TP Final - Programación II

Ariel Leonardo Fideleff

12 de diciembre de 2023

Índice

1. Descripción de la solución

1.1. Una decisión general

A excepción de esta descripción del proyecto, encontrará que todo el código y la documentación están escritos en inglés. Esta es una decisión que tomé frente a que estoy acostumbrado a programar en este idioma. Dada la cantidad de información y recursos disponibles en inglés con los que he aprendido a programar, me resulta más fácil que programar en español.

Me sentí ante la obligación dejar esto en claro, para justificar el motivo por el cual aclararé en algunos casos la relación entre los conceptos desarrollados y los términos en inglés para relacionarlos en el código. En especial veo esto importante en el apartado de detalles de implementación, donde deberé hacer referencias a nombres de variables y estructuras utilizadas.

Dicho esto, explicaré mis decisiones para tanto el programa en C como el hecho en Python.

1.2. El programa en C

Para la primera parte del TP, se nos pide desarrollar un programa en C que:

1. Tome por argumento el nombre de una persona.
2. Lea un conjunto de archivos `.txt` contenidos en un carpeta cuyo nombre se corresponde con el pasado por argumento. Cada uno contiene textos escritos por la persona en cuestión.
3. Limpiar los textos bajo un conjunto de pautas que permita separarlos en oraciones y evitar que contengan caracteres no deseados.
4. Unificar los textos "sanitizados" en un único archivo bajo el nombre de la persona que los escribió (formato también `.txt`).
5. Llamar al programa en Python con el nombre de la persona como argumento, para continuar a la siguiente etapa de nuestro programa en el TP.

1.2.1. Tomar el nombre de la persona por argumento

Obtener los argumentos que recibe un programa en C es sencillo y no presenta mayores inconvenientes. Más información en detalles de implementación.

Lo único a tener en cuenta es que nos debemos asegurar de que la cantidad de argumentos recibida por la terminal sea la correcta, con el fin de estar seguros de recibir el nombre de la persona necesario para saber en dónde debemos leer los textos a procesar.

1.2.2. Leer los archivos `.txt` escritos por la persona

El enunciado exige el uso de la función `system()` y los comandos `cd` y `ls` para leer los contenidos de un directorio. Esta forma de leer los contenidos de un directorio presenta la desventaja de que no nos permite efectivamente determinar si el directorio deseado, `Textos/[nombre persona]`, exista (o que tengamos los permisos suficientes para leer su contenido).

De hecho, si probamos el comando sugerido en el trabajo directamente en la terminal, observaremos que no se crea el archivo `archivos.txt` si no existe la carpeta, ya que el comando `cd` con el cual intentamos acceder a la carpeta, falla en primera instancia.

El valor de retorno de la función `system()` podría servirnos como indicación de algún error en la ejecución de los comandos... sin embargo observando la documentación se indica que este valor de retorno depende de la implementación de la función, aunque se pueda "esperar" que se corresponda con el código de estado resultante de la ejecución de los comandos.[1]

Ante la incertidumbre, optamos por elegir un método con el cual podemos tener mayor seguridad de su correcto funcionamiento. Sabemos bajo haber probado los comandos en la terminal previamente, cómo es el comportamiento de `cd` y `ls` si la carpeta no existe. Entonces, considerando que `archivos.txt` no es creado, podríamos inicialmente comprobar si el archivo existe intentando abrirlo como sabemos hacerlo en C, usando `fopen()`. Al fin y al cabo, si `archivos.txt` no existe,

entonces `fopen` devolvería `NULL`, no? Hay un problema, y radica en que también se nos exige que no eliminemos `archivos.txt` tras haberlo utilizado. En consecuencia, el archivo puede existir como producto de una ejecución anterior correcta, llevándonos a pensar que la carpeta buscada si existe.

Por lo tanto, para resolver esto, antes de ejecutar los comandos asociados al listado de los archivos de la carpeta con `system()`, intentaremos primero sobreescribir `archivos.txt` por uno vacío (o crearlo vacío si no existe). De esta forma, si...

- ... la carpeta deseada **no** existe, al leer `archivos.txt` veremos que se encuentra vacío, y podremos informarle al usuario con un error apropiado deteniendo la ejecución del programa.
- ... la carpeta deseada **sí** existe, el `archivos.txt` sólo resultaría vacío si no se presentan archivos en la carpeta o si no se puede acceder a ella. En este caso, no habría textos de la persona que sanitizar los cuales el programa pueda leer, y no se produciría información para la segunda etapa del programa en Python. Por lo tanto, debemos de informarle de esto al usuario y tampoco queremos que el programa continúe.

Con esto resuelto, si la carpeta existe y hay archivos para procesar, podemos leer el contenido de `archivos.txt` para obtener los *nombres* de los archivos dichos. Como es necesario abrirlos para poder procesarlos, debemos primero anteponer la carpeta en la que éstos se encuentran relativa al directorio de ejecución de nuestro programa. Por ejemplo, si la persona se llama `Pepe`, y uno de los textos tiene nombre de archivo `biografia.txt`, su ruta sería `Textos/Pepe/biografia.txt`.

Así, para cada nombre leído, podemos obtener la ruta antes descrita, leer el archivo allí ubicado, y procesar su contenido.

Ahora bien, de esto surge otra decisión importante del programa: es necesario leer los contenidos de los archivos a la memoria para procesarlos? O podemos irlos procesando "directamente" leyendo de los archivos, a la vez que escribimos el contenido sanitizado en el archivo de salida? Es decir, necesariamente el contenido de los archivos debe transitar por la memoria, pero a lo que nos referimos es si es necesario almacenar el contenido de cada archivo en alguna estructura de datos, como un arreglo.

La ventaja de leer primero todo el contenido de los archivos a arreglos, por ejemplo, es que nos permitiría definir funciones que tomen como argumento estas estructuras y que sean independientes de operaciones de lectura / escritura sobre archivos, facilitando la depuración de errores y el testing del procedimiento.

Mientras, como principal desventaja, en C presentamos complicaciones relacionadas a la administración de la memoria. Es decir, deberíamos responsabilizarnos por reservar la memoria suficiente para la información contenida en los archivos, como también de su posterior liberación una vez que no necesitemos usarla más. Esto puede conllevar a un mayor esfuerzo con el fin de evitar errores.

En esta situación particular, podemos analizar que el proceso de sanitización de los archivos no es demasiado complejo y que, desde un punto de vista personal, el esfuerzo requerido para administrar la memoria apropiadamente no vale la pena respecto a los beneficios relacionados con la modularidad de las funciones diseñadas. Es más, para una operación como la dada, personalmente creo que si el archivo a procesar es muy grande, sería un "desperdicio de memoria" cargar inicialmente todo el contenido del archivo en la memoria, ya que es un proceso que se puede realizar de forma completamente lineal sobre la entrada.

Por lo tanto, optamos entonces procesar los archivos carácter por carácter, leyendo progresivamente de cada uno de los archivos de la entrada y decidiendo según ciertas condiciones (que consideran las pautas del TP y las decisiones para el paso 2 que se especifican más adelante) qué caracteres deben escribirse al archivo de salida.

Los detalles sobre la implementación se presentarán en el apartado correspondiente.

Como un último comentario, otra de las decisiones tomadas para la lectura de los textos escritos por la persona radica en algo similar a lo anteriormente descrito para los archivos, pero en este caso, sobre `archivos.txt`. Esto vendría a ser, si es preferible leer todos los nombres de archivos contenidos en `archivos.txt` a un arreglo, o si leer progresivamente los nombres, concatenar las rutas, leer y procesar los textos uno por uno, evitando reservar memoria para las cadenas donde se almacenarían las rutas.

En este caso, si bien creo que leer los nombres de los archivos de forma progresiva resultaría en un código más corto, nuevamente esto iría en contra de la modularidad del código. La decisión al fin y al cabo se reduce a preferencia personal, pero pienso que una función que sanitice un conjunto de archivos dados por sus rutas a un único archivo dado por otra ruta, tiene un mejor diseño y modularidad que una función que deba leer un conjunto de nombres de un archivo e irlos concatenando con el directorio en el que se encuentran, para luego irlos procesando y combinando en un archivo de destino.

Es por esto que para la lectura de los nombres de los archivos de la persona a procesar, opté por leerlos en un arreglo para que luego sean procesados por otra función que opere en cada uno, sanitizándolos y escribiendo su contenido limpio en un archivo de salida.

Una vez más, los detalles de implementación se presentarán en el apartado correspondiente.

1.2.3. Limpiar los textos

Para limpiar los textos escritos por la persona, se indican en el enunciado un conjunto de pautas que el archivo combinado resultante debe seguir, junto con un ejemplo de una salida para una entrada dada. Sin embargo, hay ciertos detalles que no son claros, ante los cuales tuve que tomar ciertas decisiones a la hora de programar una función que limpiara un archivo en concreto.

En primer lugar, en teoría se indica que el archivo de salida sólo debe contener saltos de línea y caracteres del alfabeto inglés en minúscula, eliminándose cualquier otro tipo de carácter. Esto no considera los espacios, los cuales son claramente importantes para distinguir las palabras individuales de una oración. Por lo tanto, no queda del todo claro el comportamiento del programa si se presentan múltiples de estos espacios: ¿hay que dejarlos todos? ¿o se debe eliminarlos de forma que las oraciones procesadas se presenten en un formato "normal" bajo las reglas del lenguaje?

