n");
p("FSA Definition:\n");
p(" o The output format is designed to be used in Ingo Schellhammer's FSA\n");
p("   simulator 'autosim'.  It consists of an explicit header which must be\n");
p("   character perfect, containing the start state, rescue state and rescue\n");
p("   output.\n");
p(" o Deterministic FSA transitions are given one per line, separated by commas.\n");
p(" o If a tags file was specifed, inputs and outputs are translated into tags.\n");
p("   (Note that this format is not compatible with 'autosim').\n");
p("\n");
p("\n");
p("Note: MakeFSA was designed for use on an elman-type recurrent neural network\n");
p("      trained on language data to perform a one-step-lookahead task.  Therefore\n");
p("      the input and output lines correspond one-to-one, and the tags file can\n");
p("      be used to describe both.  This should hold true for most FSAs. The\n");
p("      pattern files used were generated by concatenating lines in tlearn\n");
p("      ___.teach and ___.data files.  This will only work if the tlearn files\n");
p("      are in locallist format.\n");
p("\n");
p("\n");
p("\n");
p("\n");

#undef p

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