Documentación Técnica Proyecto Final
Análisis de algoritmos

Presentado por:

María Luisa Alonso
Victor Manuel Alzate Galviz

Profesor:

Sergio Augusto Cardona Torres

2025-2

Universidad del Quindío
Facultad de Ingeniería
Análisis de Algoritmos
Ingeniería de Sistemas y Computación Nocturno

El presente documento relaciona los documentos técnicos del proyecto análisis de algoritmos 2025-2 Ingeniería de sistemas y computación nocturno Universidad del Quindío, para su correcto funcionamiento se deben instalar librerías claves las cuales detallamos en el documento requeriments.txt y explicamos a continuación:

Tabla de Contenido

¿Qué es el archivo requirements.txt?

El archivo requirements.txt es el manifiesto de dependencias de un proyecto de Python, su función principal es listar todas las librerías externas que el código necesita para funcionar correctamente.

Su propósito es garantizar la reproducibilidad del entorno, con esto dicho, si se quiere ejecutar este proyecto, solo se necesita instalarlas siguientes librerías: Ejecutar el comando:

pip install -r requirements.txt

Y pip (el gestor de paquetes de Python) se encargará de instalar automáticamente todas las librerías listadas con sus versiones correspondientes, asegurando que el entorno sea idéntico y que el proyecto se comporte de la misma manera en todas partes.

Explicación de cada librería en el proyecto Análisis Bibliométrico

A continuación, se detalla para qué se usa cada librería en la aplicación de análisis bibliométrico:

Framework del Backend y Servidor

fastapi: Es el framework web que se utiliza para construir la API del backend. Permite definir las rutas (endpoints) que el frontend puede llamar para solicitar análisis (ej. /analizar-similitud, /generar-visualizaciones). Es moderno, muy rápido y genera documentación de API automáticamente.

uvicorn[standard]: Es el servidor que ejecuta la aplicación FastAPI. uvicorn es un servidor ASGI (Asynchronous Server Gateway Interface), lo que significa que puede manejar muchas conexiones simultáneamente de manera muy eficiente, ideal para una aplicación web.

python-multipart: FastAPI lo necesita para poder recibir datos de formularios, como la subida de archivos.

jinja2: Es un motor de plantillas. Aunque FastAPI se usa principalmente para APIs, jinja2 podría usarse para renderizar páginas HTML desde el backend si fuera necesario (como la página de index.html).

Automatización y Web Scraping (Requerimiento 1)

selenium: Es la herramienta clave para la automatización del navegador. Se utiliza en web_scraper.py para controlar un navegador Chrome, iniciar sesión en las bases de datos científicas (IEEE, SAGE) y hacer clic en los botones de descarga.

webdriver-manager: Trabaja junto a selenium, su función es descargar y gestionar automáticamente el ChromeDriver correcto para la versión de Chrome que tengas instalada, evitando la necesidad de hacerlo manualmente.

Procesamiento de Datos y Análisis (Requerimientos 1, 2, 3, 4)

bibtexparser: Librería fundamental para el proyecto. Se usa en casi todos los módulos de análisis para leer, analizar y escribir archivos en formato .bib, que es el formato en que se descargan las citaciones.

scikit-learn: Es una de las librerías más importantes de Machine Learning en Python, en el presente proyecto se usa para:

TfidfVectorizer: Convertir los abstracts de texto en vectores numéricos (Requerimientos 2, 3 y 4).

cosine_similarity / cosine_distances: Calcular la similitud/distancia entre esos vectores (Requerimientos 2 y 4).

scipy: Una librería de computación científica. Su rol principal aquí es en el análisis de clústeres (Requerimiento 4).

scipy.cluster.hierarchy: Contiene las funciones linkage y dendrogram para realizar el agrupamiento jerárquico y generar la estructura del dendrograma.

numpy: Es la base para casi toda la computación numérica en Python. scikit-learn, pandas y scipy dependen de ella. Se encarga de la creación y manipulación eficiente de los arrays y matrices numéricas (como los vectores TF-IDF).

pandas: Se utiliza para la manipulación de datos estructurados. En generador_visualizaciones.py (Requerimiento 5), se usa para cargar los datos del .bib en un DataFrame, lo que facilita enormemente la agregación y el cálculo de estadísticas (ej. contar publicaciones por año o por país).