Es común el tipo de problemas que plantean "normalizar" una oración, de forma que aquellos espacios que se presenten de forma irregular, sean eliminados. Por lo tanto, decidí tomar esto en cuenta para mi función de sanitización de los archivos. Como un ejemplo, si una oración esta dada por:

```
␣␣␣␣Esta␣es␣␣una␣oracion␣␣␣de␣prueba␣␣.␣
```

La normalizaría a:

```
esta es una oracion de prueba
```

Otro aspecto similar que no está del todo claro, es exactamente a qué se refiere el enunciado con eliminar todos los símbolos. Si bien puede resultar obvio en el uso común de los signos de puntuación, puede ser menos intuitivo si hablamos de "accidentes en la escritura". Por ejemplo, si tenemos:

```
esto_$es una pru3eba, no es as!i?
```

¿Cómo deberíamos limpiar esta oración? Es cierto que el enunciado nunca menciona números, ¿pero cómo haríamos para las palabras `as!i?` o `esto_$es`? Por lo tanto, decidí tomar el criterio general que *cualquier palabra está compuesta por cualquier conjunto consecutivo de caracteres alfabéticos*, concluyendo que cualquier símbolo que se encuentre entre dos conjuntos tales, los divide en dos palabras, que deben estar separadas por un espacio. Esto considera la excepción de que estarían separadas por un salto de línea si alguno de los símbolos entre los conjuntos de caracteres alfabéticos fuera un punto final, indicando el final de una oración. Además, si se presentan muchos símbolos entre dos conjuntos de caracteres alfabéticos, bajo el criterio anteriormente establecido de los múltiples espacios, los conjuntos estarían separados con un único espacio.

Con esto establecido, el ejemplo anterior se limpiaría como:

```
esto es una pru eba no es as i
```

Por último, estableceremos la convención que el archivo de salida debe terminar en un carácter de salto de línea `\n`. Este comentario surge a que algunos de los archivos con textos de personas dados por la cátedra para probar nuestro programa carecían de un `\n` al final de estos archivos. Entonces, si bien un archivo de salida que tenga una oración en cada línea sólo requeriría tener saltos de línea entre oraciones, esto puede generar problemas en algunos casos,[2] a lo que elegiremos por incluir un salto de línea al final del archivo siempre que sea posible.

Una excepción a esto sería si se recibe un archivo vacío, en cuyo caso también sería apropiado que se genere un archivo vacío. Esto es especialmente útil si se pretenden combinar muchos archivos en uno, ya que procesar un archivo vacío en un conjunto de archivos no produciría una línea vacía en el archivo final.

1.2.4. Unificar los textos sanitizados

Como indicamos en las decisiones para el paso 2, optamos por procesar los archivos carácter por carácter, escribiendo los caracteres directamente al archivo de salida bajo ciertas condiciones que se van analizando en la entrada.

Entonces, bajo las pautas establecidas en el paso anterior (3), unificar los textos es tan fácil como procesar los archivos en orden, donde el archivo de salida siempre sea el mismo, retomando al procesar cada archivo desde donde se terminó de escribir en la salida para el anterior. Es decir, bajo las decisiones establecidas, es equivalente procesar cada uno de los archivos de entrada a archivos de salida independientes, que procesarlos todos en orden a un mismo archivo de salida, siempre y cuando se escriba el contenido sanitizado de cada archivo a continuación de donde se escribió el del anterior.

Aprovechando que esta sección es relativamente breve, aclararé sobre una decisión tomada en general en relación al manejo de archivos a lo largo del programa en C **y** en el de Python. Tenemos que tener en cuenta que puede darse una situación la cual, por ejemplo, la carpeta `Entradas`, a la cual debemos escribir el archivo de salida, sea inaccesible, sea por no tener los permisos suficientes u algún otro motivo (aunque para el ejemplo debemos asumir por el enunciado que la carpeta existe).

Es por esto que, en general, siempre que intentamos acceder a un archivo para leer o para escribir, optamos verificar además que la operación no nos devuelva una referencia nula (C) o que falle (Python), para poder mostrar en cambio un mensaje al usuario informando sobre la ruta inaccesible.

Más detalles al respecto en el apartado de detalles de implementación.

1.2.5. Llamar al programa en Python

Finalmente, para el último paso debemos llamar al programa en Python con el nombre de la persona como argumento. Esto resulta sencillo siguiendo la indicación del TP de usar una vez más la función `system()`, creando una cadena con el comando que llame al intérprete de Python `python3` con el nombre del programa (se asume que está en el mismo directorio que el ejecutable de C), y pasando como argumento el mismo nombre de la persona que fue recibido como argumento del programa en C.

Los detalles de implementación se encuentran en el apartado correspondiente, aunque no se extienden mucho más de lo indicado en el enunciado.

1.3. El programa en Python

Para la segunda parte del TP, debemos desarrollar un programa en Python que:

1. Tome por argumento el nombre de una persona.

2. Lea el archivo `.txt` generado por el programa en C con las oraciones sanitizadas escritas por la persona.

3. Lea un archivo `.txt` a nombre de la persona con frases escritas por la misma, donde las frases tienen una palabra faltante cada una, indicada con un guión bajo (_).

4. Procese las oraciones leídas en alguna estructura útil para el propósito de predecir palabras faltantes en conjunto de frases.

5. Prediga las palabras faltantes en las frases anteriormente leídas, reemplazando los guiones bajos por éstas.

6. Genere un archivo final con las frases completas.

1.3.1. Tomar el nombre de la persona por argumento

De forma similar al programa en C, esto es sencillo obtener los argumentos recibidos desde la terminal en Python. Una vez más, tendremos en cuenta de validar la cantidad de argumentos recibida, con el fin de asegurarnos recibir el nombre de la persona, necesario para conocer las rutas donde se encuentran los archivos que necesitamos para los próximos pasos. Más información en detalles de implementación.

1.3.2. Leer el `.txt` generado por el programa en C

A diferencia del programa en C, la lectura y escritura a archivos es mucho más sencilla en Python, ya que existen funciones que permiten leer un archivo en su completitud (`f.read()`, donde `f` hace referencia a nuestro archivo abierto), o incluso leerlo línea por línea (`f.readlines()`, donde `f` hace referencia a nuestro archivo abierto). Además, en Python no nos preocupa tanto la administración de la memoria, de forma que operaciones como cargar los contenidos de un archivo a un arreglo resultan mucho más sencillas que en C.

Es por esto que, considerando que la forma en la que necesitaremos procesar la información de las oraciones escritas por la persona es más compleja que cuando queríamos sanitizar la entrada en C (decisiones a explicar en los próximos pasos), decidimos que lo más apropiado para este paso es leer todas las líneas del archivo.

Recordemos que cada línea se debe corresponder con una oración escrita por la persona. Entonces, separar la entrada del archivo generado por el programa en C en sus líneas, las oraciones, resulta ideal para analizar el rol que cumplen las palabras en cada una de ellas.

Más detalles de implementación en el apartado correspondiente.

1.3.3. Leer el `.txt` de las frases con palabras faltantes

De forma similar al paso anterior, resulta más práctico en este caso también, leer todas las frases por su completitud leyendo cada una de las líneas por separado. Esto facilitará su procesamiento iterando por cada una de ellas, y usando la información analizada sobre las oraciones conocidas para poder predecir la palabra faltante.

Cabe aclarar que en el programa esto se hace después del próximo paso. Es decir, se procesan las oraciones leídas en una estructura útil, y luego se leen las frases con palabras faltantes para poder completarlas. El orden realmente no debería influir, pero opté por mencionar primero este aspecto del programa para combinar a continuación, una explicación sobre las decisiones relacionadas al algoritmo de procesamiento de oraciones y de predicción de palabras utilizado.

1.3.4. Procesar las oraciones leídas en una estructura útil y predecir las palabras faltantes

Las decisiones tomadas para este paso son de las más importantes de todo el trabajo, ya que si bien el enunciado brinda algo de orientación sobre cómo realizar estas tareas, se deja principalmente a preferencia y voluntad del alumno.

Dicho esto, subdividiremos este apartado a su vez en distintas subsecciones.

1.3.4.1 Introducción

Para determinar el método más apropiado de predicción para las palabras en este trabajo, decidí investigar al respecto del problema, y si existe alguna solución conocida al problema planteado o alguno similar.

Buscando un poco en Internet, rápidamente pude ver que es muy común encontrar soluciones para el problema de predecir *la próxima palabra* que se va a escribir en un texto. Esto es común verlo cuando buscamos en Google, por ejemplo, en el cual se nos sugieren una serie de palabras para poder continuar nuestra oración a medida que la escribimos en la barra de búsqueda. O incluso más común, cuando escribimos en un teclado como el del celular, y se nos sugieren algunas palabras para continuar con nuestra oración.

A día de hoy, la forma más común para resolver este tipo de problemas es mediante inteligencia artificial, y modelos de procesamiento de lenguaje natural (NLP, por sus siglas en inglés). En cierta forma, podemos pensar que el algoritmo que queremos desarrollar para nuestro propósito se encuentra en este área, ya que al fin y al cabo queremos manipular el lenguaje humano mediante un programa de computadora.[10]

Pero en concreto, los métodos más eficaces conocidos para este problema a día de hoy, requieren el uso de herramientas y modelos basados en Machine Learning, que se encuentran claramente fuera de los temas vistos en la materia. De hecho, existen librerías de Python enteras dedicadas a este tipo de algoritmos, incluso con ejemplos y guías para resolver exactamente el problema planteado en este trabajo.[3] Aunque nuevamente, hasta este punto en la carrera no nos hemos adentrado todavía en esta área de la computación, y resulta inapropiado elegir una solución prefabricada sin siquiera saber cómo funciona en su totalidad. Más aún en un trabajo enfocado a cerrar una materia dedicada a aprender a programar.

Es por esto que buscando por métodos más sencillos, encontré el *modelo de lenguaje de los n-gramas de palabras*, como un modelo puramente "estadístico", que asume que la probabilidad de predecir una cierta palabra en una oración depende en una cantidad fija de palabras que la preceden.[11] A veces este concepto es relacionado con las *Cadenas de Markov*, las cuales generalizan el concepto a procesos en los que un evento depende sólamente de un estado actual, obtenido a partir de acciones realizadas anteriormente.[8] Este tipo de modelos parecen ser justamente a lo que apuntaba el enunciado, en "modelo probabilístico que asigne una probabilidad a cada palabra posible".

