Intelligent Research Paper Summarization and Citation Network System

CMPE 255 – Data Mining

Project Option: 1 – Automatic Data Mining for Large PDF Files
Focused Area: AI-Driven Data Analytic Track
Instructor: Dr. Kai Kai Liu

Team 17

Name	**SJSU ID**	**Email**
Nivedita Nair	018184777	nivedita.nair@sjsu.edu
Shanmukha Manoj Kakani	018195645	shanmukhamanoj.kakani@sjsu.edu
Kalyani Chitre	017622917	kalyani.chitre@sjsu.edu

Abstract

With the rapid development of academic research in the various fields, it becomes increasingly hard for a scholar to keep abreast of the literature in their area. Automating the extraction, summarization, and semantic analysis of scholarly documents by means of end-to-end AI-powered applications is an attractive approach for such a paradigm. Given this, a researcher would be able to upload a paper in PDF and let all intelligent procedures carry on extracting key sections from that paper, summarizing concisely, identifying references, and visualizing all citation relations.

Thus, the system design integrates extractive and abstractive summarization techniques fed by state-of-the-art HuggingFace models, fused with LangChain's RAG architecture to augment question answering capabilities. As well, the interface supports semantic search over multiple documents through sentence embeddings coupled with vector similarity implemented using FAISS. The backend is designed with Python and Flask, and the frontend follows the modern principles of responsive design for an ample interactive experience. Furthermore, it generates citation graphs using NetworkX that reveal relationships between papers and research topics. In general, it is meant to smooth literature reviews, expedite academic discovery, and make scholarly research more approachable and navigable for students, researchers, and professionals.

Contents

1. Project Overview

1.1 Project Option

This project was completed under Option 1 – Automatic Data Mining for Large PDF Files. It focuses on building an intelligent AI-driven system that automates the extraction, analysis, and interpretation of content from academic research papers in PDF format.

1.2 Selected Applications and Components

A web-based Research Paper Assistant has thus been selected as an application to help researchers quickly extract meaningful insights from the fine expositions of academic literature. The principle members of the system are:

- Hybrid Summarization Module: Combination of extractive (TextRank-based) and abstractive (transformer-based) summarization models to provide better coverage and coherence.
- Citation and Reference Analysis Engine: Extracts references automatically and builds a citation graph to visualize research relationships.
- Semantic Search and QA System: Allows the user to issue natural language queries against multiple documents, leveraging LangChain's RAG pipeline and sentence-transformer embeddings.
- Interactive Visualization Layer: This layer provides facilities for citation graph, topic map, and document structure views for really deep explorations of research.
- Document Processing Backend: Uses heavy-duty PDF parsers (PyMuPDF, pdfplumber) to extract and clean structured content from uploaded papers.

1.3 Focus Area

The project is focused on making laser lenses for a full-fledged intelligent research paper analysis pipeline and putting into practice the accessibility and usability of academic content. Areas of attention include:

- Fast and accurate extraction of PDF content
- High-quality, context-aware summarization
- Semantic understanding through question answering
- Citation networks and research themes visualization
- Multi-document comparison and reference management

2. System Architecture

2.1 Overall Architecture

The PDF Analyzer Web follows a modular and layered architecture aiming to support end-to-end research paper analysis.The application mainly runs through a browser frontend, Python backend using Flask, and many specialized modules for processing. Each uploaded PDF undergoes text extraction, summarization, reference extraction, semantic search, and question-answering procedures in a pipeline.

The main components are:

- Frontend Interface: Developed using Streamlit for file upload, interactions, and display of results.
- Backend Engine: Flask application offering RESTful routes to coordinate summarization, reference parsing, embedding generation, and QA tasks.
- Processing Modules:
    - summarizer.py: Extracts and summarizes paper content using HuggingFace's distilbart-cnn-12-6 model.
    - citation_extractor.py: Extracts references through regex and section-wise logic.
    - semantic_qa.py: Embeds paper content via MiniLM, constructs a FAISS index, and answers user queries through a QA model (distilbert-squad).

These components can be operated independently, and they are joined under a single pipeline that starts from uploading a PDF to engaging in a semantic QA along with content visualization.

