FACULTAD DE INGENIERÍA DE SISTEMAS

CARRERA DE INGENIERÍA EN CIENCIAS DE LA COMPUTACIÓN

ICCD753 – RECUPERACIÓN DE LA INFORMACIÓN

PROYECTO BIMESTRAL

SISTEMA DE RECUPERACIÓN DE INFORMACIÓN BASADO EN REUTERS-21578

INTEGRANTES

- Erick Pérez
- Jeniffer Ichau
- Karen Guaña

DOCENTE

Prof. Iván Carrera

FECHA DE ENTREGA

19 de junio del 2024

ÍNDICE

Objetivo

- Desarrollar un sistema de recuperación de información basado en el dataset Reuters-21578 que permita la búsqueda y recuperación eficiente de documentos relevantes.

Introducción

El objetivo de este proyecto es diseñar, construir, programar y desplegar un Sistema de Recuperación de Información (SRI) utilizando el corpus Reuters-21578.

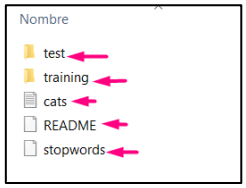
Desarrollo del proyecto

El proyecto se realizará en varias fases, cada una abordará un aspecto específico del sistema. Esto permitirá una implementación ordenada y la verificación progresiva de cada parte del sistema. A continuación, se detallan las fases del desarrollo:

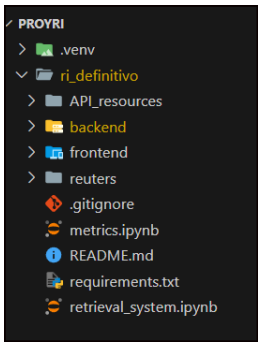
2.1 Adquisición de datos

Para dar inicio con este paso se debe descargar el corpus Reuters-21578 desde una fuente confiable que en este caso fue adquirida desde el aula virtual compartida por el ingeniero:

Una vez se haya completado la descarga se debe descomprimir, observar y comprender que contiene para poder trabajar de forma eficiente:



Luego, se inicia con la creación del proyecto que se trabajará mediante GitHub para realizar de una forma colaborativa, contiene la información del corpus como se observa en la siguiente figura:



A continuación, se adquiere del corpus la información que está en training dado que contiene la información para el entrenamiento y se procede almacenar en dos diccionarios: documents – para el texto limpio, documents_prepross – para el texto original.

```
CORPUS_DIR = "reuters/training"
documents = {}
documents_prepross = {}
```

2.2 Preprocesamiento

x

Una vez definido los pasos previamente pasamos a codificar:

Primero abrimos el archivo proporcionado con todas las stopwords a considerar y se almacenan en una variable "stop_words" para su uso posterior:

```
with open('reuters/stopwords.txt', 'r', encoding='utf-8') as file:
    stop_words = set(word.strip() for word in file.readlines())
```
✓ 0.0s

Segundo definimos una función "clean_text" en la que se realizan los pasos mencionados anteriormente por tanto eliminamos cualquier valor numérico, se convierte todo el texto a minúsculas, se separa mediante Split cada palabra, en nuestro desarrollo escogimos implementar stemming y une las palabras en stemmed_tokens en una sola cadena de texto, separadas por espacios.

```
def clean_text(*, text, stopwords):
    text = re.sub(r'\d+', '', text)
    tokens = text.lower().translate(str.maketrans('', '', string.punctuation)).split(" ")
    stemmer = SnowballStemmer("english")
    no_stw = [token for token in tokens if token not in stopwords]
    stemmed_tokens = [stemmer.stem(token) for token in no_stw]
    text_cleaned = " ".join(stemmed_tokens)
    return text_cleaned
```

Luego, está función se utilizará para la limpieza de archivos que contiene el corpus en la cual se utilizó un for que recorra y crea una lista con el corpus verificando su extensión abriendo y leyendo el contenido de cada archivo los limpia usando la función "clean_text" y por último guarda tanto el texto limpio como el texto original en dos diccionarios con el nombre del archivo como clave.

```
for filename in os.listdir(CORPUS_DIR):
    if filename.endswith(".txt"):
        filepath = os.path.join(CORPUS_DIR, filename)
        with open(filepath, 'r', encoding='utf-8') as file:
            text = file.read()
            cleaned_text = clean_text(text=text,stopwords=stop_words)
            documents[filename] = cleaned_text
            documents_prepross[filename] = text
```

Por último, se define la creación de la ruta de carpetas para almacenar información de BoW y TF-IDF para la cual se itera sobre cada ruta y se verifica si el directorio no existe para su creación.

