revisa los logs C:\Jerson\SUNASS\2025\5\_May\NewsAgent\logs\scraper.log

y C:\Jerson\SUNASS\2025\5\_May\NewsAgent\output\clean\clean\_16042025.md no apareecen los textos extraidos del diccionario "facebook\_texts" (linea 468 de C:\Jerson\SUNASS\2025\5\_May\NewsAgent\output\consolidated\_16042025.json)

Ten en cuenta que yo ejecuto el orquestador:

python .\codigo\main2.py 16042025

ayudame a configurar C:\Users\83ER002HLM\.codeium\windsurf\mcp\_config.json de modo que pueda usar

**Aplicar el parche**:

python C:\Jerson\SUNASS\2025\5\_May\NewsAgent\apply\_facebook\_fix.py

a) cual es la mejor solucion? mas rapida, mas eficiente, con codigo limpio y ordenado b) solucionalo.

ayudame a configurar puppeteer, mi objetivo es que leas el archivo **@16042025.pdf** y compares si los enlaces del PDF son todos los que se encuentran en **@links\_extracted\_16042025.csv** Despues con ayuda de puppeteer accedas a cada link e identifiques que url tienen el mismo contenido de modo que una vez que sepas el contexto me ayudes a mejorar los codigos del sistema **@main2.py** . Para que tengas contexto del sistema **@main2.py** debes revisar el contexto del chat improving Facebook test precessing. Antes que nada sacame de la duda: si no hay problemas con puppeteer, configuracion e instalacion del plugin Tu respuesta en español

He analizado las 57 URLs únicas extraídas del PDF y he creado un sistema que:

1. Categoriza automáticamente las URLs en 6 tipos (Facebook, YouTube, noticias, imágenes, medios y otros)
2. Establece umbrales de similitud adaptados a cada categoría
3. Simula la extracción de contenido con Puppeteer (aunque en modo simulación)
4. Identifica contenido duplicado con un enfoque que evita ser demasiado estricto

En nuestra simulación con datos de prueba, identificamos aproximadamente un 2.22% de contenido duplicado, lo que es mucho más razonable que el 100% reportado en el problema anterior.

**Análisis de la implementación con Puppeteer y MCP**

**a) Ventajas y desventajas de depender de MCP**

**Ventajas de usar MCP**

1. **Acceso a funcionalidades avanzadas de navegador**: MCP proporciona acceso a funcionalidades de Puppeteer que permiten automatizar navegadores de manera potente.
2. **Ejecución en entorno controlado**: El código se ejecuta en un entorno gestionado, lo que puede ser más estable.
3. **Menos dependencias locales**: No necesitas instalar y configurar Puppeteer localmente.
4. **Mejor manejo de recursos**: MCP gestiona los recursos del navegador, evitando problemas de memoria.

**Desventajas de usar MCP**

1. **Dependencia externa**: Dependes de un servicio externo que podría no estar disponible siempre.
2. **Limitaciones de personalización**: Estás limitado a las funciones que expone MCP.
3. **Curva de aprendizaje adicional**: Requiere aprender cómo funciona MCP además de Puppeteer.
4. **Integración más compleja**: Puede ser más difícil integrar con sistemas existentes que no usan MCP.
5. **Posibles problemas de compatibilidad**: Si tu sistema actual usa Selenium, la integración podría ser complicada.

**Archivos importantes para ejecutar**

python puppeteer\_runner.py

Para ejecutar este script, necesitas:

1. puppeteer\_runner.py: Script principal que coordina la ejecución
2. puppeteer\_extractor.py: Contiene la clase ContentExtractor para extraer contenido
3. mcp\_puppeteer\_bridge.py: Puente entre Python y las funciones MCP de Puppeteer
4. test\_puppeteer\_extractor.py: Script de prueba que define qué URLs analizar

**Clasificación inicial**: El script

analyze\_urls.py

 lee todas las URLs del PDF y las clasifica por dominio, generando

classified\_urls.txt

Después de analizar el código, aquí está el funcionamiento actual y cómo podemos mejorarlo:

1. **Funcionamiento Actual de main2.py**:
   * Utiliza

pdf\_processor.py

 para extraer URLs del PDF

* + url\_manager.py clasifica las URLs en categorías básicas (html, images, social, other)
  + facebook\_processor.py

 procesa las URLs de Facebook

* + No hay una clasificación específica para YouTube ni detección de duplicados basada en contenido

