SMARTBRIDGE
Let's Bridge the Gap

Smart Internz

StudyMate: An AI-Powered PDF-Based Q&A System for Students

1. Project Description

StudyMate is an AI-powered academic assistant that enables students to interact with their study materials—such as textbooks, lecture notes, and research papers—in a conversational, question-answering format. Instead of passively reading large PDF documents or relying on manual searches for specific information, users can simply upload one or more PDFs and ask natural-language questions. StudyMate responds with direct, well-contextualized answers, referenced from the source content.

The system is built using Python and integrates several core technologies:

- PyMuPDF for accurate and efficient text extraction from PDF files.
- SentenceTransformers + FAISS for embedding and retrieving semantically similar text chunks based on user queries.
- IBM Watsonx LLM—specifically the Mixtral-8x7B-Instruct model—for generating contextual and factual answers grounded in the uploaded content.
- Streamlit for developing a visually polished, interactive interface that runs locally.

Users can drag and drop multiple academic PDFs into the interface. Once uploaded, the system preprocesses the content by splitting it into overlapping text chunks and building a semantic index. When a question is submitted, StudyMate retrieves the most relevant chunks using FAISS and feeds them, along with the query, into the LLM. The answer is returned in real-time and displayed alongside source references and a downloadable Q&A log.

This project demonstrates the power of Retrieval-Augmented Generation (RAG) in an educational context and provides a practical, student-centric use case for large language models. By combining real-time document parsing, semantic search, and generative AI, StudyMate creates an intelligent and efficient study companion that turns static academic content into an interactive learning experience.

2. User Scenarios

StudyMate is designed to support a variety of realistic academic use cases where students interact with digital study materials. The system offers a seamless pipeline, from PDF upload to contextual answer generation, making it highly suitable for self-paced learning, exam revision, and quick concept clarification. Below are key usage scenarios that demonstrate the flexibility and relevance of StudyMate in a student's workflow:

Scenario 1: Concept Clarification During Study

A student revising for an upcoming exam uploads their lecture notes in PDF format. They want to revisit a difficult concept—such as "What is overfitting in machine learning?"

- The student types this question into the query input box.
- StudyMate retrieves the most relevant explanations from the PDF.
- The LLM generates a clear, concise definition grounded in the extracted content.
- The source paragraph is shown below the answer for verification.

This enables the student to get accurate help instantly without breaking their concentration to skim through pages.

Scenario 2: Studying from Digital Textbooks

While reading a digital textbook like *"Introduction to Machine Learning with Python"*, the student encounters a term they don't fully understand.

- They upload the full textbook PDF into StudyMate.
- Instead of switching to Google or ChatGPT separately, they ask:
  "Explain the difference between classification and regression."
- The system delivers an answer directly from the book, with references.

This scenario enhances learning by providing answers strictly grounded in the student's own material.

Scenario 3: Preparing for Viva or Open-Book Tests

A user preparing for a viva exam wants to quiz themselves on key concepts.

- They upload a PDF of summarized notes.

- Then ask multiple questions like:
    - *“What is principal component analysis?”*
    - *“What is the use of sklearn in Python?”*
- StudyMate logs all questions and answers in a session history.

Before the test, the user downloads the Q&A history as a text file for quick revision.

Scenario 4: Multi-PDF Research Compilation

A research student has three papers on a specific topic.

- They upload all three PDFs into StudyMate at once.
- Then ask:
  “What do these papers say about ensemble methods?”
- StudyMate finds the most relevant excerpts across documents and synthesizes the response.

This cross-PDF reasoning allows for research-wide concept extraction and thematic learning.

3. Technical Architecture

StudyMate is built on a modular, pipeline-based architecture designed to enable fast, scalable, and traceable question answering over academic PDFs. The architecture consists of six primary layers, each fulfilling a distinct role—from document ingestion to AI response generation and UI presentation.

Input Layer:

- **PDF Upload:**
  Users can upload one or more PDF files through a drag-and-drop uploader in the Streamlit interface. Uploaded files are read and parsed using PyMuPDF.
- **Text Extraction & Chunking:**
  Each PDF is converted into clean, readable text. This text is then split into overlapping chunks (typically 500 words with 100-word overlap) to ensure context continuity during retrieval.