Creacion de directorios

```
folders = ['API_resources', 'API_resources/bow', 'API_resources/tfidf']
for folder in folders:
    if not os.path.exists(folder):
        os.makedirs(folder)
```

2.3 Representación de datos en espacio vectorial

El objetivo de esta fase es convertir los textos en una forma que los algoritmos puedan procesar para lo cual se empleará la librería scikit-learn para la aplicación de BoW y Tf-Idf.

Aplicación de Bag of Words (BoW)

Esta técnica, se utiliza para convertir los textos preprocesados en matrices de frecuencia de términos. En el siguiente código se siguieron los siguientes pasos:

- Se utilizó "CountVectorizer" para convertir texto en una matriz de conteos de palabras.
- Transforma los documentos de texto en una matriz de conteos.
- Convierte la matriz de conteos en una matriz binaria usando Binarizer.
- Imprime la lista de palabras (características) encontradas y la matriz binaria resultante.

```
vectorizer_bow = CountVectorizer()
bow_counts = vectorizer_bow.fit_transform(documents.values())
onehot = Binarizer()
bow_counts = onehot.fit_transform(bow_counts.toarray())
print(vectorizer_bow.get_feature_names_out())
print(bow_counts)
```

Finalmente, guardamos los objetos para usarlos próximamente en la API con ayuda de joblib:

```
joblib.dump(vectorizer_bow, 'API_resources/bow/vectorizer_bow.joblib')
joblib.dump(bow_counts,'API_resources/bow/bow_counts.joblib')
```

Aplicación de TF-IDF

Esta técnica es útil para resaltar términos que son distintivos de un documento en particular en comparación con el resto del corpus, se empleó el siguiente código:

- Crea un TfidfVectorizer para convertir texto en una matriz TF-IDF.
- Transforma los documentos de texto en una matriz TF-IDF.
- Imprime la lista de palabras (características) encontradas.
- Imprime la matriz TF-IDF resultante, donde cada valor representa el puntaje TF-IDF de una palabra en un documento.

```
vectorizer_tfidf = TfidfVectorizer()
tfidf_counts = vectorizer_tfidf.fit_transform(documents.values())
print(vectorizer_tfidf.get_feature_names_out())
print(tfidf_counts.toarray())
```

Similar a la técnica de BoW, tanto el vectorizador como la matriz TF-IDF se guardan en archivos utilizando joblib. Estos archivos se almacenan en el directorio para su uso posterior.

```
joblib.dump(vectorizer_tfidf, 'API_resources/tfidf/vectorizer_tfidf.joblib')
joblib.dump(tfidf_counts,'API_resources/tfidf/tfidf_counts.joblib')
```

2.4 Indexación

Se emplea la biblioteca sklearn para vectorizar el corpus mediante las técnicas de Bag of Words y Tf-idf. Esta biblioteca utiliza objetos vectorizadores que se ajustan a los datos y consideran el orden de los nombres de los documentos para efectuar la vectorización, por lo que también son necesarios para la vectorización de la consulta.

```
vectorizer_bow = CountVectorizer()
bow_counts = vectorizer_bow.fit_transform(documents.values())
onehot = Binarizer()
bow_counts = onehot.fit_transform(bow_counts.toarray())
print(vectorizer_bow.get_feature_names_out())
print(bow_counts)
```

```
vectorizer_tfidf = TfidfVectorizer()
tfidf_counts = vectorizer_tfidf.fit_transform(documents.values())
print(vectorizer_tfidf.get_feature_names_out())
print(tfidf_counts.toarray())
```

Así mismo, se recurre a la biblioteca joblib para almacenar las estructuras de datos generadas por sklearn, las cuales se convierten en nuestros índices para la búsqueda y el cálculo de la similitud correspondiente a cada técnica aplicada a la consulta.

