



UNIVERSIDAD DE LOS ANDES
FACULTAD DE INGENIERIA
INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

 **Universidad de
los Andes**
Facultad de Ingeniería

TAREA 1 – ISIS 4221

Presented to:

Rubén Francisco Manrique

Presented by

Tomas Acosta Bernal - 202011237
Santiago Pardo Morales - 202013025
Juan Esteban Cuellar Argotty - 202014258
Ayman Benazzouz El Hri - 202424848

BOGOTÁ DC - COLOMBIA

Index

1. Problemas	3
1.1. Implementación de Métricas de Evaluación de IR usando Python+NumPy	3
1.2. Comparación de Estrategias de Motores de Búsqueda	4
1.2.1. [[35p]] Recuperación ranqueada y vectorización de documentos (RRDV)	4
2. Métricas	6
2.1. Precisión (Precision)	6
2.2. Precisión en el Top K (Precision at K)	7
2.3. Recall en el Top K (Recall at K)	7
2.4. Precisión Promedio (Average Precision)	7
2.5. Mean Average Precision (MAP)	8
2.6. Discounted Cumulative Gain (DCG at K)	8
2.7. Normalized Discounted Cumulative Gain (NDCG at K)	9
3. Indice invertido	10
3.0.1. Procesamiento de Texto	12
3.0.2. read_files()	12
3.0.3. replace_title_on_text()	12
3.0.4. process_texts()	13
3.0.5. Procesamiento de Queries	13
3.0.6. read_queries()	14
3.0.7. process_query()	14
3.0.8. process_queries()	14
3.1. Algoritmo de mezcla - AND	14
3.1.1. intersect_two_lists()	14
3.1.2. intersect_multiple_lists()	15
3.2. Algoritmo de mezcla - NOT	16
3.2.1. not_word	16
3.3. Consulta booleana AND	17
3.3.1. SearchEngine.__init__()	17
3.3.2. generate_results_file()	17
4. Recuperación Ranqueada y Vectorización de Documentos (RRDV)	19
4.1. Creación de la Representación TF-IDF	19
4.1.1. Implementación de TF-IDF	19
4.1.2. Optimización: TF-IDF sin Zeros	20
4.2. Similitud del Coseno	20
4.3. Recuperación de Documentos Ordenados por Similitud	21
4.4. Evaluación de Resultados	21
4.4.1. Análisis de los Resultados	22

5. Recuperación Ranqueada y Vectorización de Documentos (RRDV-GENSIM)	23
5.1. Creación de la matriz TF-IDF	23
5.2. Cálculo de la similitud del coseno	24
5.3. Pasos del Algoritmo	24
5.3.1. Requerimientos Previos	24
5.3.2. Representación de Documentos	24
5.3.3. Cálculo de Similitud Coseno	24
5.3.4. Resultado	25
5.4. Pseudocódigo del Algoritmo	25
5.5. Recuperación de documentos clasificados	26
5.6. Pasos del Algoritmo	26
5.6.1. Inicio del Proceso y Carga del Índice Invertido	26
5.6.2. Creación de la Matriz TF-IDF	27
5.6.3. Procesamiento de Consultas y Textos	27
5.6.4. Cálculo de Similitudes Coseno	27
5.6.5. Guardado de Resultados	27
5.6.6. Resumen	27
5.7. Evaluación de resultados	27
5.7.1. Entradas	28
5.7.2. Carga de Juicios de Relevancia	28
5.7.3. Carga de Resultados de Gensim	28
5.7.4. Evaluación de las Consultas	28
5.7.5. Cálculo de Métricas	29
5.7.6. Impresión de Resultados	29

Problemas

1.1. Implementación de Métricas de Evaluación de IR usando Python+NumPy

[10p] Implemente las siguientes métricas de evaluación de IR usando python+numpy (debe usar numpy):

[1p] Precision (la relevancia es binaria)

- `relevance_query_1 = [0, 0, 0, 1]`
- `precision(relevance_query_1)`0.25

[1p] Precision at K (la relevancia es binaria)

- `relevance_query_1 = [0, 0, 0, 1]`
- `k = 1`
- `precision_at_k(relevance_query_1, k)`0

[1p] Recall at K (la relevancia es binaria)

- `relevance_query_1 = [0, 0, 0, 1]`
- `k = 1`
- `number_relevant_docs = 4`
- `recall_at_k(relevance_query_1, number_relevant_docs, k)`
- 0

[1p] Average precision (la relevancia es binaria)oSuponga que el vector binario de entrada contiene todos los documentos relevantes.

- `relevance_query_2 = [0,1,0,1,1,1,1]`
- `average_precision(relevance_query_2)`0.5961904

[2p] Mean average precision -MAP-(la relevancia es binaria)

Entrada: una lista de vectores binarios, cada uno representa un vector de resultado de la consulta.

[2p] DCG at K (la relevancia es un número natural)

- `relevance_query_3 = [4, 4, 3, 0, 0, 1, 3, 3, 3, 0]`
- `k = 6`
- `dcg_at_k(relevance_query_3, k)`
- 10.27964

[2p] NDCG at K (la relevancia es un número natural)

- `relevance_query_3 = [4, 4, 3, 0, 0, 1, 3, 3, 3, 0]`
- `k = 6`
- `ndcg_at_k(relevance_query_3, k)`
- `0.7424`

1.2. Comparación de Estrategias de Motores de Búsqueda

A continuación, implementará un motor de búsqueda con cuatro estrategias diferentes.

- **Búsqueda binaria usando índice invertido (BSII)**
- **Recuperación ranqueada y vectorización de documentos (RRDV)**

Debe hacer su propia implementación usando **NumPy** y **Pandas**.

Conjunto de Datos

Hay tres archivos que componen el conjunto de datos:

- `"Docs raws texts"` contiene 331 documentos en formato NAF (XML; debe usar el título y el contenido para modelar cada documento).
- `"Queries raw texts"` contiene 35 consultas.
- `relevance-judgments.tsv` contiene, para cada consulta, los documentos considerados relevantes para cada una de las consultas. Estos documentos relevantes fueron construidos manualmente por jueces humanos y sirven como *ground-truth* y evaluación.

Pasos de Preprocesamiento

Para los siguientes puntos, debe preprocesar documentos y consultas mediante tokenización a nivel de palabra, eliminación de palabras vacías, normalización y stemming.

[[25p]] Búsqueda binaria usando índice invertido (BSII)

[10p] Cree su propia implementación del índice invertido usando los 331 documentos en el conjunto de datos.

[10p] Cree una función que lea el índice invertido y calcule consultas booleanas mediante el algoritmo de mezcla. El algoritmo de mezcla debe ser capaz de calcular: **AND** y **NOT**.

[5p] Para cada una de las 35 consultas en el conjunto de datos, recupere los documentos utilizando consultas binarias **AND** (i.e., `termino_1 AND termino_2 AND termino_3...`). Escriba un archivo (`BSII-AND-queries.results`) con los resultados siguiendo el mismo formato que `relevance-judgments`:

`q01 dXX,dYY,dZZ...`

Nota: pueden resultar archivos vacíos.

1.2.1. [[35p]] Recuperación ranqueada y vectorización de documentos (RRDV)

[10p] Cree una función que, a partir del índice invertido, cree la representación vectorial ponderada `tf.idf` de un documento o consulta. Describa en detalle su estrategia, ¿es eficiente? ¿por qué sí, por qué no?

[10p] Cree una función que reciba dos vectores de documentos y calcule la similitud del coseno.