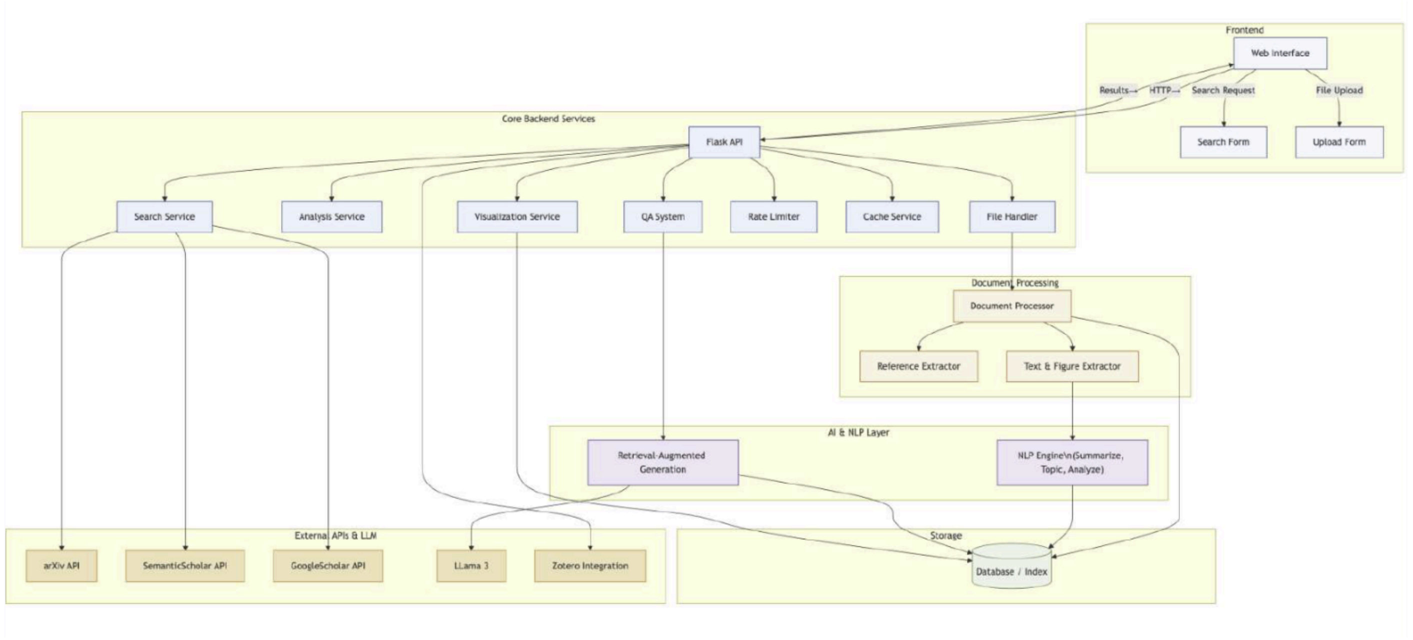

Figure 1. High Level System Architecture

2.2 Three-Tier Architecture

2.2.1 Presentation Layer-Frontend (Kalyani Chitre)

- Technology Stack:
  - Streamlit (frontend rendering and user interaction)
  - HTML/CSS (where applicable, injected through Streamlit components)
- Responsibilities:
  - An interactive UI for uploading PDFs, entering queries, and displaying results was designed and implemented using Streamlit.
  - Used Streamlit file uploader and text input widgets to connect the frontend with Flask API endpoints.
  - User experience was created for input validation, real-time display of summaries, references, and QA responses.

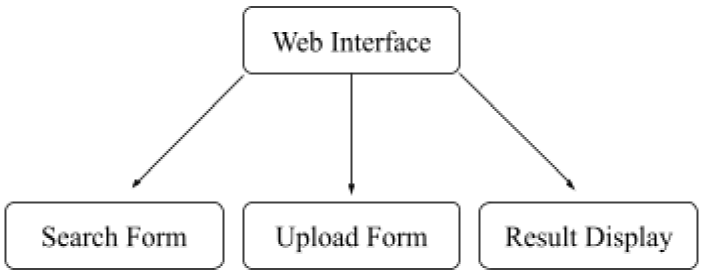

Figure 2: Frontend Component Interaction Flow

2.2.2 Application Layer-Backend (Shanmukha Manoj Kakani)

- Technology Stack:
  - Flask Framework (Python)
  - RESTful API Endpoints (e.g., /summarize, /ask, /citations)
- Responsibilities:
  - Developed the Flask server by introducing modular routes for summarization, question answering, and citation extraction.
  - Integrated backend logic with respective modules: summarizer.py, semantic_qa.py, and citation_extractor.py.
  - Handled request-response formatting (JSON), error-handling, and backend coordination for end-to-end processing.