2.5 Diseño del motor de búsqueda

Para este paso se debió realizar una API con ayuda de la librería Flask para conectar el front-end y back-end implementando algoritmos de similitud en este caso aplicando el coseno y Jaccard.

Los pasos determinados a seguir fueron:

Primero, se debe cargar los archivos que previamente se almacenó en directorios con ayuda de joblib para BoW ,TF-IDF, también donde se encuentra las stop words y los documentos limpios.

```
# Carga de lo necesario para BoW
vectorizer_path_bow = os.path.join(os.path.dirname(__file__), '../API_resources/bow/vectorizer_bow.joblib')
matrix_path_bow = os.path.join(os.path.dirname(__file__), '../API_resources/bow/bow_counts.joblib')
onehot_path = os.path.join(os.path.dirname(__file__), '../API_resources/bow/onehot.joblib')

# Carga de lo necesario para TF-IDF
vectorizer_path_tfidf = os.path.join(os.path.dirname(__file__), '../API_resources/tfidf/vectorizer_tfidf.joblib')
matrix_path_tfidf = os.path.join(os.path.dirname(__file__), '../API_resources/tfidf/tfidf_counts.joblib')

# Stopwords
stpw_path = os.path.join(os.path.dirname(__file__), '../reuters/stopwords.txt')

# doc names
docs_path = os.path.join(os.path.dirname(__file__), '../API_resources/documents.joblib')
```

Se verifica que los archivos necesarios existen y si es así lo carga utilizando “joblib.load”

```python
if os.path.exists(vectorizer_path_bow) and os.path.exists(matrix_path_bow) and os.path.exists(vectorizer_path_tfidf) and os.path
    vectorizer_bow = joblib.load(vectorizer_path_bow)
    bow_loaded = joblib.load(matrix_path_bow)
    vectorizer_tfidf = joblib.load(vectorizer_path_tfidf)
    tfidf_loaded = joblib.load(matrix_path_tfidf)
    documents = joblib.load(docs_path)
    document_names = list(documents.keys())
    documents_contents = list(documents.values())
else:
    raise FileNotFoundError("Vectorizer or matrix file not found in API_resources/bow or API_resources/tfidf")
```

A continuación, convierte cada documento de BoW cargado en una matriz dispersa (csr_matrix).

```python
corpus_list = [csr_matrix(doc) for doc in bow_loaded]
```

Se realiza la función para la similitud Jaccard para calcular entre un vector de consulta y una matriz de documentos.

Los pasos fueron:

Convierte el vector de la consulta en un conjunto de índices de términos presentes.

Convierte cada documento en el corpus en un conjunto de índices de términos presentes.

Calcula la similitud de Jaccard entre la consulta y cada documento mediante su fómula.

Devuelve un arreglo Numpy con las similitudes de Jaccard para cada documento en el corpus.

```python
def jaccard_similarity(query_vector, corpus_matrix):
    # Convertir la consulta a un conjunto de índices de términos presentes
    query_set = set(query_vector.indices)

    # Convertir el corpus a un conjunto de conjuntos de índices de términos presentes
    corpus_sets = [set(doc.indices) for doc in corpus_matrix]

    jaccard_similarities = []

    for doc_set in corpus_sets:
        intersection = len(query_set & doc_set)
        union = len(query_set | doc_set)
        similarity = intersection / union if union != 0 else 0
        jaccard_similarities.append(similarity)

    return np.array(jaccard_similarities)
```

Ahora para cada proceso de query tanto para BoW y TF-IDF se definió una ruta:

Explicación para el proceso de query de TF-IDF:

Recibe una consulta en formato JSON.

Limpia y preprocesa la consulta.

Transforma la consulta en un vector TF-IDF.

Calcula las similitudes coseno entre la consulta y los documentos.

Filtra y ordena las similitudes que superan un umbral.

Devuelve las similitudes y los contenidos de los documentos relevantes en formato JSON.

