Taller-3-4 Buscador

1st Hugo Alejandro Latorre Portela
Fundación Universitaria Konrad Lorenz
Bogotá, Colombia
hugoa.latorrep@konradlorenz.edu.co

***Abstract*—Esta publicación proporcionará detalles sobre el desarrollo y la implementación de ambos algoritmos. Además, se explorará el papel de los índices invertidos en la recuperación de información, destacando su papel fundamental en los motores de búsqueda y el procesamiento de textos a gran escala. Utilizando estos algoritmos y su comprensión de los índices invertidos, se pretende mejorar la capacidad para analizar y extraer información valiosa de grandes cantidades de datos de texto.**

***Index Terms*—Optimización, recursos, lógica, complejidad, análisis**

I. Introduction

El procesamiento eficiente de grandes conjuntos de datos de texto es esencial en los campos del procesamiento del lenguaje natural y la recuperación de información. Dos tareas básicas que suelen surgir en este ámbito son la clasificación de las palabras más frecuentes y la recuperación de documentos que contienen palabras clave específicas. Para abordar estas cuestiones, este artículo analiza el desarrollo de algoritmos que utilizan el concepto de memorización y explora el concepto de indexación inversa.

II. Análisis codigo

A. Ocurrencia de palabras

```
def contar_ocurrencias(palabra, diccionario):
    if palabra not in diccionario:
        return 0
    ocurrencias = 0
    for documento in diccionario[palabra]:
        ocurrencias += documento.count(palabra)
    return ocurrencias
```

```
import collections

def contar_ocurrencias(palabra, diccionario):
    # Verifica si la palabra existe en el diccionario. Si no existe, devuelve 0.
    if palabra not in diccionario:
        return 0

    # Cuenta la ocurrencia de la palabra
    ocurrencias = 0
    for documento in diccionario[palabra]:
        ocurrencias += documento.count(palabra)

    return ocurrencias

def clasificar_palabras(diccionario):
    # Crea una lista de palabras ordenadas por número de ocurrencias, de mayor a menor.
    palabras_ordenadas = sorted(diccionario.keys(), key=lambda palabra: contar_ocurrencias(palabra, diccionario), reverse=True)

    # Imprime la clasificación de las palabras más repetidas
    print("Clasificación de palabras más repetidas:")
    for palabra in palabras_ordenadas:
        ocurrencias = contar_ocurrencias(palabra, diccionario)
        print(f"'{palabra}': {ocurrencias} veces")
```

Fig. 1. Análisis code 1.

B. Clasificación palabras

```
def clasificar_palabras(diccionario):
palabras_ordenadas = sorted(diccionario.keys(),
key=lambda palabra: contar_ocurrencias
(palabra, diccionario), reverse=True)
    print("Clasificación de palabras más repetidas:")
    for palabra in palabras_ordenadas:
        ocurrencias = contar_ocurrencias
        (palabra, diccionario)
        print(f"'{palabra}': {ocurrencias} veces")
```

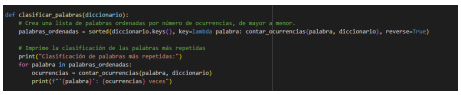

Fig. 2. Análisis code 2.

C. Ranking palabras más concurridas

```
for documento in my_documents:
        for palabra in documento.split():
            if palabra not in diccionario:
                diccionario[palabra] = []
            diccionario[palabra].append(documento)
    clasificar_palabras(diccionario)
```

Fig. 3. Análisis code 3.

D. Buscar palabra en específico

```
def buscar(diccionario, palabra_a_buscar):
    if palabra_a_buscar in diccionario:
        return diccionario[palabra_a_buscar]
    else:
        return []
```

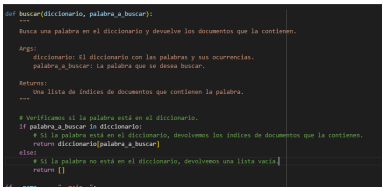

Fig. 4. Análisis code 4.