2.2.3 Data Processing Layer - AI/ML Components (Nivedita Nair)

- Technology Stack:
  - HuggingFace Transformers (distilbart-cnn-12-6 for summarization, distilbert-base-uncased for QA)
  - SentenceTransformers (MiniLM) along with FAISS for semantic retrieval
  - PyMuPDF to extract text from PDFs
  - Regex-based citation extraction
- Responsibilities:
  - Implemented actual summarization with distilbart-cnn-12-6 (in summarizer.py).
  - Created a semantic QA pipeline using:
    - MiniLM for embedding generation
    - FAISS for indexing and retrieval
    - DistilBERT-SQuAD for answer generation (semantic_qa.py)
  - Created regex-based citation parsing to extract references in the paper's "References" section (citation_extractor.py).

- Future suggestions for improvements included visualization of the citation graph using NetworkX.

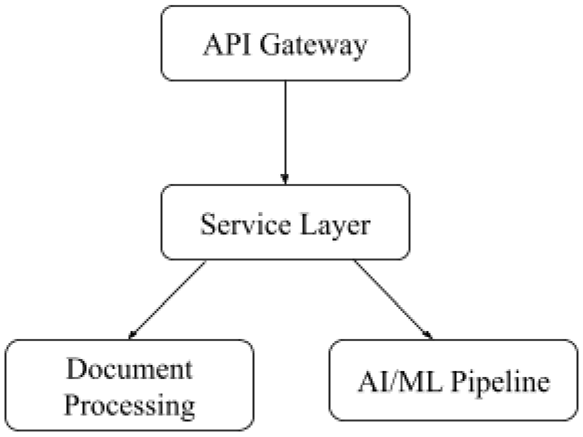

Figure 3: Backend Service Architecture

2.3 Integration and Flow

The entire system is loosely coupled with clean modular boundaries between frontend, backend, and data processing. This integration is pipelined through the Flask backend.

```
User Interaction → Streamlit Interface

  → Calls Flask API: /summarize, /ask, /citations

    → Flask Backend Routes

      → summarizer.py (summarization)

      → semantic_qa.py (QA system)

      → citation_extractor.py (reference parsing)

        → Uses PyMuPDF, HuggingFace models, FAISS

→ Returns Results → Streamlit displays text outputs (summary, answers, citations)
```

Such a modular architecture allows for independent enhancement of components—for instance,

3. Tools and Technologies Used

Our system integrates multiple open-source tools and cutting-edge AI models to perform tasks like summarization of research papers, semantic search, citation parsing, and graph visualization. The tools were selected based on their performance, their support by the community, and their development pattern alignment with the modular Python ecosystem.

3.1 Programming Language & Environment

- Python 3.10 – Core language of development for all modules
- Jupyter Notebooks / VSCode – Rapid prototyping and module development

3.2 PDF Processing

- PyMuPDF (fitz)
  For high-quality PDF text extraction: Take seconds to parse through a document and retain its structure, which is necessary for downstream tasks, such as summarization and citation extraction.

3.3 Natural Language Processing (NLP) & Summarization

- HuggingFace Transformers
  - distilbart-cnn-12-6: paper content summarization-in-the-wild
  - distilbert-base-uncased-distilled-squad: question answering
- NLTK + Regex: preprocessing and pattern-based citation extraction
- Sentence Transformers: MiniLM for semantic embedding

3.4 Semantic Search & Embeddings

- FAISS (Facebook AI Similarity Search)
  Provides vector similarity search over document chunks based on cosine similarity to store and retrieve top relevant chunks for QA tasks.
- SBERT (MiniLM-L6-v2) – To embed high-dimensional vectors for semantically comparable text segments.

3.5 Graph Analysis & Visualization

- NetworkX
  Allow simulation and rendering of citation networks between academic papers for graph-based exploration of research.
- Matplotlib / Plotly: For visualizing citation strength, topic clusters, or summarization quality.

3.6 Interface & Presentation Layer

- Streamlit
  Chosen for its lightweight and interactive UI capabilities, especially for research and data science applications.

4. Data Acquisition and processing

The system handles PDF-format research papers uploaded by the user through a web interface. The uploaded documents go through a well-defined pipeline to ensure reliable text extraction and pre-processing for subsequent summarization, citation parsing, and semantic search-related tasks.