Por supuesto no estamos obligados a usar este modelo en particular ya que, según el enunciado, "existen diversos métodos para implementar la predicción de texto", como justamente hemos visto. Pero al final opté por profundizar en este tipo de modelos dado que, como veremos, presentan una alternativa relativamente sencilla para una solución que cumpla el propósito de predecir palabras, sin presentar demasiadas complicaciones en la implementación.

1.3.4.2 Sobre el *modelo de lenguaje de n-gramas de palabras*

Cabe aclarar que a este punto en la carrera tampoco presentamos todos los conocimientos para manejarnos con libertad con probabilidades, por lo que traté de enfocar la solución al propósito del trabajo lo más cercano a soluciones existentes cuyos resultados hayan sido probados. Tampoco esto me impide de ejercer modificaciones con el fin de adaptar las soluciones encontradas al problema planteado, pero no creo óptimo "inventar una solución" completamente nueva siguiendo el sentido común.

Primero y principal, como podemos ver en base a la impronta de este modelo de lenguaje, su enfoque principal está sobre un problema similar, el de predecir la *próxima* palabra en una oración. Esto difiere de nuestro problema en predecir una palabra en cualquier lugar de una oración. Por supuesto, si la palabra faltante a predecir estuviera al final de la oración, nuestro problema se reduciría al que el modelo trata resolver, pero en muchos otros casos no estaríamos aprovechando toda la información que tenemos sobre el contexto de una oración. Sin embargo, si tratamos de entender los principios del problema de predecir la próxima palabra, podemos tomar información con el fin de orientar mejor nuestra búsqueda de una solución.

El modelo se basa, como dice su nombre, en el uso de *n-gramas*. Este término hace referencia a una serie de n componentes consecutivas de un texto, sean éstas por ejemplo, letras, sílabas o palabras.[9] Es por esto que en el nombre del modelo se aclara que nos enfocamos en los n-gramas de *palabras*.

De esta forma, si tenemos un modelo basado en n-gramas donde $n = 2$, tenemos un modelo de *bigramas*, y podemos calcular la probabilidad de que una palabra aparezca en una posición en base a la palabra anterior a ella. Típicamente se nota esto con $P(W_i|W_{i-1})$, es decir, que la probabilidad de que aparezca la palabra W_i en la posición i depende de la palabra W_{i-1} en la posición anterior. De forma análoga, si $n = 3$ tenemos un modelo de trigramas, y notamos la probabilidad de que aparezca una palabra en una posición i en base a sus dos palabras anteriores, de la forma $P(W_i|W_{i-2}W_{i-1})$.

¿Cómo se calculan estas probabilidades? En estos modelos debemos disponer de un *corpus*, un conjunto de datos o textos ordenados de forma conveniente para poder usarlos en el cálculo de nuestras probabilidades. Entonces, dado un corpus dado por un conjunto de oraciones, podemos analizar las apariciones de dos o tres palabras consecutivas (bigramas y trigramas), con el fin de calcular las probabilidades que luego nos permitirían predecir palabras para los casos donde lo requerimos. Esto coincide con una situación como la del trabajo, donde nuestro corpus vendrían a ser el conjunto de oraciones combinadas por el programa en C, escritas por una persona.

Es así especialmente importante que tanto los textos como las frases con palabras faltantes estén escritos por una misma persona, ya que estamos usando para nuestras probabilidades un corpus con información basada sobre "la forma de escribir" de una persona particular. En consecuencia, la información obtenida de los textos de la persona tendrán más eficacia en frases escritas por la misma. Si quisiéramos desarrollar un modelo más general que aplicara a cualquier o un grupo de personas, deberíamos disponer de un corpus más amplio, proveniente de textos escritos por el grupo de personas al cual queremos apuntar nuestro modelo. Como veremos más adelante, las soluciones que involucran este modelo intentan incluir un corpus amplio y relacionado con el tipo de frases cuyas palabras se intentan predecir.

Entonces, si tenemos una frase como "las vacas vuelan", en un modelo de bigramas calcularíamos para la palabra "vuelan", $P(\text{vuelan}|\text{vacas})$ como la probabilidad de que la palabra "vacas" aparezca después de la palabra "vuelan". En un modelo de trigramas, esto sería $P(\text{vuelan}|\text{las vacas})$, la probabilidad que la palabra "vuelan" aparezca después de las palabras "las vacas". Para calcular esta probabilidad en un modelo de trigramas, obtendríamos el porcentaje de apariciones de trigramas de la forma "'las vacas vuelan", respecto de todos los trigramas que *empiezan* con "las vacas" en nuestro corpus. O, más bien, sería el porcentaje de apariciones de trigramas de la forma "las vacas vuelan" respecto a la cantidad de apariciones de los *bigramas* "las vacas", así considerando también casos en los cuales las dos palabras anteriores puedan no tener una palabra sucesora. Es decir:

$$P(\text{vuelan}|\text{las vacas}) = \frac{C(\text{las vacas vuelan})}{C(\text{las vacas})}$$

donde $C(x)$, donde x es un n-grama, es la cantidad de apariciones de x en el corpus.[6]

1.3.4.3 Mejorando el modelo para predecir la próxima palabra

Con todo esto en cuenta, una primera pregunta sería qué modelo usar, ¿un modelo de bigramas? ¿trigramas? ¿4-gramas? Es intuitivo pensar que mientras nuestro modelo considere secuencias más largas de texto, más preciso va a ser, ya que va a tener más información acerca de las palabras y los contextos que anteceden a la palabra. Sin embargo, esto tiene una contraprestación, y es que no es aplicable para todos los casos. Pensemos que en un modelo de trigramas, si queremos predecir la siguiente palabra en una oración que sólo empiece con "la", no tendremos información para poder completar la próxima palabra. Podríamos elegir alguna palabra de forma aleatoria, pero no parece la decisión más razonable dejarlo al azar.

Es por esto que lo mejor es utilizar múltiples modelos en simultáneo, por ejemplo el de bigramas y el de trigramas. Entonces, siempre que podamos usaremos el modelo de trigramas que sabemos que es más preciso. Pero si tenemos algún ejemplo como el anterior, en donde el modelo de trigramas no es aplicable, usaremos el modelo de bigramas. Más aún, si no tenemos siquiera una palabra con la que comenzar, podemos usar un modelo de *unigramas*, donde básicamente completaríamos con

la palabra más frecuente del corpus. Esta forma de usar "el mejor modelo aplicable" se lo conoce como *back-off*, o sea, como "retroceder" a la opción, a un modelo de menor precisión, pero que sea aplicable en nuestro caso.

Pero podemos mejorar esta forma de usar los modelos. Pensemos que si tenemos un trigrama que le gana a otro en todo el corpus indicando que que cierta palabra es la más probable, pero en el modelo de bigramas tenemos muchos que indican que la otra palabra es la más probable, al usar el modelo de trigramas primero (cuando es aplicable) estaríamos favoreciendo una opción que tampoco es la más ideal. Por ende, lo mejor es combinar los distintos modelos con *interpolación*. Esto sería, en vez de usar los modelos por separado, para cada palabra generamos un "puntaje" combinando las probabilidades de cada modelo, de forma que se favorezca a los modelos más precisos, pero sin dejar de lado la información de otros modelos de n-gramas de menor longitud.

Combinando los modelos de unigramas, bigramas y trigramas, podemos asignar valores $\lambda_1 = 0{,}05$, $\lambda_2 = 0{,}4$ y $\lambda_3 = 0{,}55$ (donde $\lambda_1 + \lambda_2 + \lambda_3 = 1$), tal que en el caso en el que sea aplicable el modelo de trigramas, consideramos que el puntaje para la palabra W_i si anteceden las palabras $W_{i-2}W_{i-1}$ resulta:

$$\text{Puntaje}(W_i|W_{i-2}W_{i-1}) = \lambda_1 P(W_i) + \lambda_2 P(W_i|W_{i-1}) + \lambda_3 P(W_i|W_{i-2}W_{i-1})$$

donde, repasando, $P(W_i)$ sería la probabilidad de la palabra W_i en el modelo de unigramas (notar que para este modelo no depende de que la anteceda alguna palabra), $P(W_i|W_{i-1})$ es la probabilidad para que la palabra W_i suceda a W_{i-1}, y $P(W_i|W_{i-2}W_{i-1})$ es la probabilidad para que la palabra W_i suceda a las palabras $W_{i-2}W_{i-1}$i.[6]

Notar que este puntaje, así, se encontraría en un intervalo $[0, 1]$ pues los valores $\lambda_1, \lambda_2, \lambda_3$ suman 1, y las probabilidades se encuentran en $[0, 1]$.

1.3.4.4 ¿Y la información de las palabras siguientes?

El modelo visto podría ser sin problemas utilizado en nuestro programa, pero como hemos antes mencionado, no estaría usando toda la información que disponemos. Existen casos donde saber la palabra siguiente puede ser muy útil para predecir la palabra faltante en una oración. Por ejemplo, sea la frase "la _ velada", es bastante claro que lo ideal sería utilizar un adjetivo en el espacio vacío, como por ejemplo, "la *dulce* velada". Sin embargo, usando el modelo anterior, sólo tenemos conocimiento de la palabra "la" para completar la palabra. Entonces es posible que, dependiendo de nuestro corpus, el modelo prediga una oración como "la *comida* velada", la cual apenas tiene sentido.

Es por ello que decidí buscar más a fondo, y me encontré con un par de papers escrito por Bhuyan et al.,[5] y un paper al que hace referencia, de Samanta et al.[7] (el cual expande en un proceso para la detección de palabras faltantes, lo cual en nuestro caso no nos es necesario). En el primero, se describe de hecho un problema bastante similar al que queremos solucionar en el TP: detectar y generar palabras faltantes en una oración en base a un corpus de bigramas y trigramas, usando el modelo de lenguaje de n-gramas de palabras como base. Así, la única diferencia a nuestro problema a resolver para el TP, es que en nuestro caso no necesitamos detectar la palabra faltante, porque ya nos es dada la palabra a reemplazar con un guión bajo. Esto nuevamente no nos impide de explorar las ideas del paper para tomarlas en la solución a nuestro problema.

