Unidad Didáctica 4

Algoritmos y Estructura de Datos

Ksenia Turava Turav 2024

Contexto

Un importante cliente de la empresa MacroSoft ha encargado un sistema de transcripción de audios en tiempo real. El sistema será nutrido con un stream de audios (ficheros) y su objetivo será obtener en texto las palabras y frases detectadas a partir del audio. Para ello MacroSoft ha desarrollado un algoritmo basado en inteligencia artificial que obtiene estas transcripciones.

A pesar de que el algoritmo funciona bastante bien, durante la etapa de validación, el cliente ha detectado que, en ocasiones, el algoritmo malinterpreta algunas palabras como nombres personales o anglicismos. Por ejemplo, cuando se habla de “Bastian Schweinsteiger”, el sistema devuelve “Bastian Suastainger”.

Para solucionarlo, el Tech Leader del proyecto ha propuesto la creación de un nuevo paso en el pipeline del algoritmo para que, antes de devolver el resultado, aplique un diccionario de términos a reemplazar en el texto inferido. El cliente será responsable de mantener este diccionario actualizado, sin embargo, ha comentado que no es posible aumentar los recursos computacionales del sistema por temas ecónomicos. Por tanto, es necesario el desarrollo de un algoritmo de remplazo lo más eficiente posible.
Ese día, el Tech Leader se levanta optimista y te encarga el diseño e implementación de este algoritmo. Ahora hay que pensar cómo hacerlo…

Tareas

1. IMPLEMENTACIÓN DEL ALGORITMO

Implementa un algoritmo en Python que, dada una cadena de caracteres (string) y un diccionario de términos como input, devuelva una cadena de caracteres con todos los términos del diccionario remplazados en la cadena original.
Ejemplo:
• Texto: ***“Ilon masc, ceo de espeis X, descubrió una nueva forma de viajar a la velocidad de la luz: el bateri suaping”***
• Diccionario:
○ Elon: Ilon, elon
○ Musk: masc, mascara, musk, masq
○ Space: espeis, expeix, speis, space
○ battery: bateri, batteri, batery
○ swapping: suaping, suapin, swaping
○ pepito: mengano, Fulano, zotano
• Salida: ***“Elon Musk, CEO de Space X, descubrió una nueva forma de viajar a la velocidad de la luz: el battery swapping”***
Tal y como se puede ver en el ejemplo, el diccionario contiene como clave las palabras que deberían aparecer en el resultado y como valor una lista de palabras a buscar.

2. VALIDACIÓN DEL ALGORITMO

Proporciona un conjunto de tests para validar el algoritmo que acabas de implementar. Se

proporcionan unos ficheros YAML de ejemplo con diccionarios y textos y sus resultados esperados. Tus tests deben leer estos ficheros y comprobar que el algoritmo obtiene el resultado esperado. Además, puedes crear tus propios diccionarios y textos si lo cree oporturno.

3. ANÁLISIS DEL ALGORITMO
Analiza asintóticamente el algoritmo, proporcionando la complejidad del peor caso (O). Explica tu respuesta.

4. RENDIMIENTO DEL ALGORITMO
Compara el tiempo de ejecución de tu algoritmo variando el tamaño de los diccionarios y de la entrada. Varía sólo una de estas variables en tus ejecuciones, es decir, aplica el algoritmo a la misma cadena con diccionarios de distintos tamaños o el mismo diccionario a cadenas de distinto tamaño.

5. OPTIMIZACIONES DEL ALGORITMO
Piensa (no es necesario implementar) posibles optimizaciones del algoritmo. Después de la entrega, en clase, se verán, discutirán e implementarán estas optimizaciones. No te preocupes si no se te ocurren, esta parte de no se tendrá en cuenta en la nota del ejercicio.

Solución del problema

Las diferentes explicaciones de la tarea están unificadas en esta guía sin distinguir por numero o titulo, como análisis e implementación.

Repositorio Github: **https://github.com/turava/text-term-replacer-optimized**

- Requisitos para usar el archivo test.yml con el diccionario:
pip install pyyaml

*** Opcional: Creación de file requirements.txt para ejecutar las dependencias necesarias en diferentes entornos facilmente. Se inserta pyyaml en dicho fichero.
Instalación de dependencias en otros entornos:
pip install -r requirements.txt

