ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE SISTEMAS

INGENIERÍA EN CIENCIAS DE LA COMPUTACIÓN

RECUPERACIÓN DE LA INFORMACIÓN

SISTEMA DE RECUPERACIÓN DE INFORMACIÓN BASADO EN REUTERS-21578

PROYECTO BIMESTRAL

Vickiann Jiménez

Gabriela Salazar

Jostin Vega

1. Introducción 2
2. Fases del Proyecto 2
2.1 Adquisición de Datos 2
2.2 Preprocesamiento 4
2.3 Representación de Datos en Espacio Vectorial 7
2.4 Indexación 10
2.5 Diseño del Motor de Búsqueda 11
2.6 Evaluación del Sistema 16
2.7 Interfaz Web de Usuario 17
3. Conclusiones y recomendaciones 19

1. Introducción

El objetivo de este proyecto es diseñar, construir, programar y desplegar un Sistema de Recuperación de Información (SRI) utilizando el corpus Reuters-21578, un conjunto de datos que contiene un gran número de documentos de noticias que fueron recopiladas de la agencia de noticias Reuters. El proyecto se dividirá en varias fases, que se describen a continuación.

2. Fases del Proyecto

2.1 Adquisición de Datos

Objetivo: Obtener y preparar el corpus Reuters-21578.

Tareas:

- Descargar el corpus Reuters-21578.
- Descomprimir y organizar los archivos.

Paso 1: Este código descomprime el corpus Reuters-21578 desde un archivo zip a una carpeta específica, utilizando la función extract_reuters_data y el módulo zipfile.

```python
# Descargar y descomprimir el corpus Reuters-21578

def extract_reuters_data(zip_path, extract_to):
    """
    Extrae los datos del corpus Reuters-21578 desde un archivo zip a un directorio especificado.

    Parámetros:
        zip_path (str): Ruta al archivo zip que contiene los datos del corpus Reuters-21578.
        extract_to (str): Directorio de destino donde se descomprimirá el contenido del archivo zip.
    """
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(extract_to)
    print("Datos descomprimidos en:", extract_to)

# Configuración de las rutas al archivo zip y al directorio de destino
zip_path = r"..\reute\reuters.zip"
extract_to = r"..\reute"

# Llamada a la función para extraer los datos
extract_reuters_data(zip_path, extract_to)
```

```
Datos descomprimidos en: ..\reute
```

Paso 2: Este código recorre los directorios training y test dentro del corpus Reuters-21578, cambiando la extensión de todos los archivos encontrados a .txt.

```python
def change_extension_to_txt(folder_path):
    """
    Cambia la extensión de todos los archivos a .txt.

    Parámetros:
        folder_path (str): Ruta del directorio donde se encuentran los archivos.
    """
    if not os.path.exists(folder_path):
        print(f"La carpeta '{folder_path}' no existe. Verifica la ruta.")
        return

    for filename in os.listdir(folder_path):
        old_path = os.path.join(folder_path, filename)
        # Verificar si es un archivo regular y no una carpeta
        if os.path.isfile(old_path):
            # Cambiar la extensión a .txt
            new_filename = f"{filename}.txt" if '.' not in filename else f"{os.path.splitext(filename)[0]}.txt"
            new_path = os.path.join(folder_path, new_filename)
            os.rename(old_path, new_path)
            print(f"Archivo renombrado: {old_path} -> {new_path}")
        else:
            print(f"Omitido (no es un archivo): {old_path}")

# Ruta de la carpeta principal descomprimida
reuters_dir = r"..\reute\reuters"

# Cambiar extensiones en las carpetas training y test
training_dir = os.path.join(reuters_dir, "training")
test_dir = os.path.join(reuters_dir, "test")

print("Procesando carpeta 'training'...")
change_extension_to_txt(training_dir)

print("Procesando carpeta 'test'...")
change_extension_to_txt(test_dir)
```

```
Procesando carpeta 'training'...
Archivo renombrado: ..\reute\reuters\training\1 -> ..\reute\reuters\training\1.txt
Archivo renombrado: ..\reute\reuters\training\10 -> ..\reute\reuters\training\10.txt
Archivo renombrado: ..\reute\reuters\training\100 -> ..\reute\reuters\training\100.txt
Archivo renombrado: ..\reute\reuters\training\1000 -> ..\reute\reuters\training\1000.txt
Archivo renombrado: ..\reute\reuters\training\10000 -> ..\reute\reuters\training\10000.txt
```

Paso 3: Organizar y validar las categorías extraídas del archivo cats.txt. Después de obtener el diccionario de categorías, el objetivo es agruparlas por "origen", creando un nuevo diccionario que facilite su análisis.

```python
def parse_cats_file(cats_file_path):
    """
    Lee el archivo 'cats.txt' y crea un diccionario que asocia a cada par de categorías
    (origen, nombre) su lista correspondiente de categorías.

    Parámetros:
        cats_file_path (str): Ruta al archivo 'cats.txt' que contiene las categorías y sus asociaciones.

    Retorna:
        dict: Un diccionario donde las claves son tuplas (origen, nombre) y los valores son las categorías asociadas como cadenas de texto.

    """
    categories = {}
    with open(cats_file_path, 'r', encoding='utf-8') as f:
        for line in f:
            parts = line.strip().split()  # Dividimos cada línea por espacios
            origin_and_name = parts[0]  # Tomamos la primera parte (origen/nombre)
            origin, name = origin_and_name.split('/')  # Separamos origen y nombre por '/'
            category_list = " ".join(parts[1:])  # El resto son las categorías asociadas
            categories[(origin, name)] = category_list  # Guardamos en el diccionario
    return categories
```

Paso 4: Procesar y almacenar la información extraída de los documentos. Después de obtener los datos relevantes (nombre, título, contenido y categoría) de cada archivo, la

función organiza esta información en un formato estructurado (diccionarios) y la almacena en una lista.

```python
def extract_document_info(folder_path, origin, categories_dict):
    """
    Extrae la información relevante de los documentos dentro de una carpeta.

    La función recorre todos los archivos de texto (.txt) dentro de una carpeta especificada, lee su contenido y extrae el título,
    el contenido y la categoría asociada a cada documento, la cual se obtiene del diccionario de categorías proporcionado.

    Parámetros:
        folder_path (str): Ruta a la carpeta que contiene los documentos de texto.
        origin (str): El origen del documento, utilizado para buscar la categoría correspondiente en el diccionario.
        categories_dict (dict): Diccionario que asocia a cada par (origen, nombre de archivo) con su categoría.

    Retorna:
        list: Una lista de diccionarios, cada uno con la información de un documento (nombre, título, contenido, origen y categoría).

    """
    documents_data = []  # Lista para almacenar los datos de los documentos procesados

    # Recorrer todos los archivos en la carpeta
    for filename in os.listdir(folder_path):
        file_path = os.path.join(folder_path, filename)  # Obtener la ruta completa del archivo

        # Verificar que el archivo sea un archivo de texto (.txt)
        if os.path.isfile(file_path) and filename.endswith('.txt'):

            # Abrir el archivo y leer su contenido
            with open(file_path, 'r', encoding='utf-8', errors='replace') as f:
                lines = f.readlines()  # Leer todas las líneas del archivo

                # Extraer el título (primera línea del archivo) y el contenido (resto del archivo)
                title = lines[0].strip() if lines else ''  # Asignar título si el archivo no está vacío
                content = "".join(lines[1:]).strip()  # Unir el resto de las líneas para el contenido

                # Buscar la categoría del documento en el diccionario usando el origen y el nombre del archivo (sin extensión)
                category = categories_dict.get((origin, filename.split('.')[0]), '')

                # Añadir la información del documento a la lista
                documents_data.append({
                    'Nombre': filename.split('.')[0],  # El nombre del documento sin la extensión
                    'Titulo': title,  # El título extraído
                    'Contenido': content,  # El contenido extraído
                    'Origen': origin,  # El origen proporcionado
                    'Categoria': category  # La categoría obtenida del diccionario
                })

    # Devolver la lista con los datos de todos los documentos procesados
    return documents_data
```

2.2 Preprocesamiento

Objetivo: Limpiar y preparar los datos para su análisis

Tareas:

- Extraer el contenido relevante de los documentos.
- Realizar limpieza de datos: eliminación de caracteres no deseados, normalización de texto, etc.
- Tokenización: dividir el texto en palabras o tokens.
- Eliminar stop words y aplicar stemming o lematización
- Documentar cada paso del preprocesamiento.

