Utilities

Carl Edward Rasmussen

January 18th 1996

1 openPredFile

Open a file for writing. Terminate program is opening fails. Return pointer to the open file.

⟨openPredFile 1⟩ ≡

```
FILE *openPredFile(char *name, char *instance)
{
  char df[10];
  FILE *fp;

  sprintf(df, name, instance);
  if ((fp=fopen(df, "w")) == NULL) {
    fprintf(stderr, "Could not open file \"%s\" for writing... bye!\n", df);
    exit(-1);
  }
  return fp;
}◇
```

Macro referenced in scrap 9.

2 createMatrix

Allocate space for a two-dimensional matrix. The matrix is implemented as an array of pointers to arrays. The entire array is allocated by a single call to malloc.

⟨createMatrix 2a⟩ ≡

```
real **createMatrix(int rows, int cols)
{
  int  i;
  real **x;

  x = (real **) malloc((size_t) rows*sizeof(real *));
  x[0] = (real *) malloc((size_t) rows*cols*sizeof(real));
  for (i=1; i<rows; i++) x[i] = x[i-1]+cols;
  return x;
}◇
```

Macro referenced in scrap 9.

3 square

⟨square 2b⟩ ≡

```
real sq(real x) { return x*x; } ◇
```

Macro referenced in scrap 9.

4 loadExamples

This section describes a function which reads example sets from ascii files into array structures. All attributes must have numerical values. The function is a utility that is used by many learning algorithms.

The function either reads inputs only, or both inputs and targets. The inputs and targets can either be read from the same file or from separate files. The number of inputs and targets need not be pre-specified; if they are, consistency with the input files is checked. If an error occurs, the function will exit with an error message.

⟨function head 2c⟩ ≡

```
void loadExamples(ex, numInp, numTar, df, df2)
  struct exampleset *ex;      /* examples will be returned in this structure */
  int  *numInp,                       /* specifies & returns number of inputs */
       *numTar;                      /* specifies & returns number of targets */
  char *df,                             /* name of file to read examples from */
       *df2;                 /* optional second file if targets are separate */
◇
```

Macro referenced in scrap 9.

Appropriate amounts of memory for the examples will be allocated in `ex` by the function. Any of the variables `ex->num`, `numInp` and `numTar` may be set to `-1` if their correct values are unknown to the caller; otherwise their values are checked. If a single file is read and both `numInp` and `numTar` are `-1` then a single target attribute is assumed. If `df2` is `NULL`, then only a single file is read; otherwise inputs are read from the `df` file and targets are read from the `df2` file.

$\langle$function body 3$\rangle \equiv$

> $\langle$open example files 4b$\rangle$
> $\langle$check syntax of first example file 5a$\rangle$
> $\langle$check syntax of target file 5b$\rangle$
> $\langle$check consistency of number of attributes 7a$\rangle$
> $\langle$allocate arrays 7b$\rangle$
> $\langle$read examples into arrays 7c$\rangle$ $\diamond$

Macro referenced in scrap 9.

The example values in the files must appear in ascii format, with the attributes ordered in an array; one column for each attribute and one line per example. Any kind of spacing is allowed between the columns (typically spaces, commas, semicolons, etc.) excluding the characters "new-line" and any of the characters `+-.0123456789` (for obvious reasons). The syntactic format of a number is the common form[1]:

$$[+ \mid -] \; \{ \; digit \; \} \; [\; . \; \{ \; digit \; \} \;] \; [\; (\; e \mid E \;) \; [+ \mid -] \; \{ \; digit \; \} \;]$$

The numbers are read by the low-level routine **`read_num`**, which takes an int "c" (character) and an input stream "df"; the last read character is returned in "c". Usually this routine is called with the value of "c" which was returned on the last invocation; this is done to avoid pushing back characters to the input. The function distinguishes three types of input: "new-line" characters, numbers and spacing symbols. When called it will read the remainder of the input of that type (plus one additional character), and if it was a number, its value is returned.

[1] Formally, this allows for silly numbers (like "-.e-") which contain no digits; these are all interpreted as zero.

