# Data Shell

**Jesús Enrique Domínguez Barrios.**
**Rodrigo García Díaz.**

## ❖ Introducción.

La inteligencia artificial automatiza el aprendizaje y descubrimiento repetitivos a través de datos. La inteligencia artificial es diferente de la automatización de robots basada en hardware. En lugar de automatizar tareas manuales, la inteligencia artificial realiza tareas computarizadas frecuentes de alto volumen de manera confiable y sin fatiga. Para este tipo de automatización, la investigación humana sigue siendo fundamental para configurar el sistema y hacer las preguntas correctas. Se adapta a través de algoritmos de aprendizaje progresivo para permitir que los datos realicen la programación. Además, encuentra estructura y regularidades en los datos de modo que el algoritmo adquiere una habilidad: el algoritmo se convierte en un clasificador o predictor.

Para ello, los datos que requiere deben ser los precisos y adecuados para el contexto determinado para lo que se esté aplicando la IA.

## ❖ Descripción del sistema.

Para este ejercicio se solicitó la implementación de un DataShell. Como se menciona en la introducción, este sistema llega a ser de suma importancia cuando se planea el manejo de grandes cantidades de datos para su análisis bajo una estricta generalidad.

Los algoritmos que usaremos más adelante requieren recibir los datos bien configurados y asignados para un correcto aprendizaje y correctas predicciones y recomendaciones, por lo mismo, la captura de datos provenientes de archivos es algo que nuestros algoritmos no conocerán de manera preliminar, por ello se busca mediante la implementación de este DataShell el capturar y configurar de forma correcta y general diversas cantidades y tipos de datos.

Con lo anterior podemos asegurar que los datos requeridos por los futuros algoritmos serán los correctos con los cuales dicho proceso podrá trabajar.

Para ello, el sistema requerido es capaz de leer *n* líneas de datos de un archivo csv con *m* cantidad de datos separados por una coma. Así, al final de la captura, el DataShell deberá entregar una matriz de apuntadores a arreglos, en donde cada posición “contiene” cada línea que se encuentra en el archivo csv proporcionado por el usuario.

- **Metodología de desarrollo.**

Primeramente, se planteó la validación del nombre del archivo que se debía recibir por la terminal. En este caso, el diseño del sistema está hecho para que acepte únicamente archivos con extensión .csv, sin embargo, con una modificación al módulo de validación puede aceptar otro tipo de extensiones de archivo para que no se vea limitado a solo archivos con la extensión mencionada.

La manera en la que la validación funciona es sencilla. Supongamos un arreglo que el usuario ingresa la cadena *data.csv*, en este primer caso, la extensión es correcta. En nuestro sistema este dato anterior se vería de la siguiente manera:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| d | a | t | a | . | c | s | v |

La validación encuentra el primer punto de la cadena y revisa que las posiciones +1, +2 y +3 a partir de dónde se encontró el punto correspondan con los caracteres ‘c’, ‘s’ y ‘v’ respectivamente.

Si la condición se cumple, se sigue con el flujo del programa. Sin embargo, caemos en un problema: cuando no hay punto la validación no se cumple nunca.

Para solucionar el problema anterior, usamos la misma variable que usamos para guardar la posición del punto. Esto más adelante se descubrió que sirve para solucionar dos problemas: 1) Al tener iniciada la variable de la posición del punto en cero, si no encuentra ningún punto, es decir, la cadena ingresada no contiene una extensión y 2) Si se encuentra un punto en la posición cero del arreglo, quiere decir que no hay nada a la izquierda del punto, es decir, no hay nombre del archivo, por lo que de igual manera es un error. Con esta validación se cumplen esos dos problemas.

Ahora bien, supongamos que el usuario es un vándalo e ingresa la cadena ‘data..csv’, sea por error de dedo o a propósito. ¿Qué sucede con este caso? Ya que técnicamente la extensión está bien. El sistema en cuestión no acepta ese tipo de archivos, ya que SIEMPRE se queda con la posición del primer punto, por lo que al revisar la posición +1, +2 y +3, se encontrará con un ‘.’, ‘c’ y ‘s’, por lo que la validación principal no lo acepta y termina el programa.