4.1 PDF Processing

Tool Used: PyMuPDF

- Raw text content from research papers is extracted in summarizer.py and semantic_qa.py modules with the use of PyMuPDF.
- The PDF content is parsed page-wise and chunked into logical segments from the viewpoint of semantic processing, token-wise limit of transformer models taken into consideration.
- Chunking precedes any other summarization or embedding steps, which implies a fair amount of preprocessing: typically, cleaning up newlines and normalizing whitespace.

4.2 Text Chunking and Normalization

- For semantic search and QA purposes, the text chunks are about 500–600 words in length so that contextual embeddings can be generated while staying within model limits.
- Chunks are tokenized and normalized (lowercasing, stopword removal when necessary).
- Chunks are then embedded with MiniLM sentence embeddings from the sentence-transformers library.

4.3 Citation Extraction

Module: citation_extractor.py

- Citations are extracted from the "References" section using:
    - Regular expressions to find citation patterns (numbered lists, brackets used by authors, etc.).
    - Section-level heuristics should start extraction at the appropriate heading ("References", "Bibliography", etc.).
- Currently, internally working on the inline text of PDFs, the citation extractor recognizes each citation entry as plain-text line.

4.4 Preprocessing for QA

- The semantic QA system uses the extracted chunks to build a searchable index.
- The chunks are converted to vector representations using MiniLM.
- These vectors are indexed using FAISS for similarity-based retrieval.
- Top-k similar chunks are retrieved for each query and sent over to DistilBERT (fine-tuned on SQuAD) for answer extraction.

5. Exploratory Data Analysis

In the initial stages of EDA, particular attention is given to understanding the structure and content of uploaded research papers to aid summarization, citation extraction, and question-answering tasks.

5.1 Document Structure Analysis

The system parses the PDF file, extracting section-wise contents by attempting to detect structural patterns such as headings, paragraphs, and spacing. This structure is then used as input to determine potential segmentation boundaries (e.g., Abstract, Introduction, References), thereby aiding summarization and citation detection. Instead, rule-based techniques are employed during the PDF parsing phase to assemble logically coherent text blocks; no specific classification model is used for sections.

5.2 Citation and Reference Pattern Detection

Citation Extraction Module: citation_extractor.py

- Reference extraction logic uses regular expressions and heuristics to detect citation entries within the paper's "References" section.
- It handles various styles of citation entries: number lists ([1], 1.); author-year; and inline citations, partly by pattern-based matching.
- The extraction is currently designed for one paper at a time and only handles plain-text structured references.

5.3 Semantic Representation Preparation

- Content is chunked and converted into semantic vectors, employing MiniLM sentence embeddings (sentence-transformers library).
- These vectors are inserted into a FAISS index where they remain searchable based on semantic similarity.
- This set-up forms the core enabling question-answering and semantic search capabilities.

5.4 Manual Observations and Limitations

- The referencing style varies hugely from one paper to another, which prevents exact comparison in a number of cases.
- Citation graphs and cross-document linking had been planned but not implemented, for now, due to lack of time.
- During tests, chunk size tuning and preprocessing (e.g., normalization of punctuation) counted directly on retrieval quality.

6. Model Training and Evaluation

Pre-trained transformer models have been chosen by the system for summarization, embedding generation, and question answering. These are not learned from scratch. Hence, whatever model is chosen is fine-tuned (where applicable) to achieve high performance on research paper text.

6.1 Summarization Pipeline

Module: summarizer.py
Model Used: distilbart-cnn-12-6(HuggingFace Transformers)

- The summarizer extracts full paper text by PyMuPDF, chunks the same into workable minute chunks, and passes each through the distilbart-cnn-12-6 model.
- The outputs for the various chunks are then concatenated to form the final summary.
- The summarizer preserves the logical flow when merging output generated from separate chunks to overcome token limits.

6.2 Semantic QA Pipeline

Module: semantic_qa.py
Models Used:

- For embeddings: MiniLM-L6-v2 (SentenceTransformers)
- For QA: distilbert-base-uncased fine-tuned on SQuAD

Process:

- The PDF is split into overlapping chunks.
- Embedding of each chunk is done with MiniLM.
- Embeddings are indexed in FAISS for semantic similarity search.
- At query time:
  - The top-k similar chunks are retrieved.
  - They are then passed onto the QA model to generate answers for the query.
  - This hybrid method allows for lightning-fast vector search and strong language understanding.