Una vez definido lo que se plantea realizar pasamos a codificar obteniendo lo siguiente:

```python
@app.route('/process/tfidf/', methods=['POST'])
def process_query_tfidf():
    data = request.get_json()

    query = data['query']

    preprocessed_query = clean_text(text=query, stopwords=stop_words)
    query_vector = vectorizer_tfidf.transform([preprocessed_query])

    cosine_similarities = cosine_similarity(query_vector, tfidf_loaded).flatten()

    umbral = 0.2

    non_zero_similarities = []

    for index, similarity in enumerate(cosine_similarities):
        if similarity > umbral:
            doc_name = document_names[index]
            doc_content = documents_contents[index]
            non_zero_similarities.append((doc_name, float(similarity), doc_content))

    # Ordenar por similitud de mayor a menor
    non_zero_similarities.sort(key=lambda x: x[1], reverse=True)

    response = {
        'query': query,
        'cosine_similarities': non_zero_similarities
    }

    return jsonify(response)
```

Explicación para el proceso de query de BoW:

Recibe una consulta en formato JSON.

Limpia y preprocesa la consulta.

Transforma la consulta en un vector BoW.

Calcula las similitudes de Jaccard entre la consulta y los documentos.

Filtra y ordena las similitudes que superan un umbral.

Devuelve las similitudes y los contenidos de los documentos relevantes en formato JSON.

```python
@app.route('/process/bow/', methods=['POST'])
def process_query_bow():
    data = request.get_json()
    query = data['query']

    preprocessed_query = clean_text(text=query, stopwords=stop_words)
    query_vector = vectorizer_bow.transform([preprocessed_query])

    jaccard_similarities = jaccard_similarity(query_vector, corpus_list).flatten()

    umbral = 0.010
    non_zero_similarities = []
    for index, similarity in enumerate(jaccard_similarities):
        if similarity > umbral:
            doc_name = document_names[index]  # Asumiendo que document_names está cargadc
            doc_content = documents_contents[index]
            non_zero_similarities.append((doc_name, float(similarity), doc_content))  # C

    # Ordenar por similitud de mayor a menor
    non_zero_similarities.sort(key=lambda x: x[1], reverse=True)

    response = {
        'query': query,
        'jaccard_similarities': non_zero_similarities
    }

    return jsonify(response)
```

2.6 Evaluación del sistema

Se realizó la evaluación de las predicciones obtenidas tanto de BoW como de TF-IDF, así como de todo el sistema en conjunto.

BoW

El análisis general de los resultados muestra que el modelo presenta un rendimiento variado en términos de recall y precision. Para términos como "cocoa", "sorghum", "barley", y "corn", el modelo demuestra un buen desempeño con un balance razonable entre recall y precision, indicando que es capaz de recuperar una buena cantidad de elementos relevantes y, en su mayoría, los elementos recuperados son relevantes. Sin embargo, hay términos como "oat" que, aunque tienen un recall aceptable, presentan una precision más baja, sugiriendo que aunque se recuperan muchos elementos relevantes, también se incluyen muchos elementos no relevantes.

Por otro lado, existen términos como "rand", "nkr", y "sun-meal" que tienen tanto recall como precision de 0%, indicando que el modelo no es capaz de recuperar elementos relevantes para estos términos. Además, términos como "coconut" y "castor-oil" presentan una anomalía con un recall de 100% pero precision de 0%, lo que sugiere posibles inconsistencias en los datos o en el proceso de evaluación. Estos resultados generales indican que, aunque el modelo funciona bien para ciertas consultas, hay áreas significativas que requieren revisión y mejora, especialmente en la precisión de la recuperación de términos específicos y la corrección de inconsistencias en los datos.

	query	recall	precision
0	cocoa	65.454545	94.736842
1	sorghum	66.666667	76.190476
2	oat	75.000000	50.000000
3	barley	97.297297	76.595745
4	corn	59.116022	86.290323
...	...	...	...
85	rand	0.000000	0.000000
86	coconut	100.000000	50.000000
87	castor-oil	0.000000	0.000000
88	nkr	0.000000	0.000000
89	sun-meal	0.000000	0.000000

TF-IDF

El modelo TF-IDF mejora significativamente el rendimiento. En términos de recall, el modelo presenta un rendimiento mixto, con algunas consultas obteniendo altos valores de recuperación, como "cocoa" y "coconut" con 76.36% y 100% respectivamente, mientras que otras consultas tienen un rendimiento nulo, como "castor-oil" y "nkr". La precision también varía significativamente, con algunas consultas alcanzando una precisión del 100%, lo que indica que todos los elementos recuperados son relevantes, mientras que otras tienen una precisión muy baja, como "rand" con solo 11.76%, sugiriendo un alto número de falsos positivos.