Modelos de IA (Requerimiento 2)

sentence-transformers: Es la librería que permite descargar y utilizar modelos de lenguaje profundos (como BERT) para convertir texto en embeddings semánticos. Se usa para el análisis de similitud basado en IA.

torch: Es el framework de Deep Learning (desarrollado por Facebook/Meta) sobre el cual se ejecutan los modelos de sentence-transformers. Es una dependencia necesaria para que estos modelos de IA puedan funcionar.

Visualización de Datos y Reportes (Requerimiento 5)

matplotlib: La librería clásica para generar gráficos estáticos en Python. Se usa para crear la línea de tiempo de publicaciones por año y el dendrograma.

wordcloud: Específica para crear las nubes de palabras a partir de la frecuencia de los términos en los abstracts y keywords.

plotly: Se usa para crear los gráficos interactivos, como el mapa de calor geográfico y la línea de tiempo de publicaciones por revista.

pycountry: Utilidad para convertir nombres de países (ej. "Germany") a sus códigos estándar (ej. "DEU"), necesarios para que plotly pueda dibujar el mapa de calor.

fpdf2: Permite crear y escribir archivos PDF desde Python. Se usa para generar el reporte final que consolida todas las visualizaciones.

kaleido: Es un motor que plotly utiliza internamente para poder exportar sus gráficos interactivos a imágenes estáticas (como PNG), que son las que se incrustan en el PDF.

Utilidades Adicionales

networkx: Aunque no lo hemos discutido en detalle, esta librería se usa para crear, manipular y estudiar la estructura de grafos y redes complejas. Probablemente se utilice en el analizador_grafos.py (Requerimiento 6 o 7) para analizar redes de co-autoría o de citaciones.

rapidfuzz: Es una implementación rápida de algoritmos de similitud de cadenas, como la distancia de Levenshtein. Es probable que se use como una alternativa más optimizada que una implementación manual.

Requerimientos técnicos de cada uno de los requerimientos

Requerimiento 1: Automatización de Proceso de Descarga y Unificación de Datos

Arquitectura y Diseño

El diseño de este requerimiento se basa en un pipeline de dos fases: extracción y procesamiento.

1. Fase de Extracción (Scraping): Se emplea un enfoque modular donde un orquestador principal (`web_scraper.py`) invoca a sub-módulos especializados para cada base de datos (`scrapers/ieee_scraper.py`, `scrapers/sage_scraper.py`, etc.). Se utiliza la librería `Selenium` para controlar un navegador web, simulando la interacción de un usuario para realizar búsquedas y descargar los archivos de citaciones en formato `.bib`. La arquitectura permite escalar fácilmente a nuevas bases de datos simplemente añadiendo un nuevo módulo de scraping.

2. Fase de Procesamiento (Unificación y Deduplicación): Un script independiente (`unificador_deduplicador.py`) se encarga de procesar los datos descargados. Este script lee todos los archivos `.bib` de un directorio, los unifica en una sola base de datos y elimina registros duplicados.

Detalles de Implementación

Extracción de Datos (web_scraper.py)

Librerías:`selenium`, `webdriver-manager`.

Proceso:

1. El script recibe como argumentos la base de datos a atacar, el email y la contraseña del usuario.

2. Se instancia un driver de Chrome con `Selenium`, configurado para evitar la detección de automatización y para descargar archivos automáticamente en un directorio específico (`datos/descargas/<nombre_base_de_datos>`).

3. Se realiza el login en el portal de la base de datos.

4. Se ejecuta una búsqueda con el término "generative artificial intelligence".

5. Se itera sobre las páginas de resultados, descargando los datos de cada página en formato `BibTeX`.

Manejo de Sesión: Se utiliza `getpass` para solicitar la contraseña de forma segura.

Unificación y Deduplicación (unificador_deduplicador.py)

Librerías: `bibtexparser`, `os`, `unicodedata`.

Proceso:

1. El script escanea el directorio `datos/descargas` en busca de todos los archivos `.bib`.

2. Cada archivo es parseado con `bibtexparser`.

3. Para la deduplicación, se genera una **clave única** por cada artículo. La estrategia para esta clave es:

Prioridad 1: El `DOI` del artículo, si existe.

Prioridad 2: Una combinación normalizada del autor, año y título.

Prioridad 3: El ID interno del registro BibTeX.

4. Se utilizan dos estructuras de datos: un diccionario para guardar las entradas representativas y un contador para registrar las repeticiones.

5. Finalmente, se generan dos archivos en la carpeta `datos/procesados`:

articulos_unicos.bib: Contiene todas las entradas únicas.

articulos_duplicados.csv: Un resumen de los artículos que se encuentran duplicados y cuántas veces aparecieron, este formato ayuda a reducir el tamaño del archivo.

