2025

Tarea 1: Modelos de Recuperación de Información

Santiago Martínez Novoa
202112020
Diego Alejandro González Vargas
202110240

Contenido

1. Introducción

El presente informe se realiza con el fin de dar a conocer las formas de implementación de los diferentes patrones de construcción de modelos de recuperación rankeada, así como también el cálculo de las métricas de evaluación que permiten conocer la fortaleza de cada uno de estos modelos implementados. Así las cosas, a continuación, se realiza la enunciación del código de cada uno los elementos anteriormente mencionados, junto con explicaciones que permitan un mejor entendimiento del mismo. Para empezar, se presentarán las funciones simples de evaluación del modelo con la obtención de métricas. Luego, se expondrá los pasos para la construcción del índice invertido, así como los métodos de la utilización del mismo para ejecutar búsquedas binarias con operadores booleanos AND y NOT. Luego, se realizará la presentación de una implementación del cálculo de vectores TFIDF a partir del índice invertido ya conseguido. Finalmente, se presentarán las búsquedas realizadas a partir de estos vectores con similitud coseno.

2. Métricas de evaluación

2.1. Precisión

Para el cálculo de la precisión, en el caso de una lista de relevancias binarias, basta con calcular el promedio de dichas relevancias. Para ello, siguiendo con las recomendaciones dad en el enunciado, se utiliza la librería numpy para convertir la lista nativa a array de numpy y obtener el promedio de los elementos numéricos de dicho array, como sigue:

```
def precision(relevance_query:List[int]) -> float:
    """
    Calcula la precision de una consulta dada una lista de relevancia binaria
    Args:
        relevance_query (List[int]): Lista de relevancia binaria (1=relevante,
0=no relevante)
    Returns:
        float: Precision de la consulta
    """
    relevance_query = np.array(relevance_query)
    return np.mean(relevance_query)
```

2.2. Precision @ K

Para el caso de la precisión en K, en el contexto de listas de resultados con relevancia binaria, basta con extraer el arreglo de las primeras K posiciones de la lista nativa original. Una vez más, utilizando el método “mean” de la librería numpy, se obtiene el promedio del arreglo con las posiciones de interés

```
def precision_at_k(relevance_query: List[int], k: int) -> float:
```

```
    """
    Calcula la precision en el top-k de una consulta dada una lista de
relevancia binaria
    Args:
        relevance_query (List[int]): Lista de relevancia binaria (1=relevante,
0=no relevante)
        k (int): Número de elementos a considerar en el top-k
    Returns:
        float: Precision en el top-k de la consulta
    """
    if k <= 0:
        raise ValueError("k debe ser un entero positivo menor o igual al
tamaño de la lista de relevancia")

    relevance_query = np.array(relevance_query)[:k]
    return np.mean(relevance_query)
```

2.3. Recall @ K

Para el caso del recall en K, en el contexto de una lista de resultados de relevancia binaria, nuevamente se hace necesario la extracción del arreglo de los K primeros elementos de los resultados de la lista nativa original. Adicionalmente, para este caso se necesita el conocimiento del número total de documentos relevantes en la colección. Para esto, siguiendo las indicaciones dadas en el enunciado, se añade un parámetro adicional para la función, como sigue:

```
def recall_at_k(relevance_query: List[int], number_relevant_docs: int,  k:
int) -> float:
    """
    Calcula la recall en el top-k de una consulta dada una lista de relevancia
binaria
    Args:
        relevance_query (List[int]): Lista de relevancia binaria (1=relevante,
0=no relevante)
        number_relevant_docs (int): Número total de documentos relevantes
        k (int): Número de elementos a considerar en el top-k
    Returns:
        float: Recall en el top-k de la consulta
    """
    if k <= 0:
        raise ValueError("k debe ser un entero positivo menor o igual al
tamaño de la lista de relevancia")
    if number_relevant_docs <= 0:
        raise ValueError("El número de documentos relevantes debe ser un
entero positivo")
```

```
    relevance_query = np.array(relevance_query)[:k]
    return np.sum(relevance_query) / number_relevant_docs
```

2.4. Average Precision

Para poder calcular la precisión promedio de una consulta, se necesita calcular todas las precisiones de aquellos K en los que el recall cambie. En otras palabras, la precisión se calcula, en el contexto de una lista de relevancias binarias, en aquellas posiciones en las que haya un documento relevante. En este sentido, para este caso las bondades de numpy se ven reducidas, ya que se necesita hacer un recorrido tradicional del arreglo (con un ciclo for), y utilizar la función precision_at_k ya explicada para calcular las precisiones en las posiciones pertinentes. Finalmente, se calcula el número de posiciones en las que se hizo el cálculo para lo cual basta con sumar las posiciones con 1 en el vector binario. Con este número, se calcula el promedio a retornar, como sigue:

```
def average_precision(relevance_query: List[int]) -> float:
    """
    Calcula la precisión promedio de una consulta dada una lista de relevancia
binaria
    Args:
        relevance_query (List[int]): Lista de relevancia binaria (1=relevante,
0=no relevante)
    Returns:
        float: Precisión promedio de la consulta
    """
    relevance_query = np.array(relevance_query)
    cumulative_precision = 0.0
    for i in range(len(relevance_query)):
        if relevance_query[i] == 1:
            cumulative_precision += precision_at_k(relevance_query, i + 1)
    if np.sum(relevance_query) == 0:
        return 0.0
    return cumulative_precision / np.sum(relevance_query)
```

2.5. Mean Average Precision (MAP)

Para poder calcular el MAP de una serie de consultas, se hace necesario la utilización de la función descrita en el punto anterior, para cada de las listas binarias que componen la lista de entrada de la función propuesta. Una vez se tiene cada uno de estos valores, se calcula el promedio de ellos con el método correspondiente de numpy.

```
def mean_average_precision(relevance_queries: List[List[int]]) -> float:
    """
    Calcula la precisión promedio de un conjunto de consultas
```

```
    Args:
        relevance_queries (List[List[int]]): Lista de listas de relevancia
binaria
    Returns:
        float: Precisión promedio de todas las consultas
    """
    return np.mean([average_precision(query) for query in relevance_queries])
```