En general, el modelo TF-IDF muestra un buen desempeño para algunas consultas específicas, indicando su capacidad para recuperar y precisar elementos relevantes en esos casos.

	query	recall	precision
0	cocoa	76.363636	100.000000
1	sorghum	20.833333	83.333333
2	oat	50.000000	57.142857
3	barley	62.162162	85.185185
4	corn	41.988950	95.000000
...	...	...	...
85	rand	100.000000	11.764706
86	coconut	100.000000	80.000000
87	castor-oil	0.000000	0.000000
88	nkr	0.000000	0.000000
89	sun-meal	0.000000	0.000000

Promedio

Análisis Comparativo:

Recall: El modelo BoW tiene un recall significativamente mayor (38.55%) en comparación con TF-IDF (25.14%). Esto sugiere que BoW es mejor para recuperar una mayor cantidad de elementos relevantes, aunque puede incluir más elementos irrelevantes.

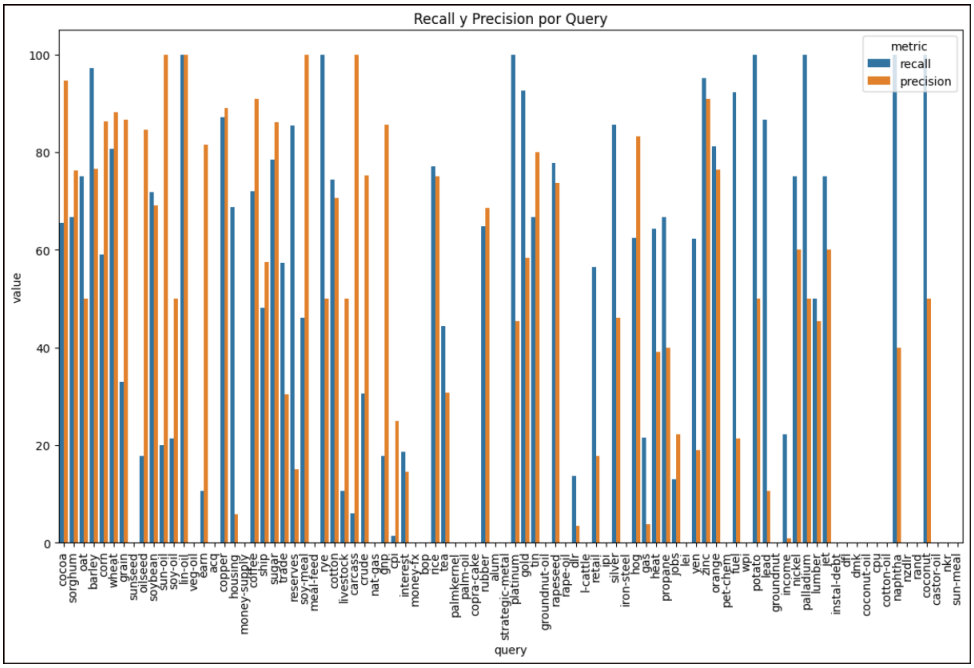
Precision: El modelo TF-IDF supera a BoW en precisión (41.31% frente a 36.14%). Esto indica que los elementos recuperados por TF-IDF son más propensos a ser relevantes, aunque puede recuperar menos elementos en general.

	Recall	Precision
BoW	38.551323	36.144797
TF-IDF	25.144300	41.305972

BoW es más efectivo en la recuperación de elementos relevantes en general, pero con un mayor riesgo de incluir elementos irrelevantes.

TF-IDF es más preciso en la recuperación de elementos relevantes, lo que significa que, aunque recupera menos elementos, aquellos que recupera son más probables de ser relevantes.

Representación gráfica



2.7 Interfaz web de usuario

La aplicación "InfoFinder" está construida utilizando tecnologías web como HTML, CSS y JavaScript.

La estructura de la página se creó con HTML, mientras que CSS se empleó para estilizar la interfaz y darle un diseño visual atractivo. JavaScript se utilizó para añadir interactividad y gestionar las búsquedas mediante la conexión con la API, permitiendo que la aplicación enviara consultas y recibiera resultados de búsqueda.