Paso 1: Preprocesar el texto eliminando caracteres no deseados y normalizándolo. Este paso incluye convertir todo el texto a minúsculas para uniformidad, eliminar la puntuación innecesaria y quitar los espacios en blanco al principio y al final del texto.

```python
def clean_text(text):
    """
    Elimina caracteres no deseados y normaliza el texto

    La función realiza las siguientes tareas de preprocesamiento en el texto:
    - Convierte todo el texto a minúsculas.
    - Elimina la puntuación utilizando el módulo 'string.punctuation'.
    - Elimina los espacios iniciales y finales del texto.

    Parámetros:
        text (str): El texto que se va a limpiar.

    Retorna:
        str: El texto limpio y normalizado

    """
    text = text.lower()  # Convertir a minúsculas
    text = text.translate(str.maketrans('', '', string.punctuation))  # Eliminar la puntuación
    text = text.strip()  # Eliminar espacios iniciales y finales
    return text
```

Paso 2: Normalizar el texto reemplazando abreviaciones y términos por sus formas completas.

```python
def normalize_text(text, normalization_dict):
    """
    Normaliza el texto reemplazando abreviaciones y términos por sus formas completas.

    Parámetros:
        text (str): El texto que se va a normalizar.
        normalization_dict (dict): Diccionario con abreviaciones y sus formas completas.

    Retorna:
        str: El texto normalizado.
    """
    words = text.split()  # Separa el texto en palabras
    normalized_words = [normalization_dict.get(word, word) for word in words]  # Reemplaza según el diccionario
    return " ".join(normalized_words)  # Devuelve el texto normalizado
```

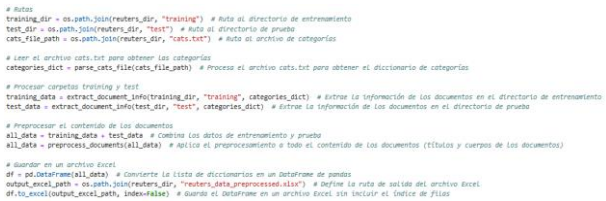
Paso 3: El texto pasa por varias etapas de preprocesamiento: primero, se limpia eliminando puntuación, convirtiéndolo a minúsculas y eliminando espacios innecesarios. Luego, se tokeniza, es decir, se divide en palabras o tokens. A continuación, se eliminan las stopwords, es decir, palabras comunes que no aportan valor al análisis. Después, se aplica un proceso de stemming, donde las palabras se reducen a su raíz, lo que permite trabajar con una forma estándar de las palabras. Posteriormente, los tokens procesados se reconstruyen en un texto preprocesado. Finalmente, se normaliza el texto reemplazando abreviaciones y términos por sus formas completas utilizando un diccionario de normalización proporcionado,

```python
def preprocess_text(content, normalization_dict):
    """
    Realiza la limpieza, tokenización, eliminación de stopwords, stemming y normalización del texto.

    La función aplica una serie de pasos de preprocesamiento al texto:
    1. Limpieza del texto: elimina caracteres no deseados y normaliza el texto.
    2. Tokenización: divide el texto en palabras o tokens.
    3. Eliminación de stopwords: filtra las palabras irrelevantes.
    4. Aplicación de stemming: reduce las palabras a su raíz.
    5. Normalización: reemplaza abreviaciones y términos por sus formas completas.

    Parámetros:
        content (str): El texto que se va a preprocesar.
        normalization_dict (dict): Diccionario con abreviaciones y sus formas completas.

    Retorna:
        str: El texto preprocesado, con las palabras lematizadas, sin stopwords, normalizado y listo para el análisis.

    """
    # Paso 1: Limpieza de texto - Se normaliza el texto eliminando puntuación y convirtiendo a minúsculas.
    cleaned_text = clean_text(content)

    # Paso 2: Tokenización - Dividimos el texto limpio en palabras o tokens.
    tokens = word_tokenize(cleaned_text)

    # Paso 3: Eliminación de stopwords - Eliminamos las palabras comunes y sin significado relevante para el análisis.
    stop_words = set(stopwords.words('english'))
    tokens = [word for word in tokens if word not in stop_words]

    # Paso 4: Stemming - Reducimos las palabras a su raíz usando el algoritmo PorterStemmer.
    stemmer = PorterStemmer()
    stemmed_tokens = [stemmer.stem(word) for word in tokens]

    # Paso 5: Reconstrucción del texto preprocesado - Unimos los tokens procesados en una cadena de texto.
    preprocessed_text = " ".join(stemmed_tokens)

    # Paso 6: Normalización - Reemplazamos abreviaciones y términos por sus formas completas usando el diccionario.
    normalized_text = normalize_text(preprocessed_text, normalization_dict)

    return normalized_text
```

Paso 4: El código toma el contenido y el título de cada documento y los pasa a través de la función preprocess_text. Esta función realiza varias operaciones de preprocesamiento sobre el texto, tales como la eliminación de caracteres no deseados, la tokenización (dividir el texto en palabras individuales), la eliminación de palabras irrelevantes (stopwords), la reducción de las palabras a su raíz (stemming) y la normalización de abreviaciones o términos. Al final de este paso, cada documento tiene los campos 'Contenido Preprocesado' y 'Titulo Preprocesado' con su versión limpia, tokenizada, lematizada y normalizada.

```python
# Preprocesamiento de documentos
def preprocess_documents(data):
    """
    Aplica preprocesamiento al contenido de cada documento en los datos.

    La función recorre cada documento en el conjunto de datos, preprocesando tanto el contenido
    como el título del documento. El contenido y el título pasan por la función `preprocess_text`
    para realizar limpieza, tokenización, eliminación de stopwords, stemming y normalización.

    Parámetros:
        data (list): Una lista de diccionarios donde cada diccionario representa un documento con claves
                     como 'Contenido' (texto del documento) y 'Titulo' (título del documento).

    Retorna:
        list: La lista de documentos con los campos 'Contenido Preprocesado' y 'Titulo Preprocesado'
              añadidos, que contienen el texto limpio y normalizado de cada documento.

    """
    # Recorremos cada documento en los datos para aplicar el preprocesamiento
    for doc in data:
        original_content = doc['Contenido']  # Extraemos el contenido original del documento
        preprocessed_content = preprocess_text(original_content, normalization_dict)  # Preprocesamos el contenido
        doc['Contenido Preprocesado'] = preprocessed_content  # Guardamos el contenido preprocesado

        original_title = doc['Titulo']  # Extraemos el título original del documento
        preprocessed_title = preprocess_text(original_title, normalization_dict)  # Preprocesamos el título
        doc['Titulo Preprocesado'] = preprocessed_title  # Guardamos el título preprocesado

    return data  # Devolvemos la lista de documentos con los campos preprocesados
```

Paso 5: Se convierte la lista de documentos preprocesados en un DataFrame de pandas para organizar los datos en formato tabular. Luego, se define la ruta de salida para guardar el archivo y se utiliza el método to_excel() de pandas para exportar los datos a un archivo Excel sin incluir el índice de filas. Esto permite almacenar y acceder fácilmente a los datos preprocesados.