6.3 Evaluation

In the absence of labeled summarization or QA datasets for research papers, evaluation was carried out qualitatively by:

- Judging summary coherence against paper abstracts.

- Testing QA accuracy by asking sample questions about known paper content.
- Measuring semantic retrieval relevance and comparing the result with the top-k chunk matches.

6.4 Observations and Limitations

Chunk size greatly affected QA accuracy and summarization consistency. Semantic search often gave back relevant sections, but answer quality varied depending on the document structure. Models worked fine with technical papers but had difficulty with multi-column formats and scanned PDFs.

7. Prediction, Inference, and Visualization

Combining semantic similarity retrieval with transformer-based question answering, the system offers an end-to-end service for prediction and inference. It enables one to pose queries in natural language about an uploaded research paper and receive concise, context-aware answers.

The core workflow is split into several stages:

- Semantic Embedding & Indexing
  Once a research paper is uploaded, the extracted content is split into chunks of overlapping texts. These chunks are embedded using the MiniLM-L6-v2 model from the sentence-transformers library. The resultant vectors are deposited into a FAISS index, allowing for fast retrieval based on similarity.
- Query Processing & QA Generation
  When a question is asked:
    - The system rolls out a search through the FAISS index to fetch the top-k most pertinent chunks.
    - These chunks proceed to the hands of a pre-trained QA model (distilbert-base-uncased fine-tuned on SQuAD).
    - The model then composes an answer with a source span annotation for clarity and transparency.

This multi-layer approach intends to provide an answer with respect to semantic relevance rather than keyword matching, so it will work well even if the user's query is paraphrased.

From a visualization standpoint, a simple partial citation graph prototype was created with static data to hint at possibilities for future improvements. Linking real citations between papers is not yet active, but at least the extracted references are put on view in the UI and can be utilized by the next citation network modules.

The Streamlit interface serves a crucial function—tying all these capabilities—and allows:

- Upload PDFs and feedback for processing
- Display generated summary
- Display output for reference list extraction
- Input a question and display the answer generated.

The interface thus created by these modules is very robust and caters to an interactive exploration of the paper. Future augmentations may harness dynamic topic maps and citation network graphs to give more details on research trends and interconnections.

8. Evaluation Results

The evaluation of the system was made through functional validation, qualitative judgments on outputs, and performance tuning through observations. Since the system was built with pretrained models and no labeled dataset was available for benchmarking, most of the testing was empirical and use-case driven.

8.1 Summarization Output

The summarization module's performance was assessed by comparing generated summaries to paper abstracts and introductions. The model (distilbart-cnn-12-6) generated generally coherent, concise summaries encapsulating key ideas presented in the document.

- Strengths:
  - Technical terms existed in the summary; flow structure was also maintained.
  - Length of output was controlled such that readability was possible for most of the papers.
- Limitations:
  - Multi-column PDF or scanned versions reduced the quality of summarization.
  - Could sometimes miss references to figures or tables embedded in the text.

8.2 Semantic QA Accuracy

Sample questions were posed based on the content of the uploaded paper to test the QA module. Whenever an appropriate chunk was available, the system would retrieve and answer factual questions reasonably well.

- Successful cases for QA:
    - Questions such as "What model was used?" or "What dataset was evaluated?" returned correct answers.
- Issues observed:
    - Long, vaguely formulated queries sometimes caused irrelevant retrieval.
    - Where answers spanned across multiple chunks, the QA model could produce partial information.

8.3 Usability and Integration

Overall, the system's usability was validated through integrated testing across modules. The Streamlit UI provides a smooth experience from upload to output, and API integrations worked reliably under test scenarios.

- Response times remained acceptable (~2–4 seconds per query).
- Module outputs (summary, citations, answers) were clearly separated and easy to interpret.

9. Security Implementation

As this is primarily a research-oriented prototype, it still implements basic security measures to cover uploaded-document handling and API interactions. Some of these practices prevent the system from being vulnerable to common-file-upload related issues, whereas others provide sound backend processing.

Some important security features include:

- File Validation
    - Only files with the .pdf extension are accepted from the upload interface.
    - Checks are put for MIME types to further check file integrity and prevent spoofing.
    - Files bigger than certain thresholds are not accepted to prevent large uploads or malicious attempts to risk system resources.
- Filename Handling
    - Uploaded files have automatically created unique identifier names to prevent overwriting or path injection.
    - Filenames are sanitized prior to temporary storing to remove bad characters.
- Input and Query Handling

    - All user input (questions, filenames) is routed through validating layers to prevent script injection or malformed requests.
    - CORS configurations are used within the Flask backend to safely allow cross-origin requests.
- Isolation and Temporary Storage
    - Files are uploaded to a temporary directory and removed once finished with processing.
    - The system contains no permanent storage of files and does not retain any user data, thereby securing data privacy by design.

