UNIVERSIDAD DE GRANADA

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

DECSAI

Metodología de la Programación

Curso 2022/2023

Guion de prácticas

Language5

Práctica final

Junio de 2023

Índice

1. Definición del problema

En esta práctica final vamos a desarrollar una aplicación sobre ficheros de texto que nos permita averiguar automáticamente el idioma en el que está escrito un determinado texto. Tanto los textos de partida de idiomas conocidos como el texto de idioma desconocido, son documentos textuales plenamente legibles, (`*txt`), en contraste con los ficheros de language, (`*bgr`), que contienen los bigramas y las frecuencias encontradas en el texto fuente (o los textos fuentes) del (los) que procede(n).

Para realizar la práctica se deben implementar tres programas que se podrán ejecutar de forma independiente. La ejecución combinada de todos ellos logrará nuestro objetivo inicial de predicción del idioma de un texto de un idioma desconocido. Las tareas están modularizadas de modo que, las salidas de un programa podrán ser entradas de otro y sus salidas entradas para el siguiente. Los programas son: **learn**, **joinLanguages** y **classify**.

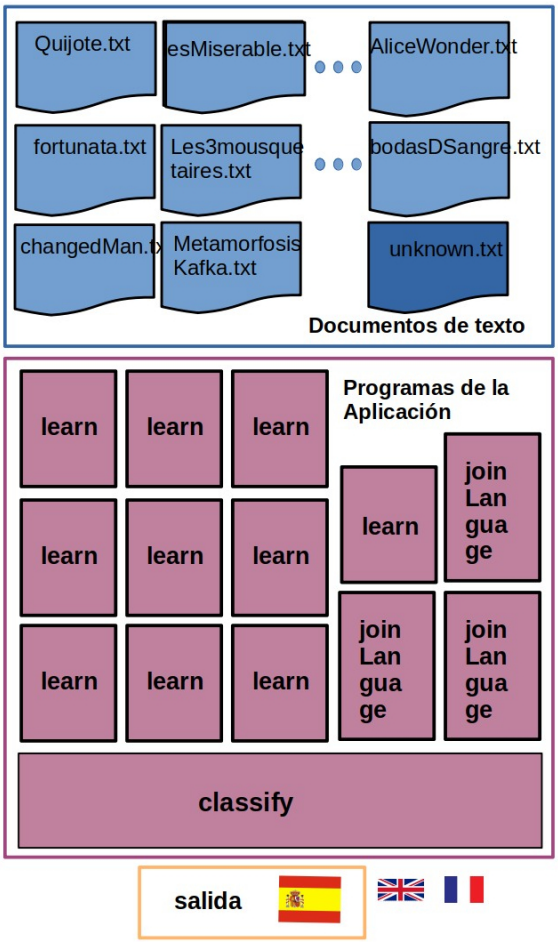

Figura 1: Flujo de entrada salida de los programas

En la figura 1 podemos identificar: los documentos de texto de entrada, las ejecuciones del software a desarrollar para la realización de una clasificación completa, por último, podemos identificar como salida la predicción final, el idioma más adecuado para el documento `unknown.txt`,

seleccionado entre los idiomas candidatos. En azul podemos reconocer una lista de documentos de texto escritos en su idioma original [1] que se van a utilizar como entrada. En color malva, podemos observar las ejecuciones sucesivas de cada uno de los programas de la aplicación que van a ser necesarias para llevar a cabo la predicción del idioma español para el documento `unknown.txt`. Pero, vamos a detallar cada etapa del proceso.

El primer paso, consiste en utilizar el programa **learn**. Dicho programa construye un fichero language (`*bgr`) a partir de un documento de texto (`*txt`), contabilizando las frecuencias de todos los bigramas que se han hallado en el texto fuente. En la figura 2 podemos ver cómo se aplica **learn** sobre diversos documentos de textos clásicos escritos en diferentes idiomas. Así pues, mediante este programa obtendremos tantos ficheros languages como documentos de texto de partida.

No obstante, se pueden crear languages enriquecidos, una suerte de language aglutinado, que integre varios ficheros languages de un mismo idioma, obtenido por fusión [2]. Así, podremos obtener un fichero language más representativo para el español como fusión de textos de diferentes épocas como por ejemplo: Quijote + Fortunata.

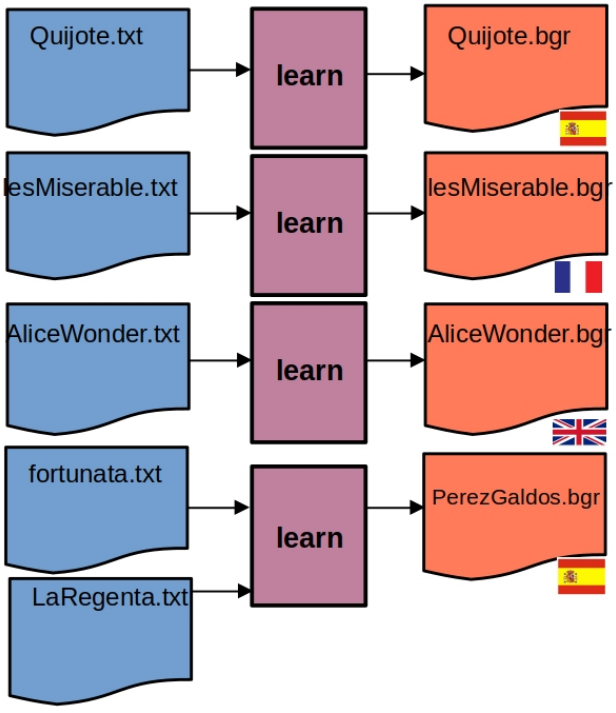

Figura 2: Ejecución del programa **learn**.

