Informe tarea 01 - Recuperaciòn de Informaciòn

Juan Esteban Rodríguez[1,†], **Camilo Barreto**[2,†], **Luis Alejandro Medina**[3,†] **and Julian Camilo Garcia**[4,†]

[†]*Estos autores contribuyeron igualmente a este trabajo*

Abstract

Este proyecto implementa un motor de búsqueda que es capaz de comparar cuatro estrategias diferentes en la recuperación de información, de las cuales las dos más básicas son Búsqueda Binaria Usando Índice Invertido (BSII) y Recuperación Ranqueada y Vectorización de Documentos (RRDV). Se utilizó Python con las bibliotecas numpy y pandas para preprocesar 331 documentos y 35 consultas del conjunto de datos, lo cual conlleva una serie de tareas, incluyendo operaciones de tokenización, eliminación de palabras vacías, normalización y stemming. La eficacia de cada enfoque se evalúa frente a un conjunto de mediciones de relevancia, incluidos valores de NDCG, P@M y R@M que estiman la precisión de recuperar los documentos relevantes.

1. Métricas de Evaluación de IR

En este punto, se implementaron métricas clave de evaluación de sistemas de recuperación de información utilizando Python y Numpy, centradas en la precisión y efectividad de las consultas en un contexto binario y numérico. Más adelante veremos como nos serán de utilidad para verificar que tan bien estan funcionando nuestros motores de busqueda.

1.1. Precision y Precision at K

Implementadas para evaluar la fracción de documentos relevantes recuperados y la precisión hasta un corte K respectivamente.

Ejemplo de Precision: precision([0, 0, 0, 1]) retorna 0.25. Ejemplo de Precision at K: precision at k([0, 0, 0, 1], 1) retorna 0.0.

1.2. Recall at K

Mide los documentos relevantes recuperados hasta la posición K.

Ejemplo: recall at k([0, 0, 0, 1], 4, 1) retorna 0.0.

1.3. Average Precision

Average Precision: Calcula la precisión promedio donde cada documento relevante es recuperado.

Ejemplo: Para la consulta [0, 1, 0, 1, 1, 1, 1], la precisión promedio es 0.5961904761904762.

1.4. DCG y NDCG at K

Evalúan la ganancia acumulativa con descuentos y su versión normalizada para comparaciones más equitativas entre diferentes sets de resultados.

Ejemplo de DCG at K: dcg at k([4, 4, 3, 0, 0, 1, 3, 3, 3, 0], 6) retorna 10.27964. Ejemplo de NDCG at K: ndcg at k([4, 4, 3, 0, 0, 1, 3, 3, 3, 0], 6) retorna 0.7424.

2. Búsqueda Binaria usando Índice Invertido (BSII)

El objetivo de este punto es desarrollar un sistema de búsqueda binaria utilizando un índice invertido. Este enfoque permite realizar consultas eficientes en grandes volúmenes de datos textuales, aplicando técnicas de procesamiento de lenguaje natural para mejorar la recuperación de información.

Para desarrollar este punto se siguieron los siguientes pasos en la codificación:

2.1. Preprocesamiento de Textos

Utilizando la biblioteca nltk, se realizaron pasos de tokenización, eliminación de palabras vacías, normalización y lematización. Estos procesos son esenciales para reducir la complejidad del texto y aumentar la relevancia de las búsquedas.

La clase procesamientotexto se creó para encapsular las funciones de preprocesamiento. Esto facilita la gestión del código y mejora la modularidad.

2.2. Construcción del Índice Invertido

Se implementó una función indiceinvertido que crea un diccionario en Python donde cada término es una clave y los valores son listas de documentos organizadas de mayor a menor para poder implementar de mejor forma el algoritmo de mezcla. Estas listas contienen que contienen el término, junto con la frecuencia del término.

```
def indiceinvertido(doc_lemalist: dict,terminos: dict):
    """
    Input: doc_lemalist, terminos.
        doc_lemalist:  objeto que contiene los indices de cada documento junto con su texto normalizado.
        terminos: vocabulario que contiene todos los textos.
    output: indiceinvertido
        indiceinvertido: objeto 'diccionario' de la forma
        {'termino':
            'Idocs':[...] # lista de documentos que contiene el termino
            'len': x     # la cantidad de documentos que contienen el termino
        }
    """
    indiceinvertido = {}
    for termino in terminos:
         indiceinvertido[termino] = {'IDdocs':[],'len':0}

    for documento in doc_lemalist.values():   ## no me gusta el doble for
        set_texto = set(documento['text'])
        for termino in set_texto:
            indiceinvertido[termino]['IDdocs'].append(documento['index'])
            indiceinvertido[termino]['len'] +=1
    return indiceinvertido
```

Code 1. Implementación Indice Invertido.

2.3. Algoritmos de Consulta

Se desarrollaron funciones para realizar operaciones de búsqueda AND y NOT utilizando el índice invertido. Estas funciones utilizan el algoritmo de mezcla para determinar la intersección o diferencia de listas de documentos, optimizando la búsqueda.

La función And toma dos términos y encuentra documentos que contienen ambos términos. Por otro lado, la función Not busca docu-

mentos que no contienen un término específico.

La función and implementa el algoritmo de mezcla sobre el índice invertido, es eficiente y hace menos uso de memoria si se compara con otro algoritmo que se apoya de la función 'set' y trata la búsqueda como un problema de conjuntos.