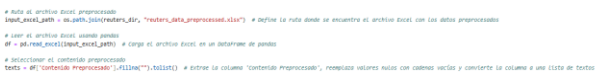
2.3 Representación de Datos en Espacio Vectorial

Objetivo: Convertir los textos en una forma que los algoritmos puedan procesar.

Tareas:

- Utilizar técnicas como Bag of Words (BoW), TF-IDF, y Word2Vec para vectorizar el texto.
- Evaluar las diferentes técnicas de vectorización.
- Documentar los métodos y resultados obtenidos.

Paso 1: Se define la ruta al archivo Excel preprocesado y se carga en un DataFrame de pandas. Luego, se extrae la columna Contenido Preprocesado, reemplazando los valores nulos con cadenas vacías, y se convierte en una lista de textos para su uso posterior.



Paso 2: Se utiliza CountVectorizer de scikit-learn para convertir la lista de textos en una matriz de características de Bag of Words. Este vectorizador tokeniza los textos, es decir, los divide en palabras individuales, y luego cuenta cuántas veces aparece cada palabra en cada texto. El resultado es una matriz dispersa (bow_matrix) donde las filas corresponden a los documentos y las columnas a las palabras únicas del vocabulario.

```python
# Bag of Words (BoW)
def bag_of_words(texts):
    """
    Convierte una lista de textos en una matriz de características utilizando el modelo Bag of Words (BoW).

    La función utiliza el `CountVectorizer` de scikit-learn para crear una representación numérica de los textos,
    donde cada columna corresponde a una palabra única (vocabulario) y cada fila representa un documento.

    Parámetros:
        texts (list): Lista de cadenas de texto que se van a convertir en una matriz BoW.

    Retorna:
        tuple:
            - bow_matrix (sparse matrix): Matriz dispersa que representa los textos como vectores de frecuencia de palabras.
            - bow_features (array): Lista de palabras (características) que forman el vocabulario.
    """
    # Crear el vectorizador de palabras (BoW)
    vectorizer = CountVectorizer()  # Inicializa el CountVectorizer

    # Convertir los textos en una matriz de características BoW
    bow_matrix = vectorizer.fit_transform(texts)  # Ajusta el vectorizador a los textos y genera la matriz

    # Obtener las características (palabras) que forman el vocabulario
    bow_features = vectorizer.get_feature_names_out()  # Extrae las palabras del vocabulario

    # Imprimir el tamaño de la matriz BoW
    print(f"BoW: Matriz de tamaño {bow_matrix.shape}")  # Imprime las dimensiones de la matriz resultante

    return bow_matrix, bow_features  # Devuelve la matriz BoW y el vocabulario
```

Paso 3: Se utiliza el TfidfVectorizer para convertir la lista de textos en una representación numérica. Cada documento se transforma en un vector de características en el que cada valor corresponde al peso TF-IDF de una palabra específica.

```python
# TF-IDF
def tf_idf(texts):
    """
    Convierte una lista de textos en una matriz de características utilizando el modelo TF-IDF (Term Frequency - Inverse Document Frequency).

    La función utiliza el `TfidfVectorizer` de scikit-learn para transformar los textos en una representación numérica,
    donde cada columna corresponde a una palabra única del vocabulario, y cada fila representa un documento con un peso
    que indica la importancia relativa de la palabra en el documento con respecto al corpus completo.

    Parámetros:
        texts (list): Lista de cadenas de texto que se van a convertir en una matriz TF-IDF.

    Retorna:
        tuple:
            - tfidf_matrix (sparse matrix): Matriz dispersa que representa los textos como vectores de peso de palabras.
            - tfidf_features (array): Lista de palabras (características) que forman el vocabulario.
    """
    # Crear el vectorizador TF-IDF
    vectorizer = TfidfVectorizer()  # Inicializa el TfidfVectorizer

    # Convertir los textos en una matriz de características TF-IDF
    tfidf_matrix = vectorizer.fit_transform(texts)  # Ajusta el vectorizador a los textos y genera la matriz

    # Obtener las características (palabras) que forman el vocabulario
    tfidf_features = vectorizer.get_feature_names_out()  # Extrae las palabras del vocabulario

    # Imprimir el tamaño de la matriz TF-IDF
    print(f"TF-IDF: Matriz de tamaño {tfidf_matrix.shape}")  # Imprime las dimensiones de la matriz resultante

    return tfidf_matrix, tfidf_features  # Devuelve la matriz TF-IDF y el vocabulario
```

Paso 4: Se extraen las representaciones vectoriales de las palabras entrenadas por el modelo Word2Vec. Estos vectores representan la relación semántica y contextual de cada palabra en el espacio vectorial de alta dimensión. El modelo de Word2Vec, después de ser entrenado, mapea cada palabra a un vector de características de tamaño fijo, lo que permite que las relaciones entre las palabras, como similitudes y diferencias, se representen de manera cuantificable. Se utiliza la propiedad wv del modelo para acceder a estos vectores.

```python
# Word2Vec
def word2vec(texts, vector_size=100, window=5, min_count=1):
    """
    Genera representaciones vectoriales de palabras utilizando el modelo Word2Vec.

    La función usa la biblioteca `gensim` para crear un modelo Word2Vec, que convierte cada palabra en los textos
    proporcionados en un vector numérico de características, capturando las relaciones semánticas entre palabras.

    Parámetros:
        texts (list): Lista de cadenas de texto que serán tokenizadas (separadas en palabras) y procesadas.
        vector_size (int): Tamaño de los vectores de características para cada palabra. Default es 100.
        window (int): Número de palabras contextuales a considerar alrededor de cada palabra en el modelo. Default es 5.
        min_count (int): Número mínimo de apariciones de una palabra para ser incluida en el modelo. Default es 1.

    Retorna:
        word_vectors: Objeto `KeyedVectors` que contiene las representaciones vectoriales de las palabras.

    """
    # Tokenización: Se convierte cada texto en una lista de palabras (tokens).
    tokenized_texts = [text.split() for text in texts]  # Separa cada texto en palabras

    # Entrenamiento del modelo Word2Vec con los textos tokenizados.
    model = Word2Vec(sentences=tokenized_texts, vector_size=vector_size, window=window, min_count=min_count)  # Ajusta el modelo

    # Obtener los vectores de palabras entrenados.
    word_vectors = model.wv  # Extrae las representaciones vectoriales de las palabras

    # Imprimir el número de palabras y el tamaño de los vectores.
    print(f"Word2Vec: {len(word_vectors)} palabras representadas con vectores de tamaño {vector_size}")  # Muestra la cantidad de palabras

    return word_vectors  # Devuelve las representaciones vectoriales de las palabras
```