2.6. Discounted Cumulative Gain @ K (DCG@K)

Para poder calcular la ganancia descontada acumulada en k para una lista relevancias de los resultados de una consulta, se propuso utilizar, en primer lugar, una transformación del input de una lista nativa de python por un array de numpy. Luego de esto, se realiza una conversión con la función arange de numpy, que permite generar los índices del ranking que se necesitan para los cálculos de la sumatoria posterior. Luego de esto, se realiza el cálculo de los denominadores de la sumatoria, o mejor llamados “descuentos” a partir del uso de la función logaritmo de numpy. Finalmente, se genera la sumatoria de los valores de la lista divididos por los descuentos pertinentes, con la funcionalidad de sum de numpy, como se muestra:

```
def dcg_at_k(relevance_scores: List[int], k: int) -> float:
    """
    Calcula el Discounted Cumulative Gain (DCG) en la posición k
    Args:
        relevance_scores (List[int]): Lista de relevancia (1=relevante, 0=no
relevante)
        k (int): La posición hasta la cual calcular el DCG
    Returns:
        float: El valor del DCG en la posición k
    """
    relevance_scores = np.array(relevance_scores)[:k]
    positions = np.arange(1, len(relevance_scores) + 1)
    discounts = np.log2(np.maximum(positions, 2))
    dgc = np.sum(relevance_scores / discounts)
    return dgc
```

2.7. Normalized Discounted Cumulative Gain @ K (DCG@K)

Para el caso del cálculo normalizado de la ganancia acumulada descontada, se necesita ordenar la lista native que se recibe, para poder tener el escenario ideal que permita hacer la normalización. Para esto, se utiliza el método sort de numpy. Sin embargo, dado que este devuleve el orden de menor a mayor, se agrega el sentido deseado para recorrer el arreglo. Con este arreglo, se calculan los DCG ideales. Una vez se tienen estos valroes, junto con los valores de los DCG calculados con la function anterior, se puede realizar el cálculo del DCG normalizado con las divisiones restantes necesarias, como sigue:

```
def ndcg_at_k(relevance_scores: List[int], k: int) -> float:
    """
    Calcula el Normalized Discounted Cumulative Gain (NDCG) en la posición k
    Args:
        relevance_scores (List[int]): Lista de relevancia (1=relevante, 0=no
relevante)
        k (int): La posición hasta la cual calcular el NDCG
    Returns:
        float: El valor del NDCG en la posición k
    """
    dcg = dcg_at_k(relevance_scores, k)
    ideal_relevance = np.sort(relevance_scores)[::-1][:k]
    idcg = dcg_at_k(ideal_relevance, k)
    if idcg == 0:
        return 0.0
    return dcg / idcg
```

3. Procesamiento de los datos

Para la construcción de cada uno los modelos de búsqueda rankeada que se van a evaluar en el presente taller, se hace necesaria la realización de labores de pre procesamiento de los documentos, de manera que el texto contenido en ellos quede expresado en un formato de tokens que aporte un mayor valor a la hora de ejecutar las búsquedas, y por ende se permita tener unas métricas de evaluación de mucha mejor calidad. A continuación, se describen todas las labores realizadas para obtener los tokens limpios con los que se alimentó cada una de las implementaciones realizadas

3.1. Normalización

En primer lugar, para tener una correcta tokenización de los documentos, se hace necesaria la normalización de los textos a evaluar, dicha normalización se compone de 3 tareas principales: En primer lugar, la conversión a minúsculas de todos los caracteres alfabéticos. En segundo lugar, la eliminación de espacios dobles. Finalmente, se realiza la eliminación de las cadenas numéricas, como sigue:

```
def normalize_text(text: str) -> str:
    """
    Normaliza el texto eliminando caracteres no deseados y convirtiendo a
minúsculas.

    Args:
        text (str): El texto a normalizar.
```

```
    Returns:
        str: El texto normalizado.
    """
    text = text.strip().lower()
    text = re.sub(r"\s+", " ", text)  # Reemplaza múltiples espacios por uno
solo
    text = re.sub(r'\[\d+\]', '', text)  # Elimina referencias numéricas
    return text
```

3.2. Tokenización

En este punto, se ingresan los textos resultados de la función de normalización, para poder hacer una labor de tokenización del contenido de los textos. Esta tokenizacion consiste en pequeñsa transformación del texto recibido para poder aprovechar algunas características puntuales de la ortografía del idioma de los textos (inglés) sin perder información útil para las consultas en el camino. En este escenario, dichas transformaciones consistieron en omitir ciertos patrones o acomodar palabras para evitar la creación de tokens innecesarios: Para empezar, se van a tener en cuenta las abreviaciones como si fuesen texto tokenizable. Además, se van a considerar las palabras con guiones como elementos tokenizables. Adicionalmente, se considera también los números simples como ítems tokenizables sin importar si son decimales. Finalmente, se determinan algunos signos de puntuación como elementos tokenizables.

```
def tokenize_text(text: str) -> List[str]:
    """
    Tokeniza el texto en una lista de tokens utilizando expresiones regulares.

    Args:
        text (str): El texto a tokenizar.

    Returns:
        List[str]: La lista de tokens.
    """
    text = normalize_text(text)
    pattern = r'''(?x)
        (?:[A-Z]\.)+[A-Z]?                    # abreviaturas: U.S.A, U.S.A.
    | [A-Za-z]+(?:-[A-Za-z]+)*                # palabras con guiones
    | [A-Za-z]+(?:'[A-Za-z]+)?                # contracciones: don't, we're
    | \$?\d+(?:\.\d+)?%?                      # números simples
    | \.\.\.                                  # puntos suspensivos
    | [\[\].,;"'?():_`!-]                     # puntuación expandida
    '''
    tokens = nltk.regexp_tokenize(text, pattern)
```

```
    return tokens
```

Finalmente, se borran las palabras de parada que sean necesarias, para lo cual se usa librería con palabras de parada en el idioma de interés, en este caso, el inglés. Para esto, se realiza un recorrido simple por las palabras ya tokenizadas:

```
def remove_stopwords(tokens: List[str]) -> List[str]:
    """
    Elimina las stopwords de una lista de tokens.

    Args:
        tokens (List[str]): La lista de tokens a procesar.

    Returns:
        List[str]: La lista de tokens sin stopwords.
    """
    stop_words = set(stopwords.words("english"))
    return [token for token in tokens if token not in stop_words]
```

3.3. Stemming

Para poder abarcar un mayor número de consultas, en lugar de utilizar las palabras exactas, se usan exclusivamente las raíces de las mismas. Para esto, se realiza el stemming de los tokens con la librería pertinente. En este caso, se hace el stemming con el patrón Snowball, uno de los más robustos y fuertes en la conversión de las palabras a raíces. Una vez más, para hacer esto se realiza un recorrido simple por los tokens restantes y se retornan los resultados, como sigue:

```
def stem_tokens(tokens: List[str]) -> List[str]:
    """
    Aplica stemming a una lista de tokens.

    Args:
        tokens (List[str]): La lista de tokens a procesar.

    Returns:
        List[str]: La lista de tokens con stemming aplicado.
    """
    stemmer = SnowballStemmer("english")
    return [stemmer.stem(token) for token in tokens]
```