Requerimiento 2: Implementación de Algoritmos de Similitud Textual

Arquitectura y Diseño

La arquitectura para este requerimiento se centra en un módulo principal (`analizador_similitud.py`) que expone una función para cada algoritmo de similitud. Para los modelos de IA, se implementa un sistema de caché simple para evitar cargar los modelos pesados en memoria más de una vez. Cada función de análisis toma como entrada los IDs de dos artículos y devuelve un diccionario con el resultado de la similitud y una explicación detallada del algoritmo utilizado.

Detalles de Implementación

Librerías: `scikit-learn`, `sentence-transformers`, `bibtexparser`.

Algoritmos Implementados:

1. Distancia de Levenshtein:

   Implementación: Se calcula la distancia de edición entre dos `abstracts`. La similitud se normaliza con la fórmula: `1 - (distancia / longitud_maxima)`.

2. Similitud de Coseno con TF-IDF:

   Implementación: Se utiliza `TfidfVectorizer` de `scikit-learn` para convertir los `abstracts` en vectores TF-IDF. Luego, `cosine_similarity` calcula la similitud entre estos vectores.

3. Índice de Jaccard:

   Implementación: Los `abstracts` se convierten en conjuntos de palabras. La similitud es `|A ∩ B| / |A ∪ B|`.

4. Coeficiente de Sørensen-Dice:

   Implementación: Similar a Jaccard, pero la fórmula es `2 * |A ∩ B| / (|A| + |B|)`.

5. IA - Sentence Transformers (`all-MiniLM-L6-v2` y `paraphrase-mpnet-base-v2`):

   Implementación: Se usa la librería `sentence-transformers`.

   Se carga un modelo pre-entrenado (e.g., `all-MiniLM-L6-v2`).

   Cada `abstract` se codifica en un "embedding" (vector semántico).

   Se calcula la similitud del coseno entre los dos embeddings.

Justificación de IA: Estos modelos se utilizan porque capturan el significado semántico del texto, no solo las palabras clave. Esto permite identificar artículos conceptualmente similares aunque no compartan el mismo vocabulario exacto, superando las limitaciones de los métodos clásicos.

Requerimiento 3: Cálculo de Frecuencia de Palabras

Arquitectura y Diseño

El diseño se basa en un módulo (`analizador_frecuencias.py`) que contiene funciones para dos tareas principales: (1) contar la frecuencia de una lista de palabras predefinidas y (2) generar un nuevo listado de palabras clave relevantes a partir de los textos. Los resultados se presentan como diccionarios de frecuencias y se generan visualizaciones (gráficos de barras) para facilitar la interpretación.

Detalles de Implementación

Librerías: `scikit-learn`, `matplotlib`, `bibtexparser`.

Proceso:

1. Carga de Datos: Se leen los `abstracts` del archivo `articulos_unicos.bib`.
2. Frecuencia de Palabras Dadas:

   Se recibe una lista de palabras clave.

   Se itera sobre los `abstracts` y se utiliza `regex` (`re.findall`) para contar las apariciones de cada palabra clave. Se usa `\b` para asegurar que se cuenten palabras completas.

3. Generación de Nuevas Palabras Clave:

   Justificación de IA (TF-IDF): Se utiliza el algoritmo `TfidfVectorizer` de `scikit-learn` como una técnica de "Machine Learning no supervisado" para identificar las palabras más significativas del corpus. TF-IDF asigna un peso a cada palabra que refleja su importancia en un documento dentro de una colección de documentos, permitiendo extraer términos clave de forma automática.

Se ajusta el TfidfVectorizer con los `abstracts`, configurándolo para que extraiga las 15 palabras con el `score` más alto (excluyendo "stop words" en inglés).

Estas 15 palabras se convierten en la nueva lista de palabras clave.

4. Cálculo de Precisión:

Se compara la lista de palabras generadas automáticamente con la lista de palabras original.

La precisión se calcula como: `|palabras_comunes| / |palabras_generadas|`.

5. Visualización:

Se utiliza `matplotlib` para generar gráficos de barras que muestran las frecuencias de ambos conjuntos de palabras.

Los gráficos se guardan en un buffer en memoria y se codifican en `base64` para poder ser enviados fácilmente a través de una API a un frontend.

Requerimiento 4: Agrupamiento Jerárquico de Artículos

Propósito: Implementar algoritmos de agrupamiento jerárquico para construir un dendrograma que represente visualmente la similitud entre los abstracts de los artículos científicos. Esto permite identificar clústeres temáticos de forma no supervisada.

