ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE SISTEMAS

Estudiantes: Danna Zaldumbide y Mishel Ramirez

Fecha: 09/06/2025

Informe - Técnico - Sistema de Recuperación de Información

1. Descripción del Corpus Utilizado

Se utilizó un corpus del dominio de videojuegos obtenido desde la biblioteca ir_datasets, específicamente del subset gaming. Este corpus contiene documentos con campos clave como _id (identificador) y text (contenido textual). Los documentos fueron cargados desde un archivo corpus.jsonl alojado en Google Drive y transformados en un DataFrame de Pandas para facilitar su manipulación posterior.

2. Decisiones de Diseño

2.1. Plataforma y entorno de ejecución

El sistema fue desarrollado en Google Colab para aprovechar su integración con Google Drive, su soporte nativo para notebooks y su entorno basado en Python. Esto facilita la experimentación, colaboración y replicabilidad del experimento.

2.2. Preprocesamiento del Texto

Para la representación ordenada del contenido, se diseñó un módulo de preprocesamiento completamente personalizado, almacenado en un archivo separado llamado preprocessing.py.

Este módulo utiliza bibliotecas estándar de NLP como NLTK, contractions y string, y realiza lo siguiente:

1. Expansión de contracciones: Reemplaza formas como "don't" por "do not".
2. Tokenización: Separa el texto en tokens usando nltk.word_tokenize.
3. Normalización: Convierte todos los tokens a minúsculas, elimina puntuación y símbolos especiales estándar del inglés.
4. Eliminación de stopwords: Quita palabras funcionales sin carga semántica como "the", "and", etc... usando la lista de stopwords de NLTK.
5. Stemming: Aplica el algoritmo de Porter para reducir palabras a su raíz morfológica.
6. Lematización: Reduce las palabras a su forma base válida en el diccionario con WordNetLemmatizer de NLTK

```
    # ====================== #
    #   1. TOKENIZACIÓN      #
    # ====================== #
    tokens = word_tokenize(doc)
    tokenized.append(tokens)

    # ========================== #
    #   2. NORMALIZACIÓN         #
    # ========================== #
    normalized_tokens = [
        token.lower() for token in tokens
        if token.isalpha() and token not in special_chars
    ]
    normalized.append(normalized_tokens)

    # ========================== #
    #   3. STOPWORDS FILTER      #
    # ========================== #
    filtered_tokens = [
        token for token in normalized_tokens if token not in stop_words
    ]
    filtered.append(filtered_tokens)

    # ========================== #
    #   4. STEMMING              #
    # ========================== #
    stemmed_tokens = [stemmer.stem(token) for token in filtered_tokens]
    stemmed.append(stemmed_tokens)

    # ========================== #
    #   5. LEMMATIZACIÓN         #
    # ========================== #
    lemmatized_tokens = [lemmatizer.lemmatize(token) for token in filtered_tokens]
    lemmatized.append(lemmatized_tokens)

if return_all:
    return tokenized, normalized, filtered, stemmed, lemmatized
```

Ilustración 1. Preprocesamiento

```
for doc in corpus:
    doc = doc.replace("\n", " ").replace("\r", " ").strip()
    doc = contractions.fix(doc)
```

Ilustración 2. Reemplazo de contracciones

La función preprocess_dataframe permite aplicar estas operaciones directamente sobre un DataFrame con una columna 'raw' de texto.

```
# Función para aplicar el preprocesamiento sobre un DataFrame
def preprocess_dataframe(df, lang='english', return_all=False):
    processed_docs = preprocess(df['raw'], lang=lang, return_all=return_all)

    if return_all:
        df[['tokenized', 'normalized', 'filtered', 'stemmed', 'lemmatized']] = pd.DataFrame(processed_docs).T
    else:
        df['stemmed'] = processed_docs[0]
        df['lemmatized'] = processed_docs[1]

    return df
```

Ilustración 3. Vista de preprocess_dataframe

Esto resulta en un DataFrame extendido con las siguientes columnas:

- tokenized: tokens originales.
- normalized: tokens normalizados y filtrados.
- filtered: sin stopwords.
- stemmed: tokens con stemming.
- lemmatized: tokens lematizados.