3. Recuperación Ranqueada y Vectorización de Documentos (RRDV)

En este punto implementamos un sistema de recuperación de información basado en la vectorización de documentos y consultas utilizando TF-IDF y medimos la similitud mediante el coseno. Este enfoque busca mejorar la precisión de las búsquedas y ofrecer una evaluación sistemática de los resultados obtenidos.

Para hacer lo anteriormente mencionado seguimos partimos de el desarrollo de Procesamiento de Textos e Indice invertido desarrollado en el punto anterior.

A partir de aquí realizamos el siguiente desarrollo:

3.1. Cálculo de TF-IDF

Se implementó una función que calcula la representación vectorial ponderada TF-IDF de los documentos, permitiendo una comparación numérica precisa entre textos basada en su contenido informativo.

```
def calcular_tf_idf(nltk_lemmaList, dicterminos,
     list_indiceinvertido, N):
    tf_idf = {}

    for doc_name, doc_data in nltk_lemmaList.
    items():
        tf_idf[doc_name] = {}

        for term in dicterminos:
            tf = doc_data['term_count'].get(term
    , 0)
            df = list_indiceinvertido[term]['len
    ']

            if tf > 0 and df > 0:
                # C lculo de TF y DF seg n las
     f rmulas proporcionadas
                tf_value = np.log10(1 + tf)
                df_value = np.log10(N / df)

                # C lculo de TF-IDF
                tf_idf[doc_name][term] =
    tf_value * df_value
            else:
                tf_idf[doc_name][term] = 0.0

    return tf_idf
```

Code 2. Implementación representación vectorial ponderada tf idf.

Además de esto, se desarrolló una función para calcular la similitud del coseno entre dos vectores de documentos, lo cual es esencial para determinar qué tan relevantes son los documentos en relación con una consulta dada. Esta función siguió la siguiente fórmula donde A y B son los Vectores a analizar.

$$\text{similitud coseno}(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{||\mathbf{A}||||\mathbf{B}||} \quad (1)$$

3.2. Evaluación de Resultados

Se realizaron cálculos de P@M, R@M, y NDCG@M para evaluar la efectividad del sistema de recuperación. Estas métricas proporcionan una comprensión detallada de la precisión, el recall y la ganancia acumulativa descontada normalizada respectivamente.

Con base en la implementación que hicimos y los resultados que tuvimos podemos concluir de esta implementación que la estrategia TF-IDF con similitud coseno para comparar corpus de textos con queries específicas es un enfoque clásico y sencillo en la recuperación de información. Es eficiente y fácil de implementar, especialmente en conjuntos de datos pequeños.Sin embargo, presenta limitaciones importantes, como la falta de consideración por aspectos subjetivos del lenguaje, como el sarcasmo y el contexto que da sentido al texto. Estas limitaciones se reflejan en las métricas de evaluación al compararse con el ground-truth. Además, este enfoque puede ser menos eficaz cuando se trata de conjuntos de datos grandes o complejos, lo que representa una gran desventaja. En comparación con modelos más avanzados, estos últimos suelen ofrecer mejores resultados en contextos donde la precisión y la comprensión semántica son cruciales.

4. Recuperación Ranqueada y Vectorización de Documentos Utilizando Gensim (GESIM)

Este punto tiene como objetivo replicar las funcionalidades del punto anterior utilizando la librería Gensim, una herramienta especializada en el procesamiento de textos y modelado de tópicos. Se espera que el uso de Gensim mejore la eficiencia y escalabilidad del sistema de recuperación de información.

Para el desarrollo de este punto seguimos los siguientes pasos en nuestra codificación.

4.1. Preprocesamiento de Textos con Gensim

Utilizamos la función preprocess string de Gensim para aplicar una serie de filtros de preprocesamiento de texto, como la eliminación de puntuación, números, y palabras cortas, además de stopwords.

4.2. Construcción de Corpus y Vocabulario

La clase MyCorpus se creó para iterar sobre el conjunto de documentos, aplicando el preprocesamiento "on the fly", lo cual es eficiente en términos de uso de memoria.

También vale resaltar que se construyó un corpus tokenizado y se generó un vocabulario utilizando las funcionalidades de Gensim, lo que facilita la transformación de textos a representaciones numéricas como bolsa de palabras y TF-IDF.

4.3. Modelado TF-IDF con Gensim

Se implementó el modelo TF-IDF sobre el corpus preprocesado para obtener representaciones vectoriales de los documentos, donde cada tupla representa la frecuencia y la importancia de cada término en el documento.

4.4. Similitud del Coseno

Se utilizó el índice de similitudes de Gensim para calcular eficientemente la similitud del coseno entre vectores de documentos, facilitando la comparación y clasificación de documentos respecto a las consultas.

4.5. Evaluación de Resultados

Se replicaron las métricas de evaluación del punto anterior (P@M, R@M, y NDCG@M) utilizando las representaciones vectoriales generadas por Gensim, y se compararon con los resultados obtenidos con la implementación manual.

4.6. Ventajas de usar GENSIM

GENSIM como alternativa

Gracias a las dos implementaciones hechas, podemos rescatar las siguientes ventajas de hacer uso de la librería de Gensim:

Gensim ofrece una representación más eficiente en memoria debido a su manejo de corpus como iteradores y su almacenamiento de vectores en formato de tuplas, solo para tokens presentes en el documento.

Gensim proporciona herramientas integradas para el preprocesamiento y modelado de documentos, lo que simplifica el código y reduce la cantidad de manejo manual de datos.

Mientras que la implementación manual en el punto anterior ofrece control total sobre los procesos, Gensim ofrece una variedad de funciones preconstruidas que pueden ser más robustas y optimizadas para tareas específicas de procesamiento de lenguaje natural.