El segundo paso consiste en utilizar el programa **joinLanguages**. Dicho programa se encarga de unir en un solo fichero language, los bigra-

[1] Los ficheros que mencionamos los puede encontrar en Books, donde se ha puesto a disposición algunos clásicos de la literatura en diferentes idiomas.

[2] Tarea ya realizada en una práctica anterior, Language2.

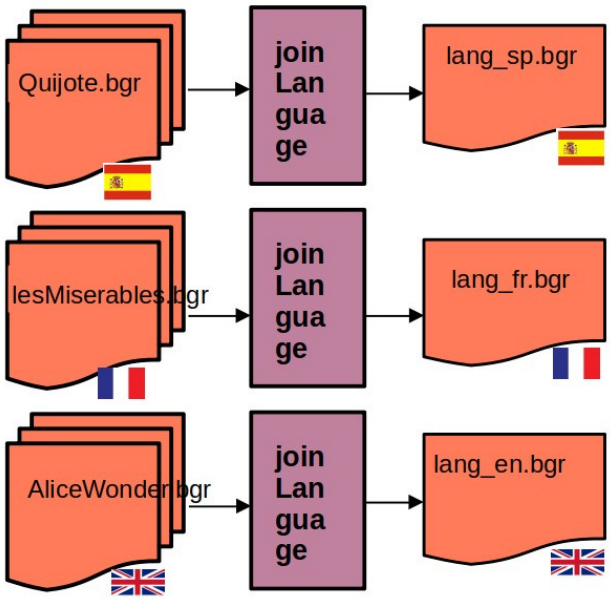

Figura 3: Ejecución del programa **joinLanguages**.

mas y las frecuencias de los languages proporcionados para la fusión. Toma como entrada un conjunto de ficheros de language, ignora aquellos que no se corresponden a un mismo idioma y da como salida un fichero como `lang_nombreIdioma.bgr`. La discriminación de los language con idéntico idioma se hace con la lectura de las cabeceras de los ficheros languages, donde el `id` del idioma se encuentra en la segunda línea.

En la figura 3 se observa como **joinLanguages** se ejecuta una vez para cada uno de los languages de referencia con los que queramos contrastar como español, francés e inglés respectivamente `lang_sp.br`, `lang_fr.br` y `lang_en.br`.

Por último, el tercer programa, **classify**, figura 4, toma como entrada: un documento de texto en un idioma desconocido y un conjunto de ficheros language de referencia, que representan los idiomas candidatos a ser asignados al documento de texto. La salida del programa se hace por pantalla, y nos informa del idioma más plausible en el que está escrito el fichero de texto.

2. Arquitectura de las prácticas

Como ya sabemos, la práctica `Language` se ha diseñado por etapas, las primeras con clases más sencillas, sobre las que se asientan otras más complejas. Este será el caso de la práctica final dónde aparece la última clase necesaria y se revisitan las clases anteriores.En la Figura 5 identificamos los cambios a realizar en `Language5`. Aparece la clase nueva `BigramCounter` y se añaden algunas funcionalidades nuevas a las clases `Bigram, BigramFreq` y `Language`, con la finalidad de permitir usar los objetos de las respectivas clases como tipos elementales, mediante la sobrecarga de operadores.

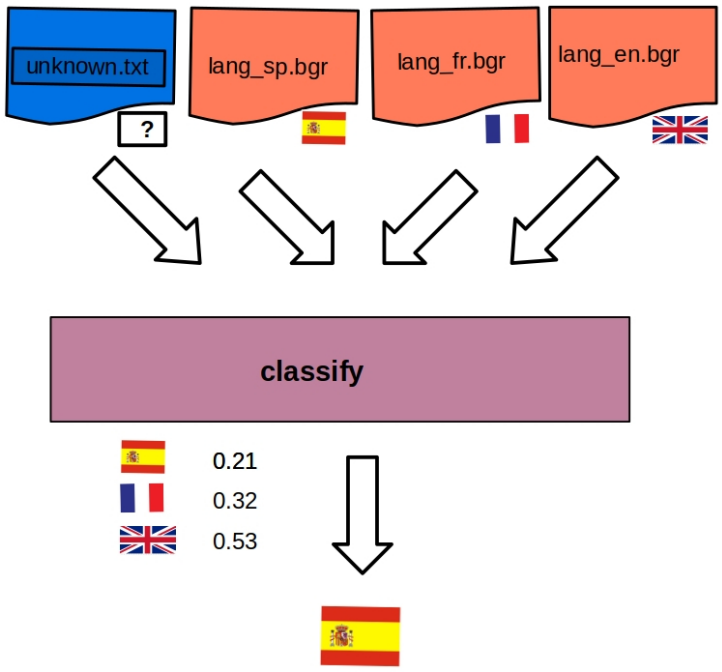

Figura 4: Ejecución del programa **classify**.