3. Construcción del Índice Invertido

Una vez realizado el preprocesamiento del corpus se construye un índice invertido. Este índice permite acceder rápidamente a los documentos que contienen un término determinado, lo cual mejora significativamente el rendimiento en la fase de búsqueda.

3.1. Representación del índice

El índice invertido se implementó utilizando un diccionario anidado (defaultdict(dict)), donde:

- Las claves principales son los términos del vocabulario (ya lematizados).
- Cada valor asociado es un subdiccionario que representa la posting list, es decir, los documentos en los que aparece el término junto con su frecuencia.

3.2. Construcción del índice

Para construir el índice, se recorrieron los documentos del corpus lematizado y se calculó la frecuencia de cada término en cada documento. Este conteo se hizo de forma independiente para cada documento, y los resultados fueron agregados al índice invertido.

```
# Crear índice invertido
inverted_index = defaultdict(dict)

# Recorrer documentos lematizados
for doc_id, tokens in enumerate(processed_df['lemmatized']):

    # Contar la frecuencia de cada término en el documento actual
    term_freq = defaultdict(int)
    for token in tokens:
        term_freq[token] += 1

    # Agregar las frecuencias de los términos al índice invertido
    for term, freq in term_freq.items():
        inverted_index[term][doc_id] = freq
```

Ilustración 4. Inverted_index

```
# Imprimir las primeras 20 entradas del índice invertido
for i, term in enumerate(sorted(inverted_index.keys())):
    if i >= 20:  # Imprimir solo los 20 primeros
        break
    print(f"Término: {term}")
    print(f"Posting list: {inverted_index[term]}")
    print("---")
```

Ilustración 5. Impresión 20 primeros resultados

4. Implementación del Modelo de Recuperación

Con el índice invertido y los documentos preprocesados, se implementaron dos modelos de recuperación de información: TF-IDF con similitud de coseno y BM25

4.1. Modelo Vectorial con TF-IDF

El modelo TF- pondera los términos de cada documento tomando en cuenta su frecuencia local y su distribución global en el corpus. Para aplicar esto se hizo:

1. Unificación del texto lematizado: Se concatenaron los tokens lematizados en una sola cadena de texto por documento.
2. Vectorización del corpus: Se utilizó la clase TfidfVectorizer de scikit-learn para transformar el corpus lematizado en una matriz TF-IDF.

```
# Unir tokens lematizados en strings
processed_df['lemmatized_text'] = processed_df['lemmatized'].apply(lambda x: " ".join(x))

# Crear matriz TF-IDF
vectorizer = TfidfVectorizer()
corpus_tfidf = vectorizer.fit_transform(processed_df['lemmatized_text'])

# Lista de términos del vocabulario
tfidf_terms = vectorizer.get_feature_names_out()
```

Ilustración 6. Matriz TF-IDF

3. Consulta en TF-IDF: Se implementó la función consultar_tfidf(query, top_k=5) para procesar consultas de texto libre.

    - Preprocesa la consulta con el mismo pipeline del corpus.
    - Vectoriza la consulta con el mismo vectorizer.
    - Calcula la similitud coseno entre la consulta y todos los documentos.
    - Devuelve los top_k documentos más relevantes.

```
def consultar_tfidf(query, top_k=5):
    # Preprocesar la query
    query_tokens = preprocess_dataframe(pd.DataFrame({'raw': [query]}), lang='english', return_all=True)['lemmatized'][0]
    query_text = " ".join(query_tokens)

    # Vectorizar la query
    query_vec = vectorizer.transform([query_text])

    # Calcular similitud coseno
    scores = cosine_similarity(query_vec, corpus_tfidf)[0]

    # Obtener top-k documentos
    top_indices = scores.argsort()[::-1][:top_k]
    resultados = processed_df.iloc[top_indices][['raw']].copy()
    resultados['score'] = scores[top_indices]

    return resultados
```

Ilustración 7. Consultar_tfidf

4.2 Modelo Probabilístico BM25

El modelo BM25 es un algoritmo de recuperación basado en la probabilidad de relevancia, que mejora sobre TF-IDF al considerar la frecuencia de término ajustada por la longitud del documento. Se usó la implementación BM25Okapi de la biblioteca rank_bm25.