[5p] Para cada una de las 35 consultas en el conjunto de datos, recupere los documentos clasificados -ordenados por el puntaje de similitud del coseno- (incluya solo los documentos con un puntaje superior a 0 para una consulta determinada). Escriba un archivo (`RRDV-consultas.resultados`) con los resultados siguiendo el siguiente formato:

`q01 dXX:cos_simi(q01,dXX),dYY:cos_simi(q01,dYY),dZZ:cos_simi(q01,dZZ)...`

[10p] Evaluación de resultados. Calcule $P@M$, $R@M$, $NDCG@M$ por consulta. M es el número de documentos relevantes encontrados en el archivo de juicios de relevancia por consulta. Luego calcule MAP como una métrica general.

- **NOTA I:** Para $P@M$ y $R@M$ suponga una escala de relevancia binaria. Los documentos que no se encuentran en el archivo `relevance-judgments` NO son relevantes para una consulta determinada.
- **NOTA II:** Para $NDCG@M$ utilice la escala de relevancia no binaria que se encuentra en el archivo `relevance-judgments`.

[[30p]] Repita el punto anterior (RRDV) pero utilizando GENSIM (use como nombre de archivo `GESIM-consultas_resultados`).

Métricas

En el primer punto tocaba implementar una serie de métricas de evaluación de sistemas de Recuperación de Información (IR) utilizando Python y la biblioteca Numpy. El objetivo principal de estas métricas es medir la efectividad de un sistema de recuperación de documentos, evaluando cuán bien un conjunto de documentos recuperados corresponde a la relevancia esperada.

Las métricas implementadas fueron:

- Precisión (Precision)
- Precisión en el top K (Precision at K)
- Recall en el top K (Recall at K)
- Precisión Promedio (Average Precision)
- Mean Average Precision (MAP)
- Discounted Cumulative Gain (DCG at K)
- Normalized Discounted Cumulative Gain (NDCG at K)

2.1. Precisión (Precision)

La precisión es la métrica más básica de IR y mide la proporción de documentos relevantes entre el total de documentos recuperados. Su fórmula es:

$$\text{Precisión} = \frac{|\text{Documentos Relevantes Recuperados}|}{|\text{Documentos Recuperados}|}$$

La función Python para calcular la precisión se implementó como sigue:

```
def precision(relevance_query: list):  
    list_of_documents = np.array(relevance_query)  
    number_of_zeros = np.sum(list_of_documents == 0)  
    number_of_ones = np.sum(list_of_documents == 1)  
    precision_calculated = number_of_ones / (number_of_zeros + number_of_ones)  
    return precision_calculated
```

Este código calcula la precisión a partir de una lista de relevancias binarias (1 para documentos relevantes y 0 para irrelevantes).

2.2. Precisión en el Top K (Precision at K)

La precisión en el top K mide la proporción de documentos relevantes entre los primeros K documentos recuperados. Su fórmula es:

$$\text{Precisión at K} = \frac{|\text{Documentos Relevantes Recuperados en el top K}|}{K}$$

La función Python para calcular la precisión en el top K se implementó como sigue:

```
def precision_at_k(relevance_query: list, k: int):
    list_of_documents = np.array(relevance_query)
    k = min(k, len(list_of_documents))
    k_documents = list_of_documents[:k]
    number_of_zeros = np.sum(k_documents == 0)
    number_of_ones = np.sum(k_documents == 1)
    precision_calculated = number_of_ones / (number_of_zeros + number_of_ones)

    return precision_calculated
```

Este código permite evaluar la precisión limitando la cantidad de documentos considerados al top K.

2.3. Recall en el Top K (Recall at K)

El recall en el top K mide la proporción de documentos relevantes recuperados en el top K en comparación con el total de documentos relevantes disponibles. Su fórmula es:

$$\text{Recall at K} = \frac{|\text{Documentos Relevantes Recuperados en el top K}|}{|\text{Total de Documentos Relevantes}|}$$

La función Python para calcular el recall en el top K se implementó como sigue:

```
def recall_at_k(relevance_query: list, number_relevant_docs: int, k: int):
    query_at_k = relevance_query[:k]
    relevant = query_at_k.count(1)
    return (relevant / number_relevant_docs)
```

Este código nos permite evaluar cuántos documentos relevantes recupera el sistema dentro de los primeros K resultados.

2.4. Precisión Promedio (Average Precision)

La precisión promedio mide la precisión en los puntos donde se recuperan documentos relevantes, y luego promedia estos valores. Su fórmula es:

$$\text{Average Precision} = \frac{1}{|\text{Documentos Relevantes}|} \sum_{k=1}^n \text{Precisión en K} \cdot \text{Relevancia}(k)$$

La función Python para calcular la precisión promedio se implementó como sigue:

```
def average_precision(relevance_query: list):
    relevance_query = np.array(relevance_query)
    nb_relevant = np.sum(relevance_query == 1)
    nb_relevant_found = 0
    sum_precisionss = 0
    k = 0

    while nb_relevant != nb_relevant_found and k < len(relevance_query):
        if relevance_query[k] == 1:
            nb_relevant_found += 1
            sum_precisionss += precision_at_k(relevance_query, k+1)
        k += 1

    return sum_precisionss / nb_relevant if nb_relevant > 0 else 0
```

Esta implementación acumula la precisión en los puntos donde se encuentran documentos relevantes y luego calcula el promedio de esas precisiones.

2.5. Mean Average Precision (MAP)

El MAP (Mean Average Precision) es el promedio de la precisión promedio para un conjunto de consultas. Esta métrica nos permite evaluar el rendimiento global del sistema de recuperación de información en diferentes consultas. Su fórmula es:

$$\text{MAP} = \frac{1}{|\text{Consultas}|} \sum_{q=1}^{|\text{Consultas}|} \text{Average Precision}(q)$$

La función Python para calcular MAP se implementó como sigue:

```
def MAP(relevance_queries: list):
    relevance_queries = [np.array(query) for query in relevance_queries]
    map_value = np.mean([average_precision(query) for query in relevance_queries])
    return map_value
```

Esta función toma un conjunto de consultas y calcula el promedio de la precisión promedio para cada una de ellas.

2.6. Discounted Cumulative Gain (DCG at K)

La ganancia acumulada descontada (DCG) mide la relevancia de los documentos recuperados, penalizando aquellos que aparecen en posiciones inferiores. Su fórmula es:

$$\text{DCG}_k = \sum_{i=1}^k \frac{\text{relevancia}(i)}{\log_2(\max(i, 2))}$$

La función Python para calcular el DCG at K se implementó como sigue:

```
def dcg_at_k(relevance_query, k):
    relevance_query = np.array(relevance_query)[:k]
    discounts = np.log2(np.maximum(np.arange(1, k + 1), 2))
    dcg = np.sum(relevance_query / discounts)
    return dcg
```

Este código calcula la ganancia acumulada descontada para los primeros K documentos, tomando en cuenta la relevancia de cada uno.

2.7. Normalized Discounted Cumulative Gain (NDCG at K)

El NDCG (Normalized Discounted Cumulative Gain) es una métrica que normaliza el DCG, dividiéndolo por el mejor resultado posible (IDCG). Esto permite comparar la calidad de diferentes resultados de búsqueda, independientemente de la cantidad de relevancia que exista en el conjunto de documentos. Su fórmula es:

$$\text{NDCG}_k = \frac{\text{DCG}_k}{\text{IDCG}_k}$$

La función Python para calcular NDCG at K se implementó como sigue:

```
def ndcg_at_k(relevance_query: list, k: int):
    dcg = dcg_at_k(relevance_query, k)
    sorted_relevance_query = sorted(relevance_query, reverse=True)
    idcg = dcg_at_k(sorted_relevance_query, k)
    ndcg = dcg / idcg if idcg > 0 else 0
    return ndcg
```

Esta función normaliza el DCG dividiéndolo por el IDCG, asegurando que los valores estén entre 0 y 1, donde 1 representa la recuperación ideal de los documentos relevantes.