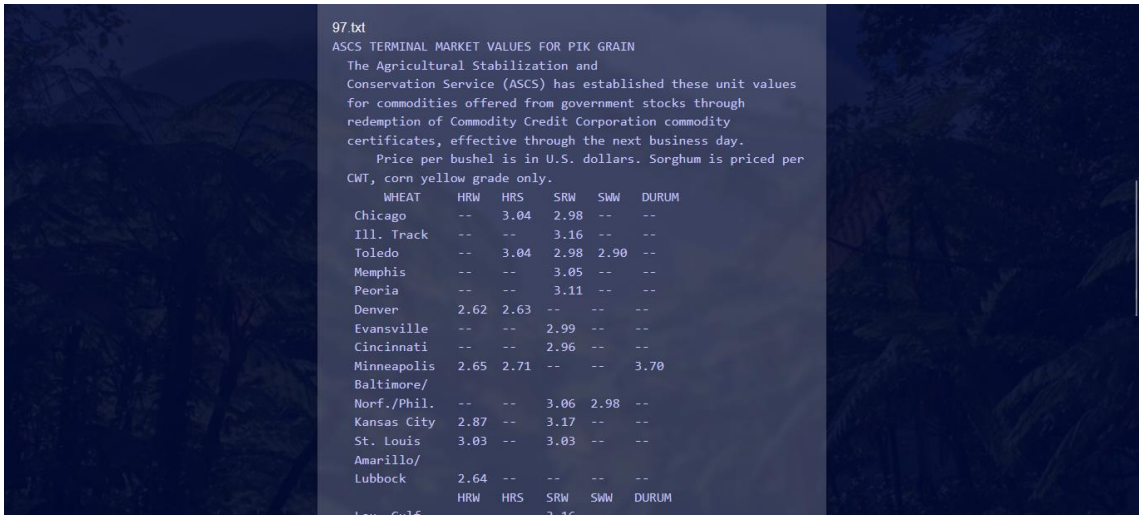
Funcionalidades de la Aplicación

1. **Barra de Búsqueda:** Los usuarios pueden ingresar sus consultas de búsqueda en el campo designado.
2. **Botón de Búsqueda:** Inicia el proceso de búsqueda cuando se hace clic en el ícono de la lupa, ejecutando el algoritmo seleccionado para encontrar resultados relevantes.
3. **Métodos de Búsqueda:** La aplicación ofrece dos métodos diferentes para realizar búsquedas. Los métodos disponibles son:
   - **TF-IDF Coseno:** Utiliza la técnica de "Term Frequency-Inverse Document Frequency" combinada con la similitud coseno para buscar documentos relevantes. Este método ayuda a encontrar documentos similares basándose en la frecuencia de términos.

- **BoW Jaccard:** Utiliza el enfoque de "Bag of Words" junto con el índice de Jaccard para las búsquedas. Este método es útil para comparar documentos basándose en la presencia o ausencia de palabras sin tener en cuenta el orden.

Ejemplo de uso

El usuario ingresa la consulta que desea en la barra de búsqueda, por ejemplo, "rye". Inicialmente, el botón de buscar se encuentra deshabilitado. El usuario debe seleccionar un método de búsqueda, como "TF-IDF Coseno" o "BoW Jaccard". Al seleccionar el método de búsqueda, el botón de buscar se habilita. Una vez hecha la búsqueda, dentro de la misma página, se visualiza un cuadro con los resultados de la búsqueda, mostrando una lista de documentos relevantes, tal como se ve en la imagen. Los documentos se enumeran con sus identificadores, permitiendo al usuario identificar rápidamente los archivos que contienen la información relacionada con su consulta, además, se muestra el contenido de cada documento, proporcionando más contexto para cada resultado.



Conclusión

En conclusión, se logró implementar el sistema de recuperación de información basado en el corpus Reuters-21578, realizando el preprocesamiento del corpus, aplicando técnicas de vectorización y diseñando el motor de búsqueda mediante algoritmos de similitud de coseno y Jaccard. Las métricas de evaluación utilizadas permitieron observar la efectividad del sistema, mostrando que es capaz de realizar búsquedas y recuperaciones de documentos relevantes de manera eficiente, estableciendo una base sólida para futuras mejoras y optimizaciones.