D BigramCounter.cpp
: Implementa la clase BigramCounter, la estructura que va a permitir alojar una matriz bidimensional en memoria dinámica; nos será de utilidad para llevar a cabo el conteo de bigramas de forma eficiente para aprendizaje de language.

Además de la aportación principal de la clase `BigramCounter` se van a revisitar las clases anteriores.

A' Bigram.cpp
: Implementa la clase Bigram, con un c-string, incorporando algunos métodos adicionales, como la sobrecarga de operadores **[]**, << y >> .

B BigramFreq.cpp
: Implementa la clase BigramFreq una composición de un bigrama y un entero para el registro de la frecuencia de un bigrama. Se incorporan algunos métodos adicionales, como la sobrecarga de operadores << y >> y todos los operadores relacionales **<**, **<=**, **==**, **>**, **>=**, **!=**.

C' Language.cpp
: Implementa la clase Language, una estructura para almacenar las frecuencias de un conjunto de bigramas utilizando memoria dinámica. Se incorporan algunos métodos adicionales, como la sobrecarga de operadores **[]**, << y >>. Se refactorizan los métodos `load()` y `save()` para que hagan uso de << y >>, además deben extenderse para admitir ficheros binarios. Refactorizar el método `sort` para que use alguno de los operadores relacionales recién definidos en Language. Por último, se incorpora la sobrecarga del operador += con

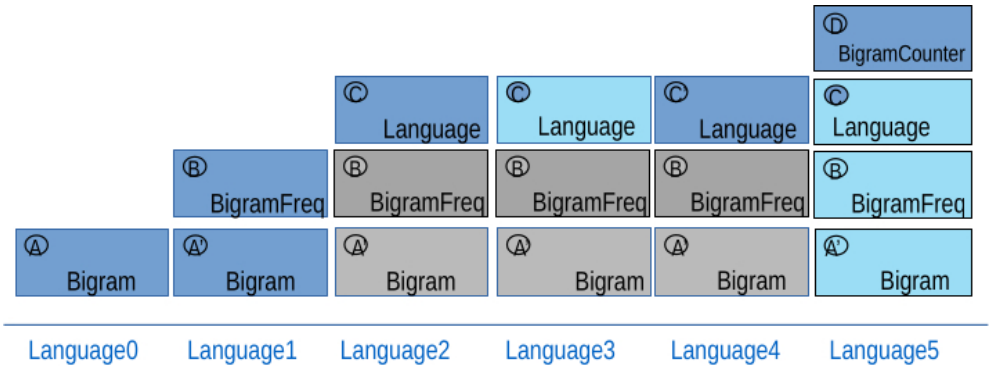

Figura 5: Arquitectura de las prácticas de MP 2023. Los cambios esenciales en las clases (cambio en estructura interna de la clase) se muestran en azul intenso; los que solo incorporan nuevas funcionalidades en azul tenue. En gris se muestran las clases que no sufren cambios en la evolución de las prácticas.

un Language como parámetro. Eso hace que ya no sean necesarios los métodos `join()` ni `append()` [3].

Este trabajo progresivo, se ha planificado en hitos sucesivos y podrá comprobar que parte del contenido de esta práctica final coincide con prácticas anteriores, por lo que podrá reutilizar material ya elaborado y depurado, así el tiempo para su elaboración se verá reducido de forma significativa.

3. Objetivos

El desarrollo de la práctica Language5 persigue los siguientes objetivos:

- Practicar con punteros y memoria dinámica dentro de una clase para la implementación de una matriz 2D.
- Profundizar en el desarrollo de los métodos básicos de una clase con memoria dinámica: constructor de copia, destructor y operador de asignación.
- Practicar con herramientas como el depurador de NetBeans y `valgrind` para rastrear errores en tiempo de ejecución.
- Practicar la sobrecarga de operadores tanto monarios como binarios.
- Elaborar un proyecto con múltiples `main()`.
- Practicar con ficheros binarios para su lectura y escritura.

[3] Son métodos obsoletos que no se deben usar, (deprecated) poner entre comentarios.

4. Programa learn. Aprendizaje de un language a partir de un conjunto de documentos de texto

A partir de un documento de texto o de varios, el programa **learn** genera un fichero language de salida de idioma `nombreIdioma` que contiene la lista de los bigramas diferentes junto con sus frecuencias hallados en la lista de documentos de la entrada. Cada bigrama se obtiene a partir de cualesquiera dos caracteres que aparezcan juntos dentro de una palabra del documento de texto de la entrada.