4. Búsqueda binaria usando Índice Invertido (BSII)

4.1. Construcción del índice invertido

Para poder realizar la construcción del índice invertido, se realiza una función que reciba por parámetro un diccionario que tiene como llaves los identificadores de los documentos, y como valores los tokens que quedaron de resultado del proceso de pre procesamiento descrito en el punto anterior, y se retorna un diccionario de diccionarios con el índice invertido ya realizado. Así pues, las llaves del diccionario principal corresponden a los tokens, mientras que los valores son diccionario que tienen a su vez, como llave la frecuencia de los tokens, y como valores los postings de estos. Ahora bien, el proceso de construcción de este índice binario es el siguiente: En primer lugar, se sacan valores únicos de cada lista de tokens con la función “set” y se agregan a una lista “global”. De manera paralela, se va construyendo la lista de postings para cada token. Para esto, antes de agregar el token a la lista global, se revisa que tenga lista de postings. En caso de que no, se crea con el docid del documento actual; en caso contrario, se realiza la adición del docid a la lista de postings conrrespondiente. Posteriormente, se realiza el ordenamiento de los tokens únicos con la función “sorted”. Finalmente, se construye el diccionario de diccionarios con la estructura propuesta para el retorno:

```
def build_inverted_index(documents: dict) -> dict:
    """
    Construye un índice invertido a partir de los documentos tokenizados.

    Args:
        documents (dict): Un diccionario con los IDs de los documentos como
claves y listas de tokens como valores.

    Returns:
        dict: Un índice invertido donde cada término tiene 'df' (document
frequency) y 'postings' (lista ordenada de doc_ids).
    """
    inverted_index = {}

    for doc_id, tokens in documents.items():
        unique_tokens = set(tokens)
        for token in unique_tokens:
            if token not in inverted_index:
                inverted_index[token] = []
            inverted_index[token].append(doc_id)

    # Convertir a formato con df y postings ordenados
    for token in inverted_index:
        postings = sorted(inverted_index[token])  # Ordenar posting list
        inverted_index[token] = {
            'docfreq': len(postings),                # Document frequency
```

```
            'postings': postings                    # Posting list ordenada
        }

    return inverted_index
```

4.2. Búsqueda binaria en el índice invertido

Para poder realizar las búsquedas binarias de cada uno de los queries, se tuvieron en cuenta 2 operadores: AND y NOT. Para el caso de AND, se realizó una función que recorre las listas de los postings de 2 tokens. Así pues, se van recorriendo paralelamente en un bucle while y se van identificando coincidencias. Dichas coincidencias se agregan al resultado, que es una lista de strings con los identificadores de los documentos:

```
def merge_and(list1: List[str], list2: List[str]) -> List[str]:
    """
    Mezcla dos listas ORDENADAS de resultados de búsqueda utilizando la
operación AND.

    Se emplean dos punteros para recorrer ambas listas de manera eficiente.

    Args:
        list1 (List[str]): La primera lista de resultados.
        list2 (List[str]): La segunda lista de resultados.

    Returns:
        List[str]: Una lista ordenada de resultados que están en ambas listas.
    """
    i, j = 0, 0
    result = []
    while i < len(list1) and j < len(list2):
        if list1[i] == list2[j]:
            result.append(list1[i])  # Documento en ambas listas
            i += 1
            j += 1
        elif list1[i] < list2[j]:
            i += 1  # Avanza en list1
        else:
            j += 1  # Avanza en list2

    return result
```

Por otra parte, para el caso del operador NOT, se realizó también un recorrido paralelo de las 2 listas con un ciclo while, en este caso agregando todos los elementos de la primera lista que no estén en la segunda, y regresando el resultado como una lista de strings:

```
def merge_not(list1: List[str], list2: List[str]) -> List[str]:
    """
```

```
    Mezcla dos listas ORDENADAS de resultados de búsqueda utilizando la
operación NOT.
    Implementa una variación del algoritmo del recorrido con dos punteros.

    Args:
        list1 (List[str]): La primera lista de resultados.
        list2 (List[str]): La segunda lista de resultados.

    Returns:
        List[str]: Una lista ordenada de los documentos que están en la
primera lista pero no en la segunda.
    """
    i, j = 0, 0
    result = []

    while i < len(list1):
        if j < len(list2) and list1[i] == list2[j]:
            # Documento está en ambas - excluir
            i += 1
            j += 1
        elif j < len(list2) and list1[i] > list2[j]:
            # Avanzar j hasta alcanzar o pasar list1[i]
            j += 1
        else:
            # list1[i] no está en list2 (o j se agotó) - incluir
            result.append(list1[i])
            i += 1

    return result
```

4.3. Evaluación de consultas y resultados

Query ID	Documentos Recuperados	Cantidad
q01	-	0
q02	d291, d293	2
q03	d105, d147, d152, d283, d291, d318	6
q04	d286	1
q06	d026, d029, d069, d257, d297, d303, d329	7
q07	d004, d034	2
q08	d108, d110, d117, d205, d251	5
q09	d198, d205, d223	3
q10	d231	1
q12	d250, d277	2
q13	-	0
q14	-	0
q16	d132, d150, d176, d184, d229, d250, d277	7
q17	d121, d271	2

q18	d192, d194, d203, d210	4
q19	d179	1
q22	-	0
q23	-	0
q24	d129, d221, d240, d282	4
q25	-	0
q26	-	0
q27	-	0
q28	d136, d174	2
q29	d037, d046, d294	3
q32	d025, d031, d090, d139, d254	5
q34	-	0
q36	-	0
q37	d169	1
q38	-	0
q40	-	0
q41	d150, d174	2
q42	-	0
q44	-	0
q45	d105	1
q46	d094, d133	2

5. Recuperación de documentos basado en vectores (RRDV)

5.1. Construcción del TF-IDF

La función emplea vectores esparsos en lugar de vectores densos para optimizar el uso de memoria y mejorar la eficiencia computacional. En un vector esparso, solo se almacenan los términos con valores TF-IDF no nulos, omitiendo los ceros implícitos del vocabulario completo.

Esta representación es especialmente ventajosa cuando el vocabulario es extenso y los documentos contienen únicamente una fracción pequeña de todos los términos posibles. Por ejemplo, un documento típico puede contener 100-1000 términos únicos de un vocabulario de 10,000+ términos.

Eficiencia:

- Memoria: O(términos_únicos_por_documento) vs O(tamaño_vocabulario_completo)
- Cálculo: Las operaciones posteriores (similaridad coseno) solo procesan términos no nulos, reduciendo significativamente las operaciones aritméticas

La equivalencia matemática entre vectores esparsos y densos se mantiene para el cálculo de similaridad coseno, ya que los términos ausentes contribuyen con 0 × 0 = 0 al producto punto.

5.2. Búsqueda de documentos por similitud coseno

Esta función aprovecha la representación esparsa para optimizar el cálculo de similaridad. El producto punto se computa únicamente sobre términos presentes en ambos vectores (intersección de vocabularios), evitando iteraciones innecesarias sobre términos ausentes que contribuirían con 0.