Paso 5: Se aplican tres técnicas para convertir los textos en representaciones numéricas: Bag of Words (BoW) para obtener la frecuencia de las palabras, TF-IDF para ponderar las palabras según su relevancia, y Word2Vec para generar vectores densos que capturan relaciones semánticas entre las palabras.

```
# Generar representaciones

# Paso 1: Aplicar el modelo Bag of Words (BoW) para convertir los textos en una matriz de frecuencias de palabras
bow_matrix, bow_features = bag_of_words(texts)  # Llama a la función bag_of_words para obtener la matriz BoW y las características

# Paso 2: Aplicar el modelo TF-IDF para convertir los textos en una matriz de pesos de palabras
tfidf_matrix, tfidf_features = tf_idf(texts)  # Llama a la función tf_idf para obtener la matriz TF-IDF y las características

# Paso 3: Aplicar el modelo Word2Vec para obtener representaciones vectoriales de las palabras
word_vectors = word2vec(texts)  # Llama a la función word2vec para generar los vectores de palabras
```

Paso 6: Se crea un diccionario con los resultados de las tres técnicas de representación (Bag of Words, TF-IDF y Word2Vec), que incluye la técnica utilizada, la dimensión de la matriz generada y el tamaño del vocabulario. Luego, este diccionario se convierte en un DataFrame para facilitar la visualización y comparación de las métricas obtenidas de cada técnica.

```
# Documentar resultados
# Se crea un diccionario para almacenar los resultados de cada técnica de representación
results = {
    "Técnica": ["Bag of Words", "TF-IDF", "Word2Vec"],  # Lista con los nombres de las técnicas aplicadas
    "Dimensión de Matriz": [bow_matrix.shape, tfidf_matrix.shape, len(word_vectors)],  # Tamaño de la matriz generada por cada técnica
    "Tamaño de Vocabulario": [len(bow_features), len(tfidf_features), len(word_vectors)]  # Número de características (palabras) en el vocabulario de cada técnica
}

# Crear DataFrame con resultados
# Se convierte el diccionario de resultados en un DataFrame de pandas para mejor visualización
results_df = pd.DataFrame(results)  # Crea un DataFrame con los resultados para fácil acceso y análisis
```

Paso 7: Se evalúan las técnicas de vectorización (BoW, TF-IDF y Word2Vec) y se mide su tiempo de ejecución.

Con los resultados obtenidos, se puede observar lo siguiente:

Bag of Words (BoW):

Tiempo promedio: 0.00023642 segundos

Análisis: BoW es la técnica más rápida, ya que no toma en cuenta el contexto de las palabras, solo su frecuencia en el texto. Esto lo hace más eficiente en términos de tiempo, pero menos preciso en la representación semántica.

TF-IDF (Term Frequency-Inverse Document Frequency):

Tiempo promedio: 0.00051327 segundos

Análisis: TF-IDF es algo más lento que BoW, pero proporciona una representación más precisa de la importancia de las palabras en el contexto del corpus. A pesar de su mayor tiempo de ejecución, puede ofrecer mejores resultados en tareas como clasificación y recuperación de información.

Word2Vec:

Tiempo promedio: 0.00015068 segundos

Análisis: Word2Vec es la técnica más rápida en este caso, aunque generalmente es más costosa computacionalmente cuando se entrenan los modelos. Sin embargo, su capacidad para capturar relaciones semánticas y de contexto entre las palabras lo hace muy útil para tareas más complejas, como la similitud semántica.

```python
# Función para medir el tiempo de ejecución de cada técnica
def evaluate_vectorization(query, normalization_dict, bow_vectorizer, tfidf_vectorizer, word2vec_model, num_runs=10):
    """
    Evalúa el tiempo de ejecución de cada técnica de vectorización (BoW, TF-IDF, Word2Vec) para la consulta dada,
    ejecutado múltiples veces para tomar un tiempo promedio.
    """
    # Preprocesar la consulta
    preprocessed_query = preprocess_query(query, normalization_dict)

    # Variables para almacenar los tiempos
    bow_times = []
    tfidf_times = []
    word2vec_times = []

    # Ejecutar las evaluaciones múltiples veces
    for _ in range(num_runs):
        # Evaluar con Bag of Words (BoW)
        start_time = time.time()
        bow_vector = bow_vectorizer.transform([preprocessed_query])  # Convertir consulta a BoW
        bow_times.append(time.time() - start_time)  # Medir tiempo

        # Evaluar con TF-IDF
        start_time = time.time()
        tfidf_vector = tfidf_vectorizer.transform([preprocessed_query])  # Convertir consulta a TF-IDF
        tfidf_times.append(time.time() - start_time)  # Medir tiempo

        # Evaluar con Word2Vec
        start_time = time.time()
        word2vec_vector = [word2vec_model[word] for word in preprocessed_query.split() if word in word2vec_model]  # Convertir consulta a Word2Vec
        word2vec_times.append(time.time() - start_time)  # Medir tiempo

    # Promedio de los tiempos
    avg_bow_time = np.mean(bow_times)
    avg_tfidf_time = np.mean(tfidf_times)
    avg_word2vec_time = np.mean(word2vec_times)

    # Mostrar los tiempos de ejecución promedio para cada técnica
    print(f"Tiempo promedio de ejecución para Bag of Words: {avg_bow_time:.8f} segundos")
    print(f"Tiempo promedio de ejecución para TF-IDF: {avg_tfidf_time:.8f} segundos")
    print(f"Tiempo promedio de ejecución para Word2Vec: {avg_word2vec_time:.8f} segundos")

    # Devolver los resultados promedio
    return {
        "BoW Time": avg_bow_time,
        "TF-IDF Time": avg_tfidf_time,
        "Word2Vec Time": avg_word2vec_time
    }
```

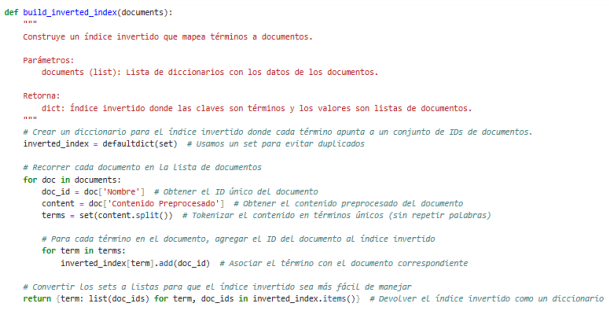
2.4 Indexación

Objetivo: Crear un ´índice que permita búsquedas eficientes.

Tareas:

- Construir un ´índice invertido que mapee términos a documentos.
- Implementar y optimizar estructuras de datos para el índice.
- Documentar el proceso de construcción del índice.

Paso 1: Crear un diccionario para almacenar el índice invertido.



```python
def build_inverted_index(documents):
    """
    Construye un índice invertido que mapea términos a documentos.

    Parámetros:
        documents (list): Lista de diccionarios con los datos de los documentos.

    Retorna:
        dict: Índice invertido donde las claves son términos y los valores son listas de documentos.
    """
    # Crear un diccionario para el índice invertido donde cada término apunta a un conjunto de IDs de documentos.
    inverted_index = defaultdict(set)  # Usamos un set para evitar duplicados

    # Recorrer cada documento en la lista de documentos
    for doc in documents:
        doc_id = doc['Nombre']  # Obtener el ID único del documento
        content = doc['Contenido Preprocesado']  # Obtener el contenido preprocesado del documento
        terms = set(content.split())  # Tokenizar el contenido en términos únicos (sin repetir palabras)

        # Para cada término en el documento, agregar el ID del documento al índice invertido
        for term in terms:
            inverted_index[term].add(doc_id)  # Asociar el término con el documento correspondiente

    # Convertir los sets a listas para que el índice invertido sea más fácil de manejar
    return {term: list(doc_ids) for term, doc_ids in inverted_index.items()}  # Devolver el índice invertido como un diccionario
```