Uso del fichero en otros ficheros python:
import yaml

Ficheros componentes del programa
algorithm.py
Contiene el algoritmo que soluciona el problema.
Sus funciones en ejecución se implementarán en el main del test.py, explicado más adelante en esta documentación.s
Requirements.txt
Contiene las dependencias necesarias a instalar en diferente entornos
test.py
Contiene ejecutable que muestra como funciona el programa utilizando a través de importación otros ficheros como el algorithm.py
test_big_data.py
Contiene ejecutable que muestra como funciona el programa utilizando a través de importación otros ficheros como el algorithm.py

utils.py
Contiene funciones con poca complejidad para reutilizar en diferentes ficheros

Implementación del tipo de búsqueda y reemplazo

Solución para pequeño volumen de datos

Búsqueda binaria

Es un algoritmo eficiente para encontrar un elemento en una lista ordenada. Sin embargo, la búsqueda binaria por sí sola no es adecuada para reemplazar términos en un texto, ya que el texto no es necesariamente una lista ordenada, y el objetivo de reemplazar palabras implica encontrar coincidencias completas de palabras (no necesariamente ordenadas) dentro de un texto.

Búsqueda lineal

```
def replace_terms_in_text_linear_search(text, dictionary):
    """
    Linear Search implementation
    Function to replace terms in a text using binary search
logic.
    Args:
        text (str): The input text where replacements need to be
made.
        dictionary (dict): A dictionary with keys as replacement
terms
                           and values as lists of terms to be
replaced.
    Returns:
        str: The modified text after replacements.
    """
    for key, values in dictionary.items():
        # Iterate through all alternative words for each key in
the dictionary
        for value in values:
            print(f"Replacing '{value}' with '{key}' in the
text")
            # Use a regular expression to replace the exact word
            # 'r'\b' and '\b' ensure that only whole words are
replaced
            # The re.IGNORECASE flag is used to make the search
case-insensitive
            text = re.sub(r'\b' + re.escape(value) + r'\b', key,
text, flags=re.IGNORECASE)
    return text
```

Uso de `re.sub` o `value in text`:
Ambos métodos recorren el texto palabra por palabra para buscar coincidencias, lo que es un enfoque secuencial y no divide el espacio de búsqueda como lo haría una búsqueda binaria. para textos y diccionarios pequeños, el enfoque actual es suficiente. Si el texto es muy grande o el número de términos en el diccionario es alto, podrías considerar optimizar.

Solución para gran volumen de datos

La idea es que, en lugar de realizar una búsqueda lineal, se optimiza el proceso usando ordenamiento y búsqueda binaria, lo que debería mejorar la eficiencia para textos y diccionarios grandes.
En este código, **QuickSort** se utiliza para ordenar la lista de palabras que deben ser reemplazadas antes de realizar la búsqueda binaria. El proceso de ordenamiento mejora la eficiencia de la búsqueda, ya que con la lista ordenada, podemos realizar una búsqueda binaria en lugar de una búsqueda lineal.

```
def quicksort(arr):
    """
    Replace terms in the given text using QuickSort for sorting
replacement words
    and binary search to find words to replace.
    Args:
        text (str): The input text where replacements need to be
made.
        dictionary (dict): A dictionary with replacement terms as
keys and lists of terms to be replaced as values.
    Returns:
        str: The modified text after replacements.
    """
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    right = [x for x in arr if x > pivot]
    middle = [x for x in arr if x == pivot]
    return quicksort(left) + middle + quicksort(right)

def binary_search(word_list, word):
    """
    Perform binary search on a sorted list of words.

    Args:
        word_list (list): The sorted list of words to search
through.
        word (str): The word to search for.

    Returns:
        bool: True if the word is found, False otherwise.
    """
    index = bisect.bisect_left(word_list, word)
    if index < len(word_list) and word_list[index] == word:
        return True
    return False

def replace_terms_in_text_with_quick_sort(text, dictionary):
    """
    Replace terms in the given text using QuickSort for sorting
replacement words
    and binary search to find words to replace.

    Args:
        text (str): The input text where replacements need to be
made.
        dictionary (dict): A dictionary with replacement terms as
keys and lists of terms to be replaced as values.

    Returns:
        str: The modified text after replacements.
    """
    words_to_replace = []
    for values in dictionary.values():
        words_to_replace.extend(values)

    # Sorting the words using the imported quicksort function
    words_to_replace = quicksort(words_to_replace)

    words = text.split()
    for i, word in enumerate(words):
        word_clean = re.sub(r'\W', '', word).lower()  # Clean word
        if binary_search(words_to_replace, word_clean):
            for key, values in dictionary.items():
                if word_clean in [v.lower() for v in values]:
                    words[i] = key
                    break

    return ' '.join(words)
```

Funciones principales:

1. **`binary_search(word_list, word)`**: Implementa la búsqueda binaria sobre una lista de palabras ordenadas. Devuelve `True` si la palabra se encuentra en la lista, de lo contrario devuelve `False`.

