Informe trabajo práctico

Programación 2

Tomás Federici

Programa en C.

El programa en C consta de 3 archivos.

1. comandos.c:

Funciones:
a. `buscar_textos`: Utiliza la función `system` para listar los nombres de archivos en "archivos.txt" de la persona cuyo nombre se pasa como argumento. Luego verifica si "archivos.txt" está vacío. Si está vacío, imprime un mensaje de error y utiliza la función `exit` para terminar el programa.
b. `comando_python`: Ejecuta el comando que llama al programa en Python.

2. limpiar_textos.c:

Funciones:
a. `corregir_char`: Recibe un carácter. Si el carácter es una letra, se convierte a minúscula y se retorna. Si es un punto, retorna un salto de línea. Para cualquier otro carácter, retorna -1.
b. `escribir_char`: Utiliza las variables **`caracter`** y **`caracter_siguiente`** para realizar correcciones y escribir caracteres en el texto según ciertas condiciones:
I. Si **`caracter`** es un punto y **`caracter_siguiente`** no es EOF, se toman caracteres del texto hasta que **`caracter_siguiente`** sea letra o EOF, esto para asegurar que después de un punto comience la siguiente oración o termine el texto, luego se escribe el salto de línea.
II. Si **`caracter`** es un espacio, se toman caracteres hasta que **`caracter_siguiente`** sea una letra o un punto, esto para asegurar que después de un espacio se siga con la siguiente palabra, luego se escribe el espacio.
III. Si **`caracter`** es un salto de línea (excepto los posteriores a un punto), se toman caracteres hasta que **`caracter_siguiente`** sea una letra, y se escribe un espacio.
IV. Si **`caracter`** una vez corregido no es -1 (es una letra válida), se escribe en el texto.

c. `limpiar_textos:` En esta función se crea el archivo **`textos_filtrados`** y recorre los textos de la persona, corrigiendolos y escribiendolos en **`textos_filtrados`**, también se usa la variable **`arvhicos_leidos`** que se inicializa en 0, y cada vez que un archivo se termina de leer, se aumenta en 1, la utilizo para colocar un salto de línea al final de cada texto siempre que **`archivos_leidos`** sea mayor que 1.

3. main.c:

En la función `main` se verifica si se pasaron correctamente los argumentos, luego se llaman a las funciones `buscar_textos`, luego `limpiar_textos` y finalmente se llama al programa en python con `comando_python`.

Programa en Python.

La idea para el programa es comenzar con un preprocesado del texto, utilizo una estructura de n-gramas, se puede elegir cualquier n y se puede cambiar en el archivo "completar_frases.py", el programa por defecto tiene n = 3 (las predicciones siguen siendo buenas con n = 4 o n = 5) ya que obtuve mejores resultados con trigramas, la forma que opté para guardar los n-gramas es la siguiente:
(*) Utilizo un diccionario que tiene como clave una tupla con dos palabras, y de valor una lista con los n-gramas que comienzan y terminan con esas dos palabras, de modo que identifico a cada n-grama con las palabras con las que comienzan y terminan, elegí esta estructura ya que utilizo dichas palabras para realizar comparaciones, y de esta forma es más fácil acceder a los n-gramas que tienen la información que necesito.
Luego a la hora de analizar las frases, distingo 3 casos:

1. Si el guión de la frase está al final de la frase, voy a tomar las palabras que están a distancia 1 y 2 del guión hacia la izquierda. En el caso de que la frase solo conste de una palabra y luego el guión, solo voy a tomar esa palabra.
2. Si el guión de la frase está al principio de la frase, voy a tomar las palabras que están a distancia 1 y 2 del guión hacia la derecha. En el caso de que la frase solo conste del guión y luego una palabra, solo voy a tomar esa palabra.
3. Si el guión está en medio de la frase (tiene al menos una palabra hacia cada lado) voy a tomar la palabra a la derecha y la palabra a la izquierda.

Existe la opción de tomar un número "n" de palabras, en lugar de dos palabra, decidí no implementarlo porque con la forma de tomar las palabras que elegí obtuve buenas predicciones y no vi necesario aumentar la cantidad de comparaciones

Con las palabras obtenidas de las frases, busco coincidencias en los n-gramas utilizando las claves que son tuplas con las palabras que comienzan y terminan los n-gramas,

en el caso de que las dos palabras usadas de referencia coincidan en la misma oración la palabra que corresponde al guión, se guarda en la lista **ambas,** y además se guarda la coincidencia en dos listas **posibles1** y **posibles2**, asociadas a cada una de las dos palabras por separado,
luego las oraciones donde coincida solo una palabra, se guarda la palabra que corresponde al guión en la lista de la palabra con la cual se hizo la coincidencia.
(**)Finalmente para elegir la lista, el orden de prioridad es:
1. Si la lista **ambas** es no vacía, se retorna.
2. Si **ambas** es vacía, se realiza la intersección entre **posibles1** y **posibles2**, si esta es no vacía, se retorna.
3. Si la intersección entre **posibles1** y **posibles2**, es vacía, y ambas son no vacías, se retorna la lista que tenga mayor frecuencia en algún elemento.
4. Si alguna de las dos listas es vacía, se retorna la que no esté vacía.
5. Si ambas están vacías, se retorna una lista vacía.
Una vez se retorne una lista, la predicción será el elemento más frecuente en la lista retornada.

Para este programa, utilizo 5 archivos:

1. func_listas.py:

Funciones:
a. `interseccion:` Realiza la intersección entre dos listas, no hago uso de conjuntos para realizar intersecciones, ya que me interesa que cada elemento de la lista mantenga su frecuencia.
b. `elemento_mas_frecuente:` Calcula el elemento más frecuente de una lista y retorna una tupla que tiene como primer elemento, el elemento más frecuente de esa lista y como segundo elemento, su frecuencia.