Semantic Retrieval Layer:

- **Embedding Model:**
  StudyMate uses the all-MiniLM-L6-v2 model from SentenceTransformers to convert each chunk and the user's query into semantic vectors.
- **Vector Indexing with FAISS:**
  The vectors are stored in a FAISS index for fast similarity search. When a user asks a question, FAISS retrieves the top-k most semantically similar chunks (typically k = 3).

LLM Inference Layer:

- **Prompt Construction:**
  A dynamic prompt is assembled by combining the user's question with the retrieved context chunks.
- **Watsonx Integration:**
  The prompt is sent to the IBM Watsonx LLM API using the ibm-watsonx-ai SDK.
  - **Model Used:** mistralai/mixtral-8x7b-instruct-v01
  - **Settings:** Greedy decoding, moderate temperature, and a token limit of 300
- **Response Handling:**
  The model returns a concise, contextual answer. The response is linked with the original chunks for source traceability.

Data Persistence Layer:

- **Session History Tracking:**
  All questions and corresponding answers are logged in memory per session.
- **Export Functionality:**
  The Q&A log can be downloaded as a .txt file. Future versions may support CSV or Markdown exports.

Frontend & UI Layer (Streamlit)

- **Modern Interface:**
  The interface is styled with a custom background and soft-themed containers.
- **Components Include:**
  - PDF Uploader (multi-file support)
  - Text input for asking questions
  - Answer card with clear formatting
  - Expandable "Referenced Paragraphs" section
  - Session-wide Q&A history
  - One-click download for answer logs
- **Live Feedback:**
  Status banners show upload success, readiness, or failure in real time.

Configuration & Security Layer

- **Environment Variables (.env):**
  - IBM_API_KEY
  - IBM_PROJECT_ID
  - IBM_URL
    These are securely loaded using python-dotenv and passed to the Watsonx client.
- **Model Switching:**
  The system can easily be adapted to work with other IBM models like Granite, FLAN-T5, or LLaMA by updating a single model ID in the backend.

4. Pre-requisites

- Python Programming: https://docs.python.org/3/
- Streamlit Framework: https://docs.streamlit.io/
- IBM Watsonx.ai (Mixtral Foundation Model): https://www.ibm.com/cloud/watsonx
- IBM Watson Machine Learning SDK (Python): https://ibm.github.io/watson-machine-learning-sdk/foundation_models/
- Sentence Transformers (Text Embedding): https://www.sbert.net/
- FAISS CPU Library (Semantic Retrieval):https://github.com/facebookresearch/faiss
- PyMuPDF for PDF Parsing: https://pymupdf.readthedocs.io/
- Python Dotenv for Environment Variables: https://pypi.org/project/python-dotenv/
- Local Development Environment: Visual Studio Code (https://code.visualstudio.com/), OS: Windows 11, Python 3.10.x
- Command Line Execution: All components are executed locally using the terminal

5. Project Workflow

The development of the StudyMate system was organized into three key milestones, each focused on implementing and validating a specific subsystem. The methodology emphasized modular code design, iterative testing, and end-to-end integration of all major components including the user interface, retrieval engine, and IBM Watsonx LLM backend.

Milestone 1: PDF Parsing and Chunk Preparation

This milestone focused on building a robust text extraction and chunking pipeline to convert unstructured academic PDFs into semantically meaningful units ready for embedding and retrieval. The process accounted for document diversity—ranging from textbooks and lecture notes to research articles—and prioritized the retention of contextual coherence during segmentation.

Activity 1.1 – Text Extraction from PDFs

The system leveraged the PyMuPDF library (imported as fitz) for extracting text from PDF files. This library provided page-level access to all visible text elements while preserving paragraph breaks and reading order. During development, multiple formatting inconsistencies were encountered across PDF types—such as split headers, repeated footers, and column layouts. These were manually analyzed and heuristically handled by normalizing the extracted string. Each file was opened as a stream, processed page-by-page, and concatenated into a master document buffer. This stage ensured that the text used for downstream semantic search was clean and minimally noisy.

Activity 1.2 – Chunk Segmentation for Retrieval-Augmented Generation

Once extracted, the document content was segmented into overlapping text chunks to be used as retrievable units. The chosen configuration was 500 words per chunk with a 100-word overlap. This overlap was critical to maintaining context continuity across boundaries—especially in academic material where explanations or equations span multiple paragraphs. The segmentation algorithm iterated through the tokenized word list with a sliding window approach and appended all generated segments to a list for embedding. Each chunk was stored alongside metadata (source filename, chunk number) for future traceability.

