Proyecto Integrador

Materia: **Análisis de Algoritmos y Estructuras para Datos Masivos**

Alumno: **Luis Fernando Izquierdo Berdugo**

Fecha: **4 de Diciembre de 2024**

Introducción

En este proyecto se buscó construir y utilizar un índice invertido para analizar varios tuits relacionados con la pandemia de COVID-19. Gracias al índice, se puede hacer búsquedas eficientes y precisas sobre el contenido de los tuits.

En este caso, el índice es una estructura de datos que mapea cada palabra a una lista de documentos (o tuits) en los que aparece. Cada entrada en la lista de posteo indica la posición de la palabra en el documento que corresponde, por lo cual, se puede determinar de manera veloz y eficaz si una palabra está presente en un conjunto de tuits y en que posición.

Las consultas conjuntivas se realizaron mediante la intersección de las listas de posteo correspondientes a cada término de la consulta, la elección del algoritmo de intersección se hizo conforme a lo desarrollado en la unidad 6, para este proyecto se implementó la búsqueda binaria, ya que es bastante instantáneo el resultado y no involucra demasiado procesamiento.

El algoritmo de búsqueda binaria intenta encontrar la posición de un elemento específico en un arreglo ordenado. Este es muy eficiente para arreglos grandes y ordenados, en el peor de los casos tiene una complejidad de O(log n), siendo n el tamaño del arreglo.

Este algoritmo sigue el siguiente proceso:

- Compara el elemento a buscar con el elemento de enmedio del arreglo, si son iguales lo devuelve.
- Si el elemento a buscar es menor a la mitad, se repite la búsqueda en la mitad izquierda del arreglo.
- Si el elemento buscado es mayor a la mitad, se repite la búsqueda en la mitad derecha del arreglo.
- Se repite el proceso en bucle hasta encontrar el elemento o ya no haya dónde buscar.

El algoritmo de intersección binaria busca los elementos comunes entre dos arreglos ordenados. Este es muy eficiente cuando ambos arreglos están ordenados porque no compara elementos innecesarios, sin embargo, su complejidad depende del tamaño de los arreglos y los elementos que tienen en común.

Este algoritmo sigue el siguiente proceso:

- Se usa un puntero para cada arreglo
- Se comparan los elementos de los punteros, si son iguales se agrega el elemento a la lista de resultados y ambos punteros avanzan.
- Si el elemento del primer arreglo es menor, el puntero del primer arreglo avanza
- Si el elemento del segundo arreglo es menor, el puntero del segundo arreglo avanza.
- Se repite el proceso hasta que se hayan recorrido por completo ambos arreglos.

Planteamiento del Problema

Para construir el índice invertido es necesario seguir los siguientes pasos fundamentales:

1. Preprocesamiento del texto
2. Asignación de identificadores
3. Creación de listas de posteo

Preprocesamiento del texto

Para que los resultados sean de mayor calidad y más fiables, es necesario hacer una serie de preprocesamientos al texto, con la finalidad de normalizar este y reducir el ruido.

Dentro de este paso se pasará todo el texto a minúsculas, se eliminarán las palabras comunes (stop words) en español, los signos de puntuación y acentos, así como la separación por palabras individuales (o tokenización) usando el espacio como separador.

Asignación de identificadores

Por medio de la función `hash` de Python se asignará un identificador único (en este proyecto se utilizaron números)

Creación de listas de posteo

Para cada una de las palabras, se generará una lista ordendad de identificadores (los creados en el paso anterior), en la cual se indica las posiciones en las que aparece el término en cada documento.

Posterior a la construcción del índice invertido, se implementará el **algoritmo de búsqueda binaria** para las consultas conjuntivas.

Código

Lo primero que está presente en el código es la importación de las librerías necesarias para el funcionamiento correcto. Estas son `json` para importar el texto, `re` y `nltk` para el preprocesamiento del texto, así como `collections` para la creación del índice invertido.

In [57]:
```
# Importar las bibliotecas necesarias
import json
import re
from collections import defaultdict
from nltk.corpus import stopwords
import nltk
# Descargar las stopwords en español
#nltk.download('stopwords')
```

Lectura y preprocesamiento del texto

En este parte del código se crea una función `leer_corpus` para abrir el archivo `pcovid2020.json` y leer línea por línea los json dentro de este. De igual manera, se creo la función `preprocesar_texto` que hace los siguientes preprocesamientos:

- Conversión a minúsculas
- Eliminación de signos de puntuación y acentos
- Tokenización del texto (división por palabra)
- Eliminación de palabras comunes en español (stopwords)

Ya con las funciones creadas, se lee el archivo y se preprocesa el texto.