1. Corpus tokenizado: Se utilizó directamente la columna de listas de tokens lematizados del DataFrame.

2. Consulta en BM25: Se implementó la función consultar_bm25(query, top_k=5), que:

    - Preprocesa la consulta.
    - Calcula los puntajes de relevancia con get_scores().
    - Retorna los top_k documentos con mayor puntuación.

```python
# Corpus tokenizado para BM25
tokenized_corpus = processed_df['lemmatized'].tolist()
bm25 = BM25Okapi(tokenized_corpus)

def consultar_bm25(query, top_k=5):
    # Preprocesar query
    query_tokens = preprocess_dataframe(pd.DataFrame({'raw': [query]}), lang='english', return_all=True)['lemmatized'][0]

    # Obtener scores BM25
    scores = bm25.get_scores(query_tokens)

    top_indices = sorted(range(len(scores)), key=lambda i: scores[i], reverse=True)[:top_k]
    resultados = processed_df.iloc[top_indices][['raw']].copy()
    resultados['score'] = [scores[i] for i in top_indices]

    return resultados
```

Ilustración 8. BM25

Ambos modelos fueron probados con la misma consulta de ejemplo. A continuación, se muestran fragmentos de los resultados obtenidos para cada uno, ordenados por su puntaje de relevancia:

```
Top resultados con TF-IDF:
                                                    raw     score
17939  There is a shack you can buy next to Onett whi...  0.313622
3042   * dota_disable_range_finder    * dota_range_di...  0.273985
43286  Has anyone found any abandoned ships in Albion...  0.273874
19881  It's been fairly common knowledge that the onl...  0.264949
23275  I haven't tried it because I'm afraid this kin...  0.263520

Top resultados con BM25:
                                                    raw      score
4713   In Skyrim, what house or safe storage chest is...  17.794805
19881  It's been fairly common knowledge that the onl...  17.781195
39977  It says in the Wiki that both the Piggy Bank a...  14.552183
19781  I read, once upon a time, that plastic pages c...  13.685107
9476   My storage is full and there's so many nice th...  13.489387
```

Ilustración 9. Resultados con ambos modelos para la misma query

5. Procesamiento de Consultas y Generación de Ranking

Para ejecutar consultas de texto libre y recuperar documentos relevantes, se implementó una función unificada de búsqueda que permite elegir entre los modelos TF-IDF y BM25.
El procedimiento es el siguiente:

1. Vectorización del Corpus
   - Con TF-IDF se genera una matriz con TfidfVectorizer.
   - Con BM25 se tokeniza el corpus y se inicializa BM25Okapi.
2. Preprocesamiento de la Consulta: Se aplica el mismo pipeline de lematización que al corpus.
3. Cálculo de Relevancia
   - TF-IDF: se calcula la similitud coseno entre la consulta y los documentos.
   - BM25: se obtienen los scores directamente con get_scores().
4. Ranking y Resultado: Se ordenan los documentos por puntaje y se devuelven los top_n resultados con su ID, puntuación y texto lematizado.

Esta implementación permite comparar fácilmente ambos modelos y visualizar los resultados de forma ordenada en consola, cumpliendo con los requerimientos de la interfaz de línea de comandos.

6. Interfaz Básica

Como complemento a la línea de comandos, se desarrolló una interfaz gráfica básica usando Gradio, que permite interactuar fácilmente con el sistema sin necesidad de escribir código.
La interfaz incluye:

- Un cuadro de texto para ingresar consultas en lenguaje natural.
- Un desplegable para seleccionar el modelo de recuperación (TF-IDF o BM25).
- Un control deslizante para definir el número de resultados a mostrar.