Activity 1.3 – Multi-PDF Aggregation and Dynamic Index Handling

In addition to single-document support, the system was extended to handle multiple PDFs uploaded together. Each file was independently processed through the extraction and chunking pipeline, and all resultant chunks were merged into a single corpus. This design allowed the user to ask a question across all uploaded documents simultaneously—enabling multi-source semantic retrieval. The FAISS index was then constructed over the aggregated embeddings, with internal references maintained to map retrieved chunks back to their original document context.

Activity 1.4 – Intermediate Testing and Debugging

To verify the integrity of the chunking process, intermediate outputs were printed and manually checked for logical completeness. Key focus areas included sentence cutoffs at boundaries, duplicate content due to overlapping windows, and retention of mathematical or symbolic notations. Representative examples from both technical and non-technical documents were tested to ensure generalizability.

Milestone 2: Embedding, Indexing, and Retrieval

This milestone was dedicated to building the core retrieval engine responsible for connecting a user's question to the most semantically relevant sections of the uploaded PDFs. It involved transforming text into vector representations, indexing them efficiently, and retrieving relevant chunks using vector similarity methods.

Activity 2.1 – Embedding Generation using Sentence Transformers

The sentence-transformers library was used to embed each chunk into a fixed-size vector. The selected model, all-MiniLM-L6-v2, provided a good tradeoff between performance and embedding quality. It generates 384-dimensional embeddings optimized for semantic similarity tasks. The model was preloaded and cached to accelerate repeated inference. Each text chunk was passed through this encoder, and the output vectors were appended to a NumPy array representing the full document embedding matrix. Testing revealed that even short academic paragraphs were effectively captured by the model, with conceptually similar chunks having noticeably closer vector distances.

Activity 2.2 – FAISS Index Construction and Optimization

Once embeddings were generated, they were indexed using Facebook AI Similarity Search (FAISS). A flat L2 index (IndexFlatL2) was used for its simplicity and effectiveness in CPU-only environments. Each embedding vector was inserted into the index alongside a reference to its original chunk. The index was serialized and loaded into memory at runtime for fast access. The cosine similarity scores were computed internally during queries, enabling top-k retrieval of contextually similar segments. Retrieval performance was tested with varying k-values (k=3, 5, 10), and k=3 was selected as optimal for balancing relevance and token length in the prompt.

Activity 2.3 – Query Embedding and Chunk Retrieval

When a user submitted a question, the same embedding model was used to convert the question into a vector. This query vector was compared against the FAISS index, and the closest chunks were returned. The retrieval logic included score normalization and chunk de-duplication across files. Each result was returned in ranked order and included metadata for display and prompt formatting. This approach ensured that only the most relevant, high-confidence segments were passed to the language model.

Activity 2.4 – Relevance Validation and Performance Tuning

Retrieved chunks were manually examined against various academic queries to validate their appropriateness. Questions like "What is overfitting?", "Explain classification vs regression", and "List evaluation metrics in ML" were tested. The returned context was consistently aligned with the intent of the query, demonstrating strong semantic matching. To reduce latency in multi-document indexing, redundant chunk filtering and embedding reuse were implemented in later iterations.

Milestone 3: Watsonx LLM Integration and Prompt Construction

This milestone focused on the generation of final answers using IBM Watsonx foundation models. It covered model selection, credential setup, prompt formatting, and response handling. The objective was to ensure that the answers generated were coherent, relevant, and contextually grounded in the content retrieved from the uploaded PDFs.

Activity 3.1 – IBM Credential Setup and Environment Configuration

To securely interact with IBM Watsonx APIs, environment variables were stored in a .env file. These included IBM_API_KEY, IBM_PROJECT_ID, and IBM_URL. The python-dotenv package was used to load these values during runtime. The credentials were passed into the Model object from the ibm-watsonx-ai SDK, which initialized the LLM client. The endpoint URL was configured to match the regional deployment of the Watsonx instance (e.g., us-south.ml.cloud.ibm.com). This setup enabled authenticated and persistent access to the foundation model.