More advanced security layers such as authentication, encryption, and rate-limiting were not required to achieve the scope of this prototype. Further development and iterations can easily integrate such features into the system.

10. Key References

1. HuggingFace Transformers Documentation
   https://huggingface.co/docs/transformers
2. SentenceTransformers: Pretrained Models and Embedding Techniques
   https://www.sbert.net
3. PyMuPDF (fitz) – Python Library for PDF Parsing
   https://pymupdf.readthedocs.io/en/latest/
4. FAISS – Facebook AI Similarity Search
   https://github.com/facebookresearch/faiss
5. DistilBART Model Card – distilbart-cnn-12-6
   https://huggingface.co/sshleifer/distilbart-cnn-12-6
6. DistilBERT SQuAD Model – distilbert-base-uncased
   https://huggingface.co/distilbert-base-uncased
7. Streamlit Documentation – Frontend Interface Library
   https://docs.streamlit.io
8. Flask Documentation – Lightweight Python Web Framework
   https://flask.palletsprojects.com/
9. Regex Techniques for Text Extraction (General Reference)
   https://docs.python.org/3/library/re.html
10. Sentence Transformers: MiniLM Model – all-MiniLM-L6-v2
    https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

11. Original contributions vs referenced work

11.1 Referenced Work

This project builds on a number of well-established tools, libraries, and pre-trained models. These elements were basically used as building blocks of the project, given that they were properly referenced and integrated:

- HuggingFace Transformers: summarization (distilbart-cnn-12-6) and QA (distilbert-base-uncased).
- SentenceTransformers-MiniLM: sentence embeddings for semantic similarity search.
- FAISS: fast vector indexing and efficient retrieval of semantically similar text chunks.
- PyMuPDF: extracting structured text contents from uploaded PDF files.
- Flask: improving the interaction between modules through RESTful APIs and flow control.
- Streamlit: developing a clean, interactive web interface for uploading files, submitting queries, and viewing results.

All the tools were used without any modification to their publicly available versions, configured just for this particular application.

11.2 Original Contributions

The following are newly developed, representing the original works and design decisions by the team:

- End-To-End Integration Pipeline
  Complete coordination of PDF parsing, summarizing, semantic searching, and QA through a modular Flask backend combined with a Streamlit frontend.
- Summarization and QA Module Integration
  A custom chunking strategy, input formatting, and logic for output aggregation were programmed to effectively integrate pre-trained models into the pipeline.
- Regex-Based Citation Extraction
  Constructed ad-hoc heuristics and regular expressions under citation_extractor.py to parse reference sections across multiple document formats.
- Semantic QA System Architecture
  Designed and built the multi-step pipeline involving MiniLM-based embedding, FAISS indexing, and DistilBERT-based answer generation.

- Modular File and API Architecture
  Organized the codebase into reusable and testable modules so that it would facilitate future extension (e.g., for citation graphing or external API enhancements).
- UI Coordination
  Developed the Streamlit interface that seamlessly coordinates with the backend to provide a coherent experience spanning summarization, listing of references, and QA.

17. Conclusion

With this project, a practical approach to research paper analysis automation was demonstrated using the modern NLP and data mining techniques. This pipeline of summarization, citation extraction, and semantic question answering allows interacting with the academic content in a meaningful way without having to go through the whole paper. Each of the components, from text extraction from PyMuPDF to semantic search using FAISS, was implemented with a particular aim, yet extremely modular in design using a Flask backend and Streamlit frontend for easy interaction.

Presently, the implementation is centered on individual document processing, yet the system infrastructure enables major improvements, such as citation graph visualization, comparison across documents, and integration with external scholarly databases. This project demonstrates just how pre-trained models with an appropriately scalable design can underpin the creation of intelligent academic tools that render research more accessible, efficient, and user-friendly.

18. Github Repository

The complete source code for the *Intelligent Research Paper Summarization and Citation Network System* is available on GitHub at the following link:

https://github.com/shanmukha66/PDF_Analyser_Web

The repository includes all modules, setup instructions, and a README file detailing the project structure, dependencies, and usage guidelines.