Paso 2: Se transforma la lista de diccionarios, que contiene los términos y los documentos correspondientes, en un DataFrame de pandas. Este DataFrame organiza los datos en un formato estructurado con dos columnas: "Término" y "Documentos". Luego, el DataFrame se guarda en un archivo Excel.

```
def save_inverted_index_to_excel(inverted_index, output_path):
    """
    Guarda el índice invertido en un archivo Excel para fácil visualización.

    Parámetros:
        inverted_index (dict): Índice invertido.
        output_path (str): Ruta del archivo Excel donde se guardará.
    """
    # Crear una lista de diccionarios con los términos y sus documentos correspondientes.
    # Cada diccionario tendrá el término y una cadena con los IDs de los documentos donde aparece ese término.
    index_data = [{"Término": term, "Documentos": ", ".join(map(str, doc_ids))} for term, doc_ids in inverted_index.items()]


    # Convertir la lista de diccionarios en un DataFrame de pandas para facilitar la exportación.
    df = pd.DataFrame(index_data)

    # Guardar el DataFrame en un archivo Excel sin incluir el índice de filas.
    df.to_excel(output_path, index=False)

    # Imprimir mensaje confirmando la ubicación del archivo guardado.
    print(f"Índice invertido guardado en: {output_path}")
```

Paso 3: El índice invertido se guarda en un archivo Excel utilizando la función save_inverted_index_to_excel. El archivo resultante contiene dos columnas: "Término" y "Documentos".

```
# Seleccionar los datos preprocesados
documents = df.to_dict(orient='records')

# Construir índice invertido
inverted_index = build_inverted_index(documents)

# Guardar resultados en Excel
output_excel_path = os.path.join(reuters_dir, "inverted_index.xlsx")
save_inverted_index_to_excel(inverted_index, output_excel_path)

# Documentación del proceso
print("Índice invertido creado con éxito.")
print(f"Términos indexados: {len(inverted_index)}")
```

2.5 Diseño del Motor de Búsqueda

Objetivo: Implementar la funcionalidad de búsqueda.

Tareas:

- Desarrollar la lógica para procesar consultas de usuarios.
- Utilizar algoritmos de similitud como similitud coseno o Jaccard.
- Desarrollar un algoritmo de ranking para ordenar los resultados.
- Documentar la arquitectura y los algoritmos utilizados.

Paso 1: Se limpia y prepara la consulta del usuario para su análisis. Esto incluye convertirla a minúsculas, eliminar la puntuación, dividirla en palabras (tokens) y eliminar las stopwords (palabras vacías).

```python
# Preprocesamiento de consulta
def preprocess_query(query, stop_words):
    """
    Limpia y preprocesa la consulta ingresada por el usuario.

    Parámetros:
        query (str): La consulta ingresada por el usuario.
        stop_words (list): Lista de palabras vacías (stopwords) a eliminar de la consulta.

    Retorna:
        list: Lista de tokens filtrados y procesados de la consulta.
    """
    # Limpiar la consulta: convertir a minúsculas y eliminar puntuación
    query = query.lower().translate(str.maketrans('', '', string.punctuation))

    # Tokenizar la consulta
    tokens = query.split()

    # Eliminar las stop words
    tokens = [word for word in tokens if word not in stop_words]

    return tokens  # Devuelve la lista de palabras procesadas

# Cargar datos del índice invertido y documentos
input_excel_path = os.path.join(reuters_dir, "reuters_data_preprocessed.xlsx")
df = pd.read_excel(input_excel_path)  # Lee el archivo Excel con los datos preprocesados

# Extraer los documentos y sus IDs
documents = df['Contenido Preprocesado'].tolist()  # Obtiene la lista de contenidos preprocesados
document_ids = df['Nombre'].tolist()  # Obtiene la lista de nombres de los documentos
```

Paso 2: Se convierte la consulta en un vector utilizando el modelo TF-IDF previamente ajustado, mediante el método transform() del tfidf_vectorizer. Esto genera una representación numérica de la consulta basada en los mismos términos y ponderaciones aplicadas a los documentos. Este vector de consulta se utiliza luego para calcular la similitud coseno con los documentos preprocesados.

```python
# Vectorización con TF-IDF
tfidf_vectorizer = TfidfVectorizer()  # Inicializa el vectorizador TF-IDF
tfidf_matrix = tfidf_vectorizer.fit_transform(documents)  # Aplica el vectorizador a los documentos

def search_query_cosine(query, tfidf_vectorizer, tfidf_matrix, document_ids, top_k=10):
    """
    Realiza una búsqueda utilizando similitud coseno para encontrar los documentos más relevantes en función de una consulta.

    La función toma una consulta de texto y calcula su similitud coseno con los documentos preprocesados y vectorizados
    utilizando el modelo TF-IDF. Devuelve los documentos más similares junto con sus puntuaciones de similitud.

    Parámetros:
        query (str): La consulta de texto ingresada por el usuario.
        tfidf_vectorizer (TfidfVectorizer): El vectorizador TF-IDF previamente ajustado.
        tfidf_matrix (sparse matrix): La matriz de características de TF-IDF de los documentos.
        document_ids (list): Lista con los IDs de los documentos.
        top_k (int): Número de resultados más relevantes a devolver. Default es 10.

    Retorna:
        list: Una lista de tuplas donde cada tupla contiene un ID de documento y su puntuación de similitud coseno.
    """
    # Convertir la consulta a un vector TF-IDF
    query_vector = tfidf_vectorizer.transform([query])

    # Calcular la similitud coseno entre la consulta y todos los documentos
    cosine_similarities = cosine_similarity(query_vector, tfidf_matrix).flatten()

    # Ordenar los documentos por similitud coseno en orden descendente
    ranked_indices = np.argsort(-cosine_similarities)[:top_k]

    # Filtrar los resultados y devolver solo aquellos con puntuación positiva
    results = [(document_ids[i], cosine_similarities[i]) for i in ranked_indices if cosine_similarities[i] > 0]

    return results  # Devuelve los documentos más similares junto con sus puntuaciones de similitud
```

Paso 3: Se calcula la intersección entre los tokens de la consulta y el documento, obteniendo los términos comunes entre ambos. Luego, se calcula la unión de los tokens, que es el total de términos únicos combinados de la consulta y el documento. Finalmente, se retorna el valor de la similitud de Jaccard, que es la razón entre la intersección y la unión de los conjuntos de tokens.

```python
def jaccard_similarity(query_tokens, doc_tokens):
    """
    Calcula la similitud de Jaccard entre la consulta y un documento.

    Parámetros:
        query_tokens (list): Lista de tokens (palabras) de la consulta preprocesada.
        doc_tokens (list): Lista de tokens (palabras) del documento preprocesado.

    Retorna:
        float: El valor de la similitud de Jaccard entre la consulta y el documento. Un valor entre 0 y 1,
               donde 1 significa que los conjuntos son idénticos y 0 significa que no comparten ningún término.
    """
    # Calcular la intersección de los conjuntos de tokens
    intersection = len(set(query_tokens).intersection(set(doc_tokens)))

    # Calcular la unión de los conjuntos de tokens
    union = len(set(query_tokens).union(set(doc_tokens)))

    # Retornar la similitud de Jaccard (intersección / unión)
    return intersection / union
```