Activity 3.2 – Prompt Construction using Retrieved Chunks

The prompt fed to the LLM was dynamically assembled using the retrieved top-k text chunks and the user's question. The prompt followed a consistent format:

- A system message that framed the task ("Answer based strictly on the following context").
- Each chunk presented as a bullet point or paragraph block.
- The user question appended at the end.
- A final instruction to avoid hallucination or unsupported claims.

The formatting emphasized clarity and token efficiency. Each prompt was encoded and validated to ensure it remained under the model's token limit (typically 2048 tokens), with the most relevant chunks prioritized if truncation was needed.

Activity 3.3 – Model Invocation and Parameter Tuning

The chosen model was mistralai/mixtral-8x7b-instruct-v01, hosted via IBM Watsonx.ai. This model was selected for its strong performance on open-ended question answering, low latency, and support for factual reasoning. The generation parameters were:

- Max new tokens: 300
- Decoding method: Greedy (no sampling)
- Temperature: 0.5 (for mild variation while retaining factual tone)

The model call was wrapped in exception-handling logic to manage failures, timeouts, or misformatted responses. Returned answers were decoded, stripped of extraneous whitespace, and stored alongside their corresponding queries.

Activity 3.4 – Output Structuring and Traceability

The final LLM answer was displayed in the user interface, accompanied by the retrieved chunks used for context. This design allowed users to verify the grounding of the response. Additionally, the answer was appended to a session dictionary for persistent storage and later export.

Testing was performed using sample questions from the “Introduction to Machine Learning with Python” textbook, confirming that the system produced clear, accurate, and referenced answers across various topics.

Milestone 4: Streamlit Interface Development and Session Handling

This milestone involved the implementation of the front-end interface for the StudyMate system using Streamlit. The goal was to create a clean, intuitive, and responsive UI that would allow users to upload documents, ask questions, view AI-generated answers, and download their full interaction history. Emphasis was placed on minimalism, accessibility, and real-time responsiveness.

Activity 4.1 – UI Layout Design and Component Structuring

The application was developed using Streamlit's layout primitives such as containers, columns, and expanders. The set_page_config() function was used at the start to define the page title, layout width, and favicon. The top section of the app featured a background image embedded using base64 encoding, which was styled via inline HTML and CSS to create a fixed, scrollable effect. The central interaction area included the following components:

- PDF uploader (st.file_uploader) with multi-file support
- Question input text box
- Submit button for triggering query execution
- Answer display area
- Reference chunk viewer in expandable form

Each component was arranged vertically, with logical groupings for input, output, and interaction history.

Activity 4.2 – Answer Display and Context Tracing

After the user submitted a question, the interface displayed the AI-generated answer in a clearly styled response block. The answer was rendered using Markdown for readability, with font adjustments for visual emphasis. Below the answer, an expandable section revealed the original text chunks used in generating the response. This traceability was critical in academic settings, ensuring users could verify the origin and relevance of every claim made by the model. Additionally, fallback messages were displayed if retrieval or generation failed, improving user feedback and error transparency.

Activity 4.3 – Session-Based Q&A History Tracking

To maintain continuity during a session, every question and corresponding answer was stored in a session_state variable. These entries were displayed sequentially at the bottom of the interface under a "Q&A History" header. Each history block was rendered with distinct styling to separate the question from its answer, preserving readability. This enabled users to scroll back through their previous queries without re-entering them.

Activity 4.4 – Downloadable Transcript Functionality

A dedicated button was provided to download the session's Q&A history as a plain text file. When clicked, the session dictionary was serialized into a structured .txt format with timestamps and delimiters between question-answer pairs. This feature was especially useful for students who wished to save the session for offline revision or share it with peers. The implementation used Streamlit's st.download_button, with data formatting handled in-memory to eliminate the need for filesystem access.

Activity 4.5 – Final Integration and Live Testing

All components—backend pipeline, LLM engine, retrieval index, and frontend—were integrated and tested as a cohesive system. Edge cases such as blank input, multiple simultaneous uploads, and long answers were tested to ensure resilience. The application was run on Windows 11 using the terminal command streamlit run streamlit_app.py and consistently produced accurate, grounded answers from academic PDFs across domains.