⟨low-level reading function 4a⟩ ≡

```
static real read_num(int *c, FILE *fp)  /* private function for loadExamples */
{
  int  neg = 0, i = 0;
  real x = 0.0;

  switch (*c) {
  case '-':
    neg = 1;
  case '+':
    *c = getc(fp);
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
    while (*c >= '0' && *c <= '9') { x = 10.0*x+(*c-'0'); *c = getc(fp); }
  case '.':
    if (*c == '.') {
      *c = getc(fp);
      while (*c>='0' && *c<='9') { i++; x = 10.0*x+(*c-'0'); *c = getc(fp); }
      x /= pow(10.0, (double) i);
    }
    if (neg) x = -x; neg = 0;
    if (*c == 'e' || *c == 'E') {
      *c = getc(fp); i = 0;
      if (*c == '+' || *c == '-') { if (*c == '-') neg = 1; *c = getc(fp); }
      while (*c >= '0' && *c <= '9') { i = 10*i+(*c-'0'); *c = getc(fp); }
      if (neg) i = -i; x *= pow(10, (double) i);
    }
    return x;
  case '\n':
    *c = getc(fp);
    break;
  default:                                    /* discard uninteresting stuff */
    while ((*c < '0' || *c > '9') && *c != '.' &&  *c != '-' && *c != '+' &&
            *c != '\n' && *c != EOF) *c = getc(fp);
  }
} ◇
```

Macro referenced in scrap 9.

We need to check that we can find the files and open them for reading. If **df2** is **NULL** then everything is read from **df**.

⟨open example files 4b⟩ ≡

```
if ((fp=fopen(df, "r")) == NULL) {
  fprintf(stderr, "Could not open data file %s for reading... bye!\n", df);
  exit(-1);
}
if (df2 && ((fp2=fopen(df2, "r")) == NULL)) {
  fprintf(stderr, "Could not open data file %s for reading... bye!\n", df2);
  exit(-1);
} ◇
```

Macro referenced in scrap 3.

We need to check that we understand the syntax of the example files. The number of examples are counted (in the variable n) and if `numEx` is not `-1` then it is checked that the number of examples is correct. The number of attributes is counted (in the variable m) and it is checked that all examples have the right number of attributes.

⟨check syntax of first example file 5a⟩ ≡

```
c = ' '; n = 0;                    /* check syntax of the first example file */
while (c != EOF) {
  i = 0;
  read_num(&c, fp);
  while (c != EOF && c != '\n') {       /* "i" counts number of attributes */
    if (strchr(NUM, c)) i++;
    read_num(&c, fp);
  }
  if (n == 0) m = i;    /* if first line, then store number of attr in "m" */
  if (i == m)        /* check that line has the right number of attributes */
    n++;                                    /* if so, increment line count */
  else if (c != EOF || i != 0) { /* otherwise, print error message and die */
    fprintf(stderr,
           "Error while reading data file %s, line %d, token %d ...bye!\n",
            df, n+1, i);
    exit(-1);
  }
}
if (ex->num == -1)
  ex->num = n;
else if (ex->num != n) {
  fprintf(stderr,
          "Error: read %d examples from file %s; %d were expected ...bye!\n",
           n, df, ex->num);
  exit(-1);
}
rewind(fp);        /* rewind the file, so we are ready to read the examples */
◇
```

Macro referenced in scrap 3.

If the `df2` file is to be read, we do a similar check of syntax for this file. It is checked that the number of examples matches the number in the first example file. Also, if possible, it is checked that the number of target attributes is as expected.

⟨check syntax of target file 5b⟩ ≡

```
if (df2) {                    /* if it exists, check syntax of target file */
  c = ' '; n = 0;
  while (c != EOF) {
    i = 0;
    read_num(&c, fp2);
    while (c != EOF && c != '\n') {       /* "i" counts the number of attr */
      if (strchr(NUM, c)) i++;
      read_num(&c, fp2);
    }
```

```
    if (n == 0) m2 = i;     /* if first line, store number of attr in "m2" */
    if (i == m2)     /* check that line has the right number of attributes */
      n++;                                  /* if so, increment line count */
    else if (c != EOF || i != 0) {       /* otherwise, print error and die */
      fprintf(stderr,
           "Error while reading data file %s, line %d, token %d ...bye!\n",
           df2, n+1, i);
      exit(-1);
    }
  }
  if (n != ex->num) {
    fprintf(stderr,
          "Different number of examples in input and target file ...bye!\n");
    exit(-1);
  }
  if (*numTar == -1)
    *numTar = m2;
  else if (*numTar != m2) {
    fprintf(stderr,
       "Error: found %d target attributes in file %s; expected %d ...bye!\n",
       m2, df2, *numTar);
    exit(-1);
  }
  rewind(fp2);
} ◇
```

Macro referenced in scrap 3.