4.1. El contador de bigramas

Se propone crear la última clase para resolver el problema del aprendizaje. En la sección A puede encontrar la declaración de la clase `BigramCounter`.

Esta clase encapsula una matriz 2D dinámica de enteros, con tantas filas y tantas columnas como caracteres válidos tengamos en cualquier idioma, de forma que cada entero guarda la frecuencia del bigrama determinado por los dos caracteres que representa la fila y columna correspondiente. Así, la frecuencia del bigrama ”de“ estará almacenada en la fila que corresponda a ”d“ y en la columna que corresponda a ”e“. El número de filas y columnas de esta matriz bidimensional viene determinado por el número de caracteres válidos que se van a considerar; una matriz cuadrada de tamaño $n * n$, donde n es el número de caracteres válidos considerados.

Los métodos de esta clase permiten:

1. Realizar el conteo de frecuencias de forma eficiente mediante la indexación por caracteres de un bigrama.

2. Traducir la matriz bidimensional a una forma más compacta, es decir, a un objeto de tipo **Language**, ordenando previamente las frecuencias de mayor a menor y eliminando aquellos bigramas con frecuencia nula.

Vamos a ver cómo se utiliza la matriz para llevar a cabo el conteo. Para cada par de caracteres consecutivos en el documento de entrada, estos pueden ser letras, dígitos, separadores (blanco, tabulador, salto de línea,) o símbolos de puntuación (comas, puntos, puntos y comas, etc.) deberemos comprobar si forman un bigrama válido. En este sentido, se dice que un bigrama es válido si está compuesto de dos caracteres ambos pertenecients al conjunto de caracteres válidos. Tal como hiciéramos en `Language0`.
Por ejemplo, supongamos que $validos = \{\texttt{abcdefghij}\}$. Así, si el contenido del fichero es el siguiente:

```
gafa: -fachada-, hija.
```

identificaremos los bigramas: `ga af fa fa ac ch .... ja`

Para cada bigrama, par de caracteres válidos, debemos de calcular la frecuencia de aparición. Para facilitar este proceso vamos a almacenar las frecuencias de cada bigrama en una matriz bidimensional, `F`, de tamaño $n \times n$, siendo n el número de caracteres válidos. En la tabla siguiente podemos ver un esquema de la matriz asociada a nuestro ejemplo, con $n = 10$. Inicialmente, todas las posiciones de la matriz tendrán el valor cero.

En la posición $F[i][j]$ se almacenará la frecuencia con la que aparece el bigrama que tiene como primer carácter el elemento (i-1)-ésimo y como segundo carácter el elemento (j-1)-ésimo en el conjunto $validos$, respectivamente [4]. Así, el bigrama `ab` se almacena en la posición $F[0][1]$ y el bigrama `ch` se almacena en la posición $F[2][7]$. Por tanto, dado un nuevo bigrama, podemos actualizar su frecuencia fácilmente si conocemos las posiciones de los caracteres en el conjunto $validos$ e incrementamos en uno su valor.

F	a 0	b 1	c 2	d 3	e 4	f 5	g 6	h 7	i 8	j 9
a 0	0	2	3	0	3	2	6	8	9	0
b 1	3	2	6	8	9	0	3	0	3	2
c 2	1	2	0	0	3	0	4	1	0	0
..	.....									
j 9	4	0	0	2	6	8	5	2	0	0

Una vez que se han procesado todos los ficheros de entrada, podemos salvar el fichero language de salida transformando esta tabla en un vector de bigramas, que será ordenado por frecuencia de aparición en orden decreciente salvándolo en el fichero de salida correspondiente.

4.2. Ejecución de learn

La sintaxis para ejecutar el programa **learn** es como sigue:

```
learn [-t|-b] [-l nombreldioma] [-o ficheroSalida] texto1.txt {texto2.txt texto3.txt}
```

Los argumentos son:

-t | **-b** paramétro opcional, respectivamente modo texto o modo binario para el fichero de salida `(-t)` es por defecto.
-l indica el nombre del idioma que se aprende, valor (`unknown` por defecto si no hay parámetro -l).
-o indica el nombre del fichero de salida <**.bgr**> (**output.bgr** por defecto, si no hay parámetro -o).
Nota: Los parámetros se pueden introducir en cualquier orden, pero solo los que comienzan con $-$.

Como ya sabemos, cuando la salida es de texto la cabecera de un fichero language es:

[4] Se busca el carácter en el string $validos$, la posición devuelta es el índice a utilizar para filas o columnas de la matriz.

- La primera línea contiene siempre la cadena "MP-LANGUAGE-T-1.0".
- La segunda línea contiene una cadena que describe qué idioma es, el introducido con **-l** o `unknown`.
- La tercera línea contiene el número de bigramas diferentes que están asociados al idioma descrito.
- Las siguientes líneas contienen la lista de bigramas, con sus frecuencias, según el número de bigramas especificados en la línea tercera.