Arquitectura y Diseño

Lenguaje: Python.

Librerías Principales:

scikit-learn: Específicamente `TfidfVectorizer` para la conversión de texto a vectores y `cosine_distances` para el cálculo de la matriz de disimilitud.

scipy: Fundamental para el análisis de clústeres, se utiliza scipy.cluster.hierarchy.linkage para aplicar los algoritmos de agrupamiento y dendrogram para la lógica de la visualización.

matplotlib: Para la renderización final del dendrograma en una imagen.

bibtexparser: Para cargar los datos de los artículos.

Diseño: El módulo `analizador_cluster.py` encapsula toda la lógica. La función principal `analizar_agrupamiento_completo` es parametrizable por el método de `linkage` a utilizar (ej. 'ward', 'complete', 'average'), lo que permite una fácil comparación entre los diferentes enfoques de agrupamiento.

Detalles de Implementación

1. Carga y Muestreo de Datos (`_cargar_y_muestrear_articulos`): Para evitar la sobrecarga computacional y visual, el sistema no procesa todos los artículos, sino que toma una muestra aleatoria (configurable, por defecto 30). Esto asegura que el dendrograma sea legible y que el análisis se complete en un tiempo razonable. Se seleccionan solo artículos que contienen un `abstract`.

2. Preprocesamiento y Vectorización del Texto:

Extracción: Se extraen los `abstracts` y los `titles` de los artículos muestreados. Los títulos se usarán como etiquetas en el dendrograma y se truncan a 75 caracteres para no saturar la visualización.

Vectorización con TF-IDF: Se utiliza `TfidfVectorizer` para convertir los abstracts en una matriz de vectores numéricos. Se configuran los siguientes hiperparámetros:

stop_words='english': Elimina palabras comunes del inglés que no aportan significado semántico (ej. "the", "a", "is").

max_features=1000: Limita el vocabulario a las 1000 palabras más importantes según su puntuación TF-IDF. Esto reduce el ruido y la dimensionalidad del espacio vectorial, enfocándose en los términos más relevantes.

3. Cálculo de la Matriz de Distancia:

El agrupamiento jerárquico no trabaja con similitud, sino con distancia (o disimilitud). Se utiliza `cosine_distances` de `scikit-learn`, que es matemáticamente `1 similitud_coseno. El resultado es una matriz cuadrada donde cada celda `(i, j)` contiene la distancia entre el artículo `i` y el artículo `j`.

4. Algoritmo de Agrupamiento (`linkage`):

Esta es la función clave de `scipy`. Toma la matriz de distancias y el método de agrupamiento seleccionado ('ward', 'complete', o 'average') como entrada.

El algoritmo procede iterativamente:

a. Inicialmente, cada artículo es su propio clúster.

b. En cada paso, se fusionan los dos clústeres más cercanos según el criterio del método seleccionado (ej. minimizando la varianza para 'ward', o basándose en la distancia máxima para 'complete').

El resultado es una `linkage_matrix`, una estructura de datos que codifica todas las fusiones realizadas, la distancia a la que ocurrieron y el número de elementos en cada clúster. Esta matriz es la definición matemática del dendrograma.

5. Generación del Dendrograma (`_generar_dendrograma_base64`):

La función `dendrogram` de `scipy` toma la `linkage_matrix` y las etiquetas (títulos de los artículos) para organizar la visualización.

Se renderiza con `matplotlib`, con la orientación a la derecha para facilitar la lectura de los títulos.

La imagen no se guarda en disco. Se guarda en un buffer de memoria (`io.BytesIO`) y se codifica en **Base64**. Esto permite que la imagen sea enviada directamente a través de una API REST al frontend, sin necesidad de gestionar archivos temporales.

Justificación del Uso de IA (Machine Learning)

El agrupamiento jerárquico es una técnica clásica de aprendizaje no supervisado, cuyo propósito es descubrir una estructura o patrón subyacente en los datos sin tener etiquetas predefinidas. En este contexto, la "IA" no es un modelo predictivo, sino un **algoritmo de descubrimiento de conocimiento**. Al agrupar los abstracts por su contenido semántico (vectorizado por TF-IDF), el sistema identifica automáticamente "escuelas de pensamiento" o sub-temas dentro del dominio de la "IA generativa". Un investigador puede usar este dendrograma para entender rápidamente qué artículos discuten temas similares, revelando la estructura temática del campo de investigación de una manera que sería extremadamente lenta y difícil de hacer manualmente.