E. Cargar diccionario con la información a iterar

```
for i, documento in enumerate(my_documents):
        palabras = documento.split()
        for palabra in palabras:
            if palabra not in diccionario:
                diccionario[palabra] = [i]
            else:
                diccionario[palabra].append(i)
```

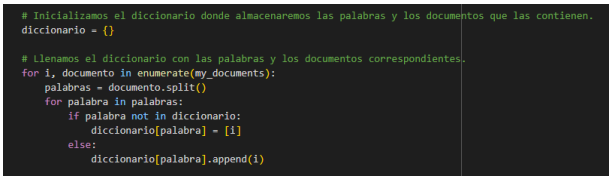

Fig. 5. Análisis code 5.

F. Palabra específica a buscar

```
palabra_a_buscar = "Python"
documentos= buscar(diccionario,palabra_a_buscar)
if documentos:
print(f"Documentos que contienen'
{palabra_a_buscar}':")
    for documento_index in documentos:
        print(my_documents[documento_index])
else:
    print(f"'{palabra_a_buscar}' no se encontró")
```

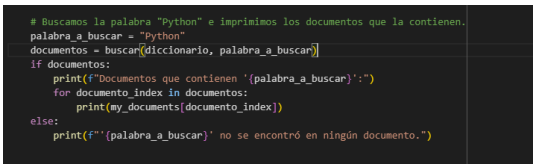

Fig. 6. Análisis code 6.

III. Resultados

A. Clasificación palabras

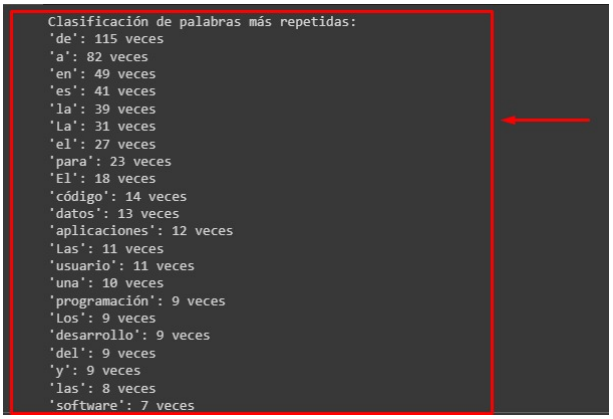

Fig. 7. Análisis clasificación.

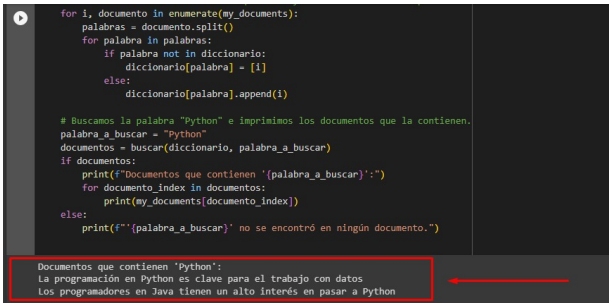

Fig. 8. Análisis busqueda palabra.

B. Búsqueda exacta

C. Descripción y complejidad Big-o

1) Define una función llamada `contar_ocurrencias` que toma dos argumentos: `palabra` y `diccionario`. Esta función se encarga de contar cuántas veces aparece una palabra específica en un conjunto de documentos representados por el diccionario.
2) Verifica si la palabra existe en el diccionario. Si no existe, devuelve 0, lo que significa que la palabra no se encuentra en ningún documento.
3) Si la palabra existe en el diccionario, se inicializa una variable `ocurrencias` en 0. Luego, se itera a través de los documentos que contienen esa palabra en el diccionario y se cuenta cuántas veces aparece la palabra en cada documento. Estas ocurrencias se suman a la variable `ocurrencias`.
4) Finalmente, la función devuelve el valor de `ocurrencias`, que representa cuántas veces aparece la palabra en total en los documentos.
5) Luego, se define otra función llamada `clasificar_palabras` que toma un argumento `diccionario`. Esta función se encarga de clasificar las palabras más repetidas en los documentos y mostrar sus ocurrencias.
6) En `clasificar_palabras`, se crea una lista llamada `palabras_ordenadas` que contiene las palabras del diccionario, ordenadas por el número de ocurrencias, de mayor a menor. Esto se logra utilizando la función `sorted` y la función `contar_ocurrencias` como clave de clasificación.
7) Luego, se imprime la clasificación de las palabras más repetidas en el formato "'palabra': ocurrencias veces" utilizando un bucle `for` para recorrer la lista de palabras ordenadas.
8) En la sección `if __name__ == "__main__":`, se inicializa un diccionario vacío llamado `diccionario`.
9) Se define una lista llamada `my_documents` que contiene una serie de oraciones o documentos de texto.
10) A continuación, se recorren los documentos en `my_documents` utilizando dos bucles `for`. El primer bucle itera a través de cada documento, y el segundo bucle divide cada documento en palabras individuales utilizando el método `split()` y verifica si cada palabra ya está en el diccionario. Si la palabra no está en el diccionario, se agrega como una clave con una lista vacía como valor. Luego, se agrega el documento actual a la lista de documentos asociados con esa palabra en el diccionario.
11) Una vez que se ha construido el diccionario con las palabras y sus documentos asociados, se llama a la función `clasificar_palabras` para clasificar y mostrar las palabras más repetidas en los documentos.