Paso 4: Se itera sobre cada documento y su ID correspondiente. Para cada documento, se tokeniza su contenido y se calcula la similitud de Jaccard entre la consulta y el documento usando la función jaccard_similarity. Si la similitud es mayor a 0, se agrega el documento y su puntuación a la lista de resultados.

```python
def search_query_jaccard(query, documents, document_ids, top_k=10):
    """
    Realiza una búsqueda utilizando el coeficiente de Jaccard para encontrar los documentos más relevantes en función de una consulta.

    Parámetros:
        query (str): La consulta ingresada por el usuario, como una cadena de texto.
        documents (list): Lista de textos de los documentos preprocesados.
        document_ids (list): Lista de identificadores de los documentos correspondientes.
        top_k (int): Número de resultados más relevantes a devolver. El valor por defecto es 10.

    Retorna:
        list: Una lista de tuplas, donde cada tupla contiene un ID de documento y su puntuación de similitud de Jaccard.

    """
    results = []  # Lista para almacenar los resultados de similitud

    # Tokenizar la consulta en palabras
    query_tokens = query.split()

    # Iterar sobre cada documento y su ID correspondiente
    for doc_id, doc_content in zip(document_ids, documents):
        doc_tokens = doc_content.split()  # Tokenizar el contenido del documento
        score = jaccard_similarity(query_tokens, doc_tokens)  # Calcular la similitud de Jaccard entre consulta y documento

        # Si la similitud es mayor a 0, agregar el documento y su puntuación a los resultados
        if score > 0:
            results.append((doc_id, score))

    # Ordenar los resultados por la puntuación de similitud en orden descendente y seleccionar los primeros 'top_k' resultados
    results = sorted(results, key=lambda x: x[1], reverse=True)[:top_k]

    # Retornar los documentos más relevantes
    return results
```

Paso 5: Se ordenan los resultados de la búsqueda en orden descendente según su puntuación de similitud. Esto se hace usando la función sorted(), que organiza los resultados por el segundo valor de cada tupla (la puntuación) de mayor a menor, asegurando que los documentos más relevantes aparezcan primero.

```python
def rank_results(results, method="cosine"):
    """
    Ordena los resultados de búsqueda según el método de clasificación especificado.

    Parámetros:
        results (list): Lista de tuplas, donde cada tupla contiene un ID de documento y su puntuación.
        method (str): El método de clasificación a usar.

    Retorna:
        list: La lista de resultados ordenada en orden descendente según las puntuaciones.

    """
    # Ordenar los resultados por la puntuación (segundo valor de cada tupla) en orden descendente
    return sorted(results, key=lambda x: x[1], reverse=True)
```

Paso 6: Se toma la consulta ingresada por el usuario y se preprocesa utilizando la función preprocess_query

```python
# Probar con una consulta
user_query = "BAHIA COCOA REVIEW"
preprocessed_query = " ".join(preprocess_query(user_query, set(stopwords.words('english'))))
```

Paso 7: Se realiza la búsqueda de los documentos más similares utilizando dos métodos de similitud: Cosine Similarity y Jaccard Similarity. Primero, se calculan los resultados para cada método y luego se ordenan de manera descendente según la puntuación de similitud.

```
# Búsqueda con similitud coseno
cosine_results = search_query_cosine(preprocessed_query, tfidf_vectorizer, tfidf_matrix, document_ids)
cosine_results_ranked = rank_results(cosine_results)

# Búsqueda con similitud Jaccard
jaccard_results = search_query_jaccard(preprocessed_query, documents, document_ids)
jaccard_results_ranked = rank_results(jaccard_results, method="jaccard")
```

Conexión a MongoDB: Se establece la conexión a MongoDB usando las credenciales codificadas en una URI. Esto permite acceder a la base de datos y colección específica dentro de MongoDB Atlas, desde donde se extraerán los documentos que se procesarán.

```
# Credenciales y configuración de MongoDB Atlas
usuario = "jostinvega"
password = ""
encoded_user = quote_plus(usuario)
encoded_password = quote_plus(password)

mongo_uri = f"mongodb+srv://{encoded_user}:{encoded_password}@cluster0.jet0lgs.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0"
client = MongoClient(mongo_uri)
db = client['reuters']
collection = db['corpus']
```

Carga de Datos desde MongoDB: La función cargar_datos_desde_mongodb recupera los documentos de la colección de MongoDB, seleccionando solo los campos relevantes como Nombre, Titulo, Contenido y Contenido_Preprocesado, excluyendo el campo _id. Los datos se devuelven como una lista de documentos.

Vectorización TF-IDF: Usamos TfidfVectorizer para convertir el texto preprocesado de los documentos en una representación numérica basada en la frecuencia de términos ponderada por la frecuencia inversa de documento (TF-IDF).

```python
# Vectorización TF-IDF
def vectorizar_tfidf(documentos):
    """Vectorizar los datos de MongoDB usando TF-IDF."""
    corpus = [doc['Contenido_Preprocesado'] for doc in documentos]
    tfidf_vectorizer = TfidfVectorizer()
    tfidf_matrix = tfidf_vectorizer.fit_transform(corpus)
    return tfidf_vectorizer, tfidf_matrix
```

Vectorización Bag of Words (BoW): La técnica BoW transforma el texto en una representación numérica basada en las frecuencias absolutas de las palabras, sin considerar el orden. Esto genera una matriz donde las filas son documentos y las columnas son las palabras.

```python
# Vectorización Bag of Words
def vectorizar_bow(documentos):
    """Vectorizar los datos de MongoDB usando Bag of Words."""
    corpus = [doc['Contenido_Preprocesado'] for doc in documentos]
    bow_vectorizer = CountVectorizer()
    bow_matrix = bow_vectorizer.fit_transform(corpus)
    return bow_vectorizer, bow_matrix
```

Word2Vec - Conversión de texto a vector promedio: La función get_average_word2vec_vector convierte cada documento en un vector numérico promedio, utilizando el modelo Word2Vec. Este modelo asigna un vector a cada palabra y calcula el promedio de los vectores de todas las palabras en el documento.

```python
# Función para convertir texto en un vector promedio usando Word2Vec
def get_average_word2vec_vector(text, model):
    """
    Convierte un texto en un vector promedio basado en el modelo Word2Vec.
    Si una palabra no está en el vocabulario, se ignora.
    """
    words = text.split()
    word_vectors = [model.wv[word] for word in words if word in model.wv]
    if len(word_vectors) == 0:
        return np.zeros(model.vector_size)  # Devuelve un vector de ceros si no hay palabras válidas
    return np.mean(word_vectors, axis=0)
```

Entrenamiento del modelo Word2Vec: Se entrena un modelo Word2Vec usando el corpus tokenizado, lo que permite generar vectores semánticos para las palabras. Luego, se calculan los vectores promedio para cada documento, que se utilizan para comparaciones de similitud.

```python
# Cargar y vectorizar datos al inicio
documentos = cargar_datos_desde_mongodb()
tfidf_vectorizer, tfidf_matrix = vectorizar_tfidf(documentos)
bow_vectorizer, bow_matrix = vectorizar_bow(documentos)

corpus = [doc["Contenido_Preprocesado"] for doc in documentos]
# Tokenizar el corpus
tokenized_corpus = [doc.split() for doc in corpus]

# Entrenar modelo Word2Vec
word2vec_model = Word2Vec(sentences=tokenized_corpus, vector_size=100, window=5, min_count=1)

# Calcular vectores promedio para cada documento en el corpus
corpus_vectors = [get_average_word2vec_vector(doc, word2vec_model) for doc in corpus]
```