In [58]:
```
stop_words = set(stopwords.words('spanish'))

# Definir la función para leer y procesar el archivo JSON
def leer_corpus(archivo):
    with open(archivo, 'r', encoding='utf-8') as f:
        corpus = [json.loads(line)['text'] for line in f]
    return corpus

# Procesar el texto de los tweets
def procesar_texto(texto):
    # Convertir a minúsculas
    texto = texto.lower()
    # Remover puntuaciones y acentos
    texto = re.sub(r'[^\w\s]', '', texto)
    # Tokenizar el texto
    palabras = texto.split()
    # Eliminar stopwords
    palabras = [palabra for palabra in palabras if palabra not in stop_words]
    return palabras

# Leer el archivo y procesar cada línea
archivo = 'pcovid2020.json'
corpus = leer_corpus(archivo)
corpus_procesado = [procesar_texto(texto) for texto in corpus]
```

Construcción de Índice Invertido

Lo primero es inicializar el índice invertido como un diccionario de listas.

Se itera sobre cada documento (tuit) del corpus procesado, asignándole un identificador único (`doc_id`)a cada uno de ellos. Dentro de cada iteración, se buelbe a iterar por cada palabra, usando la función hash para asignar un identificador a cada palabra. Finalmente, si el identificador único del documento actual no está presente en la lista de la palabra que se hasheó dentro del índice invertido, se añade.

Posterior a la creación del índice invertido, se ordenan las listas de posteo para facilitar su lectura posterior.

In [59]:
```
# Crear el índice invertido
indice_invertido = defaultdict(list)

# Asignar un número a cada palabra usando la función hash
for doc_id, texto in enumerate(corpus_procesado):
    for palabra in texto:
        palabra_hash = hash(palabra)
        if doc_id not in indice_invertido[palabra_hash]:
            indice_invertido[palabra_hash].append(doc_id)

# Ordenar las listas de posteo
for palabra_hash in indice_invertido:
    indice_invertido[palabra_hash].sort()
```

Algoritmos de busqueda e intersección

La función `busqueda_binaria` implementa el algoritmo de mismo nombre. Este toma una lista y un elemento como parámetros. Lo primero es inicializar los punteros (`izquierda` y `derecha`) que representan los límites de la porción de la lista que se está buscando. En cada paso del bucle, se calcula el índice `medio` y compara el elemento en esa posición con el que se está buscando, regresando este mismo si son iguales. Si el elemento en el medio es menor que el buscado, el puntero izquierdo se ajusta para buscar en la mitad derecha, en el caso contrario usa el puntero derecho para buscar en la mitad izquierda. Si no encuentra el elemento, regresa -1.

La función de `intersecció_binaria` usa la búsqueda binaria creada preaviamente para encontrar la intersección de dos listas ordenadas. En esta se declara una lista `resultado` para almacenar los elementos comunes de ambas listas (que son los parámetros `lista1` y `lista2`). Posteriormente se itera sobre cada elemento de la primera lista y utiliza la búsquerda binaria para verificar si el elemento está presente en la segunda lista, en el caso positivo lo añade a la lista de resultados y al final devuelve esta misma.

In [60]:
```
# Definir Funciones para Consultas Conjuntivas

# Función para realizar búsqueda binaria
def busqueda_binaria(lista, elemento):
    izquierda, derecha = 0, len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == elemento:
            return medio
        elif lista[medio] < elemento:
            izquierda = medio + 1
```

```
        else:
            derecha = medio - 1
    return -1

# Función para intersección de listas usando búsqueda binaria
def interseccion_binaria(lista1, lista2):
    resultado = []
    for elemento in lista1:
        if busqueda_binaria(lista2, elemento) != -1:
            resultado.append(elemento)
    return resultado
```

Consultas Conjuntivas

La función `resolver_consulta` toma una consulta de texto y encuentra los documentos que contienen todas las palabras de la consulta.

Lo primero que hace esta función es procesar el texto de la consulta por medio de la función `procesar_texto` para obtener una lista de palabras. Posteriormente, se obtienen las listas de posteo correspondientes a cada palabra en el índice invertido, si estas no existen, se regresa una lista vacía.

Lo siguiente es ordenar las listas de posteo por longitud para optimizar la intersección, así como la inicialización del resultado con la lista de posteo más corta. Inmediatamente después se procede a iterar sobre las demás listas, añadiendo al resultado la intersección binaria de la lista actual y el resultado acumulado previamente. Este resultado es el que se regresa con todos los identificadores de documentos que contienen todas las palabras de la consulta.

