DOCUDIGEST

Ivan Kwetey

Machine Learning Final Project Report

Abstract

I built DocuDigest as a web-based application that leverages state-of-the-art natural language processing (NLP) techniques to help users better understand and interact with lengthy documents. It provides automatic summarization, intelligent question answering, and document comparison capabilities using modern transformer-based models. In this report, I outline my motivation, the design methodology, the machine learning techniques I used, and potential future directions for the project.

1. Introduction

In today's information-rich world, navigating large volumes of text can be time-consuming and cognitively demanding. To solve this, I created a smart interface that simplifies document analysis. I integrated frontend design with backend machine learning systems to extract, condense, and interpret information from uploaded PDF and TXT files.

2. Literature Review

The growing need for document understanding has led to the rise of numerous AI-based tools that attempt to automate summarization and information retrieval. DocuDigest builds on recent advances in Natural Language Processing (NLP) and distinguishes itself from existing tools through its integrated summarization, question answering, and document comparison features all within a sleek and intuitive user interface.

2.1 Related Applications and Tools

Tool	Description	Pros	Cons
SMMRY	A rule-based web summarizer that condenses text using sentence importance scoring.	Simple to use No login needed	Extractive only- Cannot process PDFs directly- No interactivity
Scholarcy	A paid academic summarization tool that highlights key points, extracts figures, and references.	Academic citation extraction- Clean summary cards	Limited free version- No Q&A or custom queries
QuillBot	Primarily a paraphrasing tool with a summarization feature for pasted text.	Paraphrasing included- Fast response	Word limit in free tier- No document upload
ChatGPT (with Plugins)	Can summarize and answer questions about documents via plugins or file uploads.	Conversational AI- Contextual responses	Requires setup & login- Less structured summaries- Not designed solely for document navigation
Humata.ai	AI assistant that allows document uploads for Q&A and summarization.	Interactive Q&A- Good semantic search	Paid plan limits usage- May struggle with long technical PDFs
SciSpace Copilot	Academic assistant that parses papers, answers questions, and provides explanations.	Great for research papers- Citation-aware Q&A	Not general-purpose- Not optimized for diverse formats or styles

2.2 What Makes DocuDigest Different?

Unlike many of the above tools that focus on either summarization or question answering, DocuDigest combines:

- Document Upload (PDF/TXT) support
- Abstractive Summarization
- Generative + Extractive Q&A
- Section-based semantic understanding
- Document-to-document comparison

All of this is delivered via a custom-built, lightweight interface without third-party API reliance or paywalls.

3. Problem Statement

Reading and extracting information from documents is inefficient without intelligent tools. Users often need to:

- Summarize long documents quickly
- Ask specific questions and get relevant answers
- Compare multiple texts for overlap or divergence

DocuDigest aims to address all three use cases within a user-friendly web interface.

4. Methodology

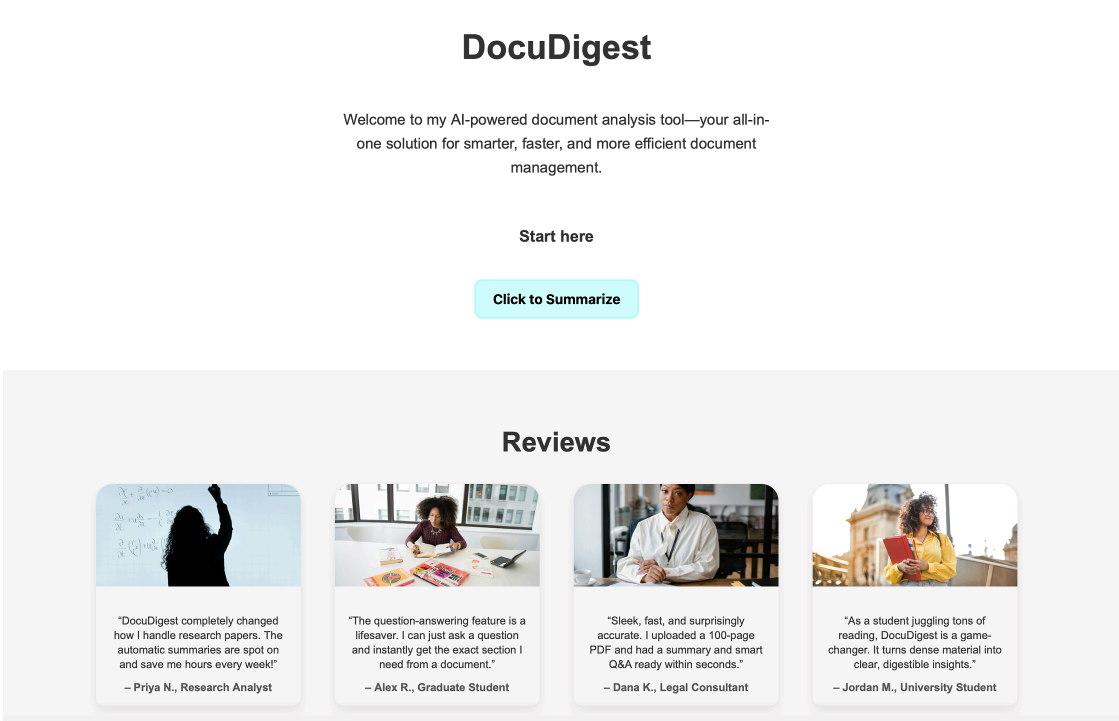
To ensure clarity, efficiency, and user value in every stage of DocuDigest's development, each methodological choice was guided by both technical feasibility and user-centric rationale. Below I explain not just what I did and how, but also why those choices were made.