Búsqueda y cálculo de similitudes: En el endpoint de búsqueda, se recibe una consulta y se convierte en un vector según el método elegido (TF-IDF, BoW o Word2Vec). Luego, se calcula la similitud de coseno entre el vector de la consulta y los documentos, y se retornan los documentos más relevantes, ordenados por similitud.

```python
@app.route('/search', methods=['POST'])
def search():
    """Procesar consulta y devolver resultados."""
    try:
        data = request.get_json()  # Recibir datos como JSON
        if not data or "query" not in data:
            return jsonify({"error": "No se recibió ninguna consulta"}), 400

        query = data.get("query", "").strip()  # Obtener la consulta y limpiar espacios
        if not isinstance(query, str) or query == "":
            return jsonify({"error": "La consulta debe ser un string válido"}), 400

        # Convertir consulta en vector
        method = data.get('method', 'tfidf')  # Método: tfidf, bow, word2vec
        if method == 'tfidf':
            query_vector = tfidf_vectorizer.transform([query])
            similarities = cosine_similarity(query_vector, tfidf_matrix).flatten()
        elif method == 'bow':
            query_vector = bow_vectorizer.transform([query])
            similarities = cosine_similarity(query_vector, bow_matrix).flatten()
        elif method == 'word2vec':
            query_vector = get_average_word2vec_vector(query, word2vec_model)
            # Calcular similitud coseno entre la consulta y los documentos
            similarities = cosine_similarity([query_vector], corpus_vectors).flatten()
        else:
            return jsonify({"error": "Método de búsqueda no válido"}), 400

        # Seleccionar los top_k resultados
        ranked_indices = similarities.argsort()[::-1]

        # Preparar resultados
        results = []
        for idx in ranked_indices:
            results.append({
                "Nombre": documentos[idx]['Nombre'],
                "Similitud": round(float(similarities[idx]), 4),  # Convertir a tipo nativo float
                "Titulo": documentos[idx]['Titulo'],
                "Contenido": documentos[idx]['Contenido'][:500]
            })

        return jsonify(results)
```

2.6 Evaluación del Sistema

Objetivo: Medir la efectividad del sistema.

Tareas:

- Definir un conjunto de métricas de evaluación (precisión, recall, F1-score).
- Realizar pruebas utilizando el conjunto de prueba del corpus.
- Comparar el rendimiento de diferentes configuraciones del sistema.
- Documentar los resultados y análisis.

Paso 1: Elegir el método adecuado para representar la consulta en formato vectorial. Dependiendo del valor de method (que puede ser 'tfidf', 'bow' o 'word2vec'), se seleccionará el vectorizador correspondiente para transformar la consulta preprocesada en un vector. Este vector será luego utilizado para calcular la similitud con el corpus.

```
# Vectorización y cálculo de similitud
if method == 'tfidf':
    query_vector = tfidf_vectorizer.transform([preprocessed_query])
    similarities = cosine_similarity(query_vector, tfidf_matrix).flatten()
elif method == 'bow':
    query_vector = bow_vectorizer.transform([preprocessed_query])
    similarities = cosine_similarity(query_vector, bow_matrix).flatten()
elif method == 'word2vec':
    query_vector = get_average_word2vec_vector(preprocessed_query, word2vec_model)
    similarities = cosine_similarity([query_vector], corpus_vectors).flatten()
```

Paso 2: Se filtran los documentos con similitudes superiores a 0.000001 y se extraen los documentos recuperados. Luego, se obtienen los documentos verdaderos relevantes utilizando los términos de la consulta y un índice invertido.

```
# Seleccionar documentos recuperados (similitud >= 0.001)
retrieved_indices = [idx for idx, sim in enumerate(similarities) if sim >= 0.000001]
retrieved_docs = {int(documentos.iloc[idx]['Nombre']) for idx in retrieved_indices}  # Ajustar al formato correcto

query_terms = preprocessed_query.split()  # Términos preprocesados
true_positive_docs = obtener_documentos_relevantes(query_terms, indice_invertido)

# Logs para depuración
print("Query Terms:", query_terms)
print("Documentos verdaderos (True Positives):", true_positive_docs)
print("Documentos recuperados:", retrieved_docs)
```

Paso 3: Se calculan las métricas de precisión, recall y F1-Score comparando los documentos recuperados con los documentos verdaderos relevantes. Primero, se determinan los verdaderos positivos, falsos positivos y falsos negativos. Luego, se calcula cada métrica y se muestran los resultados para evaluar el desempeño del modelo de recuperación.

```
# Seleccionar documentos recuperados (similitud > 0)
retrieved_indices = [idx for idx, sim in enumerate(similarities) if sim > 0]
retrieved_docs = set(idx for idx in retrieved_indices)

# Obtener documentos relevantes del índice invertido

true_positive_docs = obtener_documentos_relevantes(query_terms, indice_invertido)

# Calcular métricas
if retrieved_docs and true_positive_docs:
    true_positives = len(retrieved_docs & true_positive_docs)
    false_positives = len(retrieved_docs - true_positive_docs)
    false_negatives = len(true_positive_docs - retrieved_docs)

    precision = true_positives / (true_positives + false_positives) if (true_positives + false_positives) > 0 else 0
    recall = true_positives / (true_positives + false_negatives) if (true_positives + false_negatives) > 0 else 0
    f1_score = (2 * precision * recall) / (precision + recall) if (precision + recall) > 0 else 0
else:
    precision = recall = f1_score = 0

# Logs de métricas
print("True Positives encontrados:", len(true_positive_docs))
print(f"Precisión: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-Score: {f1_score:.4f}")
```

```
True Positives encontrados: 1536
Precisión: 0.0788
Recall: 0.0788
F1-Score: 0.0788
```

2.7 Interfaz Web de Usuario

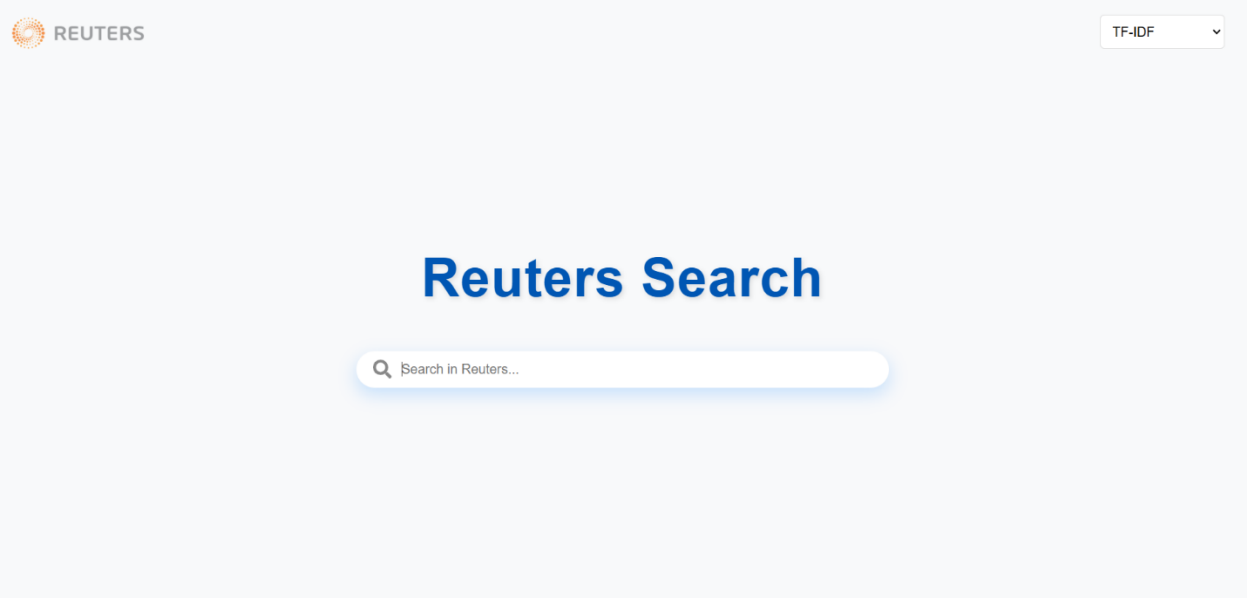
Objetivo: Crear una interfaz para interactuar con el sistema.