En el paper antes mencionado de Bhuyan et al.,[5] se explora sobre la predicción de palabras en asamés, por lo que tampoco es una referencia directa a nuestro caso en el cual queremos predecir palabras en español. Cada lenguaje tiene sus propias características que pueden hacer que un modelo tenga menor o mayor efectividad. Pero sí podemos ver que se asemeja mucho a lo que antes habíamos analizado con la predicción de palabras siguientes. En el paper se plantea un modelo en el cual, en vez de considerar la palabra anterior y las dos anteriores como vimos antes, se considera la palabra anterior y la siguiente por medio de tres elementos: dos bigramas, uno izquierdo y otro derecho, y un trigrama donde la palabra a predecir se encuentra entre las dos adyacentes conocidas.

De esta forma, en el paper se plantea que para cada palabra W_i del corpus (en nuestro caso, que se encuentra en alguna de las oraciones que tomamos como entrada), extraer los bigramas izquierdos ($W_{i-1}W_i^j$), los bigramas derechos ($W_i^j W_{i+1}$) y los trigramas ($W_{i-1}W_i^j W_{i+1}$), para todos los $j =$

$1, 2, \ldots, n$, siendo n la cantidad de palabras distintas del corpus (o sea, en las oraciones). Luego, sus probabilidades pueden ser calculadas de forma similar al problema de predecir la próxima palabra en una oración, a partir del porcentaje de la cantidad de apariciones en las oraciones de un cierto bigrama (trigrama), respecto del *total de bigramas (trigramas) que empiezan/terminan (empiezan y terminan) en* W_{i-1}/W_{i+1} (W_{i-1} *y* W_{i+1}), según sean bigramas izquierdo/derecho o trigrama.

Notar que una diferencia a no dejar pasar de este modelo respecto al del ejemplo de predecir la próxima palabra, es que el denominador del cociente, es decir, el valor respecto del cual se toma el porcentaje, es directamente el total de trigramas que empiezan con W_{i-1} y terminan con W_{i+1}, por ejemplo. Lo que quiero aclarar es que, como al tomar el trigrama tomamos a ambos lados de la palabra, no podemos considerar "todos los bigramas" formados por W_{i-1} y W_{i+1}, como sí habíamos considerado en el otro problema antes, para tener en cuenta los casos en los cuales "las palabras no puedan tener palabra sucesora". Esto ya no aplica para este problema.

Luego, en el paper, introduce también la idea que habíamos discutido de usar interpolación, asignando λ_1, λ_2 y λ_3 para darle mayor o menor significancia a los bigramas izquierdo y derecho, y el trigrama, respectivamente. Así, el "puntaje" de una palabra W_i^j si la antecede la palabra W_{i-1} y la sucede la palabra W_{i+1} resulta:

$$\text{Puntaje}(W_i^j) = \lambda_1 P(W_i^j|W_{i-1}) + \lambda_2 P(W_i^j|W_{i+1}) + \lambda_3 P(W_i^j|W_{i-1}, W_{i+1})$$

donde $P(W_i|W_{i-1})$ es la probabilidad de que la palabra W_i sucede a W_{i-1}, $P(W_i|W_{i+1})$ es la probabilidad de que la palabra W_i anteceda a W_{i+1}, y $P(W_i|W_{i-1}, W_{i+1})$ es la probabilidad de que la palabra W_i suceda a W_{i-1} **y** preceda a W_{i+1} al mismo tiempo.

Para cerrar, omitiendo el proceso de detección de palabras faltantes el cual no requerimos para nuestro caso, podemos completar las palabras en los espacios que requiramos y generar, según el paper, una lista de sugerencias en función del puntaje calculado, de mayor a menor.

1.3.4.5 El algoritmo final

Considerando toda la información y alternativas exploradas, decidí consolidar lo que personalmente consideré mejor de todo lo visto para concluir en un algoritmo final a usar para la resolución del enunciado del trabajo.

A grandes rasgos, la solución utilizada tiene muchas similitudes con el del último paper antes descrito, pero con algunas modificaciones para lograr un algoritmo más eficiente, y con el fin de considerar algunos casos que omite el algoritmo original.

El algoritmo consiste en:

1. Extraer los unigramas, bigramas y trigramas de las oraciones leídas en el paso anterior.
2. Computar, siendo n la cantidad de palabras distintas del corpus:
   a) Para cada unigrama (W^j) ($j = 1, 2, \ldots, n$) (que resulta siendo nada más y nada menos que cada palabra), la probabilidad de ser elegida respecto al resto de las palabras del corpus. O más bien, el porcentaje de las palabras del corpus que representa:

   $$P(W^j) = \frac{C(W^j)}{\sum_{k=1}^{n} C(W^k)}$$

   Esto permite luego poder utilizar la palabra "más frecuente" del corpus si se nos presenta un caso donde los bigramas o trigramas no aplican (o sea, si tenemos que predecir una palabra "suelta" sin palabras anteriores o posteriores). También, nos permitirá considerar casos en los cuales una palabra se presente con mayor frecuencia que otras, ayudando en nuestras predicciones.
   b) Para cada bigrama, computar la probabilidad de ser utilizado respecto al resto de bigramas que *comienzan* en la misma palabra. Es decir, obtenemos la probabilidad de los *bigramas izquierdos* ($W_{i-1}W_i^j$) del corpus, donde computamos la probabilidad de que

una palabra W_i^j $(j = 1, 2, \ldots, n)$ suceda a otra W_{i-1}, respecto a todas las palabras que suceden a la primera a lo largo del corpus:

$$P(W_i^j | W_{i-1}) = \frac{C(W_{i-1} W_i^j)}{\sum_{k=1}^{n} C(W_{i-1} W_i^k)}$$

c) Para cada bigrama, computar la probabilidad de ser utilizado respecto al resto de bigramas que *terminan* con la misma palabra. Es decir, de forma análoga al ítem anterior, calculamos la probabilidad los *bigramas derechos* $(W_i^j W_{i+1})$ del corpus, donde computamos la probabilidad de que una palabra W_i^j preceda a otra W_{i+1}, respecto a todas las palabras que preceden a la segunda a lo largo del corpus:

$$P(W_i^j | W_{i+1}) = \frac{C(W_i^j W_{i+1})}{\sum_{k=1}^{n} C(W_i^k W_{i+1})}$$

d) Para cada trigrama, computar la probabilidad de ser utilizado respecto al resto de trigramas que *comienzan y terminan* con las mismas palabras que éste. Es decir, calculamos la probabilidad de los trigramas $(W_{i-1} W_i^j W_{i+1})$ del corpus, donde computamos la probabilidad de que una palabra W_i^j suceda a otra W_{i-1} **y** preceda a W_{i+1} al mismo tiempo, respecto a todas las palabras que cumplen esta misma condición a lo largo del corpus:

$$P(W_i^j | W_{i-1}, W_{i+1}) = \frac{C(W_{i-1} W_i^j W_{i+1})}{\sum_{k=1}^{n} C(W_{i-1} W_i^k W_{i+1})}$$

3. Con las probabilidades anteriormente obtenidas, podemos precomputar la palabra con mayor puntaje para (siendo n la cantidad de palabras distintas del corpus):

   - Todos los unigramas, que termina siendo la palabra más frecuente del corpus.
   - Todos los bigramas izquierdos. Esto significa, sea W_{i-1}^j una palabra de una oración, cuál es la palabra con mayor puntaje para sucederla W_i, para cada $j = 1, 2, \ldots, n$, en base a las probabilidades antes computadas.
   - Todos los bigramas derechos. Esto significa, sea W_{i+1}^j una palabra de una oración, cuál es la palabra con mayor puntaje para precederla W_i, para cada $j = 1, 2, \ldots, n$, en base a las probabilidades antes computadas.
   - Todos los trigramas. Esto significa, sean W_{i-1}^j y W_{i+1}^k, cuál es la palabra con mayor puntaje para completarse entre ellas W_i, para cada $j = 1, 2, \ldots, n$ y para cada $k = 1, 2, \ldots, n$, en base a las probabilidades antes computadas.

   donde calculamos el puntaje de forma similar a cómo lo hemos visto en el paper, únicamente incluyendo los unigramas que decidimos también considerar. Sean $\lambda_1, \lambda_2, \lambda_3$ y λ_4 tal que $\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 = 1$, correspondiéndose a coeficientes para los unigramas, bigramas izquierdos, bigramas derechos y trigramas respectivamente. Calcularemos el puntaje de una palabra W_i^j (para todo $j = 1, 2, \ldots, n$) si es precedida por W_{i-1} y sucedida por W_{i+1} como:

$$\text{Puntaje}(W_i^j) = \lambda_1 P(W_i^j) + \lambda_2 P(W_i^j | W_{i-1}) + \lambda_3 P(W_i^j | W_{i+1}) + \lambda_4 P(W_i^j | W_{i-1}, W_{i+1})$$

   Aclarar que, en caso de que la palabra faltante no tenga palabras anteriores y/o posteriores conocidas (por ejemplo, para una palabra faltante al comienzo o final de una oración), calcularemos el puntaje con los valores que no correspondan en 0. De esta forma, se mantienen las prioridades que se hayan dado a las probabilidades para los modelos según la longitud de n-gramas que utilicen, pero también brinda resultados de predicciones para todas las situaciones. Veremos más información al respecto en el próximo paso del algoritmo.