Para vocabularios grandes con documentos que comparten pocos términos, esta optimización reduce significativamente la complejidad computacional de O(|vocabulario_completo|) a O(|términos_comunes|).

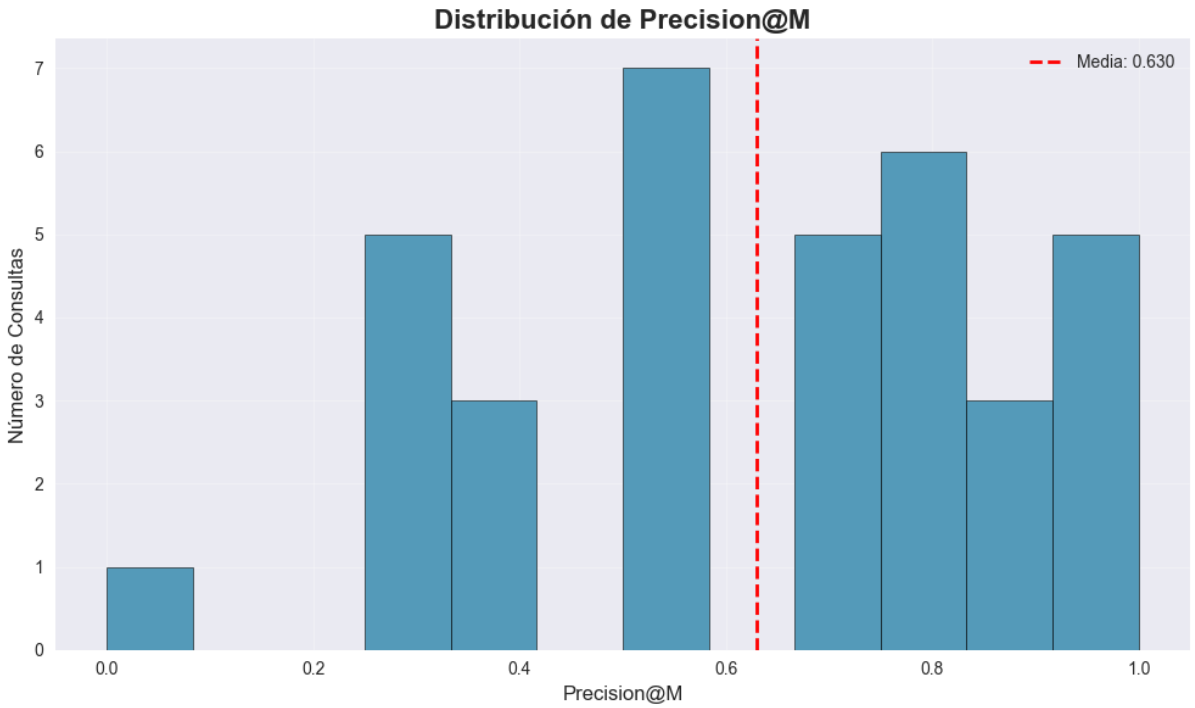
El cálculo de las normas considera todos los términos no nulos de cada vector, manteniendo la equivalencia matemática con la implementación densa.

```
def cosine_similarity_sparse(vec1: Dict[str, float], vec2: Dict[str, float]) -
> float:
    """
    Calcula la similaridad coseno entre dos vectores esparsos TF-IDF.

    Args:
        vec1, vec2: Vectores TF-IDF representados como diccionarios donde
                    las claves son términos y los valores son pesos TF-IDF

    Returns:
        float: Valor de similaridad coseno en el rango [0, 1], donde 0 indica
               vectores ortogonales (sin términos comunes) y 1 indica vectores
               idénticos en dirección
    """
    if not vec1 or not vec2:
        return 0.0

    # Términos comunes para producto punto
    common_terms = set(vec1.keys()) & set(vec2.keys())

    if not common_terms:
        return 0.0

    # Producto punto solo sobre términos comunes
    dot_product = sum(vec1[term] * vec2[term] for term in common_terms)
```

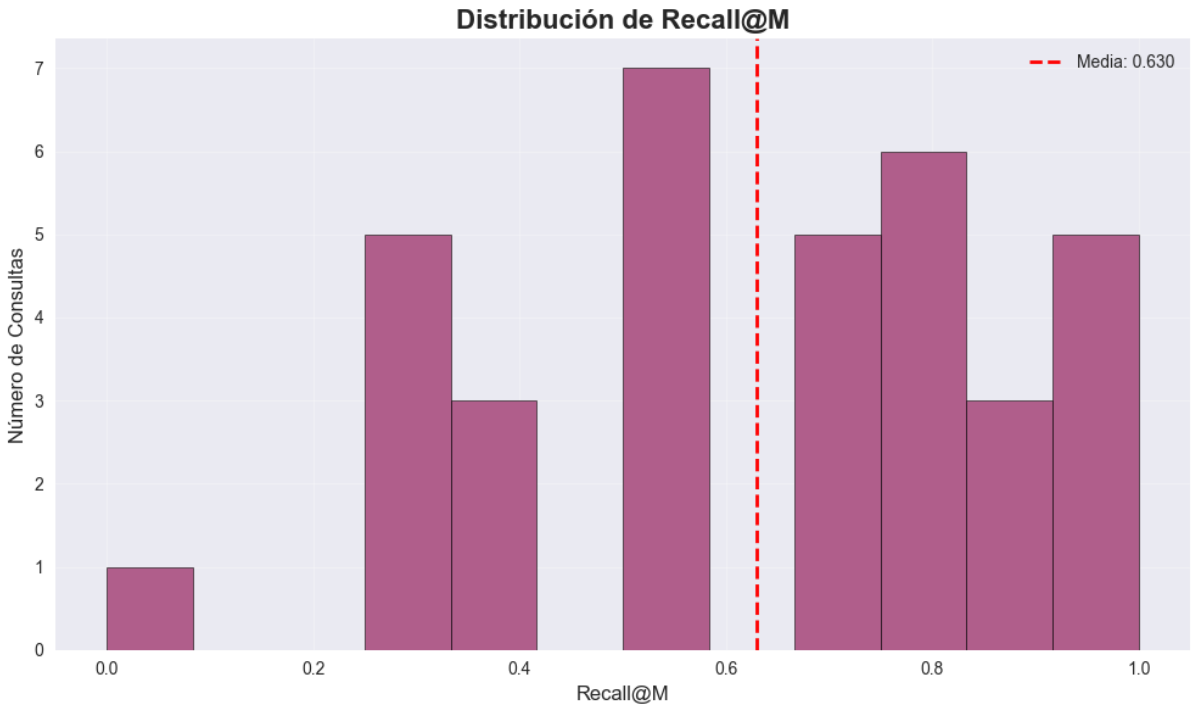
```
    # Normas euclidianas de cada vector
    norm1 = np.sqrt(sum(value ** 2 for value in vec1.values()))
    norm2 = np.sqrt(sum(value ** 2 for value in vec2.values()))

    if norm1 == 0 or norm2 == 0:
        return 0.0

    return dot_product / (norm1 * norm2)
```

