Procesamiento de Lenguaje Natural
Tarea 1

Rayo Mosquera, Jhon Stewar
j.rayom@uniandes.edu.co

De La Rosa Peredo, Carlos Raul
c.delarosap@uniandes.edu.co

Mario Garrido Córdoba
m.garrido10@uniandes.edu.co

Septiembre 2024

Punto 1: Implementación de Métricas de Evaluación de IR

En esta sección implementamos varias métricas de evaluación de recuperación de información (IR) utilizando Python y `numpy`. Estas métricas son esenciales para evaluar la eficacia de los modelos de recuperación de información, ya que permiten medir la precisión y la relevancia de los documentos recuperados en respuesta a una consulta.

1. Precision

La función `precision` calcula la precisión para un vector de relevancia binaria. La precisión se define como la fracción de documentos relevantes recuperados sobre el total de documentos recuperados.

- **Parámetros**: Una lista `relevance_query` que contiene valores de relevancia binaria (0 o 1).
- **Salida**: Un valor de tipo `float` que representa la precisión del conjunto de resultados.

```
def precision(relevance_query):
    relevance_array = np.array(relevance_query)
    precision = np.sum(relevance_array == 1) / len(relevance_array)
    return precision
```

2. Precision at K

La función `precision_at_k` calcula la precisión en los primeros `k` resultados. Esto es útil cuando solo los primeros resultados son de interés, como en motores de búsqueda donde solo se muestran las primeras entradas al usuario.

- **Parámetros**: Una lista `relevance_query` y un entero `k`.
- **Salida**: Un valor `float` que indica la precisión en los primeros `k` resultados.

```
def precision_at_k(relevance_query, k):
    relevance_array = np.array(relevance_query)
    precision_at_k = np.sum(relevance_array[:k] == 1) / k
    return precision_at_k
```

3. Recall at K

La función `recall_at_k` calcula el recall en los primeros `k` resultados. El recall mide la fracción de documentos relevantes que han sido recuperados en comparación con todos los documentos relevantes disponibles.

- **Parámetros**: Una lista `relevance_query`, un entero `number_relevant_docs` (número de documentos relevantes), y un entero `k`.
- **Salida**: Un valor `float` que representa el recall en los primeros `k` resultados.

```
def recall_at_k(relevance_query, number_relevant_docs, k):
    relevance_array = np.array(relevance_query)
    recall_at_k = np.sum(relevance_array[:k] == 1) / number_relevant_docs
    return recall_at_k
```

4. Average Precision

La función `average_precision` calcula la precisión promedio para un conjunto de resultados. Esta métrica es útil para evaluar la calidad de los resultados ordenados por relevancia.

- **Parámetros**: Una lista `relevance_query`.
- **Salida**: Un valor `float` que indica la precisión promedio.

```
def average_precision(relevance_query):
    relevance_array = np.array(relevance_query)
    relevant_ranks = np.where(relevance_array == 1)[0] + 1
    precisions_at_k = [precision_at_k(relevance_query, k) for k in relevant_ranks]
    average_precision = np.sum(precisions_at_k) / np.sum(relevance_array)
    return average_precision
```

5. Mean Average Precision (MAP)

La función `mean_average_precision` calcula la precisión promedio para un conjunto de consultas. Esta métrica se utiliza comúnmente para comparar la eficacia de diferentes modelos de recuperación de información.

- **Parámetros**: Una lista de listas `relevance_queries`, donde cada sublista representa los resultados de una consulta.
- **Salida**: Un valor `float` que representa el promedio de la precisión promedio (MAP).

```
def mean_average_precision(relevance_queries):
    mean_average_precision = np.mean([average_precision(q) for q in relevance_queries])
    return mean_average_precision
```

6. DCG at K (Discounted Cumulative Gain)

La función `dcg_at_k` calcula la ganancia acumulada descontada en `k`, que mide la relevancia de un documento en una posición específica de la lista, ponderada por la posición del documento.

- **Parámetros**: Una lista `relevance_query` con relevancia natural y un entero `k`.

- **Salida**: Un valor `float` que indica el DCG en `k`.

```
def dcg_at_k(relevance_query, k):
    REL_i = np.array(relevance_query)
    discount_factor = np.array([1 / np.log2(np.max([i, 2])) for i in range(1, len(REL_i) + 1)])
    gain = np.multiply(REL_i, discount_factor)
    dcg_at_k = np.sum(gain[:k])
    return dcg_at_k
```

7. NDCG at K (Normalized Discounted Cumulative Gain)

La función `ndcg_at_k` normaliza el DCG dividiéndolo por el DCG ideal, proporcionando una medida relativa de la calidad de la clasificación del sistema.