In [61]:
```
# Función para resolver consultas conjuntivas
def resolver_consulta(consulta):
    palabras = procesar_texto(consulta)
    listas_posteo = [indice_invertido[hash(palabra)] for palabra in palabras if

    if not listas_posteo:
        return []

    listas_posteo.sort(key=len)
    resultado = listas_posteo[0]

    for lista in listas_posteo[1:]:
        resultado = interseccion_binaria(resultado, lista)

    return resultado
```

Experimentos

Para ejecutar el experimento, se debe incluir la palabra o frase en la lista `consultas` como se observa en el código siguiente:

In [62]:
```
# Consultas conjuntivas que se desean resolver
consultas = [
    "vacuna",
```

```
    "prevención",
    "sintomas",
    "distanciamiento",
    "rascador",
    "prevencion"
]

# Resolver cada consulta y mostrar los resultados
for consulta in consultas:
    resultados = resolver_consulta(consulta)
    print(f"Consulta: '{consulta}' -> Resultados: {resultados}\n")
```

```
Consulta: 'vacuna' -> Resultados: [38, 55, 62, 113, 152, 232, 331, 338, 388, 39
9, 438, 519, 583, 650, 686, 758, 825, 828, 849, 924, 968, 1009, 1035, 1037, 103
9, 1041, 1073, 1095, 1097, 1184, 1205, 1292, 1301, 1302, 1366, 1401, 1471, 1487,
1498, 1499, 1510, 1603, 1606, 1675, 1684, 1686, 1790, 1846, 1872, 1886, 1888, 19
18, 1964, 1965, 1981, 2020, 2037, 2138, 2156, 2194, 2216, 2267, 2293, 2428, 252
0, 2535, 2615, 2705, 2730, 2733, 2740, 2753, 2798, 2811, 2926, 2956, 2965, 3030,
3061, 3086, 3283, 3305, 3319, 3411, 3437, 3449, 3489, 3509, 3530, 3541, 3619, 36
75, 3695, 3771, 3781, 3801, 3822, 3847, 4035, 4047, 4065, 4131, 4179, 4182, 420
7, 4269, 4306, 4321, 4353, 4385, 4432, 4459, 4514, 4547, 4574, 4629, 4650, 4699,
4707, 4711, 4793, 4863, 4896, 4994, 5004, 5086, 5137, 5138, 5159, 5254, 5306, 53
35, 5424, 5440, 5484, 5511, 5548, 5553, 5564, 5733, 5843, 5899, 5901, 5935, 595
9, 5971, 6104, 6201, 6211, 6236, 6240, 6294, 6324, 6343, 6356, 6376, 6382, 6491,
6567, 6617, 6632, 6635, 6636, 6699, 6735, 6771, 6851, 6879, 6903, 7042, 7174, 71
92, 7210, 7217, 7231, 7345, 7424, 7452, 7465, 7469, 7501, 7556, 7616, 7634, 766
6, 7686, 7687, 7741, 7766, 7825, 7848, 7863, 7901, 7908, 7947, 7973, 7994, 8093,
8096, 8097, 8099, 8108, 8120, 8139, 8199, 8231, 8236, 8295, 8307, 8340, 8342, 83
80, 8467, 8541, 8693, 8708, 8715, 8717, 8723, 8803, 8823, 8828, 8858, 8963, 909
9, 9127, 9147, 9153, 9167, 9175, 9237, 9285, 9339, 9355, 9356, 9426, 9491, 9535,
9540, 9652, 9666, 9754, 9822, 9875, 9907, 9949, 9997]

Consulta: 'prevención' -> Resultados: [141, 218, 253, 298, 313, 465, 522, 614, 8
23, 1020, 1142, 1175, 1322, 1361, 1363, 1495, 1534, 1539, 1566, 1620, 1648, 173
5, 1811, 1924, 2061, 2065, 2394, 2485, 2500, 2628, 2701, 2763, 3569, 3603, 3714,
3793, 3854, 3877, 3990, 4002, 4085, 4095, 4144, 4158, 4361, 4436, 4444, 4576, 46
70, 4900, 5046, 5064, 5081, 5084, 5196, 5609, 5919, 6192, 6493, 6640, 6977, 700
9, 7190, 7255, 7313, 7421, 7427, 7439, 7482, 7567, 7800, 8054, 8265, 8401, 8465,
8511, 8618, 8668, 8835, 8844, 8880, 9033, 9123, 9223, 9283, 9301, 9390, 9391, 95
85, 9713, 9723, 9808, 9911]

Consulta: 'sintomas' -> Resultados: [1319, 4054, 4143, 4352, 4369, 5766, 6549, 6
908]

Consulta: 'distanciamiento' -> Resultados: [97, 167, 674, 1216, 1550, 1574, 160
5, 1751, 2501, 3021, 3086, 3404, 3607, 3912, 4118, 4244, 4599, 4816, 5566, 5782,
5937, 6108, 6360, 6371, 6789, 7087, 7427, 7668, 8168, 8336, 8560, 9235, 9538]

Consulta: 'rascador' -> Resultados: []

Consulta: 'prevencion' -> Resultados: [2767, 3990, 5323, 6125]
```