D. Code

E. Explanation

1) Definimos una función llamada `buscar(diccionario, palabra_a_buscar)` que busca una palabra en el diccionario y devuelve los índices de los documentos que contienen esa palabra. Esta función toma dos argumentos:
   a) `diccionario`: El diccionario con las palabras y sus ocurrencias.
   b) `palabra_a_buscar`: La palabra que se desea buscar en el diccionario.
2) Comenzamos la definición de la función `buscar` con una cadena de documentación (docstring) que describe lo que hace la función, los argumentos que recibe y lo que devuelve.
3) Verificamos si la `palabra_a_buscar` existe en el `diccionario` utilizando la declaración `if palabra_a_buscar in diccionario:`. Si la palabra existe, continuamos con el siguiente paso.
4) Si la palabra existe en el diccionario, inicializamos una lista llamada `documentos` con los valores que se encuentran en el diccionario bajo la clave `palabra_a_buscar`. Estos valores son los índices de los documentos que contienen la palabra.
5) Si no se encuentra la palabra en el diccionario, asignamos una lista vacía a `documentos` usando `return []`, indicando que la palabra no se encontró en ningún documento.
6) Definimos una función llamada `buscar(diccionario, palabra_a_buscar)` que busca una palabra en el diccionario y devuelve los índices de los documentos que contienen esa palabra. Esta función toma dos argumentos:
   a) `diccionario`: El diccionario con las palabras y sus ocurrencias.
   b) `palabra_a_buscar`: La palabra que se desea buscar en el diccionario.
7) Comenzamos la definición de la función `buscar` con una cadena de documentación (docstring) que describe lo que hace la función, los argumentos que recibe y lo que devuelve:
8) Verificamos si la `palabra_a_buscar` existe en el `diccionario` utilizando la declaración `if palabra_a_buscar in diccionario:`. Si la palabra existe, continuamos con el siguiente paso.
9) Si la palabra existe en el diccionario, inicializamos una lista llamada `documentos` con los valores que se encuentran en el diccionario bajo la clave `palabra_a_buscar`. Estos valores son los índices de los documentos que contienen la palabra.
10) Si no se encuentra la palabra en el diccionario, asignamos una lista vacía a `documentos` usando `return []`, indicando que la palabra no se encontró en ningún documento.

IV. Conclusiones

1) **Funcionalidad eficiente:** El código proporcionado exhibe una funcionalidad eficiente al contar las apariciones de palabras y clasificarlas según su frecuencia en un conjunto de documentos.
2) **Utilización de diccionarios:** Se emplean diccionarios para relacionar palabras con los documentos que las contienen, lo que garantiza un acceso rápido y eficiente a la información relevante.
3) **Presentación ordenada de resultados:** La disposición ordenada de las palabras que se repiten con mayor frecuencia, junto con sus respectivas apariciones, contribuye a una comprensión más clara de los datos. Esta presentación resulta valiosa en aplicaciones relacionadas con la búsqueda y el análisis de texto.
4) **Amplio potencial de aplicación:** El código tiene el potencial de integrarse en sistemas más amplios, como motores de búsqueda o aplicaciones de análisis de texto. Además, puede servir como un ejemplo didáctico de programación para el procesamiento de texto en el entorno de Python.