5.3. Evaluación de las consultas

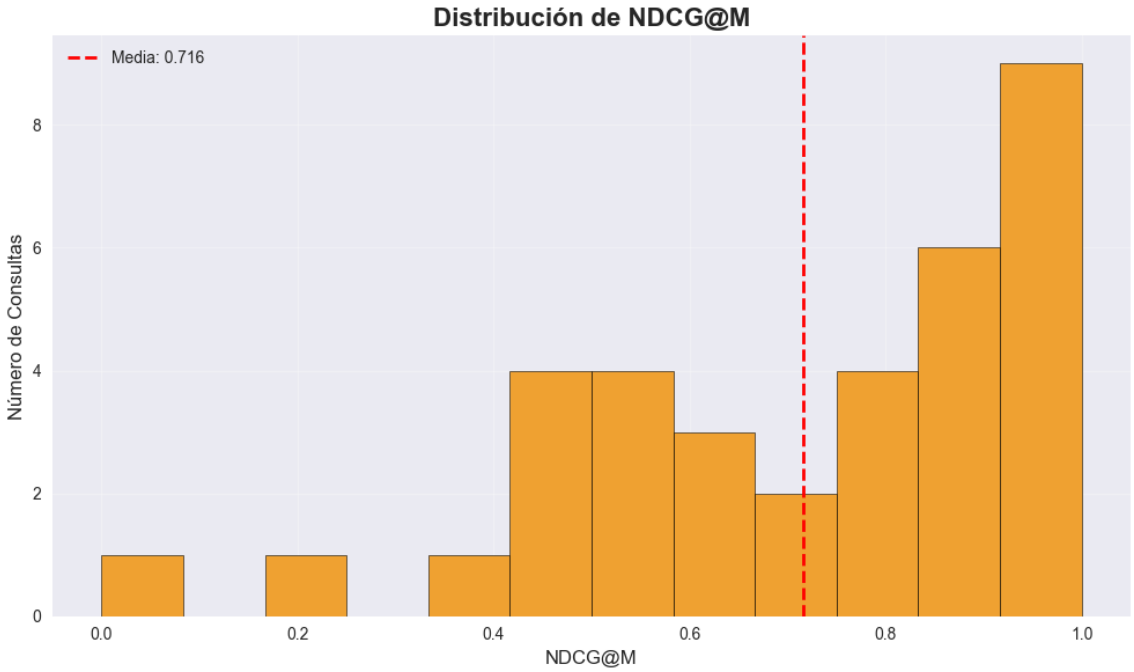
Para poder evaluar las consultas, se obtuvo, para cada una de ellas, la precisión en M, recall en M, y ndgc en M. Vale la pena mencionar que, dado que el M para la precisión y el recall se determinó en el enunciado como el número de resultados del documento de resultados ideales, los resultados de recall y precisión van a ser siempre los mismos. Esto se puede ver evidenciado en las distribuciones de los resultados de ambas métricas, como se ve a continuación



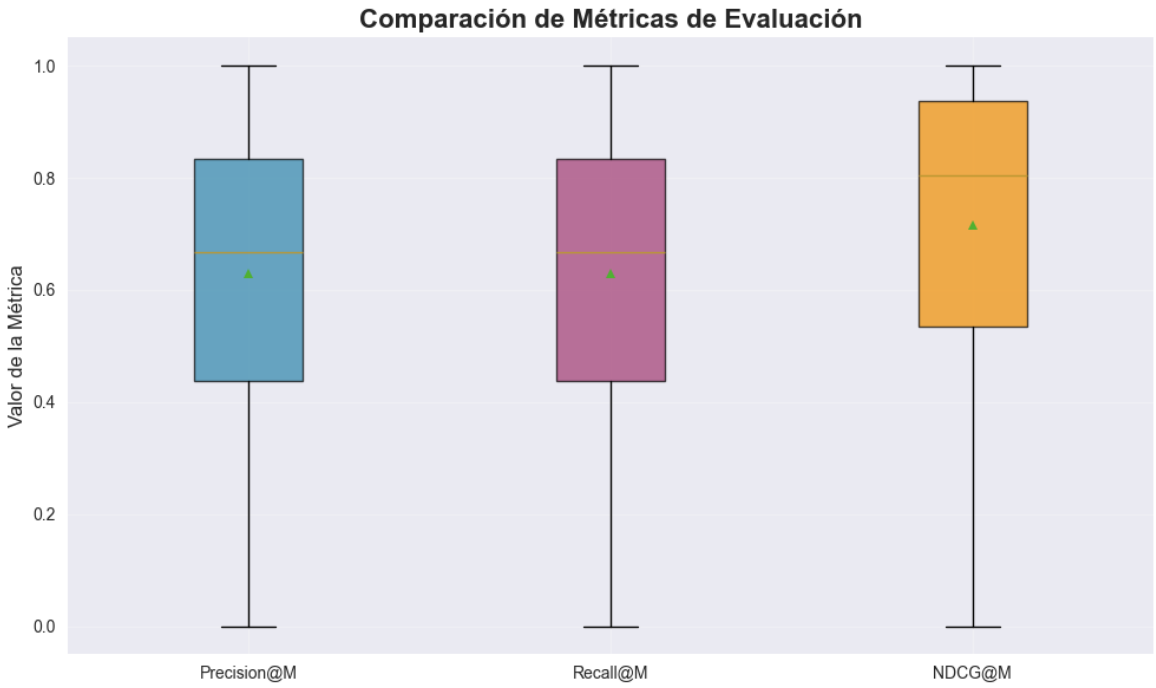
Estas gráficas describen, de las 35 consultas realizadas, cuantas se encuentran en cada slot de 0,1 de precisión y 0,1 de recall, identificando también, de acuerdo con esa distribución, donde está el promedio de dicha métrica.