Si se pasó la validación, se ingresa a una segunda validación que comprueba si el archivo se puede abrir. Si no lo hace, ya sea porque el archivo no existe o por algún error interno, el sistema se suspende.
En cambio, si el archivo se puede abrir, se comienza con el proceso de lectura del archivo.

Para la lectura, se empleó la función *fscanf* que nos permite la lectura de un formato en específico. De acuerdo a la teoría del DataShell, se lee una cadena, sin importar que los datos sean numéricos, hasta encontrar un salto de línea. La separación de estos datos se llevará a cabo en unos párrafos más en otros módulos dedicados a eso.
El siguiente paso es importante ya que, normalmente, los datos de un archivo csv vienen separados por comas, sin embargo, la forma en la que se *tokenizarán* más adelante nos pide estrictamente que estén separados por espacios. Para ello, se empleó un módulo que recorre la línea leída y cambia todas las comas que encuentre por espacios. Lo anterior garantiza la correcta separación de los datos más adelante.

El siguiente paso es *tokenizarlos*, es decir, separarlos de la cadena para convertirlos en, por fin, datos numéricos separados. Aunque suene fácil, es un tanto complejo, así que se explicará de la mejor manera posible. Primero que nada, cabe mencionar que todos los datos leídos del archivo dado serán tratados como datos de tipo *float* para asegurar una uniformidad de los mismos y disminuir la carga de trabajo que conlleva analizar si es entero, caracter, etc.
Para esta funcionalidad se empleó la función *strtof*, o *string to float*. Esta función analiza la cadena dada e interpreta su contenido como un número de punto flotante y, a su vez, devuelve ese valor como *float*.
Mencionamos que era importante cambiar las comas por espacios ya que la forma en la que funcionan las funciones de esta familia es que descartan tantos espacios en blanco encuentren hasta encontrar un caracter. El problema, es que encontrando la coma la función no sabe cómo proceder al intentar interpretar ese caracter como un número, por ello se optó por entregarle la cadena bien configurada. Una vez descartados los espacios y encontrado un caracter que puede analizar, analiza todos los siguientes caracteres válidos hasta encontrar otro espacio, cuando hace esto, retorna el valor leído hasta ese punto y asigna un apuntador en donde se quedó el análisis de la cadena. Luego, repitiendo el procedimiento hasta la longitud de la cadena, usando los mismos apuntadores, el proceso se repetirá tantas veces como datos numéricos haya en la cadena leída del archivo.

Cabe mencionar que para el almacenamiento de estos datos leídos se usó un arreglo de *float* que almacena cada dato leído de la cadena y que es arrojado tras en análisis de *strtof*. Este arreglo, a su vez, será retornado a la posición n de un índice que irá llenando un arreglo de apuntadores, donde sí, cada posición apuntará al arreglo creado por nuestro módulo donde se separan y convierten los datos. Como comentario adicional, las dimensiones de estos arreglos corresponden a la macro BUFSIZ (buffer size) de C, así que la matriz final mide BUFSIZ x BUFSIZ.

Técnicamente, con el módulo anterior se acaba el programa. Sin embargo, hay un bloque de código extra que solo sirvió para la comprobación de la matriz. Se trata de un par de *fors* que ayudan a la impresión de esta matriz. En el futuro será removido.