   Los valores para $\lambda_1, \lambda_2, \lambda_3$ y λ_4 deben tomarse con el fin de, como ya vimos, priorizar la mayor precisión que permiten los modelos de n-gramas de mayor longitud. Pero a la vez deben evitar casos como los mencionados con back-off, donde tener algún trigrama en nuestro corpus no debería de detenernos de elegir otra palabra tenga una cantidad significativa de coincidencias

en bigramas, aunque exclusivamente dentro del modelo de trigramas pueda tener una menor probabilidad.

Considerando esto, como también los valores que vimos que tienen mayores resultados en los papers mencionados[5],[7] y en algunas de las otras referencias tomadas en cuenta para este trabajo,[6] elegimos valores $\lambda_1 = 0{,}05$, $\lambda_2 = \lambda_3 = 0{,}2$ y $\lambda_4 = 0{,}55$. En el código los llamé como los "pesos" (*weights*) que le doy a las probabilidades de cada modelo.

Por un lado, tiene sentido que $\lambda_2 = \lambda_3$, ya que estamos asignando que conocer la palabra que antecede a una palabra faltante es tan importante como saber la que le sucede. Podría argumentarse que esto no es siempre así, por ejemplo con casos donde muchas veces las palabras están precedidas por artículos en el español, que no brindan mucha información única de la palabra a completar. Pero de forma análoga, también podemos encontrar muchos otros casos donde esto sucede con la palabra que le sigue a la faltante, como el uso de verbos comunes. Entonces, a rasgos generales, resulta razonable asumir esto.

También es coherente asignar un valor bajo a λ_1, pues la cantidad de veces que aparece la palabra en el corpus no nos brinda información sobre el contexto de una oración. Pero sí puede ser útil en la implementación para los casos particulares que ya hemos discutido, o para desempatar en la decisión de una palabra, para priorizar otra más frecuente.

Finalmente, tenemos que considerar que las probabilidades para los trigramas y los bigramas en nuestro modelo están sumamente vinculadas. Y es que si calculamos el puntaje de una palabra en un trigrama, para cada uno en el corpus también existen los dos bigramas izquierdo y derecho que lo conforman. Así, necesariamente $C(W_{i-1}W_i^jW_{i+1}) \leq C(W_{i-1}W_i^j)$ y $C(W_{i-1}W_i^jW_{i+1}) \leq C(W_i^jW_{i+1})$, como también $\sum_{k=1}^{n} C(W_{i-1}W_i^kW_{i+1}) \leq \sum_{k=1}^{n} C(W_{i-1}W_i^k)$ y $\sum_{k=1}^{n} C(W_{i-1}W_i^kW_{i+1}) \leq \sum_{k=1}^{n} C(W_i^kW_{i+1})$. Por lo tanto, tiene sentido asumir $\lambda_4 > \lambda_2 + \lambda_3$, pues de lo contrario, no estaríamos dándole importancia adicional a los trigramas del corpus respecto de los bigramas.

Además, el uso de estos coeficientes ha probado buenos resultados en las pruebas realizadas durante el desarrollo del programa, y con los archivos de prueba provistos por la cátedra.

4. Con la palabra de mayor puntaje ya precalculada para cada trigrama, bigrama izquierdo, bigrama derecho y unigrama (la palabra más frecuente del corpus), podremos rápidamente reemplazar los guiones bajos encontrados en las frases leídas según el contexto de cada palabra.

   Es el precómputo realizado en los pasos anteriores que nos permite obtener la palabra predicha de forma más rápida que la solución "naive" o "directa", que consistiría en iterar por todas las palabras del corpus para determinar la de mejor puntaje que correspondería a cada caso.

   En vez de eso, obtendríamos el mayor puntaje precalculado para todos los trigramas / bigramas / unigramas que contienen y aplican para la palabra restante. Es decir:

   - Si la palabra faltante W_i tiene palabras a su izquierda W_{i-1} y a su derecha W_{i+1}, obtendríamos la palabra de mejor puntaje entre todas las precalculadas del corpus que coincidan para el trigrama que empieza y termina en W_{i-1} y W_{i+1} respectivamente, para el bigrama izquierdo que empieza en W_{i-1}, para el bigrama derecho que termina en W_{i+1}, y la palabra más frecuente del corpus (el mejor unigrama).
   - Si la palabra faltante W_i tiene sólo palabras a su izquierda, tomaríamos la palabra de mayor puntaje entre todas las precalculadas del corpus que coincidan para el bigrama que empieza con W_{i-1}, y la palabra más frecuente del corpus.
   - De forma análoga, si la palabra faltante W_i sólo tiene palabras a su derecha, tomaríamos la palabra de mayor puntaje entre todas las precalculadas del corpus que coincidan para el bigrama que termina con W_{i+1}, y la palabra más frecuente del corpus.
   - Si la palabra faltante W_i no tiene palabras ni a derecha ni a izquierda (léase, para la oración "_"), reemplazaremos con la palabra más frecuente del corpus.

   Notar que usar el criterio de tomar máximos en vez de separar en casos y comprobar condiciones ayuda a una implementación más sencilla, cómo se puede observar en el programa.

Además, esto no afecta la respuesta, ya que efectivamente estamos haciendo lo mismo que la solución naive, donde tenemos ya precalculada la mejor opción para cada caso. Esto resulta cierto considerando que sólo tendrán significancia aquellas palabras que estén en unigramas / bigramas / trigramas registrados en el corpus alrededor de la palabra faltante, que son justamente para las cuales precalculamos los puntajes para obtener el mejor de todos (y su palabra correspondiente).

Los detalles faltantes relacionados principalmente a la eficiencia del algoritmo y las estructuras utilizadas para implementarlo se presentarán en el apartado de detalles de implementación.

1.3.4.6 Limitaciones del algoritmo

Uno de los interrogantes que pueden surgir alrededor del algoritmo final empleado, podría ser la validez del mismo.

Tanto para la solución de obtener la próxima palabra en una oración, como el algoritmo utilizado de obtener una palabra faltante en una oración, se toma una suposición como verdadera para poder aplicar el modelo de las cadenas de Markov. En el primer caso, que la resultante nueva palabra de una oración sólo depende de las escritas anteriormente a ella. Y la segunda, que la palabra resultante de una oración sólo depende de las escritas de forma anterior y posterior a ella.

Por supuesto, esta suposición no es realmente cierta, pero utilizarla nos brinda acceso a usar una herramienta que nos sirve como una aproximación al problema, con la intención de obtener buenos resultados. Sin embargo, como ya discutimos, existen modelos de lenguajes que obtienen resultados mucho mejores al algoritmo planteado, y debemos recordar que ni siquiera los mejores modelos generan resultados perfectos. Incluso más allá de ello, la predicción de una palabra en una oración realmente depende de muchos factores que hasta pueden resultar ambiguos para una persona, y que sin un criterio bien definido, puede ser ambiguo siquiera cuál es la respuesta correcta.

Otro punto a tener en cuenta es que el algoritmo planteado no es especialmente eficaz en los casos de los extremos de la oración que intentamos cubrir. Si bien es mejor que directamente no considerarlos, si presentamos una palabra faltante al comienzo o al final de una oración, muchas veces la palabra inmediatamente a su derecha o izquierda (respectivamente) no provee demasiada información sobre la palabra a predecir. Es por esto que la mayoría de las referencias relacionadas con solucionar el problema de predecir la próxima palabra, recomiendan usar bigramas y trigramas... de forma que se consideran hasta dos palabras hacia uno de los lados de la faltante.[6][4][9]

Podría ser de hecho una solución, considerar este modelo particular para los casos mencionados con el fin de obtener mejores resultados. Podría pensarse la oración como si se leyera de izquierda a derecha o al revés, para cada caso. Sin embargo, opté por no incluirlo por limitaciones de tiempo para la realización del trabajo, y personalmente valoré más tener consideración por los casos más frecuentes que sucederían en medio de las oraciones (considerando que las frases pueden tener una palabra faltante en cualquier posición).

Una optimización que también fue descartada en la búsqueda de soluciones para el problema planteado, es la idea de disponer de algún corpus externo incluido con el programa, que permita detectar casos en los cuales las predicciones realizadas por nuestro algoritmo no sean gramaticalmente correctas.

Esta idea se basa que, en general, mientras más grande sea el corpus de información, más preciso podrá ser nuestro algoritmo al disponer de mayor cantidad de combinaciones válidas de palabras en bigramas y trigramas. Lo único que habría que remarcar, es que este corpus ayudaría en la detección de errores de las predicciones que surjan del conjunto de oraciones provistas. Esto se debe a que, como ya hemos aclarado antes, el uso de oraciones escritas e incompletas escritas por una misma persona, mejora nuestras predicciones, ya que de esta forma nuestro algoritmo considera mejor expresiones o palabras que la persona acostumbre a utilizar más seguido.

Para este caso, nuevamente me presenté con limitaciones de tiempo para no incluirlo. Pero además, consideré que parte del objetivo del trabajo, su enfoque principal, se centraba en poder programar un algoritmo de predicción inicial, una aproximación, que logre resultados competentes.

La conclusión tras investigar los distintos aspectos del trabajo, y en particular el algoritmo mencionado, es que con este tipo de problemas siempre se pueden implementar mejoras. Estamos realmente planteando un algoritmo basado en razonamiento lógico y en base a información recopilada de otras fuentes, con el fin de obtener buenos resultados. Sin embargo, sin una definición completamente formal de "una buena predicción" o de una base teórica que nos permita obtener un resultado siempre correcto, queda a criterio personal dónde trazar la línea e incluir lo más importante dentro del período de tiempo estipulado. Es por esto que intento ser claro en mis valoraciones sobre las decisiones tomadas, para poder expresar mi motivación acerca de optar por una decisión respecto de otra.

1.3.5. Generar el archivo final con las frases completas

Tras ejecutar todo el algoritmo descrito en el paso anterior, tenemos las frases completas cargadas en un arreglo. Escribir a un archivo en Python es una tarea sencilla que no tiene mayores complicaciones, por lo que los detalles correspondientes se describen en el apartado de detalles de implementación.