Así mismo, se calcula el descuento de ganancia acumulada normalizado:



Finalmente, se comparan las 3 estadísticas con gráficas de bigotes como sigue:



MAP: 0.7175

6. RRDV con vectores de documentos usando Gensim

6.1. Construcción del TF-IDF

Para poder realizar el tfidf con Gensim, se hace necesario realizar una labor de preprocesamiento adicional, como es el caso de realizar una bolsa de palabra con los tokens ya preprocesados. Para esto, se extrae el contenido las listas de tokens con "corpora" y se reconstruyen como diccionario de lista de palabras con dictionary.doc2bow, como sigue:

```
def prepare_gensim_data(processed_documents: Dict[str, List[str]]):
    """
    Convierte documentos tokenizados al formato que espera Gensim.

    Args:
        processed_documents (Dict[str, List[str]]): Documentos procesados y
tokenizados.

    Returns:
        Tuple[corpora.Dictionary, List[List[Tuple[int, int]]], List[str]]:
Diccionario de Gensim, corpus en formato Bag-of-Words y lista de IDs de
documentos.
    """

    documents = list(processed_documents.values())
    doc_ids = list(processed_documents.keys())
```

```
    dictionary = corpora.Dictionary(documents)


    corpus = [dictionary.doc2bow(doc) for doc in documents]

    return dictionary, corpus, doc_ids
```

Posteriormente, para crear la matriz tf idf, se realiza la construcción del modelo TFIDF con la funcionalidad models.TfidfModel. Finalmente, se construye la matriz de similaridades para hacer las busquedas por distancia coseno, con el método MatrixSimilarity, todo en una misma función, como sigue:

```
def build_gensim_tfidf_system(processed_documents: Dict[str, List[str]]):
    """
    Construye sistema TF-IDF usando Gensim.
    Primero, se preparan los datos.
    Luego, se crea el modelo TF-IDF y se transforma el corpus.
    Finalmente, se crea un índice de similitud.

    Args:
        processed_documents (Dict[str, List[str]]): Documentos procesados.

    Returns:
        Tuple[Dictionary, TfidfModel, MatrixSimilarity, List[str]]: Elementos
del sistema TF-IDF.
    """
    # Preparar datos
    dictionary, corpus, doc_ids = prepare_gensim_data(processed_documents)

    # Crear modelo TF-IDF
    tfidf_model = models.TfidfModel(corpus)

    # Transformar corpus a TF-IDF
    tfidf_corpus = tfidf_model[corpus]

    # Crear índice de similitud
    similarity_index = MatrixSimilarity(tfidf_corpus)

    return dictionary, tfidf_model, similarity_index, doc_ids
```

6.2. Búsqueda de documentos por similitud coseno

Para poder realizar una búsqueda de documentos por similitud de distancias coseno, se hace necesario, al igual que en la construcción de TFIDF, la conversión de la lista de tokens al formato de bolsa de palabras de Gensim. Así mismo, se hace necesario la obtención del vector TFIDF para la query, como sigue:

```
def process_query_with_gensim(query_tokens: List[str], dictionary,
tfidf_model):
    """
    Convierte consulta a vector TF-IDF usando Gensim. Este recibe los tokens
de la consulta,
    ya procesados bajo los estándares establecidos.DirEntry

    Args:
        query_tokens (List[str]): Tokens de la consulta.
        dictionary: Diccionario Gensim.
        tfidf_model: Modelo TF-IDF Gensim.

    Returns:
        List[Tuple[int, float]]: Vector TF-IDF de la consulta.
    """
    # Convertir consulta a BOW
    query_bow = dictionary.doc2bow(query_tokens)

    # Aplicar modelo TF-IDF
    query_tfidf = tfidf_model[query_bow]

    return query_tfidf
```

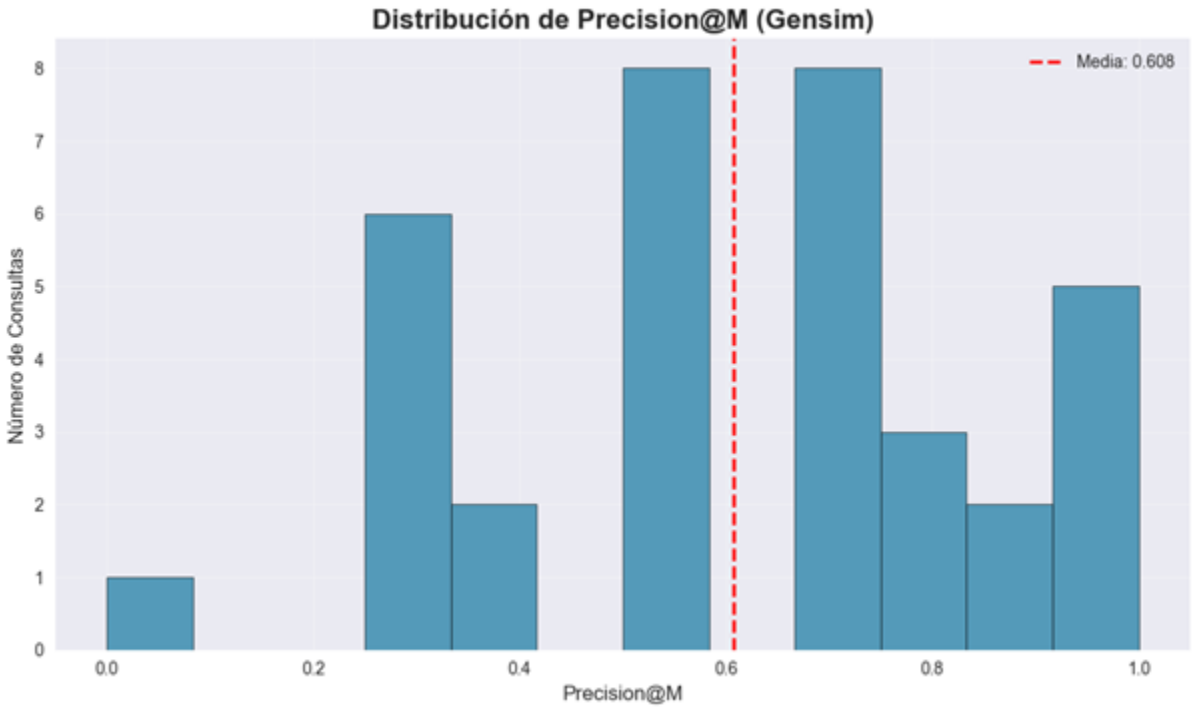
Finalmente, para el caso de Gensim, y como ya se alcanzó a esbozar en el punto anterior, se usa la funcionalidad MatrixSimilarity para obtener el índice de similaridades y realizar las búsquedas. En este caso, se rankean dichos índices para obtener las respuestas, como sigue:

```
def process_gensim_queries(
    documents_dir: str,
    queries_folder: str,
    output_file: str
):
    """
    Procesa todas las consultas usando Gensim TF-IDF
    """
    # 1. Cargar documentos
    processed_documents = load_naf_data(documents_dir)

    # 2. Construir sistema TF-IDF con Gensim
    dictionary, tfidf_model, similarity_index, doc_ids =
build_gensim_tfidf_system(processed_documents)

    # 3. Procesar consultas - QUITAR las líneas de makedirs
    with open(output_file, 'w') as f:
        query_files = sorted([file for file in os.listdir(queries_folder) if
file.endswith('.naf')])

        for query_file in query_files:
```

