9.2.7 vector

Application purpose

vector takes a data set of known tags and sentence boundaries, and writes the corresponding tlearn .teach and .data files. The output can be written in both localist and distributed representations, and the input and output lines can be either binary or normalised together.

Usage

```
*** vector ver 0.21 -- creates tlearn vector files from tag input
Usage: vector {tagfile} {inputfile} {inputType} {outputType} {fileStructure}
{basefilename}
Where: {i/oType} == [a]verage or [b]inary
       {fileStructure} == [l]ocalist or [d]istributed
       {outputfiles} -> basefilename.cf
                        basefilename.data
                        basefilename.teach
                        basefilename.reset
```

The data set consists of a sequence of whitespace-delimited tags, with optional resets denoted by '/C' within the data set.

example.txt

```
HN M H AJT *ADJ & /C
F HN M HN /S /C
ADJ *H H FI *P H FI *P /C
HN M P DD H /C
HN M P M P /C
F HN OM P DD /C
```

The file to define the tags is in the standard format: tokens separated by whitespace.

wales.claire.tags

```
&
ADJ
AJT
AL
CP
DD
DOR
DQ
F
FI
FR
G
H
HN
IE
INF
M
MOC
MOD
N
O
OM
OX
P
Q
SC
ST
T
VO
WH
/S
```

Modules used

StdDefs, TokenLst, TokScan, HTable,

Source code

Makefile

```
DISTFILES=tokenlst.c   tokenlst.h   vector.c   htable.c   htable.h   stddefs.h   test.tags   test.txt
wales.claire.tags 6yrs.5.tag.mark vector
OBJFILES=tokenlst.o vector.o tokscan.o htable.o
VERSION=021
TARGET=vector
CC=gcc

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

clean:
   rm -f *.o vector
dist: ${DISTFILES}
   tar cvf ${TARGET}_v${VERSION}.tar $(DISTFILES)
   gzip -f ${TARGET}_v${VERSION}.tar

htable.o: htable.c

tokenlst.o: tokenlst.c

vector.o: vector.c

tokscan.o: tokscan.c

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

vector.c

```
/* vector.c -- creates a set of vectors for tlearn
 *
 * Author: Dylan Muir (dr.muir@qut.edu.au)
 * Created: 17th May, 1999
 * Modified: 8th October, 1999
 * Version: 0.21
 */

#include <stdio.h>
#include <string.h>
#include "stddefs.h"
#include "htable.h"
#include "tokenlst.h"
#include "tokscan.h"


/* --- Vector defines --- */

#define  BINARY   0
#define  AVERAGE  1

#define  TAGFILE     1
#define  INPUT_TYPE  3
#define  OUTPUT_TYPE 4
#define  INPUT_FILE  2
#define  BASEFILE    6
#define  FILE_STRUCT 5

#define  EXTRAmark   '*'
#define  RESET       "/C"

#define  VERSION        "0.21"
#define  DISTFILETYPE   "distributed\n"      /* Type of output tlearn file */
#define  LOCALFILETYPE  "localist   \n"      /* MUST BE SAME LENGTH */


/* --- Vector enums --- */

typedef enum {binary, averaged} vectorType;
typedef enum {localist, distributed} fileType;


/* --- Helper functions prototypes --- */

void        Usage(FILE *output, char *argv[]);
void        Help(char *argv[]);
tokenList   *ReadWholeFile(FILE *input);
```

```
void        PrintTokenList(FILE *output, tokenList *list);
bool        SetupHashtable(tokenList *tags);
int         CountTokens(tokenList *list);
int         DoVectors(FILE *input, int vectorSize, vectorType intype, vectorType outtype, fileType
fileStruct, FILE *dvFile, FILE *tvFile, FILE *rvFile);
bool        IsExtra(tokenList *tag);
bool        IsReset(tokenList *tag);
void        AddToVector(float vector[], int vectorSize, char *tag);
void        OutputVector(FILE  *output,  float  vector[],  int  vectorSize,  vectorType  printType,
fileType fileStruct);
void        CalcVector(float vector[], int vectorSize);
void        ResetVector(float vector[], int vectorSize);
void        OpenFiles(  int argc, char *argv[],
                        FILE **tagFile, FILE **inputFile,
                        FILE **cfFile, FILE **dvFile, FILE **tvFile, FILE **rvFile);
void        InterpretCommandLine(int   cargc,   char   *argv[],   vectorType   *inputType,   vectorType
*outputType, fileType *fileStruct);
void        OutputReset(FILE *resetFile, int vectorNum);
void        CopyVector(float dest[], float source[], int vectorSize);
void        MakeCfFile(FILE *cfFile, int vectorSize);
void        CloseFiles(FILE *inFile, FILE *cfFile, FILE *dvFile, FILE *tvFile, FILE *rvFile);
void        FixDatFiles(FILE *dvFile, FILE *tvFile, FILE *rvFile, int numVectors, int numResets);
tokenList   *ReadWholeVector(FILE *input, tokenList *index, tokenList **line, float vector[], int
vectorSize, bool *inFile);