Cuando la salida es binaria la cadena mágica, primera línea, contiene la cadena: "MP-LANGUAGE-B-1.0". Las líneas 2 y 3 muestran el mismo contenido que en formato de texto. A continuación aparece la lista de parejas bigrama-frecuencia en formato binario, no sería legible con un editor pues, esta lista se podría escribir de una sola vez, de forma compacta con el mismo número de bytes para cada par, mediante la función `stream.write` [5].

El programa **learn** acepta un número variable de ficheros de entrada <*.txt>, siempre que al menos uno aparezca en la llamada. El fichero de salida <.bgr> tiene una cadena mágica específica "MP-LANGUAGE-T-1.0.º "MP-LANGUAGE-B-1.0", texto o binaria respectivamente, por lo que ambos tienen una cabecera de 3 líneas.

Ejemplo 4.1. Faltan argumentos para la ejecución del programa. El programa necesita al menos un fichero de entrada (un libro).

```
Linux> learn
```

La salida del ejemplo anterior debe dar el siguiente mensaje por la salida estándar de error:

```
Error, run with the following parameters:
Format:
learn [-t|-b] [-l languageId] [-o outputFilename] <text1.txt>
                      [<text2.txt> <text3.txt> .... ]
           learn the model for the language languageId from the
                      text files <text1.txt> <text2.txt> <text3.txt> ...

Parameters:
-t|-b: text mode or binary mode for the output file (-t by default)
-l languageId: language identifier (unknown by default)
-o outputFilename: name of the output file (output.bgr by default)
<text1.txt> <text2.txt> <text3.txt> ....: names of the input
                                    files (at least one is mandatory)
```

Un ejemplo de ejecución con argumentos correctos podría ser el siguiente:

[5] Se remite al lector a revisar los ficheros binarios en la parte de teoría.

Ejemplo 4.2. Aprendizaje del lenguage *spanish* a partir del libro `quijote.txt`, guardando el modelo aprendido en el fichero `quijote.bgr`.

```
Linux> learn -l spanish -o quijote.bgr quijote.txt
```

El resultado de esta ejecución es un fichero en disco ("quijote.bgr" en el ejemplo) de texto, con la lista de bigramas y las frecuencias halladas en el fichero de texto quijote.txt. El formato del fichero **quijote.bgr** es exactamente el que hemos estado usando en las prácticas anteriores.

Ejemplo 4.3. Aprendizaje del lenguage *spanish* a partir de los libros `quijote.txt` y `fortunata.txt`, guardando el modelo aprendido en el fichero `lang_spanish.bgr` en formato binario.

```
Linux> learn  -b -l spanish -o lang_spanish.bgr quijote.txt fortunata.txt
```

El objetivo de esta práctica es extraer los bigramas de un conjunto de documentos de texto (`*.txt`) siempre que el contenido de estos ficheros se ajuste a las siguientes reglas.

- Los ficheros de texto que vamos a considerar, están todos en la misma codificación. En concreto recomendamos usar la codificación ISO 8859-15 (también conocida como Alfabeto Latino n.º 1 o ISO Latín 1.).
- ¡Atención! Cualquier editor puede alterar la codificación de los ficheros, es necesario comprobar la codificación de los ficheros antes de aplicar `learn`, con el comando: $file$.
- Es posible que la visualización de un fichero con contenido ISO 8859-15 en distintos programas no sea la adecuada y se observen caracteres ilegibles. En language0 se indicaba como cambiar la codificación de cualquiera de ellos.
- Todos los caracteres se pasan a minúscula y se ignoran caracteres no válidos como " !=. , ; : etc.. Asociada a la nueva clase que vamos a implementar, se encuentra definida una constante `DEFAULT_VALID_CHARACTERS` que contiene todos los caracteres válidos para cualquier idioma con codificación ISO 8859-15.
- Para procesar cada uno de los ficheros de entrenamiento, se van leyendo las palabras una a una ignorando los separadores (blanco, tabulador, `\n`) y cualquier carácter no válido.

En la carpeta `Books` podrá encontrar una serie de ficheros de texto de ejemplo en el formato indicado, escritos en 4 idiomas diferentes.

5. Programa joinLanguages. Fusión de varios ficheros language en uno solo

El programa **joinLanguages** tiene por objeto gestionar diferentes ficheros language `*.bgr` (con cualquiera de los formatos indicados en la sección 4.2) pertenecientes a un mismo idioma, con el fin de obtener un único language fusión de los primeros. El resultado se guarda en un nuevo fichero language con el formato especificado para la salida.

La sintaxis para ejecutar el programa **joinLanguages** es como sigue:

```
Linux> joinLanguages [-t|-b] [-o <outputFile.bgr>]  <file1.bgr>
                                [<file2.bgr> ... <filen.bgr>]
```

Los argumentos son:

-t | **-b** paramétro opcional, respectivamente modo texto o modo binario (modo texto es por defecto) para el fichero de salida.
-o indica el nombre del fichero de salida `<.bgr>`, (**output.bgr** por defecto, si no hay parámetro -o).
Nota: Los parámetros se pueden introducir en cualquier orden, tan solo los que comienzan con `-`.