2. preprocesado.py:

Funciones:
a. Tanto la función `entradas` como la función `frases` se utilizan para cargar el texto en entradas y las frases de la persona que se pasa como argumento y se retorna una lista con cadenas tal que cada cadena es una línea de los textos.
b. `preprocesado:` Esta función recibe como argumento la lista generada con `entradas` y un número natural “n” y divide el texto en n-gramas guardandolos en un diccionario como se explicó en (*). Además en esta función se aprovecha para calcular el elemento más frecuente en cada línea y guardarlo en la lista **caso_nulo**. Se retorna una tupla que tiene como primer elemento, los n-gramas y como segundo elemento, la palabra más frecuente en la lista **caso_nulo** que será utilizado como predicción en caso de que no se encuentren coincidencias en el texto de ninguna otra forma.

3. buscar_posibles.py:

Funciones:
a. `palabras_cercanas:` Esta función recibe una de las frases con guion bajo, en forma de cadena, y retorna una tupla con tres elementos, el primer elemento es un número entero que uso para identificar si se trata de una frase en la cual el guión está al comienzo, al final, o en medio, utilizo las constantes **`TIPO_MEDIO`** = 0, **`TIPO_IZQ`** = 1, **`TIPO_DER`** = 2.
El segundo y tercer elemento son palabras cercanas al guión.
Si se trata de **`TIPO_MEDIO`** se toman las palabras a la izquierda y derecha del guión.
Si se trata de **`TIPO_IZQ`** se toman las palabras a distancia uno y dos hacia la derecha del guión.
Si se trata de **`TIPO_DER`** se toman las palabras a distancia uno y dos hacia la izquierda del guión.
En estos últimos dos casos si la frase consta solo de una palabra y el guión, se tomará solo esa palabra.
b. `palabras_posibles:` Esta función recibe el diccionario con los n-gramas, y una tupla generada en `palabras_cercanas.`
Se crean 3 listas, dos de ellas (**`posibles1`** y **`posibles2`**) van a guardar las coincidencias de cada una de las palabras usadas de referencia por separado y otra lista (**`ambas`**) que va a guardar las palabras para las cuales coinciden ambas palabras de referencia a la vez.
La función itera sobre las claves de los diccionarios y luego se divide en 3 casos:
Ⅰ. **`TIPO_MEDIO:`** En este caso, se verifica si la palabra a la izquierda (resp. derecha) del guión, coincide con la palabra a la a la izquierda (resp. derecha) del n-grama y en ese caso se guardará la palabra que corresponde al guión en la oración en la lista **`posibles1`** (resp. **`posibles2`**)
Además, si la palabra a la izquierda del guión, coincide con la palabra a la a la izquierda del n-grama y además la palabra que se encuentra en el índice 2 del n-grama, coincide con la palabra a la derecha del guión, la palabra que corresponde al guión se guarda en **`ambas.`**
Ⅱ. **`TIPO_DER:`** En este caso si la palabra que está dos lugares (resp. un lugar) a la izquierda del guión coincide con la palabra a la izquierda del n-grama la palabra se guarda la palabra que corresponde al guión en **`posibles2`** (resp. **`posibles1`**)
Además si ambas palabras coinciden en la misma oración, se guarda la palabra que corresponde la guion en **`ambas.`**
Ⅲ. **`TIPO_IZQ:`** Este caso es análogo al caso **`TIPO_DER`** solo que se buscan coincidencias a la derecha del n-gama
Luego la función retorna una tupla con **`posibles1`**, **`posibles2`** y **`ambas.`**
c. `lista_posibles:` Esta función recibe el diccionario con los n-gramas y una tupla creada en `palabras_cercanas` y llama a la función `palabras_posibles` para luego con la tupla retornada, verificar el orden de prioridad explicado en (**)

4. completar_frases.py:

Funciones:
a. `predicciones:` Esta función llama a las funciones en “preprocesado.py” para cargar la información y se crea el archivo de salida donde se van a escribir las frases completas.

Luego se itera sobre cada una de las frases, se obtiene la lista de las palabras posibles para la predicción con las funciones de “buscar_posibles.py” y se utiliza como prediccion el elemento más frecuente de la lista obtenida, en caso de ser vacía, se utiliza el elemento obtenido en el preprocesado como **`caso_nulo`**, se reemplaza cada guión con la predicción obtenida y se escribe en el archivo.

Testing en C.

Para el testing en C, decidí testear la función `limpiar_textos` ya que esta utiliza la función `corregir_char` y `escribir_char`, y ambas funciones apuntan al mismo propósito por lo que si la función escribe el texto correctamente, quiere decir que las demás funciones, tienen un correcto funcionamiento.

Para esto, utilizo la funcion `archivos_iguales` la cual compara dos archivos y retorna 1 si los archivos son iguales y 0 si los archivos no son iguales.

Además dentro de la carpeta Textos se encuentra una carpeta llamada “Testing_en_C” la cual tiene dos archivos para corregir y una carpeta llamada “TestC” la cual contiene un archivo con la salida esperada de los archivos en “Testing_en_C” usada para comparar con el archivo que se genera en la carpeta Entradas luego de ejecutar el programa.

Comando para el Testing: ./TestC.out Testing_en_C

Testing en Python.

Para el testing en Python, hice test de 6 funciones,

`preprocesado:` 1 assert (Para esta función, tuve que crear un archivo en la carpeta entradas con el cual se realiza el testing)

`interseccion:` 4 asserts

`elemento_mas_frecuente:`4 asserts

`palabras_cercanas:` 6 asserts

`palabras_posibles:` 6 asserts

`lista_posibles:` 6 asserts

Comando: python3 Testing_en_Python.py