2. Detalles de implementación

2.1. El programa en C

2.1.1. Tomar el nombre de la persona por argumento

Para asegurarnos que el programa en C reciba la cantidad de argumentos correcta, verificamos el parámetro `int argc;` recibida como argumento de la función `main`. Para nuestro caso, `argc` debe ser 2, pues el primero de los argumentos se corresponde con el nombre del ejecutable, y el segundo debería ser el nombre de la persona dado por el usuario (sin espacios).

Esto se puede hacer fácilmente incluyendo una condición al principio de la función `main()`, en la cual se advierta al usuario si la cantidad de argumentos recibidos es distinta de la esperada:

```
// ...
if (argc != 2) {
    puts("usage: ./main [person_name]");
    return 0;
}
// ...
```

2.1.2. Leer los archivos `.txt` escritos por la persona

Para primero escribir el archivo `archivos.txt` como uno vacío, podemos simplemente abrirlo en modo `"w"`, e inmediatamente cerrarlo:

```
void reset_files_list(char *filesListPath) {
    FILE *fp = fopen(filesListPath, "w");
    if (fp == NULL)
        handle_file_open_error_rwa(filesListPath, 'w');
    fclose(fp);
}
```

Donde `char *filesListPath` es la ruta a `archivos.txt`, y verificamos que podamos escribir en él verificando si nuestra referencia al archivo `fp == NULL`. Así, `handle_file_open_error_rwa()` sólo muestra un error al usuario para que sepa de la ruta inaccesible en caso de no poder realizar la escritura.

Lo relacionado a la concatenación de rutas y lectura de líneas desde `archivos.txt` se puede ver en el código provisto.

Un detalle a mencionar respecto al manejo de errores provenientes de la manipulación de archivos, es que pude comprobar que los errores generados por `cd` y `ls` para listar los archivos a partir de ser llamados con la función `system()`, **sí** se muestran en la salida de la ejecución del programa. Esto quiere decir que en los casos en los cuales la carpeta con los textos de la persona no exista en `Textos` o los archivos no sean accesibles por algún otro motivo (por ejemplo, permisos), el usuario vería mensajes de error no deseados. Si bien yo muestro un mensaje de error informando que no se pudo acceder al directorio con los Textos, éste se terminaría mostrando junto con los errores de `cd` y `ls`, que no es ideal.

Por lo tanto, tomé la decisión de modificar un poco el comando provisto por la cátedra para ejecutar con `system()`, con el fin de redirigir la salida de errores generada por los comandos a `/dev/null`, una ruta de archivos en Linux que descarta cualquier entrada que le llegue. Es común usar esta ruta para este preciso propósito, de eliminar los mensajes de error generados por comandos de la terminal. De esta forma, la función para ejecutar el comando resulta:

```
void list_texts(char *textsPath, char *filesListPath) {
    char command[MAX_BUF];
    sprintf(command, "cd %s 2>/dev/null && ls > ../../%s 2>/dev/null", textsPath,
↪   filesListPath); // Generar la cadena del comando con sprintf()
    system(command);
}
```

donde `char *textsPath` es la ruta a la carpeta de la persona en `Textos`, y `char *filesListPath` es la ruta a `archivos.txt`, donde se escribirán la lista de archivos en `textsPath` (si los comandos se ejecutan correctamente).

Notar que la parte concreta que agregué en los comandos de `cd` y `ls` es `2>/dev/null`, que le indica al procesador de comandos que la salida estándar de errores, `stderr`, identificada un *descriptor de archivo* número 2, sea redirigida a `/dev/null`. Así, los mensajes de error no se mostrarán por la salida del programa en la terminal, sino serán redirigidos a `/dev/null`, donde serán descartados.

De esta forma, en el caso en que la carpeta de la persona no exista en `Textos`, esté vacía, o sea innaccesible por algún otro motivo, sólo se mostrará el mensaje de error que yo incluí en mi programa.

Aclarar que si bien en el programa uso `fprintf()` tomando `stderr` como argumento para imprimir los mensajes de error en la salida de errores estándar, la redirección antes utilizada en los comandos ejecutados con `system()` sólo afectan a `cd` y `ls`. Es por esto que los mensajes que yo imprima a ésta salida se mostrarán sin inconvenientes.

Por otro lado, en cuanto a la lectura de los textos escritos por la persona, dijimos que los leeríamos mientras se procesan cada uno de los archivos. En consecuencia, resulta pertinente explicar este aspecto dentro de los detalles del próximo paso.

2.1.2.1 Limpiar los textos y unificarlos

Para limpiar los textos, dijimos que iríamos procesando los caracteres de cada uno de los archivos de entrada en base a las condiciones establecidas por el TP, como también en base a las decisiones tomadas en la descripción de la solución. Esto se traduce en el código en una función que ya recibe una referencia al archivo de entrada `FILE* input`, como también al archivo de salida `FILE *output`:

```
void sanitize_file(FILE* input, FILE* output);
```

Así, podemos ir leyendo los caracteres desde `input` uno a uno con `fgetc()` hasta el final del archivo, o sea, hasta que `fgetc()` devuelva `EOF`, mientras se escriben los caracteres a la salida con `fputc()`:

```
// ...
char c;
while ((c = (char)fgetc(input)) != EOF) {
```

```
    // condiciones que usan fputc() según sea necesario
    // ...
}
// ...
```

Estas referencias a los archivos son obtenidas en otra función que itera por las rutas de los archivos a combinar, llamando en cada una, la función mencionada `sanitize_file()`. Esta función se llama la misma referencia de salida en cada caso, para que se escriba a continuación de lo que se limpió para el archivo anterior:

```
void sanitize_files(int n, char **inputPaths, char *outputPath) {
    FILE* output = fopen(outputPath, "w");
    if (output == NULL)     // Manejo de errores en la escritura
        handle_file_open_error_rwa(outputPath, 'w');

    for (int i = 0; i < n; i++) {
        FILE *input = fopen(inputPaths[i], "r");
        if (input == NULL)  // Manejo de errores en la lectura
            handle_file_open_error_rwa(inputPaths[i], 'r');
        sanitize_file(input, output); // Se llama a la función con el mismo output
        fclose(input);
    }
    fclose(output); // El archivo de salida se cierra al final
}
```

donde `char **inputPaths` son las rutas de los archivos de entrada, los textos escritos por la persona, y `char *outputPath` es la ruta del archivo de salida, donde se combinan los textos sanitizados.

Por último, para el manejo de los errores antes mencionado, utilizamos una función a la cual le pasamos directamente la ruta del archivo para la cual la lectura / escritura falló, de forma que se la pueda a informar al usuario. Opté por no incluir en ésta la condición para verificar que `fopen()` haya devuelto `NULL` ya que consideré que ésta es relativamente sencilla de verificar en las funciones donde se abren los archivos, y evita tener que pasar la variable `FILE* fp` a la función de manejo de errores:

```
void handle_file_open_error_rwa(char *path, char mode) {
    switch(mode) {
        case 'r':   // Si se intentó leer el archivo con modo 'r'.
            fprintf(stderr, "There was an error reading from %s.\n",path);
            fprintf(stderr, "File may not exist or you may not have permission to
↪   read it.\n");
            break;
        case 'w':
        case 'a':   // Si se intentó escribir o concatenar al final del archivo
↪   con modos 'w' y 'a'.
            fprintf(stderr, "There was an error writing to %s.\n",path);
            fprintf(stderr, "Check the path exists and that you have permission
↪   to write in it.\n");
            break;
    }
    exit(1); // Finaliza la ejecución del programa. Evita que incurra en un error
↪   que lleve a su terminación abrupta (por ejemplo, segmentation fault).
}
```

El resto de los detalles pueden revisarse en el código del programa.

2.1.3. Llamar al programa en Python

Como dijimos, podemos formar el comando para llamar al programa en Python en una cadena, y llamar a la función `system()` con el comando como argumento:

```
void call_python_script(char *personName) { // Recibe nombre de la persona
    char command[MAX_BUF];  // Cadena en la cual almacenar el comando
    sprintf(command, "python3 %s %s", PYTHON_SCRIPT_NAME, personName); //
↪   Formamos el comando con sprintf()
    system(command); // Llamamos a system() con el comando como argumento
}
```

2.2. El programa en Python

2.2.1. Tomar el nombre de la persona por argumento

En Python, para recibir los argumentos que recibe el programa, disponemos de la variable `sys.argv`, a la cual podemos acceder importando el módulo `sys`. Ésta es una lista la cual podemos conocer su longitud usando `len(sys.argv)`. Así, usaremos este dato para verificar que la cantidad de argumentos sea 2, y mostrar instrucciones de uso del programa al usuario en caso contrario:

```
def main():
    if len(sys.argv) != 2:
        print("usage: python main.py [person_name]")
        exit(0)
    # ...
```

Aclarar que para poder usar la función `main()` como la primera que se ejecuta en nuestro programa de Python, indicamos en `main.py`:

```
# ...
if __name__ == "__main__":
    main()
```

2.3. Leer el `.txt` generado por el programa en C

Podemos leer las oraciones sanitizadas generadas por el programa en C, directamente abriendo el archivo en Python, usando la función `f.read()` tras abrir el archivo y tener una referencia a él en `f`. Luego, usamos la función `.splitlines()` que separa el texto en una lista de líneas (las oraciones), eliminando los saltos de línea `\n`. Es por esto último que preferimos este método a usar `f.readlines()`, que conserva estos saltos al final de cada línea:

```
def path_readlines_no_endline(path: str) -> list[str]:
    try:
        file = open(path,"r")  # Intentar abrir el archivo
    except:     # Manejar error en la lectura del archivo
        print(f'There was an error reading from {path}.')
        print("File may not exist or you may not have permission to read it.")
        exit(1)

    lines = file.read().splitlines() # Leer las líneas
    file.close()    # Cerrar el archivo
    return lines    # Devolver las líneas leídas
```

donde `path` es la ruta al archivo generado por el programa en C.