El programa recibe al menos un fichero language `<file1.bgr>` de entrada , que puede ser de texto o binario. Además puede recibir un nombre de fichero `<fileoutput.bgr>` para la salida.

Ejemplo 5.1. Faltan argumentos para la ejecución del programa. El programa necesita al menos un fichero language de entrada (un fichero `bgr`).

```
Linux> joinLanguages
```

La salida del ejemplo anterior debe dar el siguiente mensaje por la salida estándar de error:

```
Error, run with the following parameters:
Format:
joinLanguages [-t|-b] [-o <outputFile.bgr>] <file1.bgr>
                          [<file2.bgr> ... <filen.bgr>]
       join the Languages files <file1.bgr> <file2.bgr> ... into
                          <outputFile.bgr>

Parameters:
-t|-b: text mode or binary mode for the output file (-t by default)
-o <outputFile.bgr>: name of the output file (output.bgr by default)
<file*.bgr>: each one of the files to be joined
```

Según que el párametro sea `-t` o `-b`, la salida puede ser de texto o binario. Así, el programa `joinLanguages` se podría utilizar para convertir un fichero language de texto a binario y viceversa.

Como ejemplo, tomemos el fichero en binario `lang_spanish.bgr` que hemos obtenido como resultado de la ejecución del Ejemplo 4.3. Lo podemos convertir a formato texto según se indica en el Ejemplo 5.2.

Ejemplo 5.2. Conversión del fichero en binario `lang_spanish.bgr` (obtenido con Ejemplo 4.3) a formato texto, guardando el resultado en el fichero `lang_spanish_txt.bgr`.

```
Linux> joinLanguages -o lang_spanish_txt.bgr lang_spanish.bgr
```

El programa `joinLanguages` convierte el fichero de entrada binario `lang_spanish.bgr`, en un fichero de texto llamado `lang_spanish_txt.bgr`.

No obstante, el verdadero propósito de `joinLanguages` es fusionar varios ficheros languages. Un ejemplo de uso podría ser la ejecución del Ejemplo 5.3.

Ejemplo 5.3. Unión de tres ficheros languages guardando el resultado en el fichero binario `lang_spanish.bgr`

```
Linux> joinLanguages -b -o lang_spanish.bgr quijote.bgr
                              BodasdeSangre.bgr Fortunata.bgr
```

Se obtiene el fichero `lang_spanish.bgr` binario como la fusión de tres ficheros languages, cada uno aprendido por separado usando `learn` previamente.

Básicamente, es lo que hacía la práctica `Language2` excepto por el hecho de que ahora se pueden leer y escribir ficheros languages de texto y/o binarios, y han variado los argumentos del main.

6. Programa classify. Predicción del idioma de un documento

El programa **classify** consiste en que dado un documento de texto escrito en un idioma desconocido, y un conjunto de languages de referencia, $L_1, L_2, ..L_i$ cuyos idiomas están especificados, se quiere calcular la distancia del language que se calcula sobre el texto de entrada, a cada uno de los languages $L_1, L_2, ..L_i$ y determinar aquel con menor distancia, para asignarle su idioma al documento de texto. Una versión próxima es la que resolvimos en `Language3` y `Language4`.

A partir del documento de texto de entrada se ha de generar un language en memoria cuyo idioma inicialmente es `unknown`; este contiene la lista de los bigramas diferentes hallados con sus frecuencias. El conteo de las frecuencias de los bigramas hallados se realiza mediante los métodos de la clase `BigramCounter` desarrollados para el programa learn. Finalmente, se obtiene un language L_x y ya estamos en las mismas condiciones de `Language4`. Se calcula la distancia $distance(L_x, L_1)$, $distance(L_x, L_2)$, $distance(L_x, L_3)$... $distance(L_x, L_i)$. La predicción consiste en asignarle al documento de texto el idioma de aquel language cuya distancia sea menor.

La sintaxis para ejecutar el programa **classify** es como sigue:

```
Linux> classify <text.txt> <lang1.bgr> [<lang2.bgr> <lang3.bgr> ...]
```

El programa recibe al menos dos ficheros de entrada [6]: `<text.txt>` y `<lang1.bgr>`. El primero, `<text.txt>`, es el documento de texto de entrada del que se va a obtener un objeto language en memoria. El segundo, `<lang1.bgr>`, y el resto de ficheros (`*.bgr`) son respecto a los que se van a calcular las distancias para determinar el de menor distancia. Finalmente, la salida muestra por pantalla el idioma del language seleccionado.