/* --- Main function --- */

int main(int argc, char *argv[])
{
   FILE  *tagFile,
         *inputFile,
         *cfFile,
         *dvFile,
         *tvFile,
         *rvFile;

   tokenList   *tags;
   int         vectorSize, numVectors;
   vectorType  inputType, outputType;
   fileType    fileStruct;

   InterpretCommandLine(argc, argv, &inputType, &outputType, &fileStruct);

   OpenFiles(argc, argv, &tagFile, &inputFile, &cfFile, &dvFile, &tvFile, &rvFile);

   tags = ReadWholeFile(tagFile);
   if ((vectorSize = CountTokens(tags)) == 0) {
      fprintf(stderr, "*** Read zero tags from tags file [%s].\n", argv[TAGFILE]);
      fprintf(stderr, "    Cannot continue with tlearn training vectors of size zero.\n");
      fprintf(stderr, "Aborting...\n");
      exit(21);
   }

   printf("Read %d tags\n", vectorSize);

   if (!SetupHashTable(tags)) {
      fprintf(stderr, "*** Cannot set up lookup table for tags.\nAborting...\n");
      exit(1);
   }

   numVectors = DoVectors( inputFile, vectorSize,
                           inputType, outputType,
                           fileStruct,
                           dvFile, tvFile, rvFile);
   MakeCfFile(cfFile, vectorSize);

   printf("Wrote %d vectors total.\nFinished!\n", numVectors);
   CloseFiles(inputFile, cfFile, dvFile, tvFile, rvFile);
}


/* --- Helper function bodies --- */

void Usage(FILE *output, char *argv[])
{
   fprintf(output, "\n*** vector ver %s -- creates tlearn vector files from tag input\n", VERSION);
   fprintf(output,  "Usage:  %s  {tagfile}  {inputfile}  {inputType}  {outputType}  {fileStructure}
{basefilename}\n", argv[0]);
   fprintf(output, "Where: {i/oType} == [a]verage or [b]inary\n");
   fprintf(output, "       {fileStructure} == [l]ocalist or [d]istributed\n");
   fprintf(output, "       {outputfiles} -> basefilename.cf\n");
   fprintf(output, "                        basefilename.data\n");
```

```
   fprintf(output, "                        basefilename.teach\n");
   fprintf(output, "                        basefilename.reset\n");
   fprintf(output, "For detailed info, type %s --help\n\n", argv[0]);
}

void Help(char *argv[])
{
   FILE  *helpFile;

   if ((helpFile = fopen("vector.help", "wt")) == NULL) {
      fprintf(stderr, "\n*** Unable to create \"vector.help\"\n");
      helpFile = stdout;
   }

   fprintf(helpFile, "\n-------------------------------------------\n");
   fprintf(helpFile, " Help file for vector ver %s\n", VERSION);
   fprintf(helpFile, "-------------------------------------------\n");

   Usage(helpFile, argv);

   fprintf(helpFile, "Vector will translate a dataset of tags into vectors for\n");
   fprintf(helpFile, "traning and testing with tlearn.  Inputs are:\n");
   fprintf(helpFile, "   * a file with all the tags contained in the dataset,\n");
   fprintf(helpFile, "     separated by whitespace\n");
   fprintf(helpFile, "   * an input file with resets marked with \"/C\"\n");
   fprintf(helpFile, "   * whether the inputs and outputs are binary (0/1)\n");
   fprintf(helpFile, "     or averaged (sum to 1)\n");
   fprintf(helpFile, "   * whether output files are to be localist or distributed\n");
   fprintf(helpFile, "     (see tlearn documentation for details)\n");
   fprintf(helpFile, "   * a base file name to use\n\n");