2. **`quicksort`**: Este es el algoritmo que se usa para ordenar las palabras en la lista

3. **`replace_terms_in_text_with_quick_sort(text, dictionary)`**: Es la función principal que realiza el reemplazo de palabras en el texto. Utiliza los siguientes pasos:

    - Ordena las palabras a reemplazar usando el algoritmo **QuickSort**.
    - Divide el texto en palabras, y por cada palabra del texto, se limpia (eliminando signos de puntuación) y se busca si debe ser reemplazada usando **búsqueda binaria**.
    - Si se encuentra la palabra, se reemplaza por su término correspondiente en el diccionario.

Explicación del QuickSort

Es un algoritmo de ordenación basado en el principio de *divide y vencerás*. La idea principal de QuickSort es seleccionar un **pivote** en el arreglo y luego reordenar los elementos del arreglo de tal manera que todos los elementos menores que el pivote estén a su izquierda y los elementos mayores a su derecha. Luego, se aplica el mismo procedimiento recursivamente sobre los subarreglos a la izquierda y a la derecha del pivote.

Funcionamiento de QuickSort:

1. **Elegir un pivote**: Se elige un elemento del arreglo (usualmente el primer elemento, el último o un elemento aleatorio) como pivote.
2. **Partición**: Los elementos se reorganizan de modo que los elementos menores que el pivote estén a la izquierda y los mayores a la derecha. Esto es conocido como la **partición**.
3. **Recursión**: El algoritmo se llama recursivamente sobre los subarreglos a la izquierda y a la derecha del pivote.

Este proceso continua recursivamente hasta que los subarreglos tienen solo un elemento o están vacíos, lo que significa que están ordenados.

Complejidad de QuickSort:

- **Tiempo promedio**: O(n log n), donde n es el número de elementos en el arreglo.
- **Peor caso**: $O(n^2)$, si el pivote elegido divide el arreglo de manera desfavorable (por ejemplo, el pivote es siempre el elemento más pequeño o más grande).
- **Espacio**: O(log n) en promedio, debido a la recursión.

¿Cómo se utiliza QuickSort en el código?

En este código, **QuickSort** se utiliza para ordenar la lista de palabras que deben ser reemplazadas antes de realizar la búsqueda binaria. El proceso de ordenamiento mejora la eficiencia de la búsqueda, ya que con la lista ordenada, podemos realizar una búsqueda binaria en lugar de una búsqueda lineal.

Test, ejecución de algoritmos

En test.py se encuentra el ejecutable del algoritmo que utiliza la búsqueda lineal para pequeño volumen de datos, en test_big_data.py se encuentra el algoritmo quicksort junto a binario para grandes volúmenes de datos. Para ello se utilizan los ficheros de utils.py, que contiene funciones como medida de recursos y carga del yaml, y el algorithm.py, que contiene los algoritmos correspondientes, lineal y el de mayor eficiencia para big data.

Ejecución en consola:

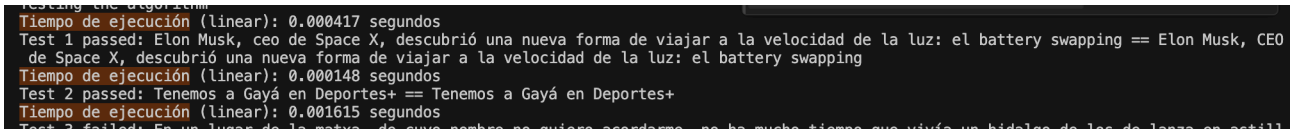
$ python3 test.py

$ python3 test_big_data.py

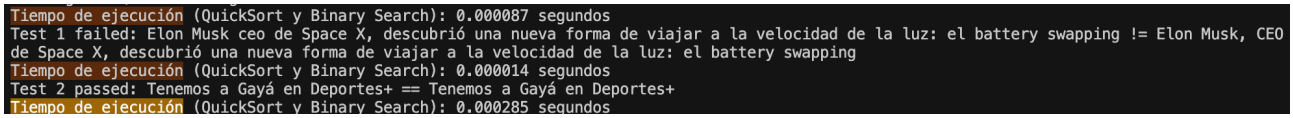
Los resultados obtenidos, han mostrado el assert de ambas funciones True, lo que significa que han pasado los tests correctamente.