Indice invertido

Para la creación del índice invertido, decidimos manejarlo de dos formas diferentes dependiendo de un parámetro de entrada llamado `occurrences`, que puede tomar el valor de `True` o `False`. Si su valor es `False`, como viene por defecto, el índice invertido se utiliza para la búsqueda sin incluir el número de ocurrencias de cada término en los documentos. Tomamos esta decisión por cuestiones de usabilidad en el código, ya que en muchos casos no es necesario saber las ocurrencias exactas de los términos para realizar búsquedas eficientes.

El proceso de creación del índice invertido sigue un orden específico de funciones, que funcionan de la siguiente manera:

```
def inverted_index_complete_pipeline(self, directory: str = "data/docs-raw-texts/", occurrences: bool =
False, filename: str = "inverted_index_without_occurrences.json") -> dict:
    """
    Executes the complete inverted index creation pipeline.

    This method processes the texts, creates the inverted index, and saves it to a file.

    Args:
        directory (str, optional): Directory containing the raw text files. Defaults to "data/docs-
raw-texts/".
        occurrences (bool, optional): Whether to include occurrence counts in the index. Defaults to
False.
        filename (str, optional): Name of the file to save the index. Defaults to
"inverted_index_without_occurrences.json".

    Returns:
        dict: Created inverted index.
    """
    self.logger.info("Starting complete inverted index pipeline")
    overall_start_time = time.time()

    self.logger.info("Step 1: Processing texts")
    processed_df = self.text_processor.process_texts(directory)

    self.logger.info("Step 2: Applying vectorizer and processing")
    vectorized_df = self.apply_vectorizer_and_process(processed_df)

    self.logger.info("Step 3: Creating inverted index")
    inverted_index_to_return = self.inverted_index(vectorized_df, occurrences)

    self.logger.info("Step 4: Saving inverted index")
    self.save_inverted_index(inverted_index_to_return, filename)

    overall_end_time = time.time()
    self.logger.info(f"Completed inverted index pipeline. Total time taken: {overall_end_time -
overall_start_time:.2f} seconds")

    return inverted_index_to_return
```

Figura 3.1: Índice invertido

Inicialización y Configuración del Logger

El método comienza con la configuración de un *logger* para registrar las actividades y el tiempo de ejecución de cada paso.

Procesamiento de Textos

Utiliza el método `process_texts` del objeto `TextProcessor` para procesar los textos crudos ubicados en un directorio específico (por defecto, `"data/docs-raw-texts/"`). Este procesamiento incluye tokenización y limpieza del texto.

Aplicación de Vectorizador y Procesamiento

Posteriormente, el método llama a `apply_vectorizer_and_process`, que aplica un `CountVectorizer` de `sklearn` para transformar el texto procesado en una matriz de ocurrencias de palabras. Este vectorizador convierte los textos en una matriz donde las filas representan documentos y las columnas representan términos. Cada celda contiene la frecuencia con la que un término aparece en el documento correspondiente.

Creación del Índice Invertido

Luego, el método `inverted_index` crea un diccionario que mapea cada término (palabra) a los documentos donde aparece. Si el parámetro `occurrences` es verdadero, también agrega el número de ocurrencias del término en el índice. Este índice invertido permite saber rápidamente en qué documentos aparece cada término.

Guardado del Índice Invertido

Una vez creado el índice invertido, el método lo guarda en un archivo JSON, cuyo nombre por defecto es `inverted_index_without_occurrences.json`. Si se desea incluir las ocurrencias de los términos, este archivo incluirá los recuentos de términos junto a los documentos.

Retorno del Índice Invertido

Finalmente, devuelve el índice invertido como un diccionario, con el término como clave y la lista de documentos donde aparece como valor.

3.0.1. Procesamiento de Texto

```
def process_texts(self, directory: str = "data/docs-raw-texts/") -> pd.DataFrame:
    """
    Executes the complete text processing pipeline.

    This method reads the files, processes the text content, and applies various
    text normalization techniques including tokenization, lowercasing, punctuation removal,
    non-ASCII character removal, stopword removal, and verb stemming.

    Args:
        directory (str, optional): Directory containing the NAF files.
                                   Defaults to "data/docs-raw-texts/".

    Returns:
        pd.DataFrame: Processed DataFrame containing 'identifier', 'text', and 'text_list' columns.
    """
    self.logger.info("Starting text processing pipeline")
    start_time = time.time()

    df = self.read_files(directory)
    df = self.replace_title_on_text(df)

    self.logger.info("Processing and normalizing text")
    df["text"] = df["text"] + " " + df["title"]
    df.drop(columns=["title"], inplace=True)

    for i in range(len(df["text"])):
        text = df["text"][i]
        processed_text = word_tokenize(text)
        processed_text = self.processor_.to_lowercase(processed_text)
        processed_text = self.processor_.remove_punctuation(processed_text)
        processed_text = self.processor_.remove_stopwords(processed_text)
        processed_text = self.processor_.stem_verbs(processed_text)
        df.at[i, "text"] = ' '.join(processed_text)

    df["text_list"] = df["text"].apply(lambda x: x.split())

    end_time = time.time()
    self.logger.info(f"Completed text processing pipeline. Total time taken: {end_time - start_time:.2f} seconds")

    return df
```

Figura 3.2: Procesamiento de Texto

La clase `TextProcessor` está diseñada para procesar textos de archivos en formato NAF. Se encarga de leer los archivos, extraer la información relevante y realizar normalizaciones sobre los textos. A continuación, se detallan los principales métodos de esta clase:

3.0.2. `read_files()`

Este método lee los archivos en formato NAF desde un directorio especificado "data/docs-raw-texts/". Sus principales pasos son:

- Leer archivos con extensión `.naf` del directorio.
- Extraer el identificador del archivo, el texto crudo y el título de cada documento.
- Crear un `DataFrame` con las columnas `identifier`, `text`, y `title`.

El método retorna un `DataFrame` con la información de los archivos.

3.0.3. `replace_title_on_text()`

Este método recibe como argumento el `DataFrame` resultante de `read_files()`. Se encarga de:

- Eliminar el título del comienzo del texto, si es que está presente.
- Limpiar el texto eliminando saltos de línea.

El resultado es un `DataFrame` con los textos limpios y sin los títulos duplicados.

3.0.4. process_texts()

Este es el método principal que ejecuta todo el pipeline de procesamiento de textos. Sus pasos incluyen:

- Llamar a `read_files()` para leer los archivos y luego a `replace_title_on_text()` para limpiar los textos.
- Aplicar las siguientes técnicas de normalización de texto:
 - **Tokenización:** Dividir el texto en palabras.
 - **Minúsculas:** Convertir todo el texto a minúsculas.
 - **Eliminación de puntuación:** Remover signos de puntuación.
 - **Eliminación de stopwords:** Remover palabras vacías (palabras comunes como artículos y preposiciones).
 - **Stemming:** Aplicar un proceso de reducción de las palabras a su raíz verbal.
- Crear una nueva columna `text_list` que contiene una lista de palabras normalizadas para cada texto.