Como se puede observar en los resultados, se obtienen listas con los tuits en los que aparecen las palabras de consulta (y una lista vacía para aquellos que no se encuentran).

Se creó una función auxiliar para mostrar fragmentos de los tuits donde aparece la consulta (solamente utiliza el resultado de la última consulta).

In [63]:
```python
def mostrar_fragmentos(resultados, corpus, consulta):
    palabras_consulta = procesar_texto(consulta)
    fragmentos = []

    for doc_id in resultados:
        texto = corpus[doc_id]
        texto_procesado = procesar_texto(texto)
        indices = [i for i, palabra in enumerate(texto_procesado) if palabra in

        if indices:
            inicio = max(0, indices[0] - 5)
            fin = min(len(texto_procesado), indices[-1] + 6)
            fragmento = ' '.join(texto_procesado[inicio:fin])
            fragmentos.append((doc_id, fragmento))

    return fragmentos

# Ejemplo de uso de la función mostrar_fragmentos
fragmentos = mostrar_fragmentos(resultados, corpus, consulta)
for doc_id, fragmento in fragmentos:
    print(f"Documento ID: {doc_id} -> Fragmento: {fragmento}")
```

```
Documento ID: 2767 -> Fragmento: web informa bastante confiablemente medidas pre
vencion covid19 nuevo coronavirus protegerse mejor
Documento ID: 3990 -> Fragmento: transporte vertical httpstcomtsdhwbtf6 segurida
dlaboral ascensores prevencion
Documento ID: 5323 -> Fragmento: ahora deberian incorporar infra medidas prevenc
ion covid19 estandar optimo pueden hacerlo
Documento ID: 6125 -> Fragmento: prevencion coronavirus consejo 1 alejate amigo
```

Se puede observar que el resultado es correcto, ya que la palabra "prevencion" si se encuentra en los 4 textos.

Conclusiones

Durante la evaluación de pruebas, se destacó es la velocidad de procesamiento, tardando una cantidad de tiempo menor a un segundo para hacer varias consultas. De igual manera, se observa que los resultados son fiables por medio de la función donde se muestran fragmentos de texto que incluyen la consulta efectuada.

Se nota un funcionamiento erróneo al utilizar o no acentos. Para las pruebas, se utilizó la palabra `“prevención”`, así como `“prevencion”`; en ambos casos se esperaba que se obtuviera el mismo resultado, sin embargo, la consulta con acento arrojó muchos más resultados que aquella sin acento. Esto se puede deber a un mal preprocesamiento del texto, así como un error de progamación durante la consulta, intersección o búsqueda de las palabras.

Para la elección de algoritmos de intersección y búsqueda se utilizaron diferentes opciones, siendo la más óptima la búsqueda binaria, mientras que para intersección fue algo muy parejo entre **“galloping” y “binaria”**, ganando esta última finalmente, ya que mostraba resultados correctos que galloping omitía, proporcionando más confianza.

Definitivamente se encuentra muy interesante la utilización de un índice invertido y la función `hash` de Python, ya que, se puede notar una eficiencia mayor en las pruebas de este experimento con los distintos realizados a lo largo del curso, lo cual se debe principalmente a la combinación de ambas técnicas.

Para continuar con este experimento se podría implementar y comparar otros algoritmos de intersección y búsqueda de manera más extensa, así como la implementación de ponderaciones a las palabras en función de importancia para mejorar los resultados. De igual manera se puede mejorar muchísimo el preprocesamiento para evitar los enlaces y otras palabras o caracteres que causan ruido en el experimento.

Referencias Bibliográficas

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms, third edition. http://portal.acm.org/citation.cfm?id=1614191

Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro, “Adaptive Set Intersections, Unions, and Differences”, in Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000), San Francisco, California, January 9–11, 2000, pages 743–752.

Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro, “Experiments on Adaptive Set Intersections for Text Retrieval Systems”, in Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments (ALENEX 2001), Lecture Notes in Computer Science, volume 2153, Washington, DC, January 5–6, 2001, pages 91–104.