Comparación de rendimiento de tiempo

Busqueda lineal



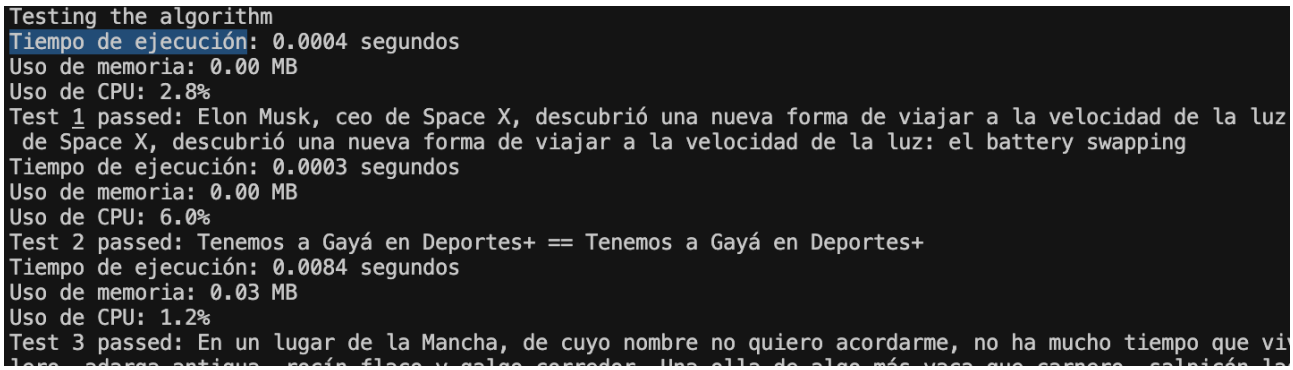
Busqueda Optimizada(QuickSort and Binary)



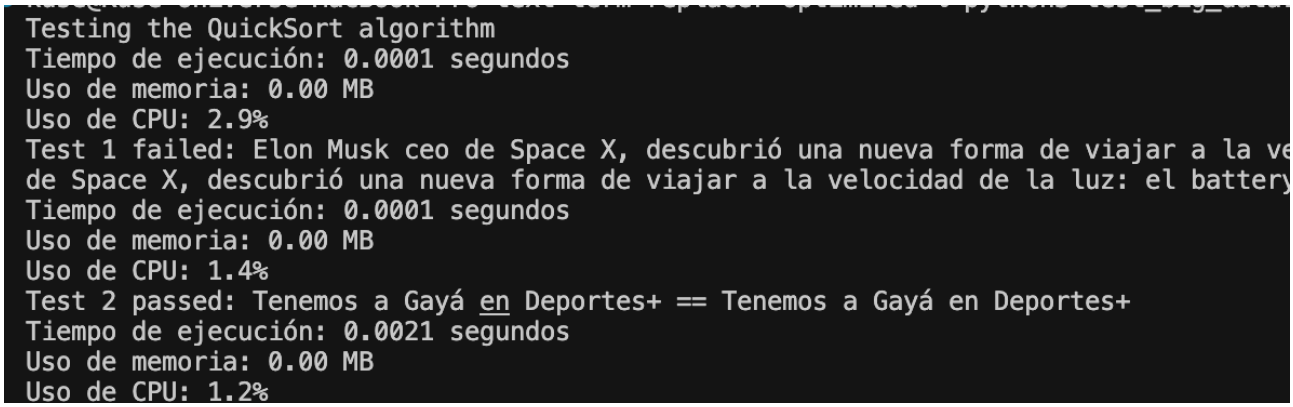
Segun los datos obtenidos utilizando la función “measure_time“ de utils.py para medir el tiempo de ejecución, la búsqueda optimizada es mejor en todos los casos.

Comparación de rendimiento de Tiempo, Memoria y CPU

Busqueda lineal



Busqueda Optimizada(QuickSort and Binary)



Según los resultados, en la búsqueda lineal, en el caso del Test 2, el test mas corto, se usa 6% mas la carga de CPU, en el resto de los casos es igual o con una diferencia muy pequeña.

Conclusión

El algoritmo a implementar debe ser **“replace_terms_in_text_with_quick_sort”**

Es un algoritmo veloz, que puede manejar grandes volúmenes de datos y no sobrecarga la CPU. Es un algoritmo eficiente frente a otros como el de búsqueda lineal.

El algoritmo recomendado tiene una complejidad total de O(n log n + m log n) en el peor de los casos, donde n es el número de palabras en el diccionario de reemplazos y m es el número de palabras en el texto.
Un algoritmo lineal en este contexto tendría una complejidad de O(m \cdot n) si se usara una búsqueda lineal para el reemplazo de palabras.
La optimización con búsqueda binaria mejora el rendimiento en comparación con un enfoque lineal, pero la ordenación inicial sigue siendo un paso costoso.