El resultado es un DataFrame con las columnas `identifier`, `text` (texto procesado) y `text_list` (lista de palabras normalizadas).

3.0.5. Procesamiento de Queries

```
def process_query(self, query: str) -> List[str]:
    """
    Process a single query through the preprocessing pipeline.

    Args:
        query (str): Raw query text.

    Returns:
        List[str]: Processed query as a list of tokens.
    """
    tokens = word_tokenize(query)
    tokens = self.processor.to_lowercase(tokens)
    tokens = self.processor.remove_punctuation(tokens)
    tokens = self.processor.remove_stopwords(tokens)
    tokens = self.processor.stem_verbs(tokens)
    return tokens

def process_queries(self) -> pd.DataFrame:
    """
    Process all queries in the DataFrame.

    Returns:
        pd.DataFrame: DataFrame with processed queries.
    """
    if self.queries_df is None:
        self.read_queries()

    self.queries_df["processed_query"] = self.queries_df["query"].apply(self.process_query)
    self.queries_df["query_list"] = self.queries_df["processed_query"].apply(lambda x: ' '.join(x).split())
    logger.info("Processed all queries")
    return self.queries_df
```

Figura 3.3: Procesamiento de Queries

La clase `QueryProcessor` está diseñada para manejar y procesar consultas (queries). A través de varios métodos, se encarga de leer los archivos que contienen las consultas, normalizar el texto y realizar los preprocesamientos necesarios para preparar las consultas para su uso en tareas de recuperación de información o análisis.

3.0.6. `read_queries()`

Este método lee archivos de consultas desde un directorio especificado. Sus principales pasos son:

- Leer archivos en formato NAF que contienen las consultas.
- Extraer el identificador de la consulta y el texto crudo de la consulta.
- Crear un `DataFrame` con las columnas `identifier` y `query`.

El método retorna un `DataFrame` que contiene las consultas crudas leídas de los archivos.

3.0.7. `process_query()`

Este método toma una consulta cruda como argumento y la procesa mediante varios pasos de preprocesamiento:

- **Tokenización:** Divide la consulta en palabras o tokens.
- **Minúsculas:** Convierte todo el texto de la consulta a minúsculas.
- **Eliminación de puntuación:** Elimina cualquier signo de puntuación del texto.
- **Eliminación de stopwords:** Remueve palabras vacías que no contribuyen al significado de la consulta.
- **Stemming:** Reduce las palabras a su raíz o forma base.

El resultado es una lista de tokens procesados que representa la consulta.

3.0.8. `process_queries()`

Este método procesa todas las consultas que han sido leídas en el `DataFrame` previamente generado. Sus pasos incluyen:

- Aplicar el método `process_query()` a cada consulta del `DataFrame`.
- Crear una nueva columna llamada `processed_query` que contiene las consultas procesadas.
- Crear una columna adicional llamada `query_list` que contiene las consultas procesadas en forma de listas de palabras.

El resultado es un `DataFrame` que contiene tanto las consultas originales como las consultas procesadas, listas para ser usadas en tareas de análisis o recuperación de información.

3.1. Algoritmo de mezcla - AND

Los métodos `intersect_two_lists` y `intersect_multiple_lists` están diseñados para realizar la operación de intersección entre listas de postings. Estos métodos buscan encontrar documentos comunes en varias listas de postings.

3.1.1. `intersect_two_lists()`

Este método realiza la intersección entre dos listas de postings. Los pasos principales son:

- Se reciben dos listas ordenadas de IDs de documentos, `p1` y `p2`, correspondientes a los postings de dos términos diferentes.
- Se inicializan dos índices, `i` y `j`, para recorrer ambas listas.
- Mientras ambos índices estén dentro de los límites de sus respectivas listas:
 - Si el ID de documento en `p1[i]` es igual al de `p2[j]`, ese ID se agrega a la lista de resultados (`answer`).
 - Si `p1[i]` es menor que `p2[j]`, se incrementa el índice `i`.
 - Si `p2[j]` es menor que `p1[i]`, se incrementa el índice `j`.
- El proceso continúa hasta que se recorren ambas listas por completo.

El resultado es una lista de IDs de documentos que están presentes tanto en `p1` como en `p2`.

```

def intersect_two_lists(self, p1, p2):
    """
    Perform the merge operation (intersection) between two posting lists.

    Args:
        p1 (list): Posting list of term 1 (sorted list of document IDs).
        p2 (list): Posting list of term 2 (sorted list of document IDs).

    Returns:
        list: A list of document IDs that are present in both p1 and p2.
    """
    answer = []
    i, j = 0, 0

    while i < len(p1) and j < len(p2):
        if p1[i] == p2[j]:
            answer.append(p1[i])
            i += 1
            j += 1
        elif p1[i] < p2[j]:
            i += 1
        else:
            j += 1

    return answer

def intersect_multiple_lists(self, lists):
    """
    Realiza la intersección de múltiples listas.

    Args:
        lists (list of lists): Lista de listas de posting a intersectar.

    Returns:
        list: Lista resultante de la intersección de todas las listas de entrada.
    """
    if not lists:
        return []

    if len(lists) == 1:
        return lists[0]

    result = lists[0]
    for i in range(1, len(lists)):
        result = self.intersect_two_lists(result, lists[i])

    return result

```

Figura 3.4: Algoritmo de mezcla - AND

3.1.2. intersect_multiple_lists()

Este método realiza la intersección entre múltiples listas de postings. Sus pasos son:

- Se recibe una lista de listas de postings, donde cada lista corresponde a un conjunto de IDs de documentos asociados con un término.
- Si no se proporciona ninguna lista, el resultado es una lista vacía.
- Si solo hay una lista, se devuelve tal cual.
- En caso de haber más de una lista:
 - Se toma la primera lista como resultado inicial.
 - Para cada lista posterior, se llama al método `intersect_two_lists()` para realizar la intersección entre el resultado actual y la siguiente lista.
 - El proceso continúa hasta que todas las listas hayan sido intersectadas.

El resultado es una lista de IDs de documentos que están presentes en todas las listas de postings.

3.2. Algoritmo de mezcla - NOT

3.2.1. not_word



```
def not_word(self, p, inv_index):
    """
    Realiza múltiples intersecciones alrededor de todo el índice

    Args:
        p: Palabra de la cual se desea obtener los documentos en los cuales no aparece
        index: Todo el índice invertido
    """
    # Obtengo todos los documentos de la palabra de entrada
    docs_with_word = inv_index.get(p, [])

    docs_without_word = []

    for i, docs in inv_index.items():
        if i != p:
            for doc in docs:
                if doc not in docs_with_word:
                    # Se añade la palabra a la respuesta si todos documentos NO coinciden con los
                    # documentos de la palabra de entrada
                    if doc not in docs_without_word:
                        docs_without_word.append(doc)

    return {p: sorted(docs_without_word)}
```

Figura 3.5: Algoritmo de mezcla - NOT

Dentro de la clase BinarySerch también, se realiza la interseccion entre una lista de posting con múltiples listas de postings mediante una comparación, sus pasos son:

- Se recibe todo el índice y una palabra en específico dentro del diccionario
- Se extraen todos los documentos en los que se encuentra la palabra de entrada
- Se recorren las demás palabras del índice, y si ninguno de los documentos de la palabra en la iteración actual contiene alguno de los documentos de la palabra seleccionada, se añade a la respuesta en caso de que no se haya añadido.
- El proceso continúa hasta que todas las listas hayan sido recorridas

El resultado es un diccionario con la palabra de entrada como clave y sus valores son una lista con todos los documentos en los cuales esta palabra no aparece.