   fprintf(helpFile, "Outputs are:\n");
   fprintf(helpFile, "   * [basefilename].cf -- contains skeleton config file\n");
   fprintf(helpFile, "   * [basefilename].data -- contains data (input) vectors\n");
   fprintf(helpFile, "   * [basefilename].teach -- contains teaching (output) vectors\n");
   fprintf(helpFile, "   * [basefilename].reset -- contains resets\n\n");

   fprintf(helpFile, "Using the output files:\n");
   fprintf(helpFile, "Outputs are specified either localist or distributed.\n");
   fprintf(helpFile, "Note that localist is only valid if binary outputs are\n");
   fprintf(helpFile, "specified.  The \".cf\" file will have to be filled out for\n");
   fprintf(helpFile, "the specific network to be trained.\n\n");

   fprintf(helpFile, "Interpreting the vectors:\n");
   fprintf(helpFile, "The vectors are created in the same order as the input tags were\n");
   fprintf(helpFile, "presented to the program.\n");

   fprintf(stderr, "Created the file \"vector.help\"\n\n");
}


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 PrintTokenList(FILE *output, tokenList *list)
{
   while (list != NULL) {
      fprintf(output, "%s ", list -> token);

      list = list -> NEXT;
   }
}

bool SetupHashTable(tokenList *tags)
{
```

```
   int   index = 0;

   if (!hashInitTable(CountTokens(tags)) * 2)
      return FALSE;

   while (tags != NULL) {
      if (!hashInsert(tags -> token, index))
         fprintf(stderr, "--- Duplicate tag: [%s]\n", tags -> token);

      index++;
      tags = tags -> NEXT;
   }

   return TRUE;
}

int CountTokens(tokenList *list)
{
   int   count = 0;

   while (list != NULL) {
      list = list -> NEXT;
      count++;
   }
   return count;
}

int  DoVectors(FILE  *input,  int  vectorSize,  vectorType  intype,  vectorType  outtype,  fileType
fileStruct, FILE *dvFile, FILE *tvFile, FILE *rvFile)
{
   float       *dvector, *tvector;
   tokenList   *line,
               *index;
   bool        inFile;
   int         vectorNum,
               resetNum;


   if ((dvector = (float *) malloc(vectorSize * sizeof(float))) == NULL) {
      fprintf(stderr, "*** Cannot allocate data vector.\nAborting...\n");
      exit(1);
   }

   if ((tvector = (float *) malloc(vectorSize * sizeof(float))) == NULL) {
      fprintf(stderr, "*** Cannot allocate teach vector.\nAborting...\n");
      exit(1);
   }

   if (fileStruct == localist) {
      fprintf(dvFile, LOCALFILETYPE);
      fprintf(tvFile, LOCALFILETYPE);
   } else {
      fprintf(dvFile, DISTFILETYPE);
      fprintf(tvFile, DISTFILETYPE);
   }

   fprintf(dvFile, "           \n");   /* \                                                    */
   fprintf(tvFile, "           \n");   /*  >--- Make space for numbers of vectors, resets, etc. */
   fprintf(rvFile, "           \n");   /* /                                                    */

   vectorNum = 0;
   inFile = TRUE;
   ResetVector(dvector, vectorSize);
   OutputReset(rvFile, 0);
   resetNum = 1;

   index = ReadWholeVector(input, NULL, &line, dvector, vectorSize, &inFile);

   do {
      if (IsReset(index)) {
         OutputReset(rvFile, vectorNum);
         ResetVector(dvector, vectorSize);
         index = index -> NEXT;
         index = ReadWholeVector(input, index, &line, dvector, vectorSize, &inFile);
         resetNum++;
      }

      index = ReadWholeVector(input, index, &line, tvector, vectorSize, &inFile);
      if (index == NULL && !inFile)
         continue;

      if (intype == averaged) CalcVector(dvector, vectorSize);
```

```
      if (outtype == averaged) CalcVector(tvector, vectorSize);
      OutputVector(dvFile, dvector, vectorSize, intype, fileStruct);
      OutputVector(tvFile, tvector, vectorSize, outtype, fileStruct);
      CopyVector(dvector, tvector, vectorSize);

      if ((vectorNum % 1000) == 0 && vectorNum > 0)
         printf("Wrote %d vectors...\n", vectorNum);

      vectorNum++;
   } while (inFile);