- **Parámetros**: Una lista `relevance_query` y un entero `k`.
- **Salida**: Un valor `float` que indica el NDCG en `k`.

```
def ndcg_at_k(relevance_query, k):
    best_ranking = np.sort(relevance_query)[::-1]
    ndcg_at_k = dcg_at_k(relevance_query, k) / dcg_at_k(best_ranking, k)
    return ndcg_at_k
```

Conclusión

Estas funciones proporcionan un conjunto robusto de métricas para evaluar sistemas de recuperación de información. Cada métrica tiene su propósito específico y permite una evaluación detallada y diversa de los modelos IR en diferentes escenarios y condiciones de recuperación. Estas implementaciones son eficientes y escalables, aprovechando las capacidades de `numpy` para realizar cálculos vectoriales de manera rápida y efectiva.

Punto 2: Implementación del Motor de Búsqueda: Búsqueda Binaria usando Índice Invertido (BSII)

En esta sección, implementamos un motor de búsqueda utilizando la estrategia de búsqueda binaria con un índice invertido (BSII). Esta técnica permite recuperar documentos basándose en consultas booleanas aplicadas sobre un índice invertido que almacena la frecuencia de ocurrencia de términos en un conjunto de documentos.

1. Preprocesamiento de Documentos y Consultas

Para construir el índice invertido, primero preprocesamos los documentos y las consultas para preparar los datos. Este proceso incluye:

- **Carga y Extracción de Texto**: Utilizamos la función `load_files` para cargar archivos en formato NAF (XML) desde un directorio especificado y extraer el texto contenido en los elementos `<raw>`. La función `extract_text_from_xml` maneja esta tarea, asegurando una gestión eficiente de memoria mediante el uso de `iterparse`.
- **Limpieza y Tokenización**: La función `process_text` convierte el texto a minúsculas, elimina tokens no alfabéticos y palabras vacías, y aplica un algoritmo de *stemming* usando la librería `nltk`. Esto reduce la dimensionalidad del texto y estandariza los términos para una indexación más eficiente.

2. Construcción del Índice Invertido

Una vez preprocesados los documentos, construimos el índice invertido utilizando la función `build_inverted_index`. Este índice es una estructura de datos que mapea cada término de la colección de documentos a un diccionario que contiene los documentos en los que aparece dicho término y la frecuencia con la que lo hace.

- **Parámetros**: Un diccionario `processed_documents` con IDs de documentos y tokens procesados.
- **Salida**: Un `defaultdict` anidado que mapea cada token a un diccionario de frecuencias por documento.

```
def build_inverted_index(processed_documents):
    inverted_index = defaultdict(lambda: defaultdict(int))

    for doc_id, tokens in processed_documents.items():
        for token in tokens:
            inverted_index[token][doc_id] += 1

    return inverted_index
```

3. Conversión de Consultas en Consultas Booleanas

Las consultas preprocesadas se convierten en consultas booleanas utilizando la función `construct_boolean_queries`. Esta función crea una cadena de términos conectados con operadores booleanos (`AND`, `NOT`) para cada consulta.

- **Parámetros**: Un diccionario `processed_queries` de consultas y tokens procesados.
- **Salida**: Un diccionario `boolean_queries` de IDs de consultas y sus respectivas expresiones booleanas.

4. Evaluación de Consultas Booleanas

La función `evaluate_boolean_queries` evalúa un conjunto de consultas booleanas aplicándolas al índice invertido. Esta función implementa el algoritmo de mezcla para combinar los resultados de las consultas basadas en los operadores booleanos (`AND`, `NOT`).

- **Parámetros**: Un índice invertido `inverted_index` y un conjunto de consultas booleanas `boolean_queries`.
- **Salida**: Un diccionario `evaluated_queries` de IDs de consultas y listas de IDs de documentos que satisfacen cada consulta.

```
def evaluate_boolean_queries(inverted_index, boolean_queries):
    evaluated_queries = {}

    for key, values in boolean_queries.items():
        tokens = values.lower().split()
        doc_sets = {}

        for token in tokens:
            if token not in {'and', 'not'} and token in inverted_index:
                doc_sets[token] = inverted_index[token].keys()

        base_token = tokens[0]
```

```
        result_docs = doc_sets.get(base_token, set()) if base_token in doc_sets else set()

        i = 1
        while i < len(tokens):
            operator = tokens[i]
            term = tokens[i + 1]

            if operator == 'and':
                if term in doc_sets:
                    result_docs &= doc_sets.get(term, set())
            elif operator == 'not':
                if term in doc_sets:
                    result_docs -= doc_sets.get(term, set())

            i += 2

        evaluated_queries[key] = sorted(list(result_docs))

    return evaluated_queries
```

5. Guardado de Resultados

Finalmente, los resultados de la evaluación de las consultas booleanas se guardan en un archivo de texto delimitado por tabulaciones (`TSV`) utilizando un objeto escritor de CSV.

```
with open(file_name, 'w', newline='', encoding='utf-8') as file:
    writer = csv.writer(file, delimiter='\t')
    for key, values in evaluated_queries.items():
        writer.writerow([key, ','.join(values)])
```

Conclusión

Esta implementación de búsqueda binaria usando un índice invertido permite una recuperación eficiente de documentos mediante la evaluación de consultas booleanas. El uso de técnicas de preprocesamiento como la tokenización, eliminación de palabras vacías y stemming, junto con el uso de estructuras de datos eficientes como `defaultdict`, asegura que el motor de búsqueda sea tanto robusto como escalable.