Notar que en el programa en Python decidimos manejar el error de lectura directamente dentro de la función. Esto se debe que el manejo de los archivos para este caso es más sencillo que en C, en el cual presentaba complicaciones en la detección del listado de los archivos de una carpeta con `ls`. Además es más sencillo el manejo de la memoria, pudiendo utilizar una función que devuelve fácilmente una lista para todas las operaciones de lectura utilizadas en el programa.

2.3.1. Leer el `.txt` de las frases con palabras faltantes

Para leer las frases escritas por la persona, utilizamos la misma función del paso anterior, pero indicándole como parámetro en este caso, la ruta del archivo de frases en vez del generado por el programa en C.

En ambos casos escribimos funciones sencillas que manejan cada caso particular, construyendo fácilmente la ruta del archivo que debe ser leído. Estas funciones como otros detalles menores, pueden verse en el código del programa.

2.3.2. Procesar las oraciones leídas en una estructura útil y predecir las palabras faltantes

En la descripción de la solución tomó un proceso de investigación hasta decidirme por un algoritmo que crea suficiente para la tarea encomendada en este paso. Durante esta descripción, hay algunos detalles de la solución que aclaré tenían como objetivo una mayor eficiencia del programa. A continuación aclararé estos detalles, como también las estructuras de de Python elegidas para implementar el algoritmo empleado.

2.3.2.1 Extraer los n-gramas

Para extraer los unigramas, bigramas y trigramas de las oraciones en el archivo generado en C, utilizamos una función que recibe una lista de oraciones, y devuelve un *diccionario*. Cada n-grama es representado como una tupla de n componentes, entonces el diccionario indica para cada n-grama extraído, la cantidad de veces que aparece en las oraciones. Opté por usar esta estructura ya que permite rápidamente ir agregando las distintas apariciones que un n-grama dado a medida se iteran por éstos en el texto. En el final, disponemos de un número de cantidad de veces que aparece cada n-grama, útil para el próximo paso del algoritmo, que utiliza estas cantidades para calcular probabilidades.

Para facilitar la lectura del código, definimos un tipo `Ngram = tuple[Word, ...]`, una tupla de hasta n componentes `Word`, con `Word = str`, para que sea también más legible que hablamos de n-gramas de palabras. Análogamente definimos para más adelante, los tipos `Unigram = tuple[Word]`, `Bigram = tuple[Word, Word]` y `Trigram = tuple[Word, Word, Word]`, para unigramas, bigramas y trigramas respectivamente.

Los detalles faltantes en relación a la obtención de los n-gramas se encuentran en el código del programa.

2.3.2.2 Calcular las probabilidades

Para calcular las distintas probabilidades del segundo paso del algoritmo, utilizamos los unigramas, bigramas y trigramas extraídos de las oraciones para computar probabilidades de unigramas, bigramas izquierdo y derechos, y trigramas, respectivamente.

Para el caso de los bigramas, debemos primero precomputar la cantidad de bigramas que empiezan o terminan con cierta palabra (para izquierdo o derecho). Y para trigramas, contar cuántos empiezan y terminan en cierto par de palabras.

Para ello, usaremos también diccionarios en Python, donde para cada comienzo (bigramas izquierdos), final (bigramas derechos) o comienzo y final (trigramas), vamos acumulando la cantidad de apariciones iterando por todos los bigramas o trigramas extraídos de las oraciones.

Luego, para el caso de los bigramas, iremos iterando por cada uno de ellos para calcular su probabilidad, utilizando el cociente entre la cantidad de veces que aparece en el corpus (obtenida en el paso anterior del algoritmo), respecto a la cantidad que antes contamos, de los que empiezan / terminan en cierta palabra (bigrama izquierdo o derecho respectivamente). Mientras, de forma similar para los trigramas, iteraremos por ellos para calcular su probabilidad con el cociente entre la cantidad de apariciones en el corpus (obtenida en el paso anterior del algoritmo), respecto a la cantidad de trigramas que comienzan y terminan con las mismas palabras que el trigrama en cuestión.

Notar que el uso de diccionarios nos permite en esta segunda iteración obtener rápidamente la cantidad de apariciones de aquellos que comienzan/terminan/comienzan y terminan con las mismas palabras, para cada bigrama/trigrama.

Finalmente, la probabilidad para cada bigrama o trigrama (según corresponda), es almacenada en otro diccionario para su posterior procesamiento en la etapa de precálculo de predicciones del algoritmo. Por conveniencia y legibilidad del código, para estos diccionarios definimos los tipos:

- `Probability = float`, para mayor claridad y dado que calculamos las probabilidades en un intervalo $[0, 1]$.
- `UnigramProbability = dict[Unigram, Probability]`, almacenando para cada unigrama, la probabilidad.
- `BigramProbability = dict[Bigram, Probability]`, y luego `LeftBigramProbability = BigramProbability` y `RightBigramProbability = BigramProbability`, almacenando para cada bigrama (izquierdo o derecho), su probabilidad. Notar que decidí primer definir la probabilidad para un bigrama general y luego utilizarlo para ambos bigramas izquierdos y derechos. Esto se debe a que me permite distinguir que no representan la probabilidad de las mismas cosas (pues claramente calculo las probabilidades de forma distinta), aunque representando que ambos utilizan una estructura idéntica para almacenar sus probabilidades.
- `TrigramProbability = dict[Trigram, Probability]`, almacenando para cada trigrama, su probabilidad definida en la descripción de la solución.

Como un último comentario, cabe aclarar que si bien podemos entender los unigramas como palabras individuales, opté por usar tuplas de un sólo elemento. Esto lo hice con el fin de generalizar la extracción de los n-gramas en el paso anterior, y porque permite representar el concepto de los n-gramas de una forma unificada sin tener que cambiar demasiado los conceptos o los tipos de datos que manejaba.

La implementación de los detalles explicados se puede observar en el código del programa.

2.3.2.3 Precalcular las predicciones

En cuanto al precálculo de las palabras de mayor puntaje, primero defino una función auxiliar que dado un trigrama $(W_{i-1}W_i^jW_{i+1})$, reciba las probabilidades de dicho trigrama, de los bigramas izquierdo $(W_{i-1}W_i^j)$ y derecho $(W_i^jW_{i+1})$ y del unigrama (W_i^j), contenidos, calcule el puntaje asignado a la palabra W_i^j. Este puntaje conglomera las probabilidades de la palabra como unigrama, como sucesor de W_{i-1}, como predecesor de W_{i+1}, y como sucesor y predecesor de ambos al mismo tiempo, utilizando los λ_i $(i = 1, 2, 3, 4)$ descritos en la descripción de la solución.

Con esto en cuenta, definimos, en base a lo descrito en la descripción de la solución:

- `WEIGHT_UNIGRAM = .05` por $\lambda_1 = 0{,}05$.
- `WEIGHT_BIGRAM = .2` y luego `WEIGHT_LEFT_BIGRAM = WEIGHT_BIGRAM` y `WEIGHT_RIGHT_BIGRAM = WEIGHT_BIGRAM`, representando $\lambda_2 = \lambda_3 = 0{,}2$.
- `WEIGHT_TRIGRAM = 0.55` por $\lambda_4 = 0{,}55$.

Entonces, nuestra función para calcular el puntaje (*score*) resulta:

```
def calc_score(unigram: Probability = 0.0, leftBigram: Probability = 0.0,
↪  rightBigram: Probability = 0.0, trigram: Probability = 0.0) -> Score:
    return WEIGHT_UNIGRAM * unigram + WEIGHT_LEFT_BIGRAM * leftBigram +
    ↪  WEIGHT_RIGHT_BIGRAM * rightBigram + WEIGHT_TRIGRAM * trigram
```

donde `Probability = float` y también `Score = float`, para mayor claridad y legibilidad del código.

Notar que como también mencionamos en la descripción de la solución, podemos utilizar esta función para calcular la palabra de mayor puntaje para bigramas izquierdos y derechos, como también unigramas. En esos casos, sólo paso por argumento las probabilidades disponibles, y el resto

resultan `0.0` dada la declaración de la función que así las establece si no se pasan argumentos. Por ejemplo, si quiero computar el puntaje para bigramas derechos, sólo pasaría por argumentos a la función `calc_score()` los valores de las probabilidades que conozco para `unigram` y `rightBigram`, llamando a la función como `calc_score(unigram = [valor], leftBigram = [valor])`.

Cada una de las funciones que calculan las predicciones para los distintos trigramas / bigrama izq. / bigrama der. / unigrama (palabra más frecuente del corpus), toman como parámetro los diccionarios con las probabilidades calculadas en el paso anterior del algoritmo, y devuelven, en cada caso:

- Para los unigramas, se devuelve la palabra más frecuente del corpus, un `PredictedWord`. Para legibilidad, se define `UnigramPredicted = PredictedWord`
- Para los bigramas derechos, se devuelve un diccionario donde para cada palabra que se encuentra al comienzo de un bigrama en el corpus, se tiene el `PredictedWord` corresondiente, la palabra de mayor puntaje predicha para ese comienzo de bigrama. Se define para mayor legibilidad `LeftBigramPredicted = BigramPredicted`, donde `BigramPredicted = dict[Word, PredictedWord]`.
- De forma análoga al punto anterior, para los bigramas izquierdos, se devuelve `Right BigramPredicted`, donde `RightBigramPredicted = BigramPredicted`, la palabra predicha de mayor puntaje para cada uno de las palabras que finalizan en algún bigrama del corpus. En ambos casos uso estos tipos definidos para mayor claridad y representar que aunque la información representada para ambos bigramas es diferente, utilizo el mismo tipo de estructura para almacenarla.
- Y para los trigramas, se devuelve un `TrigramPredicted` donde definimos `TrigramPredicted = dict[tuple[Word,Word], PredictedWord]`, ya que para cada comienzo y final de un trigrama en el corpus, se tiene la palabra predicha de mayor puntaje si se la coloca entre medio. Como se puede observar, representamos este par de palabras comienzo y final con una tupla de dos cadenas. Esto no significa que conformen un bigrama, aunque se los represente de la misma manera, pues claramente estas palabras no se encuentran de forma consecutiva, entonces **no** sería correcto usar el tipo para los bigramas que habíamos definido anteriormente.

donde `PredictedWord = tuple[Word, Score]`, una vez más para mayor legibilidad, representando la palabra predicha según corresponda, incluyendo su puntaje (*score*).