Requerimiento 5: Análisis Visual de la Producción Científica

Propósito: Generar un conjunto de visualizaciones interactivas y estáticas para ofrecer una visión general de la producción científica, incluyendo aspectos geográficos, temporales y temáticos. Finalmente, consolidar estos análisis en un reporte PDF descargable.

Arquitectura y Diseño

Lenguaje: Python.

Librerías Principales:

pandas: Para la manipulación y agregación de los datos bibliográficos.

plotly: Para la creación de gráficos interactivos (mapa de calor, línea de tiempo por revista).

wordcloud: Para generar la nube de palabras.

matplotlib: Para generar gráficos estáticos (línea de tiempo general).

pycountry: Para la conversión de nombres de países a códigos ISO 3166-1 alpha-3, necesarios para el mapa de calor.

fpdf: Para la generación del reporte en formato PDF.

Diseño: El módulo `generador_visualizaciones.py` contiene funciones específicas para cada tipo de gráfico. Una función orquestadora, `generar_visualizaciones_completo`, las invoca a todas y consolida los resultados. Los gráficos se devuelven en formato Base64 (para imágenes estáticas) o JSON (para gráficos interactivos de Plotly), permitiendo su fácil integración en un frontend. Una función separada, `exportar_visualizaciones_pdf`, se encarga de componer el reporte final.

Detalles de Implementación

1. Mapa de Calor Geográfico (`_generar_mapa_calor`):

Extracción de País: Se implementa una función `_extraer_pais` que utiliza expresiones regulares y la librería `pycountry` para identificar el país a partir del campo de afiliación del autor en el archivo BibTeX. Es una heurística robusta que busca nombres de países y códigos comunes (USA, UK).

Agregación de Datos: Con `pandas`, se agrupan los artículos por el código de país extraído, contando el número de publicaciones por país. Adicionalmente, se identifican los 3 autores más prolíficos de cada país.

Visualización con Plotly: Se utiliza `px.choropleth` para generar el mapa. Los datos clave (conteo, autores principales) se pasan al parámetro `custom_data`, y la plantilla del `hovertemplate` se personaliza para mostrar esta información de forma clara cuando el usuario pasa el ratón sobre un país. El gráfico se devuelve tanto en formato JSON (para interactividad en la web) como en imagen Base64 (para el PDF).

2. Nube de Palabras (`_generar_nube_palabras`):

Consolidación de Texto:** Se concatenan los campos `abstract` y `keywords` de todos los artículos en un único gran bloque de texto.

Generación: La librería `wordcloud` procesa este texto, calcula la frecuencia de cada palabra y genera una imagen donde el tamaño de cada palabra es proporcional a su frecuencia. El resultado es una imagen Base64.

3. Líneas de Tiempo (`_generar_linea_tiempo`, `_generar_linea_tiempo_por_revista`):

Agregación por Año: Usando `pandas`, se agrupan las publicaciones por año y se cuenta el número de artículos en cada uno. Esto se visualiza con `matplotlib` como un gráfico de líneas simple.

Agregación por Año y Revista: Para un análisis más detallado, se agrupa por año y por el nombre de la revista (`journal`). Para mantener la legibilidad, se filtran los datos para mostrar solo las 10 revistas con más publicaciones. Se utiliza `plotly.express.line` para crear un gráfico de líneas interactivo donde cada línea representa una revista.

4. Exportación a PDF (`exportar_visualizaciones_pdf`):

Composición del Documento: Se utiliza la librería `FPDF`. Se crea una página para cada visualización.

Incrustación de Imágenes: Las imágenes (previamente generadas y pasadas como strings Base64) se decodifican y se escriben en el PDF utilizando un buffer en memoria, evitando la creación de archivos intermedios.

Inclusión de Datos Tabulares: Para el mapa de calor, además de la imagen, se incluye una tabla con el top 10 de países, sus conteos de publicaciones y los autores principales, proporcionando un resumen cuantitativo.

Generación del Archivo: El PDF se guarda en el directorio `datos/reportes_visuales` con un nombre de archivo que incluye la fecha y hora (`timestamp`), asegurando que cada reporte sea único.

Justificación del Uso de IA

No se aplica IA en este componente. Es un módulo de **Business Intelligence (BI) y visualización de datos**. Su objetivo es transformar los datos bibliográficos en insights visuales y fácilmente digeribles para un usuario humano, permitiéndole entender tendencias y patrones en la producción científica de un vistazo.