4.1 System Architecture

I adopted a modular design to separate concerns and ensure maintainability across components:

Frontend:

Developed using HTML, CSS, and JavaScript. I employed Flexbox and media queries to ensure a responsive layout. Pages were kept minimal, with clearly defined upload, results, and interaction sections.



Backend:

A Flask server was implemented to manage the logic for handling file uploads, routing requests to ML models, and returning processed outputs. Uploaded documents are saved temporarily, and their content is parsed using Python libraries (e.g., PyMuPDF for PDFs).

Installations

In [3]:
```
!pip install PyMuPDF --quiet
!pip install transformers --quiet
!pip install transformers datasets --quiet
!pip install sentence-transformers --quiet
!pip install faiss-cpu --quiet
```

```
Requirement already satisfied: faiss-cpu in /opt/anaconda3/lib/python3.
12/site-packages (1.10.0)
Requirement already satisfied: numpy<3.0,>=1.25.0 in /opt/anaconda3/li
b/python3.12/site-packages (from faiss-cpu) (1.26.4)
Requirement already satisfied: packaging in /opt/anaconda3/lib/python3.
12/site-packages (from faiss-cpu) (24.1)
```

Imports

In [2]:
```
import os
os.environ["TOKENIZERS_PARALLELISM"] = "false"
os.environ["TRANSFORMERS_NO_TF"] = "1"
import json
import faiss
import torch
import evaluate
import re
import transformers
import nltk
import pandas as pd
import numpy as np
import fastapi
import flask
import fitz
import os
import pyttsx3
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
from transformers import pipeline
from transformers import AutoModelForQuestionAnswering
from sentence_transformers import SentenceTransformer, util
import joblib
from fpdf import FPDF
from sentence_transformers.util import cos_sim
import matplotlib.pyplot as plt
import evaluate
from tqdm import tqdm
```

In [3]:
```
from sentence_transformers import SentenceTransformer
```

```python
model = SentenceTransformer("all-MiniLM-L6-v2")
print("Model loaded!")
```

```
Model loaded!
```

TEXT TO SPEECH

To enable text-to-speech functionality in DocuDigest, I used the pyttsx3 library, which provides a simple interface for converting text into spoken audio. I initialized the speech engine using pyttsx3.init(), which sets up the system's default TTS engine. Then, I retrieved the list of available voice profiles by calling engine.getProperty('voices'). This gave me access to different voice options — such as male and female voices, or regional accents — depending on the system's installed speech engines (like SAPI5 on Windows or NSSpeechSynthesizer on macOS).

I looped through the available voices using enumerate(voices) to print each voice's index, name, and ID. This helped me preview the voice options available on my machine so I could select the most appropriate one for generating spoken summaries. This text-to-speech feature makes DocuDigest more accessible by allowing users to listen to summaries rather than just read them, which is especially helpful for multitasking or users with visual impairments.