3.3. Consulta booleana AND



```
import json
import logging
import pandas as pd
from algorithms.binary_search.binary_search import BinarySearch
from algorithms.binary_search.query_processor import QueryProcessor
from algorithms.binary_search.inverted_index import InvertedIndex

# Set up logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)

class SearchEngine:
    """
    A class for performing binary search on processed queries.
    """
    def __init__(self, inverted_index_path: str = "inverted_index_without_occurrences.json"):
        """
        Initialize the SearchEngine.

        Args:
            inverted_index_path (str): Path to the inverted index JSON file.
        """
        try:
            with open(inverted_index_path) as file:
                self.inverted_index = json.load(file)
        except Exception as e:
            index = InvertedIndex()
            self.inverted_index = index.inverted_index_complete_pipeline()

        self.binary_search = BinarySearch()
        logger.info("SearchEngine initialized")

    def generate_results_file(self, queries_df: pd.DataFrame, output_file: str):
        """
        Generate a results file based on binary search of processed queries.

        Args:
            queries_df (pd.DataFrame): DataFrame containing processed queries.
            output_file (str): Path to the output file.
        """
        self.binary_search.generate_results_file(queries_df, self.inverted_index, output_file)
        logger.info(f"Results file generated: {output_file}")
```

Figura 3.6: Consulta booleana AND

La clase `SearchEngine` está diseñada para realizar búsquedas binarias en consultas procesadas, utilizando un índice invertido. El índice invertido se carga desde un archivo JSON (En caso de haber uno) y las consultas se comparan contra este índice para generar archivos de resultados.

3.3.1. `SearchEngine.__init__()`

Este es el constructor de la clase y se encarga de inicializar la instancia del motor de búsqueda. Los pasos principales incluyen:

- Recibe como parámetro la ruta del archivo JSON que contiene el índice invertido. Si el archivo no puede abrirse por alguna razón, se genera un nuevo índice invertido utilizando la clase `InvertedIndex`.
- Carga el índice invertido desde el archivo JSON y lo almacena en `self.inverted_index`.
- Inicializa una instancia de la clase `BinarySearch` para realizar las búsquedas.
- Registra la inicialización correcta del motor de búsqueda utilizando el sistema de logging.

3.3.2. `generate_results_file()`

Este método se encarga de generar un archivo de resultados basado en la búsqueda binaria de las consultas procesadas. Los pasos que sigue son:

- Recibe un `DataFrame` que contiene las consultas ya procesadas, y la ruta de destino para el archivo de salida.
- Utiliza el método `generate_results_file()` de la clase `BinarySearch`, pasando como argumentos el `DataFrame` de consultas, el índice invertido y la ruta del archivo de salida.
- Una vez que el archivo de resultados se ha generado, se registra un mensaje en el sistema de logging indicando el éxito de la operación.

La clase `SearchEngine` utiliza varios paquetes y módulos importantes:

- json: Para cargar y manejar el archivo JSON que contiene el índice invertido.
- logging: Para generar mensajes de log sobre el estado de las operaciones (inicialización, generación de resultados).
- pandas: Para manejar el DataFrame que contiene las consultas procesadas.
- BinarySearch, QueryProcessor, InvertedIndex: Clases específicas que ayudan a realizar búsquedas binarias, procesar consultas y generar el índice invertido, respectivamente.

Generando los resultados que se pueden ver a continuación:

```
q01
q02 d291,d293
q03 d105,d147,d152,d283,d291,d318
q04 d286
q05 d026,d029,d069,d257,d297,d303,d329
q06 d004,d034
q07 d100,d110,d117,d205,d251
q08 d198,d205,d223
q09 d231
q10 d250,d277
q11
q12
q13 d132,d150,d176,d184,d229,d250,d277
q14 d121,d271
q15 d192,d194,d203,d210
q16 d179
q17
q18
q19 d129,d221,d240,d282
q20 d020,d032,d167,d211
q21
q22
q23 d136,d174
q24 d037,d046,d294
q25 d025,d031,d090,d139,d254
q26
q27 d257,d265
q28 d169
q29
q30
q31 d150,d174
q32
q33 d029,d185
q34 d105
q35 d004,d133
```

Figura 3.7: SearchEngine

Recuperación Ranqueada y Vectorización de Documentos (RRDV)

En este punto, se solicitaba implementar un sistema de recuperación ranqueada basado en la vectorización de documentos utilizando la técnica de TF-IDF y la similitud del coseno. Este sistema debe permitir recuperar documentos relevantes para un conjunto de consultas, ranqueando los documentos en función de su similitud con las consultas. El proceso se dividió en cuatro fases principales:

1. Creación de una representación vectorial ponderada TF-IDF para cada documento y consulta.
2. Cálculo de la similitud del coseno entre los vectores TF-IDF de los documentos y las consultas.
3. Recuperación de documentos, ordenados por el puntaje de similitud del coseno.
4. Evaluación de los resultados obtenidos utilizando las métricas solicitadas.

4.1. Creación de la Representación TF-IDF

Para representar los documentos y consultas de manera vectorial, se utilizó la técnica TF-IDF (Term Frequency - Inverse Document Frequency). Esta técnica tiene como objetivo ponderar los términos de manera que aquellos que aparezcan con frecuencia en un documento, pero que no sean comunes en todo el corpus, obtengan un peso mayor.

4.1.1. Implementación de TF-IDF

La función para calcular la representación TF-IDF de un documento o consulta es la siguiente:

```
def tfIdf(invertedIndex, N, doc):
    vectTfIdf = [0] * len(invertedIndex)
    for word in doc:
        if word in invertedIndex:
            tf = np.log10(1 + doc.count(word))
            idf = np.log10(N / len(invertedIndex[word]))
            tfIdf = tf * idf
            vectTfIdf[list(invertedIndex.keys()).index(word)] = tfIdf
    return vectTfIdf
```

Descripción de la función

- 1. Entrada: La función toma como entrada el índice invertido, el número total de documentos N y la lista de palabras del documento o consulta que se va a procesar.
- 2. TF (Frecuencia del Término): Para cada palabra en el documento, se calcula la frecuencia del término (TF) usando $\log(1 + tf)$ para evitar el crecimiento lineal con el número de repeticiones.
- 3. IDF (Frecuencia Inversa de Documentos): El IDF se calcula utilizando la fórmula $\log\left(\frac{N}{DF(t)}\right)$, donde $DF(t)$ es el número de documentos en los que aparece el término.
- 4. Vectorización: Se asigna el valor TF-IDF al índice correspondiente de la palabra en el vector de salida.

Este enfoque utiliza un vector lleno de ceros, lo que no es eficiente para grandes corpus donde la mayoría de los términos no están presentes en cada documento.

4.1.2. Optimización: TF-IDF sin Zeros

Para optimizar el espacio utilizado, se implementó una versión mejorada que almacena solo los términos que están presentes en el documento o consulta, utilizando un diccionario. Esto es especialmente útil cuando el vocabulario es extenso pero los documentos son relativamente pequeños, como suele ser el caso en sistemas de recuperación de información.

```
def tfIdfSinZeros(invertedIndex, N, doc):
    vectTfIdf = {}
    for word in doc:
        if word in invertedIndex:
            tf = np.log10(1 + doc.count(word))
            idf = np.log10(N / len(invertedIndex[word]))
            tfIdf = tf * idf
            vectTfIdf[word] = tfIdf
    return vectTfIdf
```

Este enfoque tiene varias ventajas:

- Eficiencia en Memoria: En lugar de almacenar largos vectores llenos de ceros, se almacenan solo los términos que aparecen en el documento.
- Acceso Directo: Los términos relevantes pueden accederse directamente, lo que permite realizar operaciones sobre ellos sin la necesidad de iterar sobre términos irrelevantes.
- Optimización del Espacio de Almacenamiento: Al usar un diccionario en lugar de un array, se reduce significativamente el espacio requerido, especialmente en corpus con un gran vocabulario.

