Manual de uso de la librería data_handler.h

La librería data_handler.h permite al programador cargar datos de archivos con sufijo .csv a una estructura de tipo parser_result y manipularlos para poder ser usados dentro del código y ahorrar tiempo en la programación del preprocesado de datos en C. La propia librería se encarga del manejo de los ficheros de datos y la lectura de ellos para que el programador no deba tener en cuenta estos factores y facilitar la programación de código.

Esta librería forma parte del trabajo de fin de grado de Álvaro González Méndez, estudiante de ingeniería informática en Escuela Técnica Superior de Ingenieros Informáticos de la Universidad Politécnica de Madrid. El código y la documentación de esta se encuentra en el repositorio de GitHub: https://github.com/itsTwoFive/gml_nn.

Podemos dividir los métodos y operaciones de la librería en los siguientes campos:

- **Obtención de información sobre los datos** de un fichero .csv:
  - Obtención del numero de casos en una tabla
  - Obtención del numero de atributos en una tabla
- **Lectura de datos** contenidos en un fichero .csv:
  - Parseador que recoge los datos y los guarda
- **Tratamiento de datos** y **asignación de memoria dinámica** para estos:
  - Asignar memoria para almacenar datos
  - Sustitución de un valor por otro en toda la tabla
- **Barajado de datos** de forma aleatoria:
  - Dividir los datos barajados en dos sets
  - Discriminar datos de forma aleatoria
  - Separar en clases binarias datos dependiendo de un valor entero

El tipo de datos `parser_result` implementado en la librería es definido de la siguiente forma:

```
typedef struct{
    double **data_input;
    double **data_output;
    char **atrib_names;
    int num_case;
    int num_in;
    int num_out;
}parser_result;
```

Donde `data_input` es un array de arrays que almacena toda la información de los casos que tomamos como entrada de datos. La cadena de cadenas de reales `data_output` sería los resultados esperados para cada caso de la tabla. El parámetro `atrib_names` es un array de cadenas que almacena los nombres de los atributos de una tabla. El numero entero `num_case` guarda el número de casos que hay dentro de una tabla. Otro entero llamado `num_in` cuenta el número de atributos que son entrada de datos independientes y por último, `num_out` es el número de atributos que dependen de los independientes.

Además, el `data_handler` cuenta con la macro `LOG_PARSER` que muestra datos sobre la operación de lectura del fichero, si no se desea que aparezca por pantalla nada si se establece a 0 se desactiva.

Interfaz de uso (listado de métodos)

```
int get_number_cases(char filename[]);
```

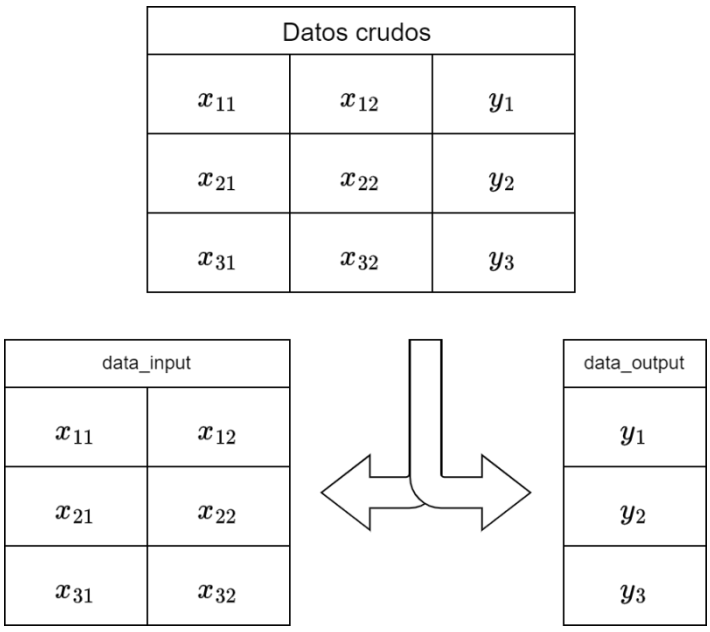
El método **get_number_lines** se encarga de abrir el fichero cuya dirección es `filename` y contar el número de casos contenidos en el archivo de valores separados por comas para después devolver la cuenta total. La rutina maneja todas las operaciones de entrada y salida de datos con los ficheros de forma que el programador no tiene que encargarse de abrir un fichero y cerrarlo. En caso de que el nombre del fichero sea erróneo o no exista en la dirección especificada se mostrara un mensaje de error por pantalla y se detendrá la ejecución del código.

```
int get_number_atributes(char filename[]);
```