6. Project Output

The final implementation of the StudyMate system results in a fully operational, user-friendly interface that enables real-time interaction with academic PDFs using natural language queries. The project output spans multiple dimensions—interface usability, backend responsiveness, semantic accuracy, and session-level continuity. This section documents the output interface and features with reference to actual use cases and interface screenshots provided during testing.

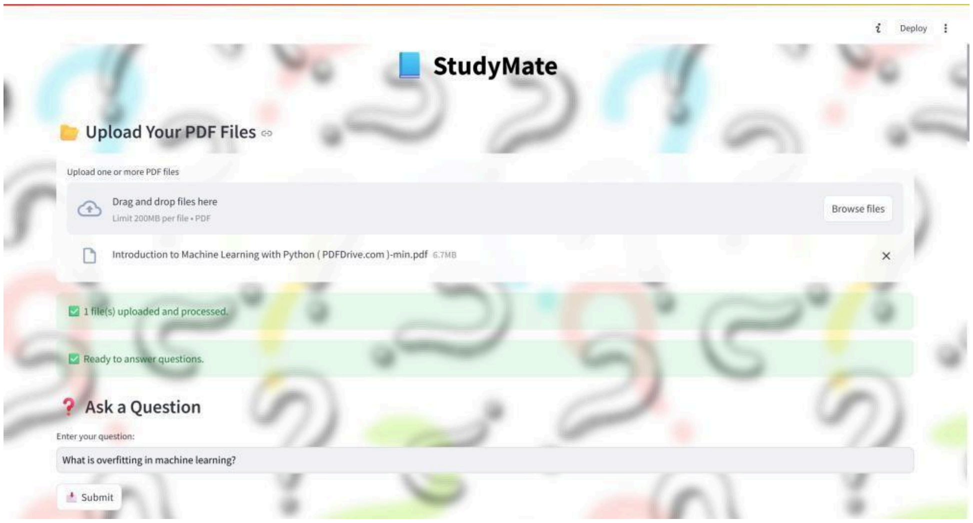
Component 1: User Interface Layout and Aesthetic

The application interface is rendered using the Streamlit framework with a custom layout and background image. The homepage is titled “H]_ StudyMate”, and includes a fixed, wide layout with a clean arrangement of inputs and outputs. Visual elements such as buttons, headers, input boxes, and scrollable sections are styled using embedded CSS to provide a modern look while maintaining readability.

The interface is divided into four major sections:

- Header area: Project title and branding
- PDF uploader section: File input area supporting drag-and-drop
- Query interaction section: Text input, submit button, and generated response
- Session history section: Scrollable log of all Q&A interactions

The layout responds well to various screen sizes and operating systems, and all user interactions are processed locally without redirection or page reloads.



StudyMate

Upload Your PDF Files

Upload one or more PDF files

Drag and drop files here
Limit 200MB per file • PDF

Introduction to Machine Learning with Python (PDFDrive.com)-min.pdf 6.7MB

☑ 1 file(s) uploaded and processed.

☑ Ready to answer questions.

Ask a Question

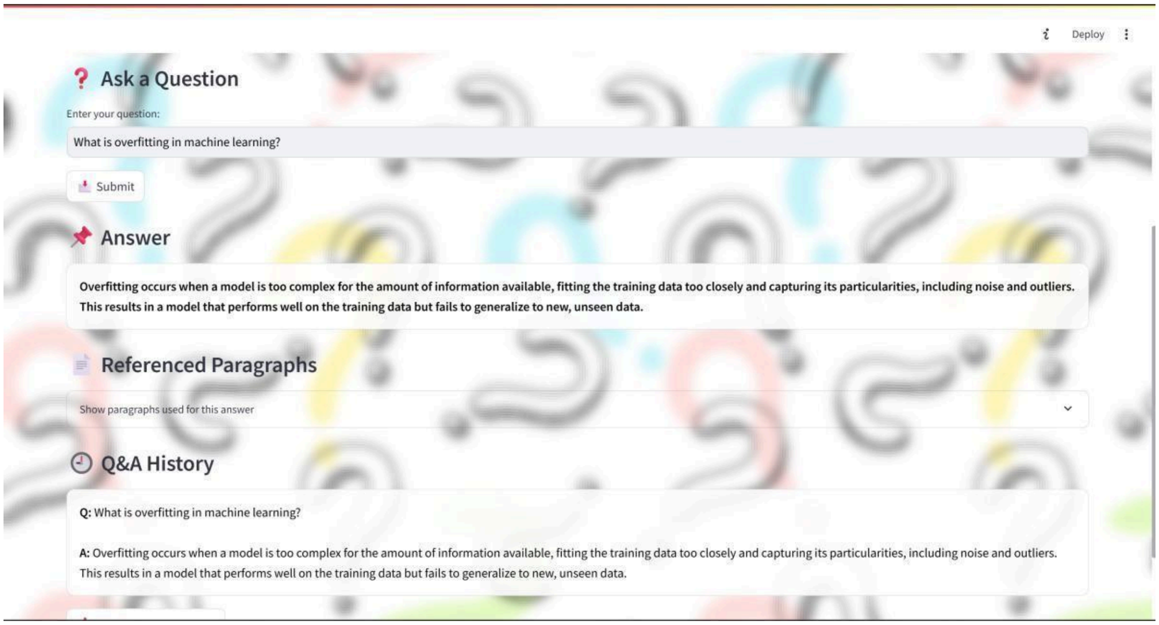
Enter your question:
What is overfitting in machine learning?