4.2. Similitud del Coseno

La similitud del coseno es una métrica que mide la similitud entre dos vectores, en este caso, entre los vectores TF-IDF de un documento y una consulta. Se calcula de la siguiente manera:

$$\cos(\theta) = \frac{A \cdot B}{||A|| \cdot ||B||}$$

Donde:

- A y B son los vectores TF-IDF.
- $A \cdot B$ es el producto punto entre los vectores.
- $||A||$ y $||B||$ son las normas de los vectores.

Implementación de la Similitud del Coseno

La similitud del coseno se implementó con la siguiente función:

```
def cosineSimilarity(tfIdf1, tfIdf2):
    producto_punto = np.dot(tfIdf1, tfIdf2)
    normas = (np.linalg.norm(tfIdf1) * np.linalg.norm(tfIdf2))
    return producto_punto / normas
```

Descripción de la función

1. Producto Punto: Se calcula el producto punto entre los dos vectores TF-IDF.
2. Normas: Se calculan las normas de los dos vectores
3. Cálculo Final: La similitud se obtiene dividiendo el producto punto entre las normas.

El resultado es un valor entre 0 y 1, donde 1 indica que los vectores son idénticos en dirección (máxima similitud).

4.3. Recuperación de Documentos Ordenados por Similitud

Una vez calculadas las similitudes entre una consulta y todos los documentos del corpus, los documentos se ordenan de mayor a menor según su similitud, y aquellos con una similitud mayor a 0 se recuperan.

La función para recuperar los documentos es:

```
def cosineSimilarityDocQuery(inverseIndex, corpus):
    N = text_df.shape[0]
    queriesProcessed = queries_df['query_list'].values
    results_file = open("RRDV-consultas_results.txt", "w")
    tfIdfDoc = {}
    for i, doc in enumerate(corpus, 1):
        tfIdfDoc[i] = tfIdf(inverseIndex, N, doc)

    for index_query, query in enumerate(queriesProcessed, 1):
        tfIdfQuery = tfIdf(inverseIndex, N, query)
        similarities = [
            [i, cosineSimilarity(tfIdfDoc[i], tfIdfQuery)] for i, doc in enumerate(corpus,
                                                                                     1)
        ]
        similarities = [s for s in similarities if s[1] != 0]
        similarities.sort(key=lambda x: x[1], reverse=True)
        results_file.write(f"q{index_query:02} " + " ", ".join([f"d{doc}:{sim:.4f}" for doc,
                                                                sim in similarities]) + "\n")
    results_file.close()
```

Descripción de la función

1. Cálculo de Similitudes: Para cada consulta, se calculan las similitudes del coseno entre la consulta y cada documento del corpus.
2. Ordenación: Los documentos se ordenan de mayor a menor similitud.
3. Recuperación: Se recuperan los documentos cuya similitud es mayor a 0, y los resultados se escriben en un archivo.

4.4. Evaluación de Resultados

Para evaluar la efectividad del sistema de recuperación ranqueada, se calcularon las métricas de Precisión (P@M), Recall (R@M) y NDCG (NDCG@M) para cada consulta.

En lugar de calcular las métricas individualmente para cada consulta, se generó un *dataframe* que almacena estas métricas para todas las consultas, permitiendo observar el rendimiento del sistema de manera consolidada.

Esto permitió obtener una tabla con las métricas de precisión, recall y NDCG para cada consulta. A continuación, se muestran algunos de los resultados obtenidos:

Query	Precision	Recall	NDCG
q1	0.000000	0.000000	0.000000
q2	0.181818	0.181818	0.084390
q3	0.000000	0.000000	0.000000
q11	0.200000	0.200000	0.359608
q12	0.083333	0.083333	0.029996
q27	0.100000	0.100000	0.080674

Tabla 4.1: Resultados de Precisión, Recall y NDCG para algunas consultas.

Finalmente, el Mean Average Precision (MAP), que promedia la precisión a lo largo de todas las consultas, fue calculado, obteniendo un valor de 0.032.

4.4.1. Análisis de los Resultados

- Bajo Rendimiento General: La mayoría de las consultas arrojaron valores de precisión, recall y NDCG muy bajos, incluso 0 en muchos casos. Esto indica que el sistema tiene dificultades para recuperar documentos relevantes o para ranquearlos de manera adecuada.
- Consultas con Desempeño Notable: Algunas consultas, como *q2*, *q11* y *q27*, lograron un desempeño aceptable en términos de precisión y NDCG, lo que sugiere que para ciertos casos el modelo puede capturar correctamente los documentos relevantes y su orden. Sin embargo, estas consultas son una minoría.
- El valor de MAP obtenido, que es de aproximadamente 0.0327, refleja un bajo rendimiento en la recuperación general del sistema. Esto sugiere que el modelo, tal como está, no es particularmente efectivo para recuperar documentos relevantes con un alto grado de precisión y ordenación.

Recuperación Ranqueada y Vectorización de Documentos (RRDV-GENSIM)

5.1. Creación de la matriz TF-IDF



```
def create_tfidf_matrix(self, inverted_index: dict):
    """
    Create a TF-IDF matrix from the given inverted index.

    Parameters:
    inverted_index (dict): The inverted index dictionary.

    Returns:
    tuple: A tuple containing:
    - corpus_tfidf (list of gensim.interfaces.TransformedCorpus): The TF-IDF matrix.
    - dictionary (gensim.corpora.Dictionary): The dictionary mapping words to their ids.
    - df (pandas.DataFrame): The processed inverted index DataFrame with an additional
      'term_id' column.
    """
    self.inverted_index_occurrences = inverted_index
    df = self.process_inverted_index()
    documentos_list = self.create_documents(df)

    self.dictionary = corpora.Dictionary(documentos_list)
    df['term_id'] = df['termno'].map(lambda x: self.dictionary.token2id.get(x, -1))
    corpus = [self.dictionary.doc2bow(doc) for doc in documentos_list]
    self.tfidf_model = TfidfModel(corpus)
    corpus_tfidf = self.tfidf_model[corpus]

    return corpus_tfidf
```

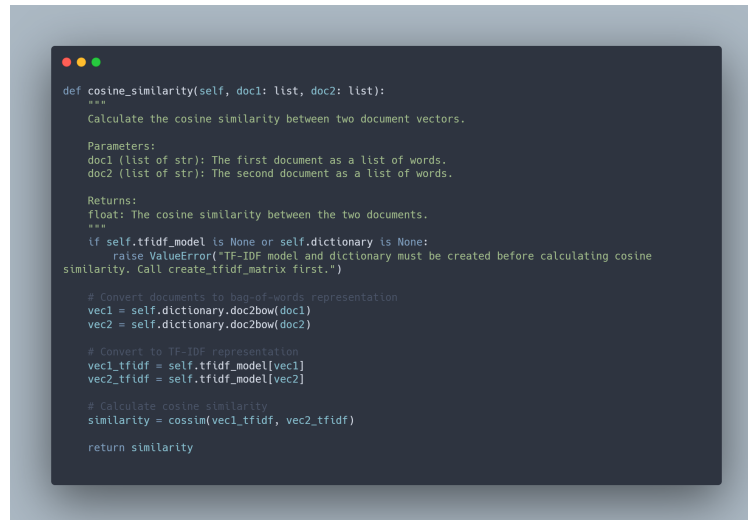
Figura 5.1: Matriz TF-IDF Gensim

Para crear la representación vectorial ponderada TF-IDF de un documento o consulta, utilizamos la clase `RRDVGensim` del archivo proporcionado. El método `create_tfidf_matrix()` se encarga de crear una matriz TF-IDF a partir de un índice invertido. Los pasos principales son:

1. Se carga el índice invertido, que contiene los términos y los documentos en los que aparecen.
2. Se genera una lista de documentos, donde cada documento es una lista de términos que aparecen en él.
3. Se crea un diccionario de términos con `gensim.corpora.Dictionary`, que asigna un ID único a cada término.
4. Cada documento se convierte en su representación Bag of Words (BoW), que cuenta la ocurrencia de cada término en el documento.
5. Finalmente, se aplica el modelo TF-IDF utilizando `gensim.models.TfidfModel` para obtener la representación ponderada TF-IDF de cada documento.