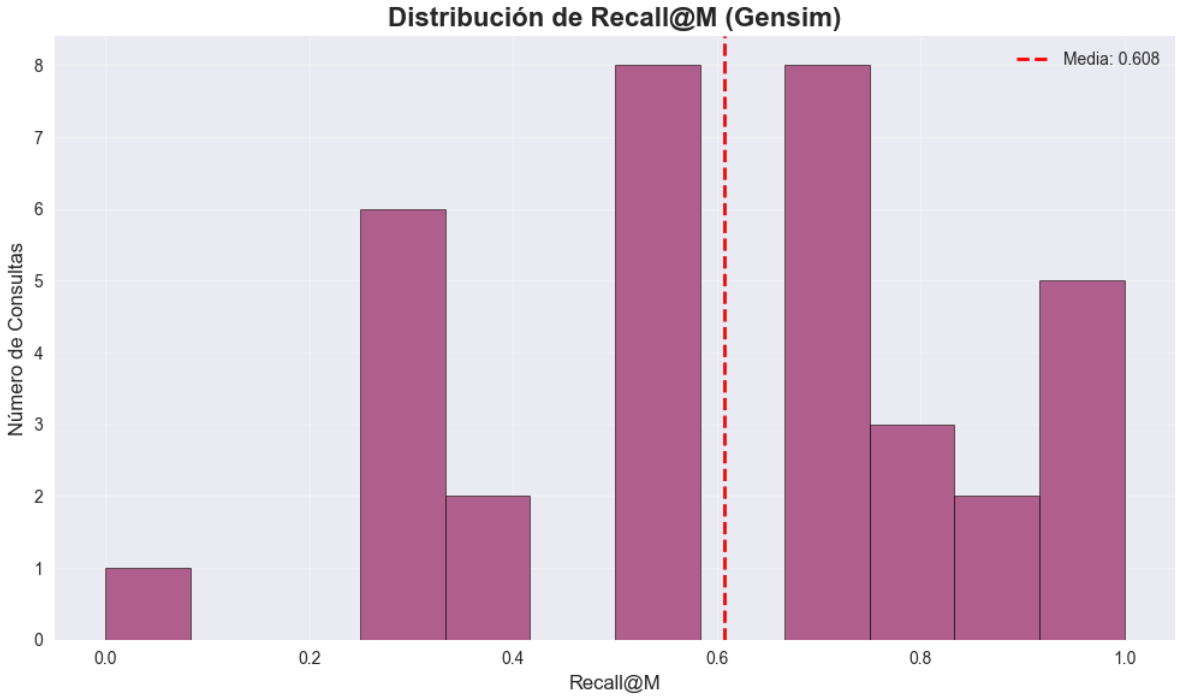
```
            # Parse consulta NAF
            tree = ET.parse(os.path.join(queries_folder, query_file))
            raw_content = tree.find('.//raw').text.strip()
            public_id = tree.find('.//public').get('publicId')

            query_tokens = preprocess_text(raw_content)

            query_tfidf = process_query_with_gensim(query_tokens, dictionary,
tfidf_model)

            similarities = similarity_index[query_tfidf]

            doc_similarities = []
            for i, sim in enumerate(similarities):
                if sim > 0:
                    doc_similarities.append((doc_ids[i], float(sim)))

            # Ordenar por similitud descendente
            doc_similarities.sort(key=lambda x: x[1], reverse=True)

            if doc_similarities:
                result_pairs = [f"{doc_id}: {sim:.4f}" for doc_id, sim in
doc_similarities]
                f.write(f"{public_id}\t{','.join(result_pairs)}\n")
            else:
                f.write(f"{public_id}\n")
```

6.3. Evaluación de las consultas y resultados

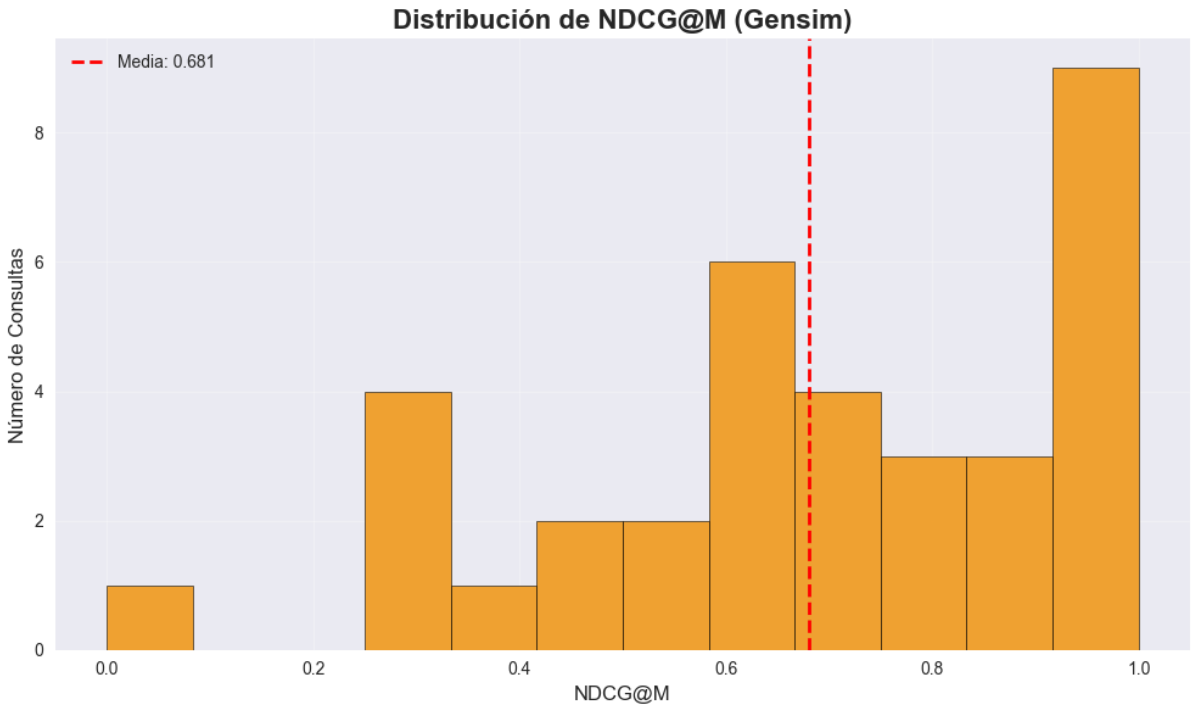
Para poder evaluar las consultas, se obtuvo, para cada una de ellas, la precisión en M, recall en M, y ndgc en M. Vale la pena mencionar que, dado que el M para la precisión y el recall se determinó en el enunciado como el número de resultados del documento de resultados ideales, los resultados de recall y precisión van a ser siempre los mismos. Esto se puede ver evidenciado en las distribuciones de los resultados de ambas métricas, como se ve a continuación