La función **get_number_atributes** mantiene un comportamiento similar a **get_number_cases**, este método cuenta el número de columnas o atributos que tiene una tabla en el fichero con la dirección `filename` y lo devuelve en forma de entero. Al igual que la rutina **get_number_cases**, esta maneja todas las operaciones de entrada de datos, por lo que el programador no debe manejar la apertura ni el cerrado del fichero .csv. En caso de que el nombre del fichero sea erróneo o no exista en la dirección especificada se mostrara un mensaje de error por pantalla y se detendrá la ejecución del código.

```
parser_result parse_data(char filename[], int num_inputs);
```

La rutina **parse_data** lee `num_inputs` número de casos de la tabla del fichero en la dirección `filename` y guarda todos los casos en una estructura `parser_result` dividiendo cada fila en dos partes: una de datos independientes `data_input` que sirven como atributos de entrada para cada caso y otra de atributos derivados o deducidos de los independientes `data_output`, separando las entradas de las salidas para poder utilizar las listas de casos como conjuntos de datos de entrenamiento y de prueba. Los datos de entrada y salida de cada caso comparten el índice dentro de los dos arrays, de esta forma el programador puede asociar estos valores posteriormente al guardado en las dos lista. En la figura de abajo se explica de forma grafica cómo funciona esta separación:



El campo `atrib_names` del `parser_result` se llena con los nombres de los atributos indicados en la primera fila del fichero .csv, y los valores `num_case`, `num_in` y `num_out` también se rellenan respectivamente con el número de casos recogido, el número de datos independientes por caso y el número de datos derivados de los independientes o esperados. Devolviendo finalmente un solo `parser_result` con todos los datos y los parámetros que la estructura maneja.

```
double **array_alloc(int rows, int cols);
```

El método **`array_alloc`** crea una cadena de cadenas de números reales de tipo double, el numero de filas y columnas de las cadenas viene determinado por los parámetros `rows` y `cols` respectivamente. Este método usa memoria dinámica por lo que se debería después de usar las cadenas realizar un liberado de estas en memoria con el método `free` de la stdlib.h para prevenir desbordamientos en el heap que puedan causar comportamientos inesperados o el fallo del sistema.

```
parser_result random_trim(parser_result data,int size);
```

La rutina **`random_trim`** reduce una estructura `parser_result` `data` a una más pequeña de tamaño `size`, copiando el numero atributos de entrada `num_in` y de salida `num_out`. El método coje de forma pseudoaleatoria `size` número de casos guardados en la pareja de cadenas `data_input` y `data_output` de `data` y los introduce en un nuevo `parser_result`.

Cuidado al usar este método ya que puede duplicar varios casos. Recomendable cuando se trabaja con set de datos muy grandes y se quiere sacar una muestra pequeña de estos. Si se necesita estrictamente que no haya duplicados se puede usar el método **`data_div`** y escoger uno de los dos `parser_result` generados por este.

```
parser_result *data_div(parser_result in, int div_size);
```

La función **`data_div`** permuta y divide los datos leídos de un archivo en dos sets de tamaño `div_size` y (`num_cases - div_size`) permitiéndonos seleccionar `div_size` número de casos como set de entrenamiento y el resto de prueba. El barajado o permutación de los casos se hace antes de la separación y es una implementación del algoritmo de barajado de Fisher-Yates, ya que garantiza que no se dupliquen casos y, además, tiene una complejidad computacional lineal lo cual es asumible al trabajar con grandes sets de datos. El siguiente seudocódigo describe el comportamiento de Fisher-Yates siendo C un vector que contiene todos los casos:

1. Se da valor a $t = |C| - 1$
2. Mientras que $t > 0$:
    a. Se selecciona aleatoriamente un entero k entre 0 y t.
    b. Se cambia el valor de $C[k]$ por el de $C[t]$.
    c. $t = t - 1$.

Una vez reorganizados los datos se separan en dos `parser_result` y se devuelven en forma de array. El primero contendrá `div_size` número de casos y el segundo el resto de los contenidos originalmente en `in`.