In [ ]:

```python
engine = pyttsx3.init()
voices = engine.getProperty('voices')
for i, voice in enumerate(voices):
    print(f"{i}: {voice.name} - {voice.id}")
```

DocuDigest Model Initialization

In building the DocuDigestModel, I started by initializing several key components in the **init** method. This setup loads all the pre-trained models required for summarization (t5-small), extractive question answering (roberta-base-squad2), generative question answering (flan-t5-small), and semantic similarity detection using sentence embeddings (all-MiniLM-L6-v2). This makes the system ready to handle document processing tasks immediately after startup.

In [4]:

```python
class DocuDigestModel:
    def __init__(self):
        os.environ["TOKENIZERS_PARALLELISM"] = "false"
        os.environ["TRANSFORMERS_NO_TF"] = "1"

        self.sum_tokenizer = AutoTokenizer.from_pretrained("t5-small")
        self.sum_model = AutoModelForSeq2SeqLM.from_pretrained("t5-sma

        self.qa_tokenizer = AutoTokenizer.from_pretrained("deepset/rob
```

```python
        self.qa_model = AutoModelForQuestionAnswering.from_pretrained(

        self.gen_qa_tokenizer = AutoTokenizer.from_pretrained("google/
        self.gen_qa_model = AutoModelForSeq2SeqLM.from_pretrained("goo

        self.embedding_model = SentenceTransformer("all-MiniLM-L6-v2")

        print("All models loaded and ready.")
```

To begin analyzing a document, I use the read_pdf method, which extracts raw text from a PDF file using the PyMuPDF library. After text extraction, I clean the content using the clean_placeholders function, which removes bracketed placeholders (e.g., "[example]") and other unwanted formatting artifacts to ensure clean input for downstream models.

In [5]:
```python
    def read_pdf(self, file_path):
        doc = fitz.open(file_path)
        return " ".join([page.get_text() for page in doc])

    def clean_placeholders(self, text):
        return re.sub(r"\[.*?\]", "", text)
```

4.2 Summarization Pipeline

I selected the T5-small model for summarization because it balances performance and speed for a real-time web application. Abstractive summarization provides more natural and flexible summaries compared to extractive methods, better serving users who need contextual insights rather than just keyword highlights.

- I used the T5-small transformer model, loaded with Hugging Face's AutoTokenizer and AutoModelForSeq2SeqLM.
- Texts were preprocessed and truncated before passing through the summarizer.

For summarization, the summarize_text function takes in a cleaned block of text and generates an abstractive summary using the T5 model. This helps condense long-form documents into a concise overview. If I want to isolate specific sections like "Conclusion" or "Findings," I use the get_heading_candidates method to identify potential section headers, followed by extract_section, which pulls out that portion of the document based on headings and line limits.

In [6]:
```python
    def summarize_text(self, text, max_len=100, min_len=30):
        input_text = "summarize: " + text
        input_ids = self.sum_tokenizer.encode(input_text, return_tenso
```

```
        summary_ids = self.sum_model.generate(
            input_ids, max_length=max_len, min_length=min_len,
            length_penalty=2.0, num_beams=4, early_stopping=True
        )
        return self.sum_tokenizer.decode(summary_ids[0], skip_special_

    def get_heading_candidates(self, text):
        lines = text.splitlines()
        return list({line.strip() for line in lines if line.strip() an
                line.strip().isupper() or ':' in line or len(line.stri
        )})

    def extract_section(self, text, section_name, max_lines=40):
        lines = text.splitlines()
        headings = self.get_heading_candidates(text)
        section_text = []
        capture = False
        count = 0
        for line in lines:
            if section_name.lower() in line.lower():
                capture = True
                continue
            if capture:
                if line.strip() in headings and line.strip().lower() !
                    break
                section_text.append(line)
                count += 1
                if count >= max_lines:
                    break
        return "\n".join(section_text).strip()
```

4.3 Question Answering

HHaving both extractive and generative QA systems allows us to deliver high-confidence factual answers while also supporting complex queries where span-based answers may not suffice. RoBERTa is robust for fact-based retrieval, whereas FLAN-T5 enhances flexibility by generating human-readable responses in open-ended contexts.