- **Pseudocódigos importantes.**

```
WHILE (i < largo de la cadena AND puntos == 0)
        IF(nombre del archivo [i] == .)
                posición del punto = i
                puntos aumenta en 1
        i aumenta en 1
IF (posicion del punto == 0)
        RETURN -1
ELSE
        IF (nombre del archivo[posicion del punto + 1] == c AND nombre del
        archivo[posicion del punto + 2] == s AND nombre del archivo[posicion del punto +
        3] ==v)
                RETURN 1
        ELSE
                RETURN 0


buffer = inicializar con malloc
WHILE (ptr != linea que se leyó + largo de la linea que se leyó) //Así el proceso continuará
        temporal = strtof de la cadena
        buffer[contador] = temporal
        contador aumenta en 1
```

RETURN buffer

archivo = abrir con fopen
IF (archivo es NULL)
    PRINT(Error al abrir archivo)
    EXIT
ELSE
    RETURN archivo

/*El siguiente módulo soluciona un problema con fgets al leer del bash. Al leer también el salto de línea, la apertura del archivo usando esa cadena leída siempre devuelve un error ya que al final de la extensión se encuentra este caracter, por ello, buscamos reemplazarlo con el caracter que finaliza una cadena*/
longitud = largo del nombre del archivo - 1
IF (nombre del archivo [longitud] == '\n')
    nombre del archivo [longitud] = '\0'

- **Código.**

- **DataShell.h**

```
//
//  DataShell.h
//
//
//  Creado por Rodrigo Garcia Diaz y Jesus Enrique Domìnguez el 27 de octubre del 2020.
//

#ifndef DataShell_h
#define DataShell_h


/*
* System headers required by the following declarations
* (the implementation will import its specific dependencies):
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
/*
* Application specific headers required by the following declarations
* (the implementation will import its specific dependencies):
*/

/* Constants declarations. */

/* Set EXTERN macro: */

#ifndef DataShell_IMPORT
   #define EXTERN
#else
   #define EXTERN extern
#endif

/* Types declarations. */


/* Global variables declarations. */


/* Function prototypes. */


/*
 *
  * La funcion vista_Menu pedira al usuario el nombre del archivo que desea ingresar al sistema para ser tokenizado y
 * formateado
 *
 * @params
 *      void

 * @returns
       none
*/


EXTERN void vista_Menu(void);



/*
 *
 * La funcion controlador_Proceso controlarà las llamadas a las funciones y las variables
 *
 * @params
 *     archivo (char *):
          nombre del archivo que se abrirà para el correspondiente proceso
```

```
 * @returns
       void
*/


EXTERN void controlador_Proceso(char * archivo);


/*
 *
 * La funcion modelo_Abre_Archivo abrirà el archivo determinado con 'nombre_archivo' en modo de lectura.
 *
 * @params
 *     nombre_archivo (char *):
          nombre del archivo que se abrirà para el correspondiente proceso
 * @returns
       Archivo abierto en el modo de lectura
*/


EXTERN FILE * modelo_Abre_Archivo(char * nombre_archivo);


/*
 *
 * La funcion modelo_Valida_Nombre validarà si el nombre de archivo ingresado por el usuario es correcto.
 *
 * @params
 *     nombre_archivo (char *):
          nombre del archivo que se validarà
 * @returns
       Entero de confirmaciòn de si es correcto o no
*/


EXTERN int modelo_Valida_Nombre(char * nombre_archivo);


/*
 *
 * La funcion modelo_Correcion_Nombre cambiarà el \n leido por fgets por un \0
 *
 * @params
 *     nombre_archivo (char *):
          nombre del archivo a corregir
 * @returns
       none
*/
```

```
EXTERN void modelo_Correccion_Nombre(char * nombre_archivo);



/*
 *
 * La funcion modelo_Tokenizer separara la cadena en cada coma.
 *
 * @params
 *     linea (char *):
       cadena que va a separar
 * @returns
       puntero con al arreglo de datos separados para ser metidos a la matriz de datos.
*/


EXTERN float * modelo_Tokenizer(char * linea, float * buffer);


/*
 *
 * La funcion modelo_Columnas nos permite calcular la cantidad de columnas presentes en el archivo leido.
 *
 * @params
 *     linea (char *):
       cadena con la que se calcularàn las columnas presentes
 * @returns
       cantidad de columnas en el archivo
*/


EXTERN size_t modelo_Columnas(char * linea);



/*
 *
 * La funcion vista_Error_Menos1 mostrarà un mensaje de error indicando que deberìa hacer para corregirlo.
 *
 * @params
 *     none
 * @returns
       none
*/
```

```
EXTERN void vista_Error_Menos1(void);


/*
 *
  * La funcion vista_Error_Apertura_Archivo mostrarà un mensaje de error indicando que deberìa hacer para corregirlo.
 *
 * @params
 *     none
 * @returns
       none
*/


EXTERN void vista_Error_Apertura_Archivo(void);

/*
 *
 * La funcion controlador_linea quita las comas de la linea y los sustituye por espacios.
 *
 * @params
 *fila (char *):
       cadena que se sustituira las comas.
 * @returns
       apuntador a al arreglo que se edito.
*/


EXTERN char *controlador_linea(char fila[]);


#undef DataShell_IMPORT
#undef EXTERN


#endif /* DataShell_h */
```