Component 2: Multi-PDF Upload and Preprocessing Pipeline

Users are provided with a file uploader widget that supports uploading multiple academic PDFs in a single session. Upon upload:

- The content from each file is extracted using PyMuPDF and processed sequentially.
- Each document is split into overlapping chunks of approximately 500 words with 100-word overlap to preserve context flow.
- All chunks across all PDFs are stored in a unified list, embedded into dense vectors, and indexed using FAISS.

A status message confirms successful upload and processing, and the backend logic ensures that duplicate files or invalid formats are rejected gracefully. This feature supports seamless integration of study notes, textbooks, and research papers into a single searchable knowledge base.



❓ Ask a Question

Enter your question:

What is overfitting in machine learning?

📌 Answer

Overfitting occurs when a model is too complex for the amount of information available, fitting the training data too closely and capturing its particularities, including noise and outliers. This results in a model that performs well on the training data but fails to generalize to new, unseen data.

Referenced Paragraphs

Show paragraphs used for this answer

Q&A History

Q: What is overfitting in machine learning?

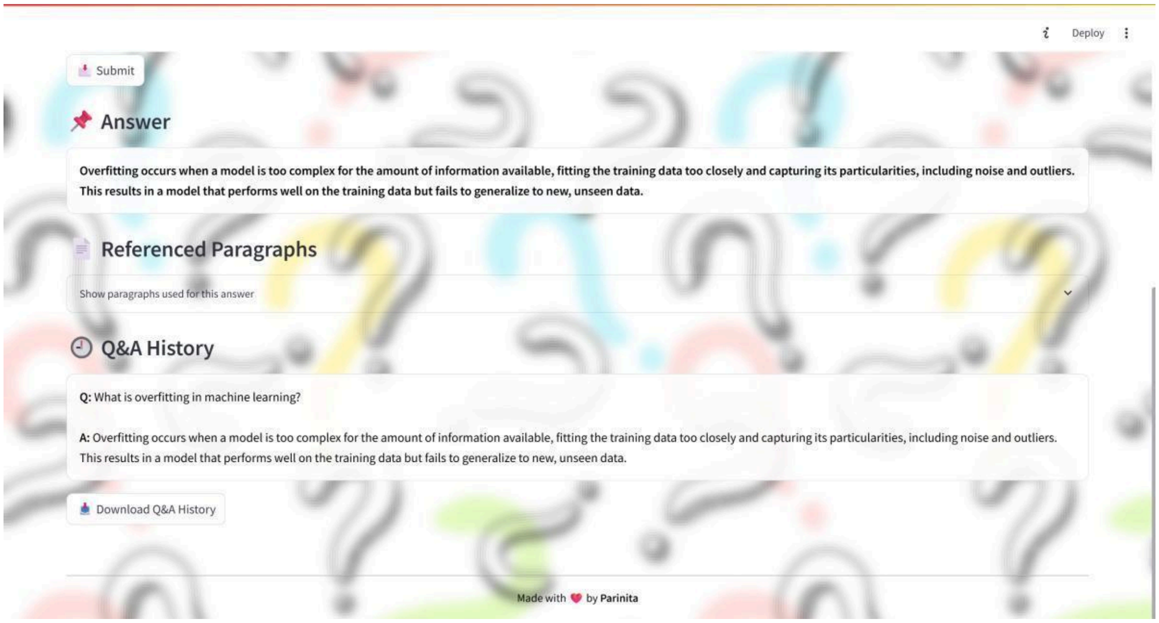
A: Overfitting occurs when a model is too complex for the amount of information available, fitting the training data too closely and capturing its particularities, including noise and outliers. This results in a model that performs well on the training data but fails to generalize to new, unseen data.