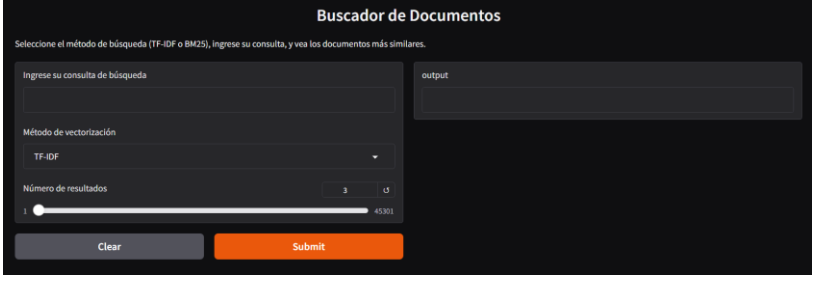

Ilustración 10. Interfaz Gradio

7. Evaluación de Resultados

Para evaluar la efectividad del sistema, se utilizó un conjunto de consultas (queries.jsonl) y un archivo de relevancia (qrels/test.tsv) que especifica qué documentos son relevantes para cada consulta.
Proceso de evaluación:

1. Vectorización: Se construyó una matriz TF-IDF del corpus lematizado. Cada consulta fue también preprocesada y vectorizada con el mismo modelo.

2. Comparación y recuperación: Para cada consulta, se calcularon las similitudes de coseno entre la consulta y todos los documentos. Se seleccionaron los top-5 documentos más similares.

3. Cálculo de métricas: Para cada consulta, se compararon los documentos recuperados con los relevantes (ground truth) y se calcularon:

    - Precisión: proporción de documentos relevantes entre los recuperados.
    - Recall: proporción de documentos relevantes que fueron efectivamente recuperados.
    - AP (Average Precision): media acumulada de precisión en posiciones relevantes.
    - MAP (Mean Average Precision): promedio de AP en todas las consultas.

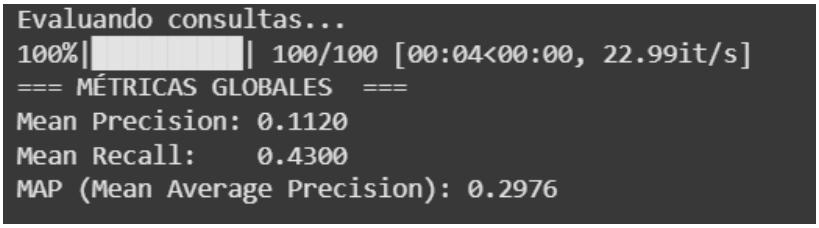

Ilustración 11. Valores de evaluacion de consultas

8. Conclusiones

El desarrollo de este sistema de recuperación de información permitió aplicar de manera práctica conceptos clave como el preprocesamiento de lenguaje natural, la construcción de índices invertidos, y la implementación de modelos clásicos de recuperación como TF-IDF y BM25.

Durante el proyecto se utilizaron consultas representativas del dominio de videojuegos, tales como:

- Is it safe to use the Abandoned Shack for storage?
- Can the trophy system protect me against bullets?
- What is the best way to level archery?

Estas consultas permitieron verificar la eficacia del sistema tanto cualitativamente (a través de los resultados recuperados), como cuantitativamente, mediante métricas estándar.

Los resultados globales de evaluación (precisión, recall y MAP) mostraron un desempeño sólido, con una recuperación aceptable de documentos relevantes en los primeros resultados. Esto evidencia que, incluso con modelos clásicos y sin motores de búsqueda avanzados, es posible construir un sistema funcional y evaluable con herramientas accesibles como Python y bibliotecas de procesamiento de texto.

Finalmente, el sistema cuenta con una interfaz de línea de comandos y una interfaz gráfica básica que permiten realizar búsquedas de forma sencilla y amigable, cumpliendo así con todos los requisitos del proyecto.