Ejemplo 6.1. Faltan argumentos para la ejecución del programa

```
Linux> classify
```

Ejemplo 6.2. Faltan argumentos para la ejecución del programa

```
Linux> classify ../Books/lesMiserables.txt
```

La salida de los dos ejemplos anteriores debe dar el siguiente mensaje por la salida estándar de error:

```
Error, run with the following parameters:
Format:
 classify <text.txt> <lang1.bgr> [<lang2.bgr> <lang3.bgr> ....]
 Obtains the identifier of the closest language to the input text file
```

Ejemplo 6.3. Clasificación del documento de texto contenido en el fichero `BodasdeSangre_FedericoGarciaLorca.txt` como *english, french, spanish* o *german*, usando un fichero `bgr` para modelar cada idioma.

```
Linux> classify BodasdeSangre_FedericoGarciaLorca.txt changedMan.bgr
       lesMiserables.bgr fortunata.bgr Die_Verwandlung.Franz_Kafka.German.bgr
```

Se clasifica como español como era de esperar. La salida obtenida sería:

```
Final decision: language english with a distance of 0.119412
```

Ejemplo 6.4. Clasificación del documento de texto `Die_Verwandlung.Franz_Kafka.German.txt` como *english, french, spanish*, o *english*, usando un fichero `bgr` para modelar cada idioma.

```
Linux> classify Die_Verwandlung.Franz_Kafka.German.txt changedMan.bgr
               lesMiserables.bgr fortunata.bgr aliceWonder.bgr
```

Se clasifica como inglés, lo cual puede ser sorprendente pero, fijémonos que no se dió ningun language en alemán, por lo que la menor distancia es con `changedMan.bgr`, uno de los ficheros language en inglés. A continuación se muestra la salida que se obtendría:

```
Final decision: language spanish with a distance of 0.238421
```

Con esto comprobamos la limitación de toda clasificación, solo se puede clasificar lo que ha sido aprendido.

[6] Mínimo dos ficheros para que no de error de sintaxis, pero no ha lugar a ninguna predicción, pues le asignará siempre el idioma del segundo. ¿Porqué?.

7. Práctica a entregar

Para la práctica final se ha de elaborar un proyecto nuevo **Language5** con la estructura habitual de directorios. En lugar de elaborar un proyecto NetBeans para cada programa, se van a desarrollar 3 programas independientes cada uno con su main propio. Así, en la carpeta `src` además de la definición de cada una de las clases se encuentran 3 ficheros fuentes `learn.cpp, joinLanguages.cpp` y `classify.cpp`, cada uno con su main correspondiente para la gestión de sus parámetros de entrada. Para la compilación y ejecución de los programas por separado, utilizará un fichero denominado `metamain.cpp`, que nos permitirá seleccionar el programa principal del proyecto. Los detalles se encuentran en B.

8. Código para la práctica

A. BigramCounter.h

```
/*
 * Metodología de la Programación: Language5
 * Curso 2022/2023
 */

/*
 * @file:   BigramCounter.h
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 *
 * Created on 12 February 2023, 11:00
 */

#ifndef BIGRAM_COUNTER_H
#define BIGRAM_COUNTER_H

#include "Language.h"

/**
 * @class BigramCounter
 * @brief It is a helper class used to calculate the frequency of each bigram in
 * a text file. It consists of a square matrix of integers. Each element contains
 * the frequency of the bigram that defines that element: the bigram formed taking
 * the characters defined by the row and column of that element.
 */
class BigramCounter {
public:

    /**
     * A const c-string with the set of characters that are considered as
     * part of a word. Any other character will be considered a separator
     * Only lowercase characters are included in this string. This c-string
     * is used in the constructor of this class, as the default value to assign
     * to the field _validChars
     */
    static const char* const DEFAULT_VALID_CHARACTERS;

    /**
     * @brief Constructor of the class. The object will contain a matrix of integers
     * with as many rows and colums as the number of characters in @p validChars
     * Each element of the matrix will be set to 0
     * @param validChars The set of characters that are considered as
     * part of a word. Input parameter
     */
    BigramCounter(std::string validChars = DEFAULT_VALID_CHARACTERS);

    /**
     * @brief Copy constructor
     * @param orig the BigramCounter object used as source for the copy. Input
     * parameter
     */
    BigramCounter(BigramCounter orig);

    /**
     * @brief Destructor
     */
    ~BigramCounter();

    /**
     * @brief Returns the number (size) of valid characters that are considered as part
     * of a word in this BigramCounter object. Query method
     * @return the number (size) of valid characters that are considered as part
     * of a word in this BigramCounter object
     */
    int getSize();

    /**
     * @brief Gets the number of bigrams with a frequency greater than 0.
     * Query method
     * @return the number of bigrams with a frequency greater than 0
     */
    int getNumberActiveBigrams();

    /**
     * @brief Sets the frequency of the given bigram using the value
     * provided with @p frequency. Modifier method
     * @param bigram The bigram in which the frequency will be set. Input
     * parameter
     * @param frequency The new frequency. Input parameter
     * @return true if the bigram was found in this object. false otherwise
     */
    bool setFrequency(Bigram bigram, int frequency);

    /**
     * @brief Increases the current frequency of the given bigram using the value
     * provided with @p frequency. If @p frequency is 0 or frequency is not
     * provided, then 1 is added to the current frequency of the bigram.
     * Modifier method
     * @throw std::invalid_argument This method throws an
     * std::invalid_argument exception if the given bigram is not valid
```

```
     * @param bigram The bigram in which the frequency will be modified.
     * Input parameter
     * @param frequency The quantity that will be added to the current frequency.
     * Input parameter
     */
    void increaseFrequency(Bigram bigram, int frequency = 0);

    /**
     * @brief Overloading of the assignment operator
     * @param orig the BigramCounter object used as source for the assignment.
     * Input parameter
     * @return A reference to this object
     */
    BigramCounter operator=(BigramCounter orig);


    /**
     * @brief Overloading of the operator +=. It increases the current
     * frequencies of the bigrams of this object with the frequencies of the
     * bigrams of the given object.
     * Modifier method
     * @param rhs a BigramCounter object
     * @return A reference to this object
     */
    BigramCounter operator+=(BigramCounter rhs);

    /**
     * @brief Reads the given text file and calculates the frequencies of each
     * bigram in that file.
     * Modifier method
     * @throw std::ios_base::failure Throws a std::ios_base::failure exception
     * if the given file cannot be opened
     * @param fileName The name of the file to process. Input parameter
     * @return true if the file could be read; false otherwise
     */
    void calculateFrequencies(char* fileName);

    /**
     * @brief Builds a Language object from this BigramCounter object. The
     * Language will contain the bigrams and frequencies for those one with
     * a frequency greater than 0.
     * Query method
     * @return A Language object from this BigramCounter object
     */
    Language toLanguage();

private:
    int** _frequency; ///< 2D matrix with the frequency of each bigram

    /**
     * Set of characters that are considered as part of a word. Any other
     * character will be considered a separator of words. Only lowercase
     * characters are included in this string
     */
    std::string _validCharacters;

    /**
     * @brief Overloading of the () operator to access to the element at a
     * given position
     * Query method
     * @param row Row of the element. Input parameter
     * @param column Column of the element. Input parameter
     * @return A const reference to the element at the given position
     */
    int operator()(int row, int column);

    /**
     * @brief Overloading of the () operator to access to the element at a
     * given position
     * Query/Modifier method.
     * @param row Row of the element. Input parameter
     * @param column Column of the element. Input parameter
     * @return A reference to the element at the given position
     */
    int operator()(int row, int column);
};

#endif /* BIGRAM_COUNTER_H */
```