Punto 3: Recuperación ranqueada y vectorización de documentos

La creación de la matriz TF-IDF a partir del índice invertido puede ser un proceso costoso en términos de tiempo, ya que requiere iterar sobre todos los documentos para calcular las frecuencias asociadas a cada término. Este paso es crucial porque, al generar los vectores TF-IDF para los documentos, es necesario que la representación tenga una dimensión consistente con el vocabulario completo, asegurando que las posiciones de los términos sean coherentes en todos los documentos. Esto permite que la matriz TF-IDF se represente con los documentos como filas y los términos como columnas (matriz término-documento), facilitando su uso en análisis posteriores.

Una mejora que podría incrementar la organización y modularidad de la implementación sería estructurar el vectorizador como una clase. Esta clase podría almacenar en sus atributos los términos, los documentos, la matriz TF-IDF y otros valores relevantes para cálculos futuros. Además, se podrían definir

métodos específicos para el cálculo de los vectores TF-IDF y la similitud coseno. Este enfoque no solo evitaría la repetición de parámetros y el uso excesivo de 'drilling' en las funciones independientes, sino que también garantizaría una mayor consistencia y facilidad de mantenimiento. Además, este enfoque es más similar al que implementan librerías reconocidas para este tipo de procesos (por ejemplo, Scikit-learn).

Dado que esta función se ejecuta solo una vez al inicio y sobre un corpus relativamente pequeño, el tiempo de ejecución no es crítico. Sin embargo, para mejorar la eficiencia y escalabilidad de la solución, sería recomendable implementar estas mejoras en el diseño.

Es importante mencionar que toda la implementación se realizó usando NumPy, lo que permite ejecutar funciones de forma vectorizada de manera muy eficiente. Esto es relevante porque en la implementación se desarrollaron dos funciones principales: $crear\_tf\_idf\_matrix$ y $crear\_vector\_tf\_idf$. La primera tiene la funcionalidad limitada a crear únicamente la matriz TF-IDF correspondiente al índice invertido, resultando en una estructura estática y optimizada para realizar consultas sobre todo el corpus a través de las operaciones matriciales propias de NumPy y sus métodos de ordenación a partir de índices. La función $crear\_vector\_tf\_idf$, por otro lado, procesa una entrada individual de texto para obtener su representación vectorial ajustada al corpus previamente procesado (componente IDF). Esta organización permitió reducir al mínimo la necesidad de iterar sobre el corpus para consultar y evaluar el resultado de las mismas.

Finalmente, se validó que los resultados de las implementaciones propias coincidieran con los obtenidos mediante la implementación de Gensim. Sin embargo, para que esto se cumpla, es necesario no escalar logarítmicamente el término TF, ya que Gensim no lo hace, y utilizar el mismo preprocesamiento especificado en el punto dos. De lo contrario, el vocabulario difiere; de hecho, con Gensim resulta aproximadamente 2,000 términos más extenso. A pesar de estas diferencias, los resultados en cada una de las evaluaciones son bastante similares en términos de valor y documentos recuperados, lo que demuestra consistencia entre ambas aproximaciones.

Punto 4: Recuperación ranqueada y vectorización de documentos con GENSIM

En esta sección se usaron las mismas funciones para cargar y leer los documentos. Sin embargo para procesarlos se utilizo la función $preprocess\_string$ de GENSIM, la cual se encarga de realizar varias técnicas de preprocesamiento como remover signos de puntuación, espacios en blanco, valores númericos, stopwords, y hacer stemming.

Luego, se creo un modelo $tfidf$ con los módulos de GENSIM que utiliza tuplas para representar la frecuencia de los términos en cada documento donde el primer valor de la tupla es el identificador de la palabra en el diccionario, y el segundo valor es su frecuencia $tfidf$.

Usando este modelo, se calcula la relevancia de cada documento por cada uno de los 46 queries del conjunto de datos. Adicionalmente, usando las métricas del primer punto, se calcula el P@M, R@M, el MAP, y el NDCG@M donde M corresponde a la cantidad de documentos relevantes de cada query. Se obtiene un MAP de 0.71 muy cercano al obtenido en el punto 3.

Algunos casos interesantes corresponden al query 3 donde se observa no solo una precisión perfecta sino también un valor de NDCG@M de 1.

Los resultados obtenidos son almacenados dentro de la subcarpeta *output/GENSIM*.