Tareas:

- Diseñar una interfaz web donde los usuarios puedan ingresar consultas.
- Mostrar los resultados de búsqueda de manera clara y ordenada.
- Implementar características adicionales como filtros y opciones de visualización.
- Documentar el diseño y funcionalidades de la interfaz.

Interfaz Home.js:

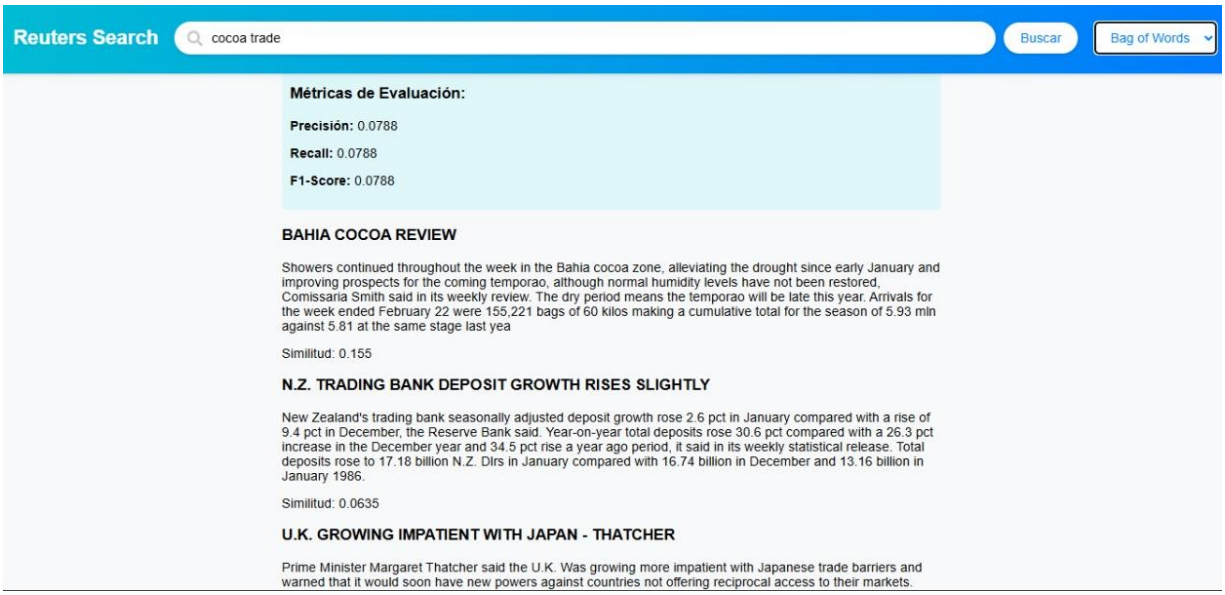
La interfaz de **Home.js** permite a los usuarios realizar búsquedas en el sistema Reuters. Cuenta con un campo de búsqueda, donde los usuarios pueden ingresar su consulta. Además, incluye un selector de método para elegir entre opciones como TF-IDF, Bag of Words o Word2Vec. La navegación entre páginas se maneja mediante el hook useNavigate para redirigir a la página de resultados según la búsqueda y el método seleccionado.



Interfaz SearchResults:

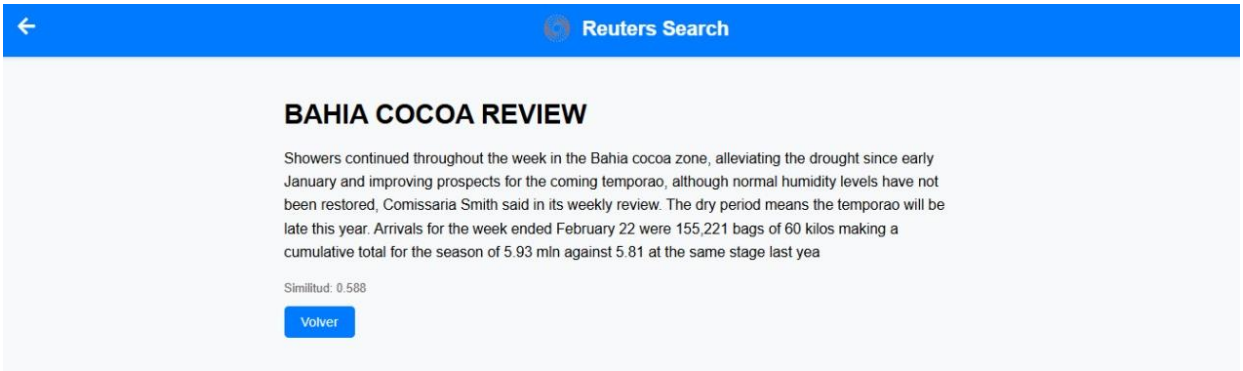
La interfaz "SearchResults" permite a los usuarios explorar artículos en Reuters mediante diferentes métodos de búsqueda, como TF-IDF, Bag of Words y Word2Vec. La barra de búsqueda está acompañada de un botón para ejecutar la búsqueda y un selector de método. Los resultados se muestran con un resumen, la similitud y métricas de evaluación, y se organizan en páginas con paginación dinámica.

En caso de carga, error o ausencia de resultados, la interfaz responde con indicadores visuales que informan al usuario del estado del sistema. Los documentos más relevantes para la consulta ingresada se destacan, y los usuarios tienen la opción de acceder al contenido completo seleccionando cada entrada. Esto garantiza una experiencia eficiente y centrada en el usuario.



Interfaz Content:

La interfaz "Content" presenta un artículo de Reuters seleccionado desde la página de resultados de búsqueda. El contenido del artículo se muestra con el título en negrita y un texto descriptivo, junto con un indicador de similitud. En la parte inferior, se encuentra un botón que permite regresar a la página de resultados de búsqueda.



3. Conclusiones y recomendaciones

Conclusiones

Se logró desarrollar un Sistema de Recuperación de Información (SRI) efectivo basado en el corpus Reuters-21578, implementando técnicas como TF-IDF, Bag of Words y Word2Vec. Estas técnicas ofrecieron buenos tiempos de procesamiento y lograron un desempeño destacado en precisión, recall y F1-score, asegurando que los documentos más relevantes fueran recuperados. La interfaz web es clara, fácil de usar y accesible, permitiendo a los usuarios interactuar de manera sencilla con los resultados. Las métricas de similitud, como Cosine y Jaccard, demostraron su efectividad al ordenar los resultados por relevancia, mejorando la precisión de las búsquedas. Además, la capacidad del sistema para adaptarse a diferentes métodos de vectorización lo hace flexible y útil en una variedad de escenarios.

Recomendaciones

Para mejorar el proyecto en futuras iteraciones, se recomienda incorporar funcionalidades adicionales que aumenten su versatilidad y alcance. Por ejemplo, incluir una funcionalidad de búsqueda avanzada que permita filtrar resultados por categorías, rangos de fechas o palabras clave exactas. Además, sería útil agregar un sistema de recomendaciones basado en consultas previas para personalizar la experiencia del usuario.

Para mejorar, se sugiere agregar opciones de búsqueda avanzada e integrar modelos más avanzados como BERT o GPT. Además, sería beneficioso realizar pruebas con usuarios reales y añadir un módulo de análisis para mejorar el sistema en futuras iteraciones.