B. NetBeans. Un proyecto con varios ejecutables

NetBeans elabora un solo ejecutable por cada proyecto. Así que para evitar tener que crear tres proyectos, uno por cada programa que hemos de elaborar se ha usado el siguiente fichero `metamain.cpp` que contiene directivas para el precompilador.

```
#ifdef LEARN
```

```
    #include "learn.cpp"
#elif CLASSIFY
    #include "classify.cpp"
#elif JOIN
    #include "joinLanguages.cpp"
#endif
```

Para seleccionar en NetBeans el fuente que se quiere compilar y ejecutar hemos de seleccionar la configuración que se desea. Para ello, nos situamos en el proyecto Language5 $->$ properties $->$ Debug $->$ Configuration y se despliega una lista como se muestra en la figura 6.

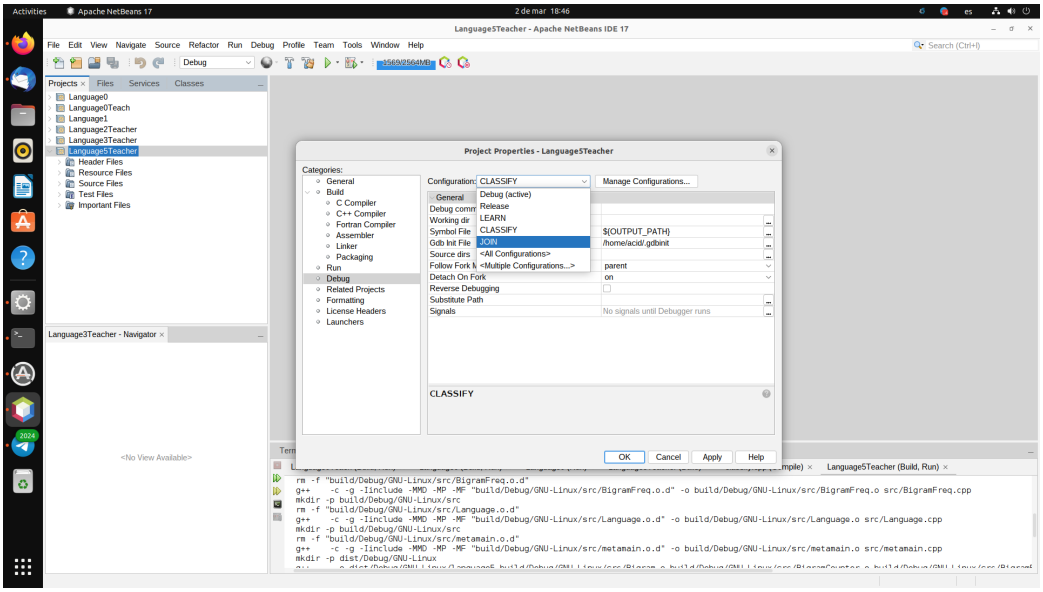

Figura 6: Seleccionando el ejecutable