La estrategia es eficiente, ya que Gensim está optimizado para manejar grandes colecciones de documentos y términos, lo que reduce el costo computacional en comparación con implementar TF-IDF desde cero. Sin embargo, la eficiencia depende de la cantidad de documentos y términos en el índice invertido.

5.2. Cálculo de la similitud del coseno



```
def cosine_similarity(self, doc1: list, doc2: list):
    """
    Calculate the cosine similarity between two document vectors.

    Parameters:
    doc1 (list of str): The first document as a list of words.
    doc2 (list of str): The second document as a list of words.

    Returns:
    float: The cosine similarity between the two documents.
    """
    if self.tfidf_model is None or self.dictionary is None:
        raise ValueError("TF-IDF model and dictionary must be created before calculating cosine similarity. Call create_tfidf_matrix first.")

    # Convert documents to bag-of-words representation
    vec1 = self.dictionary.doc2bow(doc1)
    vec2 = self.dictionary.doc2bow(doc2)

    # Convert to TF-IDF representation
    vec1_tfidf = self.tfidf_model[vec1]
    vec2_tfidf = self.tfidf_model[vec2]

    # Calculate cosine similarity
    similarity = cossim(vec1_tfidf, vec2_tfidf)

    return similarity
```

Figura 5.2: similitud del coseno

El método de `cosine_similarity()` calcula la similitud coseno entre dos documentos representados como vectores. La similitud coseno es una métrica que mide la semejanza entre dos vectores mediante el cálculo del coseno del ángulo entre ellos. Es comúnmente utilizada en tareas de recuperación de información y procesamiento de lenguaje natural.

5.3. Pasos del Algoritmo

5.3.1. Requerimientos Previos

Para calcular la similitud coseno entre dos documentos, es necesario que:

- Se haya creado un modelo TF-IDF con Gensim.
- Exista un diccionario de términos que mapee las palabras a sus identificadores.

Si no se han creado el modelo TF-IDF y el diccionario, el método lanza una excepción.

5.3.2. Representación de Documentos

- Cada documento es representado como un vector bag-of-words (BOW) mediante el uso del diccionario.
- El método `doc2bow()` de Gensim convierte cada documento en una lista de pares (término, frecuencia).
- Después, estos vectores BOW se transforman a su representación TF-IDF utilizando el modelo TF-IDF previamente creado.

5.3.3. Cálculo de Similitud Coseno

- La similitud coseno se calcula utilizando la fórmula:

$$\text{sim}(A, B) = \frac{A \cdot B}{\|A\| \|B\|}$$

donde A y B son los vectores TF-IDF de los dos documentos.

- El producto punto ($A \cdot B$) se calcula sumando los productos de los valores correspondientes de cada término en los dos vectores.

-
- Las magnitudes $||A||$ y $||B||$ se calculan como la raíz cuadrada de la suma de los cuadrados de los valores de cada vector.

5.3.4. Resultado

El método devuelve un valor numérico entre -1 y 1:

- Un valor cercano a 1 indica que los documentos son muy similares.
- Un valor cercano a 0 indica que los documentos no tienen mucha similitud.
- Un valor negativo indicaría que los documentos son completamente opuestos, aunque este caso es raro en aplicaciones de TF-IDF.

5.4. Pseudocódigo del Algoritmo

1. Verificar si el modelo TF-IDF y el diccionario han sido creados.
2. Convertir el documento 1 y el documento 2 a su representación bag-of-words.
3. Transformar los vectores BOW a vectores TF-IDF.
4. Calcular la similitud coseno utilizando la función `cossim()` de Gensim, que calcula el producto punto y las magnitudes de los vectores.
5. Devolver el valor de la similitud coseno.

5.5. Recuperación de documentos clasificados

```
def sorted_query(self, query_list):
    """
    Sorts a list received by parameter

    Parameters:
        query_list: The query list of the documents that want to be ordered
    Returns:
        list: list sorted
    """
    resultado = []

    for item in query_list:
        query, documentos_str = item.split(' ', 1)
        documentos = documentos_str.split(',')

        documentos_ordenados = sorted(documentos, key=lambda x: float(x.split(':')[1]), reverse=True)

        resultado.append(f'{query} {'.'.join(documentos_ordenados)}')

    return resultado

def process_and_save_results(self, output_filename: str = "results/GENSIM-consultas_resultado.txt"):
    """
    Process queries and texts, calculate cosine similarities, and save results to a file.

    Parameters:
        queries_file (str): Path to the queries file.
        texts_directory (str): Path to the directory containing text files.
        output_filename (str): Name of the output file to save results.

    Returns:
        None
    """
    start_time = time.time()
    self.logger.info("Starting RRDV process...")

    # Load or create inverted index
    self.load_inverted_index()
    self.logger.info("25% complete - Inverted index loaded/created")

    # Create TF-IDF matrix
    self.logger.info("Creating TF-IDF matrix...")
    self.create_tfidf_matrix(self.inverted_index_occurrences)
    self.logger.info("50% complete - TF-IDF matrix created")

    # Process queries and texts
    self.logger.info("Processing queries and texts...")
    query_processor = QueryProcessor()
    text_processor = TextProcessor()

    df_queries = query_processor.process_queries()
    df_texts = text_processor.process_texts()
    self.logger.info("75% complete - Queries and texts processed")

    # Calculate and format cosine similarities
    self.logger.info("Calculating cosine similarities...")
    formatted_result = self.format_cosine_similarities(df_queries, df_texts)

    # Save results to file
    self.logger.info(f"Saving results to {output_filename}...")
    with open(output_filename, 'w') as f:
        f.write(formatted_result)

    end_time = time.time()
    total_time = end_time - start_time
    self.logger.info(f"100% complete - Results saved in {output_filename}")
    self.logger.info(f"Total processing time: {total_time:.2f} seconds")
```

Figura 5.3: Recuperación de documentos clasificados

Para recuperar documentos clasificados en base a su similitud del coseno, el método `format_cosine_similarities()` calcula las similitudes entre las consultas y los documentos. Los pasos son:

El método `process_and_save_results()` en la clase `RRDVGensim` sigue un flujo para procesar consultas y textos, calcular las similitudes de coseno y guardar los resultados en un archivo de texto. A continuación, se describe el algoritmo paso a paso.