   FixDatFiles(dvFile, tvFile, rvFile, vectorNum, resetNum);
   return vectorNum;
}


bool IsExtra(tokenList *tag)
{
   return ((tag != NULL) && (*(tag -> token) == EXTRAmark));
}

bool IsReset(tokenList *tag)
{
   return ((tag != NULL) && (strcmp(tag -> token, RESET) == 0));
}

void ResetVector(float vector[], int vectorSize)
{
   while (vectorSize >= 0) {
      vector[vectorSize] = 0.0;
      vectorSize--;
   }
}

void AddToVector(float vector[], int vectorSize, char *tag)
{
   if (!hashIsIn(tag)) {
      fprintf(stderr, "--- Tag [%s] not defined.\n", tag);
      return;
   }

   vector[hashValue(tag)] = 1.0;
}

void CalcVector(float vector[], int vectorSize)
{
   int   entries,
      index;
   float value;

   index = entries = 0;

   while (index < vectorSize) {
      if (vector[index] != 0.0)
         entries++;
      index++;
   }

   value = 1.0 / entries;

   index = 0;
   while (index < vectorSize) {
      if (vector[index] != 0.0)
         vector[index] = value;
      index++;
   }
}


void  OutputVector(FILE  *output,  float  vector[],  int  vectorSize,  vectorType  printType,  fileType
fileStruct)
{
   int   index = 0,
         notFirst = 0;

   while (index < vectorSize) {
      if (printType == averaged)
         fprintf(output, "%4f ", vector[index]);

      else {
         if (fileStruct == localist) {
         if (((int) vector[index] != 0)) {
```

```
            if (notFirst != 0)
               fprintf(output, ",");
            fprintf(output, "%d", index + 1);
            notFirst = 1;
         }} else
            fprintf(output, "%d ", (int) vector[index]);
      }

      index++;
   }
   fprintf(output, "\n");
}


void OpenFiles(int argc, char *argv[],
               FILE **tagFile, FILE **inputFile,
               FILE **cfFile, FILE **dvFile, FILE **tvFile, FILE **rvFile)
{
   char  buffer[MAXSTRING];

   if (strlen(argv[BASEFILE]) > MAXSTRING - 10) {
      fprintf(stderr,   "***   MAXSTRING   too   short   for   base   filename.\nPlease   recompile.
Aborting...\n");
   }

   if ((*tagFile = fopen(argv[TAGFILE], "rt")) == NULL) {
      fprintf(stderr, "*** Cannot open [%s] to read tags.\nAborting...\n", argv[TAGFILE]);
      exit(1);
   }

   if ((*inputFile = fopen(argv[INPUT_FILE], "rt")) == NULL) {
      fprintf(stderr, "*** Cannot open [%s] to read input.\nAborting...\n", argv[INPUT_FILE]);
      exit(1);
   }

   strcpy(buffer, argv[BASEFILE]);
   strcat(buffer, ".cf");
   if ((*cfFile = fopen(buffer, "wt")) == NULL) {
      fprintf(stderr, "*** Cannot open [%s] to write config file.\nAborting...\n", buffer);
      exit(1);
   }

   strcpy(buffer, argv[BASEFILE]);
   strcat(buffer, ".data");
   if ((*dvFile = fopen(buffer, "wt")) == NULL) {
      fprintf(stderr, "*** Cannot open [%s] to write data file.\nAborting...\n", buffer);
      exit(1);
   }

   strcpy(buffer, argv[BASEFILE]);
   strcat(buffer, ".teach");
   if ((*tvFile = fopen(buffer, "wt")) == NULL) {
      fprintf(stderr, "*** Cannot open [%s] to write teach file.\nAborting...\n", buffer);
      exit(1);
   }

   strcpy(buffer, argv[BASEFILE]);
   strcat(buffer, ".reset");
   if ((*rvFile = fopen(buffer, "wt")) == NULL) {
      fprintf(stderr, "*** Cannot open [%s] to write reset file.\nAborting...\n", buffer);
      exit(1);
   }
}


void InterpretCommandLine(int argc, char *argv[], vectorType *inputType, vectorType *outputType,
fileType *fileStruct)
{
   if ((argc > 1) && (strcmp(argv[1], "--help") == 0)) {
      Help(argv);
      exit(0);
   }

   if (argc < 7) {
      Usage(stderr, argv);
      fprintf(stderr, "*** Invalid number of arguments.\n");
      exit(1);
   }

   if (argc > 7)
      fprintf(stderr, "*** WARNING: extra arguments ignored.\n");
```