- **DataShell.c (contiene main)**

```
#include "DataShell.h"

int main (void)
{
  vista_Menu();

  return 0;
}
```

- **controlador_Proceso.c**

```
#include "DataShell.h"

void controlador_Proceso(char * archivo)
{
 size_t count = 0, columns;

 int i,j;
 int validacion_nombre;
 FILE * archivo_lectura;

 char linea_leida[BUFSIZ];

 float * Matriz_Datos[BUFSIZ];
 float * float_ptr;

 float * buffer;

 //Corregimos el '\n' que se lee con fgets
 modelo_Correccion_Nombre(archivo);

 //Validamos si el nombre es correcto.
 validacion_nombre = modelo_Valida_Nombre(archivo);


 if(validacion_nombre == 0 || validacion_nombre == -1)
 {
  vista_Error_Menos1();
 }

 else//Si la validacion salio correctamente, podemos continuar con el proceso.
 {
  archivo_lectura = modelo_Abre_Archivo(archivo);

  while(!feof(archivo_lectura))
  {
   //Leemos una linea del archivo.
   fscanf(archivo_lectura,"%s\n",linea_leida);

   //Adecua la cadena para que sea tokenizada
   strcpy(linea_leida, controlador_linea(linea_leida));

   //La separamos
   Matriz_Datos[count] = modelo_Tokenizer(linea_leida,buffer);

   count++;

   free(buffer);
  }
```

```
    fclose(archivo_lectura);

    //Calculamos la cantidad de columnas ayudàndonos con la ultima linea leida.
    columns = modelo_Columnas(linea_leida);

    printf("%zu columns and %zu rows read\n\n",columns,count);

        //EL siguiente bloque de còdigo es para pura verificacion, se borrara en futuras implementaciones.
    for(i=0 ; i<count ; i++)
    {
      float_ptr = Matriz_Datos[i];

      for(j=0 ; j<columns; j++)
      {
           if(j==0)
           {
            printf("%.0f -> ",float_ptr[j]);
           }

           else
           {
            printf("%f ",float_ptr[j]);
           }
      }

      printf("\n");
    }

  }
}
```

- **modelo_Archivos.c**

```
#include "DataShell.h"

FILE * modelo_Abre_Archivo(char * nombre_archivo)
{
  FILE * archivo;

  archivo = fopen(nombre_archivo,"r");

  if (archivo == NULL)
  {
    vista_Error_Apertura_Archivo();
    exit(1);
  }

  else
  {
```

```
    return archivo;
  }
}
```

- **modelo_Columnas.c**

```
#include "DataShell.h"

size_t modelo_Columnas(char * linea)
{
  size_t count_data = 0;
  char * ptr = linea;
  float temp;

  while(ptr != linea + strlen(linea))
  {
    temp = strtof(ptr,&ptr);

    count_data++;
  }

  return count_data;

}
```