5.6. Pasos del Algoritmo

5.6.1. Inicio del Proceso y Carga del Índice Invertido

- El método comienza registrando en el logger que el proceso de RRDV ha comenzado.
- A continuación, intenta cargar el índice invertido desde un archivo JSON.
- Si el archivo no existe, se crea un nuevo índice invertido utilizando la clase `InvertedIndex` y se guarda.
- Cuando se completa esta operación, se registra que el 25 % del proceso ha sido completado.

5.6.2. Creación de la Matriz TF-IDF

- Se utiliza el índice invertido cargado o creado para generar una matriz TF-IDF mediante Gensim.
- Una vez que se ha creado la matriz TF-IDF, el logger indica que el 50 % del proceso ha sido completado.

5.6.3. Procesamiento de Consultas y Textos

- Se inicializan instancias de las clases `QueryProcessor` y `TextProcessor` para procesar las consultas y los textos, respectivamente.
- Se generan dos DataFrames: uno para las consultas (`df_queries`) y otro para los textos (`df_texts`), ambos con los textos ya procesados.
- Al finalizar este paso, se registra que el 75 % del proceso ha sido completado.

5.6.4. Cálculo de Similitudes Coseno

- El algoritmo calcula las similitudes coseno entre cada consulta y cada documento utilizando la matriz TF-IDF generada.
- Durante el cálculo, se registra el progreso en intervalos de 100 comparaciones, indicando el porcentaje completado del total.
- Los resultados de las similitudes se formatean para facilitar su análisis.

5.6.5. Guardado de Resultados

- Los resultados formateados se guardan en un archivo de texto, cuyo nombre se especifica en el parámetro `output_filename`.
- Una vez que se ha completado el guardado, se registra que el 100 % del proceso ha sido completado.
- También se registra el tiempo total que tomó el proceso.

5.6.6. Resumen

1. Se calcula la similitud del coseno entre cada consulta y cada documento en el conjunto de datos.
2. Se filtran aquellos documentos cuya similitud es mayor a 0.
3. Los documentos recuperados para cada consulta se ordenan por el puntaje de similitud del coseno.
4. Finalmente, los resultados se formatean en el siguiente formato: `qXX dYY:cos simi(qXX,dYY),dZZ:cos simi(qXX,dZZ)...`

El resultado se guarda en un archivo, lo que permite analizar el rendimiento del sistema de recuperación.

5.7. Evaluación de resultados

Para la evaluación de los resultados, el archivo proporciona métodos como `precision_at_m()`, `recall_at_m()`, y `ndcg_at_m()`. Los pasos son:

El método `evaluate_queries()` evalúa el rendimiento de un sistema de recuperación de información utilizando juicios de relevancia y calcula varias métricas de evaluación, como Precision at M ($P@M$), Recall at M ($R@M$) y Normalized Discounted Cumulative Gain at M ($NDCG@M$) para cada consulta (`query`).

```
def evaluate_queries(self, relevance_filepath: str = "data/relevance-judgments/relevance-
judgments.tsv", gensim_results_filepath: str = "results/GENSIM-consultas_resultado.txt"):
    """
    Evaluates the performance of a retrieval system using relevance judgments and computes the P@M,
    R@M, and NDCG@M metrics for each query.

    Args:
        relevance_filepath (str): The path to the relevance judgments TSV file.
        gensim_results_filepath (str): The path to the Gensim results file.

    Returns:
        None
    """
    df, relevance_dict = self.load_relevance_judgments(relevance_filepath)

    gensim_results = {}
    with open(gensim_results_filepath, 'r') as f:
        for line in f:
            parts = line.strip().split()
            query = parts[0]
            docs = [doc.split(':')[0] for doc in parts[1].split(',')]
            gensim_results[query] = docs

    queries = df['query'].tolist()
    for query in queries:
        retrieved_docs = gensim_results[query]
        relevant_docs = relevance_dict.get(query)
        M = len(relevant_docs)
        p_at_m = self.precision_at_m(retrieved_docs, relevant_docs, M)
        r_at_m = self.recall_at_m(retrieved_docs, relevant_docs, M)
        ndcg_at_m_score = self.ndcg_at_m(retrieved_docs, relevant_docs, M)

        print(f"Query: {query} -> P@M: {p_at_m}, R@M: {r_at_m}, NDCG@M: {ndcg_at_m_score}")
```

Figura 5.4: Evaluación de resultados

5.7.1. Entradas

- `relevance_filepath`: La ruta al archivo TSV que contiene los juicios de relevancia de las consultas.
- `gensim_results_filepath`: La ruta al archivo de resultados generados por Gensim que contiene los documentos recuperados para cada consulta.

5.7.2. Carga de Juicios de Relevancia

- El método llama a `load_relevance_judgments()` para cargar los juicios de relevancia desde el archivo TSV.
- Esto devuelve un DataFrame que contiene las consultas y un diccionario de relevancia (`relevance_dict`) que mapea cada consulta a los documentos relevantes y sus puntuaciones de relevancia.

5.7.3. Carga de Resultados de Gensim

- Los resultados generados por Gensim se cargan desde el archivo especificado en `gensim_results_filepath`.
- Para cada línea del archivo, se extrae la consulta y la lista de documentos recuperados.
- Estos resultados se almacenan en un diccionario (`gensim_results`) donde la clave es la consulta y el valor es una lista de identificadores de documentos recuperados.

5.7.4. Evaluación de las Consultas

- Se obtiene una lista de todas las consultas desde el DataFrame de juicios de relevancia.
- Para cada consulta:

- Se recuperan los documentos devueltos por Gensim y los documentos relevantes desde el diccionario de relevancia.
- Se calcula el valor de M , que es el número de documentos relevantes para la consulta.

5.7.5. Cálculo de Métricas

Para cada consulta, se calculan las siguientes métricas:

- Precision at M ($P@M$): La precisión es la proporción de documentos relevantes dentro de los primeros M documentos recuperados. Se calcula utilizando el método `precision_at_m()`.
- Recall at M ($R@M$): El recall es la proporción de documentos relevantes recuperados entre todos los documentos relevantes. Se calcula con el método `recall_at_m()`.
- NDCG at M ($NDCG@M$): La métrica $NDCG@M$ mide la ganancia acumulativa a medida que se examinan los documentos recuperados, normalizada para tener en cuenta la relevancia. Se calcula con `ndcg_at_m()`.

5.7.6. Impresión de Resultados

Para cada consulta, se imprime lo siguiente:

Query: query $\rightarrow P@M : p_at_m, R@M : r_at_m, NDCG@M : ndcg_at_m_score$

donde se muestran los valores de las métricas calculadas para cada consulta.

```
Query: q01 -> P@M: 0.3333333333333333, R@M: 0.3333333333333333, NDCG@M: 0.3992424290497487
Query: q02 -> P@M: 0.6363636363636364, R@M: 0.6363636363636364, NDCG@M: 0.6414483775963549
Query: q03 -> P@M: 1.0, R@M: 1.0, NDCG@M: 0.9846288354276679
Query: q04 -> P@M: 0.7142857142857143, R@M: 0.7142857142857143, NDCG@M: 0.7752886293631626
Query: q06 -> P@M: 0.6666666666666666, R@M: 0.6666666666666666, NDCG@M: 0.8053569325627968
Query: q07 -> P@M: 0.25, R@M: 0.25, NDCG@M: 0.2128453970090281
Query: q08 -> P@M: 0.6666666666666666, R@M: 0.6666666666666666, NDCG@M: 0.7499297796797687
Query: q09 -> P@M: 0.8333333333333334, R@M: 0.8333333333333334, NDCG@M: 0.8878789582207093
```

Figura 5.5: Impresión de Resultados