Finally, whenever possible, we check that the right number of attributes have been read, and set the variables **numInp** and **numTar** to appropriate values. First we handle the situation where the targets where in a separate file; then the more complicated situation where both inputs and (possibly) targets where in the same file. If both inputs and targets are read from the same file and both **numInp** and **numTar** are **-1**, then one target attribute is assumed.

⟨check consistency of number of attributes 7a⟩ ≡

```
if (df2) {
  if (*numInp == -1)
    *numInp = m;
  else if (*numInp != m) {
    fprintf(stderr,
        "Error: expected %d input attributes in file %s; found %d ...bye!\n",
        *numInp, df, m);
    exit(-1);
  }
}
else {                           /* everything was read from a single file */
  if (*numInp == -1) {
    if (*numTar == -1) *numTar = 1;           /* assume 1 target attribute */
    *numInp = m-*numTar;
  }
  else {
    if (*numTar == -1) {
      if ((*numTar = m-*numInp) < 0) {
        fprintf(stderr,
        "Error: found only %d input attr in file %s; expected >%d ...bye!\n",
          m, df, *numInp);
        exit(-1);
      }
    }
    else if (m != *numInp+*numTar) {
        fprintf(stderr,
             "Error: found %d attributes in file %s; expected %d ...bye!\n",
             m, df, *numInp+*numTar);
        exit(-1);
      }
  }
} ◇
```

Macro referenced in scrap 3.

Allocate space for the examples. The `target` array is only allocated if targets are actually read.

⟨allocate arrays 7b⟩ ≡

```
ex->inp = createMatrix(ex->num, *numInp);
if (numTar) ex->tar = createMatrix(ex->num, *numTar); ◇
```

Macro referenced in scrap 3.

Finally, we read the examples and store them in the arrays. The reading is again performed with the low-level reading routine. First we echo to `stderr` which files and how many numbers are being read.

⟨read examples into arrays 7c⟩ ≡

```
if (df2) {
  fprintf(stderr,
```

```
    "Reading %4d examples of %3d inputs from \"%s\"\n", ex->num, *numInp, df);
  fprintf(stderr,
  "                     and %3d target(s) from \"%s\"\n", *numTar, df2);
}
else {
  fp2 = fp;                     /* now both pointers point to the same file */
  fprintf(stderr,
      "Reading %4d examples of %3d inputs and %3d target(s) from \"%s\"\n",
       ex->num, *numInp, *numTar, df);
}
c = ' ';
for (i=0; i<ex->num; i++) {               /* read the examples into arrays */
  for (j=0; j<*numInp; j++) {
    while (!strchr(NUM, c)) read_num(&c, fp);
    ex->inp[i][j] = read_num(&c, fp);
  }
  for (j=0; j<*numTar; j++) {
    while (!strchr(NUM, c)) read_num(&c, fp2);
    ex->tar[i][j] = read_num(&c, fp2);
  }
}
fclose(fp); if (df2) fclose(fp2); ◇
```

Macro referenced in scrap 3.

The complete files ready for compilation is:

"util.h" 8 ≡

```
#include <stdio.h>
#define real double
struct exampleset { int num; real **inp; real **tar; };
extern FILE *openPredFile(char *name, char *instance);
extern real **createMatrix(int rows, int cols);
extern real sq(real x);
extern void loadExamples(struct exampleset *ex, int *numInp, int *numTar,
                         char *df, char *df2);
◇
```

"util.c" 9 ≡

```
#include <math.h>
#include <stdlib.h>
#include "util.h"

⟨openPredFile 1⟩
⟨createMatrix 2a⟩
⟨square 2b⟩
⟨low-level reading function 4a⟩
⟨function head 2c⟩
{
  FILE *fp, *fp2;
  char NUM[] = "+-0.123456789";
  int  i, j,                                        /* miscellaneous counters */
       m, m2, n,      /* counters for inputs, targets and number of examples */
       c;       /* holds characters from the input between calls to read_num */
  ⟨function body 3⟩
}
◇
```