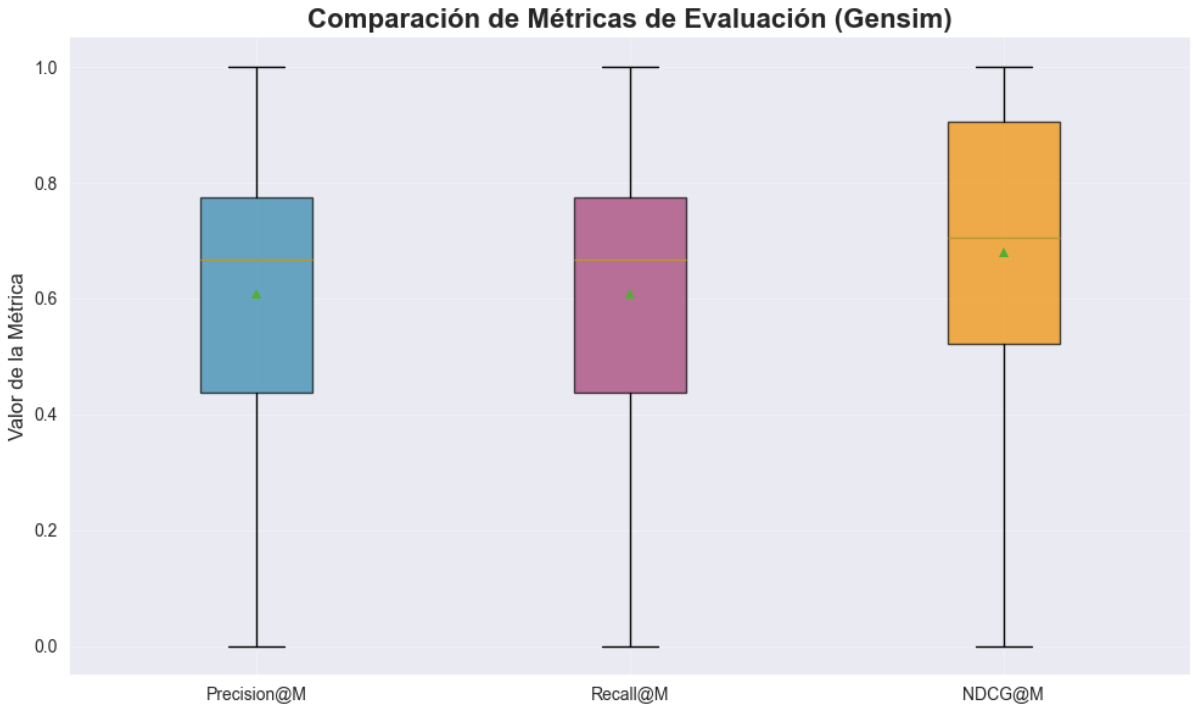
Estas gráficas describen, de las 35 consultas realizadas, cuantas se encuentran en cada slot de 0,1 de precisión y 0,1 de recall, identificando también, de acuerdo con esa distribución, donde está el promedio de dicha métrica.



Así mismo, se calcula el descuento de ganancia acumulada normalizado:



Finalmente, se comparan las 3 estadísticas con gráficas de bigotes como sigue:



MAP: 0.6760

7. Conclusiones

- En este trabajo contrastamos búsqueda binaria sobre índice invertido (BSII) y recuperación ranqueada por vectores (RRDV) bajo un *pipeline* homogéneo de normalización, tokenización con regex, eliminación de *stopwords* y *stemming*. Esta coherencia en el preprocesamiento permitió comparar con justicia los enfoques y analizar dónde gana cada uno.
- BSII cumple su propósito cuando las condiciones del enunciado son estrictas (operadores AND/NOT) y se requiere trazabilidad exacta de coincidencias. Sin embargo, su cobertura depende fuertemente de la intersección de *postings*: ante vocabularios dispersos o consultas cortas, la intersección se reduce y aparecen múltiples casos con 0 resultados; por tanto, BSII es preciso pero poco tolerante al desajuste léxico.
- RRDV (TF-IDF + coseno) aliviana esa fragilidad: al ponderar términos por idf = log(N/df) (sin suavizado) y normalizar por documento, el modelo degrada el aporte de términos frecuentes y premia coincidencias informativas, permitiendo recuperar y ordenar aun cuando la coincidencia no sea perfecta. Esto, además, habilita la exclusión operativa de documentos con puntaje 0, como pide el taller.
- La decisión de usar vectores esparsos para TF-IDF fue acertada: mantiene la equivalencia matemática con la forma densa para el coseno, pero reduce memoria y costo computacional al operar solo sobre términos presentes y, en el *dot product*, sobre la intersección de vocabularios. En colecciones con vocabularios largos y documentos relativamente cortos, esta optimización es clave.
- En métricas, con el criterio del enunciado (usar M igual al tamaño del ideal), Precision@M y Recall@M coinciden; nDCG@M aporta sensibilidad al orden dentro del *ranking*. En nuestros experimentos, el sistema vectorial propio alcanzó MAP ≈ 0.7175, mientras que la variante con Gensim obtuvo MAP ≈ 0.6760; la brecha es razonable y puede atribuirse a diferencias de tokenización/normalización y a los detalles de la ponderación interna por defecto del modelo de Gensim frente a nuestra calibración sin suavizado.
- La tokenización con regex propuesta (abreviaturas, palabras con guion, contracciones y algunos signos) funcionó bien para el corpus, pero al estar centrada en ASCII y en el apóstrofo recto ', puede omitir variantes tipográficas (’) o fenómenos lingüísticos no cubiertos —una vía clara de mejora futura.
- En conjunto, los resultados muestran que RRDV supera a BSII en calidad media de recuperación para las 35 consultas, sin perder interpretabilidad: podemos inspeccionar pesos TF-IDF, términos dominantes y contribuciones al coseno para entender por qué un documento sube o baja en el ranking.
- Líneas de mejora: (i) enriquecer el preprocesamiento (manejo de comillas tipográficas, números con separadores y lematización), (ii) explorar esquemas de TF alternativos (logarítmico/binary) y normalizaciones por longitud, (iii) probar BM25 como base

fuerte y comparador directo de TF-IDF, (iv) incorporar frases/bigramas o expansión de consultas para robustecer *recall*, y (v) evaluar sensibilidad de nDCG y MAP a variaciones de M y a distintos cortes de *top-k*.

- En síntesis, la práctica confirma que un índice invertido es la piedra angular para ambas familias, pero que la ponderación estadística del modelo vectorial es determinante para ranquear con calidad en colecciones reales: la capa de idf/normalización marca la diferencia entre “encontrar algo” y ordenar bien lo encontrado.