- **modelo_Correcion_Nombre.c**

```
#include "DataShell.h"

void modelo_Correccion_Nombre(char * nombre_archivo)
{
  size_t ln = strlen(nombre_archivo)-1;

  if(nombre_archivo[ln] == '\n')
  {
    nombre_archivo[ln] = '\0';
  }
}
```

- **modelo_Tokenizer.c**

```
#include "DataShell.h"

float * modelo_Tokenizer(char * linea, float * buffer)
{
  size_t count_data = 0;
  char * ptr = linea;
  float temp;

  buffer = malloc(sizeof(float)*BUFSIZ);

  while(ptr != linea + strlen(linea))
  {
```

```
    temp = strtof(ptr,&ptr);

    buffer[count_data] = temp;

    count_data++;

  }

  return buffer;

}
```

- **modelo_Valida_Nombre.c**

```
#include "DataShell.h"

int modelo_Valida_Nombre(char * nombre_archivo)
{
  int i = 0, pos_punto = 0, puntos = 0;

  while( i < strlen(nombre_archivo) && puntos == 0)
  {
    if(nombre_archivo[i] == '.')
    {
      pos_punto = i;
      puntos++;
    }

    i++;
  }

    /*Revisando quedandonos siempre con el primer punto encontrado, no importa cuando puntos se ingresen, siempre estarà mal.
    Independientemente de si la extension està bien.
     Del mismo modo, si no ingresa ningun punto o si solo coloca la extension sin el nombre, el programa serà detenido.*/

  if(pos_punto == 0) // En este caso, se encontrò el punto en la primer posicion de la cadena o no se encongrò ningun punto.
  {
    return -1;
  }

  else//Si no hubo error minus 1, deberemos checar si la extension es correcta.
  {
     if(nombre_archivo[pos_punto + 1] == 'c' && nombre_archivo[pos_punto + 2] == 's' && nombre_archivo[pos_punto + 3] == 'v')
    {
      return 1;
    }
```

```
    else
    {
      return 0;
    }
  }
}
```

- **modelo_sincomas.c**

```
#include "DataShell.h"

char *controlador_linea(char fila[]){
  int i=0;

  for(i=0; i < strlen(fila); i++){
    if(fila[i] == ',')
      fila[i] = ' ';
  }

  return fila;
}
```

- **vista_Errores.c**

```
#include "DataShell.h"

void vista_Error_Menos1(void)
{
  printf("\n\nIngrese un archivo correcto.\n");
  printf("\n\t> Extensiòn '.csv'.\n");
  printf("\t> No mas de un punto.\n");
  printf("\t> Un nombre valido antes de la extension.\n\n");
}

void vista_Error_Apertura_Archivo(void)
{
  printf("\n\nHubo un error. Intente de nuevo.\n");
  printf("\n\tPosibles causas:\n");
  printf("\t> El archivo que especifico no existe.\n");
  printf("\t> Error desconocido.\n\n");
}
```

- **modelo_Vista_Menu.c**

```
#include "DataShell.h"

#define TAMANO 50

void vista_Menu()
{

  char nombre_archivo[TAMANO];
```

```
  printf("\n\n\tHola y bienvenido.");
  printf("\n\nNombre del archivo: ");
  fgets(nombre_archivo,TAMANO,stdin);

  controlador_Proceso(nombre_archivo);
}
```

- **Referencias.**


- cplusplus.com. (s. f.). strtof - C++ Reference. Recuperado 7 de noviembre de 2020, de http://www.cplusplus.com/reference/cstdlib/strtof/
- SAS. (s. f.). Inteligencia artificial – Qué es y por qué es importante. Recuperado 8 de noviembre de 2020, de https://www.sas.com/es_mx/insights/analytics/what-is-artificial-intelligence.html#:%7E:text=%C2%BFPor%20qu%C3%A9%20es%20importante%20la,repetitivos%20a%20trav%C3%A9s%20de%20datos.&text=En%20lugar%20de%20automatizar%20tareas,manera%20confiable%20y%20sin%20fatiga.