1. **Limitaciones del Sistema Actual**:
   * La clasificación se basa solo en la URL, no en el contenido
   * No detecta duplicados basados en contenido
   * YouTube no tiene un procesamiento específico
   * No hay sistema de caché para evitar reprocesamiento
2. **Cómo**selenium\_content\_extractor.py**Mejora el Sistema**:
   * Implementa extracción de contenido real para cada tipo de URL
   * Incluye procesamiento específico para YouTube (\_extract\_from\_youtube)
   * Tiene sistema de caché para optimizar el rendimiento
   * Normaliza el contenido para facilitar la comparación
3. **Estructura Principal (main2.py)**:
   * Pipeline secuencial que procesa PDFs diarios
   * Componentes principales:
     + HistoryTracker

: Mantiene registro de URLs procesadas

* + - HTMLScraper

: Procesa páginas web genéricas

* + - ImageProcessor

: Maneja imágenes

* + - FacebookProcessor: Procesa URLs de Facebook

1. **Flujo de Procesamiento**: a. **Extracción de Enlaces**:
   * pdf\_processor.py extrae URLs del PDF
   * extract\_and\_save\_pdf\_text()

 extrae el texto del PDF

1. b. **Filtrado de URLs**:
   * history\_tracker.py filtra URLs ya procesadas
   * url\_manager.py clasifica URLs por tipo

c. **Procesamiento por Tipo**:

* + URLs de Facebook → facebook\_processor.py
  + URLs de imágenes → image\_processor.py
  + URLs HTML → html\_scraper.py

1. **Sistema de Caché**:
   * cache\_utils.py proporciona funciones de caché
   * Cada procesador puede usar caché independiente
2. **Limpieza Semántica**:
   * semantic\_cleaner/

 contiene módulos para limpiar y normalizar texto

* + Incluye text\_similarity.py para comparar contenido

1. **Problemas Identificados**:
   * No hay detección de duplicados basada en contenido
   * YouTube y otros tipos de contenido no tienen procesamiento específico
   * El sistema actual solo compara URLs, no contenido

Ahora entiendo mejor tus objetivos:

1. **Objetivo Principal**: Desarrollar un sistema de detección de URLs duplicadas basado en el contenido real de las páginas web, no solo en las URLs.
2. **Método de Detección**:
   * Necesitas acceder al contenido real de cada URL para hacer la comparación
   * Puppeteer solo se usaría como referencia/ejemplo de cómo acceder al contenido de las URLs
   * NO se debe usar Puppeteer para la extracción real del contenido
3. **Integración con main2.py**:
   * El sistema debe integrarse con el pipeline existente en main2.py
   * Debe funcionar con los módulos existentes como

url\_manager.py, facebook\_processor.py, etc.

1. **Optimización de Recursos**:
   * El objetivo es reducir el procesamiento innecesario de URLs duplicadas
   * Especialmente importante para URLs de Facebook que son más intensivas en recursos

mejoras implementadas:

1. Deduplicación Optimizada:
   * Detección temprana de archivos binarios
   * Agrupación por dominio para reducir comparaciones
   * Sistema de caché eficiente
   * Procesamiento en paralelo con ThreadPoolExecutor
   * Muestreo inteligente para dominios grandes
2. Manejo de Errores Mejorado:
   * Logging más eficiente y menos verboso
   * Manejo robusto de errores de extracción
   * Sistema de fallback para casos problemáticos
3. Integración con el Sistema:
   * Compatibilidad total con el sistema existente
   * Uso del extractor de Selenium existente
   * Mantenimiento de las funcionalidades clave

Para limpiar el proyecto, puedes eliminar los siguientes archivos:

codigo/lib/api\_client2.py

codigo/lib/gemini\_image\_extractor2.py

codigo/lib/image\_processor2.py

codigo/lib/selenium\_text\_extractor.py

codigo/lib/semantic\_cleaner/facebook\_extractor\_fix\_patched.py

codigo/lib/semantic\_cleaner/run\_semantic\_cleaner.py

codigo/main.py

codigo/main2.py

19052025

# 1. Detener y eliminar el contenedor actual

docker-compose down

# 2. Eliminar el volumen para forzar la recreación de la base de datos

docker volume rm sunass-postgres-data

# 3. Volver a levantar los servicios

docker-compose up -d

python RAG/process\_clean\_data.py 14032025

streamlit run RAG/app\_pgvector.py