```
   switch (*argv[INPUT_TYPE]) {
      case 'A':
      case 'a':
         *inputType = averaged;
         break;

      case 'B':
      case 'b':
         *inputType = binary;
         break;

      default:
         Usage(stderr, argv);
         fprintf(stderr, "*** Error in 'input type' field\n");
         exit(1);
   }

   switch (*argv[OUTPUT_TYPE]) {
      case 'A':
      case 'a':
         *outputType = averaged;
         break;

      case 'B':
      case 'b':
         *outputType = binary;
         break;

      default:
         Usage(stderr, argv);
         fprintf(stderr, "*** Error in 'output type' field\n");
         exit(1);
   }

   switch (*argv[FILE_STRUCT]) {
      case 'L':
      case 'l':
         if (!((*inputType == binary) && (*outputType == binary))) {
            Usage(stderr, argv);
            fprintf(stderr, "*** Error in 'file type' field\n");
            fprintf(stderr, "    For localist files, input and output must be 'binary'\n");
            exit(1);
         }
         *fileStruct = localist;    /* localist */
         break;

      case 'D':
      case 'd':
         *fileStruct = distributed;    /* distributed */
         break;

      default:
         Usage(stderr, argv);
         fprintf(stderr, "*** Error in 'file structure' field\n");
         exit(1);
   }

   printf("- Vector ver %s\n", VERSION);
   printf(" Tags file: [%s]\n", argv[TAGFILE]);
   printf(" Input file: [%s]\n", argv[INPUT_FILE]);
   printf(" Output files: [%s].cf, .data, .teach, .reset\n", argv[BASEFILE]);

   printf(" I/O type: ");
   *inputType == averaged ? printf("averaged / ") : printf("binary / ");
   *outputType == averaged ? printf("averaged\n") : printf("binary\n");

   printf(" File structure: ");
   *fileStruct == localist ? printf("localist\n") : printf("distributed\n");
}

void OutputReset(FILE *resetFile, int vectorNum)
{
   fprintf(resetFile, "%d\n", vectorNum);
}

void CopyVector(float dest[], float source[], int vectorSize)
{
   int   index = 0;

   while (index < vectorSize) {
      dest[index] = source[index];
      index++;
```

```
   }
}

void MakeCfFile(FILE *cfFile, int vectorSize)
{
   fprintf(cfFile, "NODES:\n");
   fprintf(cfFile, "nodes = ***\n");
   fprintf(cfFile, "inputs = %d\n", vectorSize);
   fprintf(cfFile, "outputs = %d\n", vectorSize);
   fprintf(cfFile, "output nodes are <node-list>\n");
   fprintf(cfFile, "\nCONNECTIONS:\n");
   fprintf(cfFile, "groups = ***\n");
   fprintf(cfFile, "\nSPECIAL:\n");
}

void CloseFiles(FILE *inFile, FILE *cfFile, FILE *dvFile, FILE *tvFile, FILE *rvFile)
{
   fclose(inFile);
   fclose(cfFile);
   fclose(dvFile);
   fclose(tvFile);
   fclose(rvFile);
}

void FixDatFiles(FILE *dvFile, FILE *tvFile, FILE *rvFile, int numVectors, int numResets)
{
   fseek(dvFile, strlen(DISTFILETYPE), SEEK_SET);
   fseek(tvFile, strlen(DISTFILETYPE), SEEK_SET);
   fseek(rvFile, 0, SEEK_SET);

   fprintf(dvFile, "%d", numVectors);
   fprintf(tvFile, "%d", numVectors);
   fprintf(rvFile, "%d", numResets);
}

tokenList *ReadWholeVector(FILE *input, tokenList *index, tokenList **line, float vector[], int
vectorSize, bool *inFile)
{
   while (index == NULL && *inFile)
      index = *line = ReadLineTokens(input, inFile);

   if (index == NULL && !*inFile)
      return NULL;

   ResetVector(vector, vectorSize);

   AddToVector(vector, vectorSize, index -> token);
   index = index -> NEXT;

   do {
      while (index != NULL) {
         if (IsExtra(index))
            AddToVector(vector, vectorSize, index -> token + 1);
         else
            return index;

         index = index -> NEXT;
      }
      *line = DestroyTokenList(*line);
      index = *line = ReadLineTokens(input, inFile);
   } while (*inFile);

   return index;
}


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