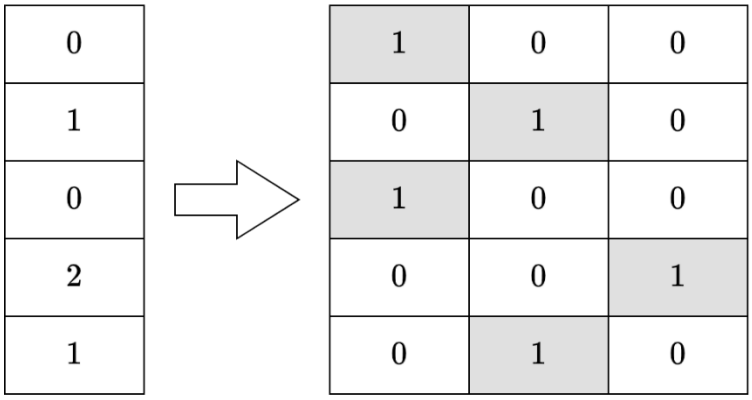
```
void change_all_values_for(double **data, int size_x, int size_y, double actual,
double new);
```

La rutina **`change_all_values_for`** recorre todos los valores almacenados en `data` y modifica los semejantes a `actual` al valor `new`, los parámetros `size_x` y `size_y` definen el tamaño de la array

de arrays. Para lidiar con los problemas que puede suponer la comparación de dos números en representación de coma flotante, se usa la macro `EPSILON_DISTANCE` definida en la cabecera por defecto igual a 10^{-5} que establece la distancia entre dos valores en coma flotante máxima para que ambos se consideren semejantes.

```
double **from_integer_to_binary_classes(double **list, int num_case, int
*num_classes);
```

El método **`from_integer_to_binary_classes`** permite transformar una vector vertical de datos enteros (reales que representan enteros) `list` que establecen la clase de un caso a una matriz que contenga tantas columnas como posibles casos y se marca con un 1 la clase a la que pertenece un caso. Es necesario declarar el número de casos que tiene la lista con el parámetro `num_case` para que funcione de forma correcta. La siguiente imagen muestra como transforma el array de arrays en otro separando las clases por columnas:



El método finalmente cambia el valor que se encuentra en el puntero `num_clases` al número de clases que existían en el vector principal.

Ejemplos de código

En esta sección se mostrarán varios ejemplos de uso de la librería con el fin de mostrar al programador que vaya a hacer uso de este código de ejemplo por el cual poder guiarse.

Programa 1. Averiguar el número de clases y de atributos en una tabla .csv

```c
#include <stdio.h>
#include <stdlib.h>
#include "data_handler.h"

int main(int argc, char const *argv[])
{
    /* Establecemos el nombre de fichero a leer */
    char filename[] = "FicheroDePrueba.csv";

    /* Obtenemos el numero de casos y atributos */
    int num_casos = get_number_cases(filename);
    int num_atrib = get_number_atributes(filename);

    /* Los mostramos por consola */
    printf("El fichero %s tiene %i casos con %i atributos\n"
        ,filename
        ,num_casos
        ,num_atrib);

    return 0;
```

```
}
```

Programa 2. Se leen los datos de un fichero y se dividen en set de entrenamiento y prueba

```c
#include <stdio.h>
#include <stdlib.h>
#include "data_handler.h"

int main(int argc, char const *argv[])
{
    /* Establecemos el nombre de fichero a leer */
    char filename[] = "FicheroDePrueba.csv";

    /* Leemos el archivo y guardamos los datos contenidos en pr */
    int num_inputs = get_number_atributes(filename)-1;
    parser_result pr = parse_data(filename,num_inputs);

    /* Dividimos los datos en dos set, de entrenamiento y prueba */
    parser_result* sets = data_div(pr,3);
    parser_result set_entrenamiento = sets[0];
    parser_result set_prueba = sets[1];

    /* ..... */

    return 0;
}
```

Programa 3. Se leen los datos de un fichero, se separan las clases de la columna de salida y se cambian todos los valores de las salidas que sean 0 por -1.

```c
#include <stdio.h>
#include <stdlib.h>
#include "data_handler.h"

int main(int argc, char const *argv[])
{
    /* Establecemos el nombre de fichero a leer */
    char filename[] = "FicheroDePrueba.csv";

    /* Leemos el archivo y guardamos los datos contenidos en pr */
    int num_inputs = get_number_atributes(filename)-1;
    parser_result pr = parse_data(filename,num_inputs);

    /* Separamos la columna final en casos */
    int num_clases = 0;
    double **separados =
       from_integer_to_binary_classes(pr.data_output,pr.num_case,&num_clases);

    /* Cambiamos el valor de 0 a -1 en todos los atributos de clase */
    change_all_values_for(separados,num_clases,pr.num_case,0.0,-1.0);

    /* Modificamos los valores del parser_result para utilizar los preprocesados */
    pr.data_output = separados;
    pr.num_out = num_clases;

    /* ..... */

    return 0;
}
```