Component 3: Question Submission and AI-Generated Answers

The central functionality of StudyMate revolves around its intelligent question-answering interface. A text input box allows the user to ask any question related to the uploaded content. Once submitted:

- The question is embedded into a vector using the same model as the chunks (all-MiniLM-L6-v2).
- FAISS returns the top three most semantically relevant chunks.
- These are inserted into a pre-defined prompt template, followed by the user's question.
- The prompt is submitted to the Watsonx Mixtral-8x7B-Instruct model via the IBM API.
- The model responds with a structured, context-aware answer.

The generated answer is rendered within a styled response card. Below the answer, the user can expand a section labeled "Referenced Paragraphs" to view the exact source chunks used for that specific question. This feature ensures that all AI-generated output is verifiable and aligned with the uploaded content.



Overfitting occurs when a model is too complex for the amount of information available, fitting the training data too closely and capturing its particularities, including noise and outliers. This results in a model that performs well on the training data but fails to generalize to new, unseen data.

Q: What is overfitting in machine learning?

A: Overfitting occurs when a model is too complex for the amount of information available, fitting the training data too closely and capturing its particularities, including noise and outliers. This results in a model that performs well on the training data but fails to generalize to new, unseen data.

Component 4: Live Session History Tracking and Downloadable Transcript

In addition to displaying real-time answers, the system maintains a complete record of all interactions within a session. Every user question and its corresponding AI response is stored in memory and rendered at the bottom of the interface in reverse chronological order. Each entry is displayed in a collapsible format with distinct styling for questions and answers.

This session history is particularly useful for revision and review. At any point, the user can click the “Download Q&A History” button to export the full conversation as a plain text file. The exported file includes clear delimiters between each Q&A pair and is compatible with any text editor. This functionality simulates a personalized AI tutor log, making the system valuable not just for real-time assistance but also for long-term study support.

Component 5: System Responsiveness and Stability

The full pipeline—from PDF upload to answer generation—was tested on a local development environment (Windows 11, Python 3.10) and demonstrated consistent responsiveness. Latency from question submission to answer display averaged under five seconds, depending on input length and model load. Error handling routines were implemented for cases such as:

- Missing PDFs
- Empty questions
- Model timeout or invalid responses
- Retrieval failure due to inadequate content

Each of these cases triggered appropriate fallback messages or warnings, thereby improving system robustness and user trust.

7. Conclusion

The StudyMate project successfully demonstrates the practical application of Retrieval-Augmented Generation (RAG) in the educational domain. It provides a seamless way for students to interact with static academic PDFs by converting them into dynamic, queryable knowledge sources. Through the integration of semantic search, prompt engineering, and large language models, the system transforms traditional reading into an engaging and efficient learning experience.

From a technical standpoint, the project integrates multiple AI components—including PyMuPDF for parsing, SentenceTransformers and FAISS for retrieval, and IBM Watsonx foundation models for answer generation—within a clean and modular pipeline. Each subsystem was individually validated and collectively integrated into a user-friendly Streamlit application. The inclusion of session history, downloadable transcripts, and reference tracking makes the tool highly usable for real-world academic study and exam preparation.

During development, particular attention was given to the accuracy, traceability, and transparency of the responses generated. The system ensures that every answer is grounded in the content of the uploaded documents and verifiable through reference paragraphs. This design choice positions StudyMate as a responsible and academically appropriate use of generative AI.

StudyMate represents a strong foundation for further development. Potential future enhancements include support for table and diagram parsing, integration of OCR for image-based PDFs, confidence scoring for retrieved answers, and deployment as a web-based service for broader access.

In summary, StudyMate meets its core objective—to serve as an intelligent, AI-powered study assistant that allows students to engage with their learning material more effectively and efficiently. It stands as a practical, innovative, and extensible solution that leverages the strengths of generative AI to enhance education.