Usar una vez más, diccionarios para las predicciones precalculadas, luego nos permitirá acceder a ellos rápidamente en la próxima etapa del algoritmo, donde completaremos las palabras faltantes de las frases escritas por la persona.

2.3.2.4 Reemplazando las palabras faltantes

Finalmente, para completar las palabras faltantes, iteramos la lista de frases incompletas leídas, y separamos sus palabras usando `.split()`, para luego buscar el índice de aquella que sea un guión bajo usando `words.index('_')`, asumiendo que todas las oraciones tienen un _, como lo indica el enunciado.

Luego, dependiendo de este índice, podemos fácilmente determinar si se tiene una palabra adelante o atrás. Para todos los casos pertinentes, descritos en la solución general, vemos de obtener el máximo `PredictedWord` en base a todos los precalculados para bigramas y trigramas que contienen a la palabra faltante, en función de las palabras disponibles antes y después de dicho guión bajo. Definiremos que un `PredictedWord` es mayor que otro si el puntaje del primero supera el del segundo.

Con la palabra predicha de mayor puntaje, podemos reemplazar el guión bajo en la lista de palabras por esta palabra, y concatenarlas con espacios para formar la oración completa.

En este caso, a diferencia de las funciones en C, consideré innecesario separar la función que cumple esta tarea para una lista de oraciones, respecto de otra que pueda resolverla para una sóla oración. El motivo de esto es que en ningún otro caso requerimos de hacer el uso de resolver una oración por separado, además que la diferencia entre resolver para una oración o para muchas,

únicamente radica en iterar elementos de una lista (estructura sencilla de manejar en Python). Es más, resulta razonable para un conjunto de datos de predicciones precalculados, aprovecharlo para múltiples frases, que de hecho es de utilidad posteriormente en el testing.

Los detalles concretos se pueden encontrar en el código del programa.

2.3.3. Generar el archivo final con las frases completas

Para terminar con el programa, podemos escribir las frases completas al archivo de salida correspondiente al nombre de la persona. Esto se reduce a una función que escriba cualquier lista de líneas en una ruta dada.

De forma similar a cuando escribíamos la salida sanitizada en C, en este caso también nos aseguraremos de agregar un salto de línea al final de las frases completas.

Otra cosa que tendremos en cuenta, es la posibilidad de que el archivo de destino sea inaccesible o no exista la carpeta que lo debe contener. Para esto, implementé una verificación muy similar a la descrita en la lectura de archivos, para "atrapar" la excepción que surja en estos casos al intentar abrir el archivo.

Así, la función resulta:

```
def path_writelines_newline(lines: list[str], path: str):
    try:
        file = open(path,"w")    # Intentar abrir el archivo
    except:      # Manejar error en la escritura del archivo
        print(f'There was an error writing to {path}.')
        print("Check the path exists and that you have permission to write in
        ↪  it.")
        exit(1)

    if lines: # Concatenar las líneas con \n entre ellas y al final, pero evitar
    ↪  escribir un \n si no hay líneas que escribir
        file.writelines('\n'.join(lines) + '\n')
    file.close()  # Cerrar el archivo
```

donde `path` es la ruta a la cual escribir las oraciones, y `lines` son las líneas a escribir en la ruta, las susodichas oraciones.

Un detalle menor a aclarar en la excepción, es el uso de `exit(1)`, en vez de `exit(0)`, pues este número indica el estado de salida del programa, el cual se le informa al programa que lo llama. Cuando se usa el `0`, se entiende que se informa de una finalización correcta del programa, mientras un número distinto de `0` es indicador de algún error. En particular, usar `1` indica esto como un error general, porque claramente experimentamos un problema al intentar leer del archivo.

Sin embargo, este detalle es justamente menor, pues no influye en el uso de nuestro programa ya que no usamos este código de error. Pero opté por hacerlo de esta forma igualmente, pues creí pertinente hacerlo dado que es lo más apropiado para la forma en la que el programa finaliza en esta situación. Esto incluye también a los `exit(1)` usados en la verificación `fp == NULL` dentro del programa en C.

Aquellos detalles no mencionados pueden verse en el código del programa.

3. Correr el programa y los tests

Para correr el programa, en el directorio raíz de los archivos del proyecto se presenta un archivo `Makefile` para automáticamente compilar los archivos fuente en C del programa. Usando el comando `make` en la consola, debería generarse en la misma carpeta el ejecutable `./main`. Así, podrá correr todo el programa con los archivos de `nombre_persona`, ejecutando el comando `./main nombre_persona`.

Dentro del directorio principal se encuentran dos carpetas, una llamada `c`, y otra llamada `python`.

La carpeta `c` tiene todos los archivos relacionados al programa en C, a excepción de `main.c` con la función `main()`, que necesita estar en el mismo directorio que las carpetas de `Textos`, `Entradas`, `Frases` y `Salidas` (o, en concreto, el ejecutable, y el programa de Python deberían, pero para este propósito me resultó conveniente también dejar el archivo `main.c` junto con estas carpetas).

De forma análoga, la carpeta `python` contiene todos los archivos relacionados al programa de Python, excepto `main.py` que se debe encontrar en el mismo directorio que las carpetas antes mencionadas.

Las carpetas de cada uno tienen una carpeta `src`, con todos los archivos de código fuente relacionado a cada programa, y una carpeta `tests`, con funciones de testing sobre las funciones más importantes de cada uno de los programas. Para correr los tests:

- En Python, usamos la herramienta usada en la materia `pytest`. Para correr los tests, es sólo cuestión de correr el comando `pytest` en la terminal, estando dentro de la carpeta `tests` de `python`.
- En C, usamos la función `assert()` de la librería `assert.h` como elemento principal de nuestro testing. En consecuencia, los tests no son más que código en C que debe ser compilado para correr. Para ello, dentro de la carpeta `tests` existe otro `Makefile`, de forma que si corremos `make` en ella, se compilará un ejecutable `./run_tests` para poder correr los tests del programa.

  Este `Makefile` realmente no tiene mucho contenido por su cuenta, y funciona como un "acceso directo" al comando `make test` del archivo `Makefile` principal en la raíz del proyecto.

Como detalle adicional, se pueden eliminar los ejecutables creados por la compilación en C, así como también archivos objeto creados por este motivo dentro de `c/build`, con el comando `make clean`. Éste se puede ejecutar tanto en el directorio principal del proyecto como en el directorio de `tests` de C.

4. Documentación (bonus)

Como un adicional a lo pedido para el trabajo, con el fin de desarrollar un código de mayor calidad, opté por comentar detalladamente el propósito de las distintas funciones y variables del programa, tanto en C como Python. Esta documentación se encuentra en los archivos cabecera `.h` en el caso de C, y directamente arriba de las funciones en Python. El formato respetado para los comentarios va en línea con el aceptado por Doxygen, una herramienta para poder generar documentación de código en múltiples lenguajes de programación, entre los que se encuentran C y Python.

Por esto, en la carpeta `doc` dentro del directorio principal del proyecto, se ha generado en la carpeta `html`, una interfaz web de documentación de los archivos del programa, accesible mediante `index.html` (abrir en cualquier navegador). Esta documentación incluye detalles sobre el funcionamiento de las distintas partes del código, y repite necesariamente algunas de las decisiones explicadas en este trabajo.

También puede optar por generar la documentación de forma local, usando el comando `doxygen config` dentro de la carpeta `doc` (requiere de instalar el paquete `doxygen` claramente). Éste hace uso del archivo de configuración allí contenido para generar la interfaz web de documentación en base a la versión más actual del código fuente.

Puede acceder en dicha interfaz a la documentación de cada uno de los archivos, accediendo a la opción `Files` → `Files List` para elegir entre los distintos archivos fuente documentados.

Referencias

[1] C reference: system(). https://cplusplus.com/reference/cstdlib/system/. Accedido: 8/12/2023.

[2] Why should text files end with a newline? https://stackoverflow.com/questions/729692/why-should-text-files-end-with-a-newline. Accedido: 8/12/2023.

[3] word2vec. https://www.tensorflow.org/text/tutorials/word2vec. Accedido: 9/12/2023.

[4] Lecture notes in prediction and part-of-speech tagging. https://www.cl.cam.ac.uk/teaching/2004/NatLangProc/slides3.pdf, 2004. Accedido: 10/12/2023.

[5] M. P. Bhuyan and S. K. Sarma. Generation of missing words in assamese text using n-gram based model. *Journal of Physics: Conference Series*, 1706(1):012166, December 2020.

[6] Vitou Phy. Language model concept behind word suggestion feature. https://towardsdatascience.com/sentence-generation-with-n-gram-21a5eef36a1b. Accedido: 9/12/2023.

[7] Pratip Samanta and Bidyut Baran Chaudhuri. A simple real-word error detection and correction using local word bigram and trigram. In *ROCLING/IJCLCLP*, 2013.

[8] Wikipedia contributors. Markov chain — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Markov_chain&oldid=1187542218, 2023. Accedido: 9/12/2023.

[9] Wikipedia contributors. N-gram — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=N-gram&oldid=1189017569, 2023. Accedido: 9/12/2023.

[10] Wikipedia contributors. Natural language processing — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Natural_language_processing&oldid=1187330712, 2023. Accedido: 9/12/2023.

[11] Wikipedia contributors. Word n-gram language model — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Word_n-gram_language_model&oldid=1180183848, 2023. Accedido: 9/12/2023.