- I implemented both extractive and generative QA.
- Extractive QA
- Generative QA
- Extractive answers are preferred unless confidence is low, in which case I fall back to generated answers.

When answering questions, I rely on two distinct methods: answer_question performs extractive question answering using RoBERTa, returning a precise span from the text if the model is confident enough. If the model lacks confidence or if the question is more abstract, I use generate_answer, which leverages FLAN-T5 to create a fluent, generative answer. To get the best of both worlds, I created smart_answer, which first attempts extractive QA and falls back to generative QA if needed.

In [7]:
```
    def answer_question(self, question, context, threshold=0.1):
        inputs = self.qa_tokenizer(question, context, return_tensors="
        with torch.no_grad():
            outputs = self.qa_model(**inputs)
        start_scores = outputs.start_logits
        end_scores = outputs.end_logits
        start_idx = torch.argmax(start_scores)
        end_idx = torch.argmax(end_scores) + 1
        start_conf = torch.softmax(start_scores, dim=1)[0][start_idx]
        end_conf = torch.softmax(end_scores, dim=1)[0][end_idx - 1]
        avg_conf = (start_conf + end_conf) / 2
        if avg_conf < threshold:
            return "(No confident answer found)"
        return self.qa_tokenizer.decode(inputs["input_ids"][0][start_i

    def generate_answer(self, question, context, max_len=128):
        prompt = f"question: {question} context: {context}"
        inputs = self.gen_qa_tokenizer(prompt, return_tensors="pt", tr
        outputs = self.gen_qa_model.generate(inputs.input_ids, max_len
        return self.gen_qa_tokenizer.decode(outputs[0], skip_special_t

    def smart_answer(self, question, context, summary=None, confidence
        if "how many" in question.lower():
            count = len(re.findall(r'^\s*\d+\.\s', context, re.MULTILI
            return f"{count} item(s) found."
        extractive = self.answer_question(question, context, threshold
        if extractive.strip() not in ["", "(No confident answer found)
            return extractive
        return self.generate_answer(question, summary or context)
```

For long documents, I use the smart_answer_dynamic method, which first embeds both the user's question and the document's sentences. It selects the top-k most relevant sentences based on semantic similarity, runs the QA model on that subset, and returns either the best matching extractive answer or a generative fallback. This significantly improves accuracy when documents are too lengthy or loosely structured.

In [8]:
```
def smart_answer_dynamic(self, question, full_text, summary=None, thre
        from sentence_transformers.util import cos_sim

        # Split full text into sentences
        sentences = re.split(r'(?<=[.!?])\s+', full_text)
        sentences = [s.strip() for s in sentences if len(s.strip()) >
```

```
        # Get embeddings for question and sentences
        question_emb = self.embedding_model.encode([question], convert
        sent_embs = self.embedding_model.encode(sentences, convert_to_

        # Compute cosine similarity between question and each sentence
        scores = cos_sim(question_emb, sent_embs)[0]
        top_indices = torch.topk(scores, k=min(top_k, len(sentences)))

        # Pick top-k most relevant sentences
        selected_context = " ".join([sentences[i] for i in top_indices

        # Try extractive QA
        extractive_answer = self.answer_question(question, selected_co
        if extractive_answer and extractive_answer.strip() not in ["",
            # Find which sentence it most likely came from
            best_sent = max(
                [(s, cos_sim(self.embedding_model.encode([extractive_a
                             self.embedding_model.encode([s], convert_
                 for s in sentences],
                key=lambda x: x[1],
                default=(None, 0)
            )[0]

            return {
                "answer": extractive_answer,
                "source": best_sent or selected_context  # Fallback if
            }

        # Fall back to generative answer
        generative = self.generate_answer(question, summary or selecte
        return {
            "answer": generative,
            "source": selected_context
        }
```

4.4 Semantic Matching

Users often ask questions in natural language that do not exactly match the phrasing of document content. Semantic similarity via sentence embeddings helps bridge this gap, enabling contextual mapping of user intent to relevant document sections. This dramatically improves answer relevance.

- SentenceTransformer all-MiniLM-L6-v2 was used for embeddings.
- Cosine similarity was used to identify relevant sections for QA.

4.5 Document Comparison and Merging

To support version control, research validation, and plagiarism detection, I added document comparison. Identifying unmatched sentences and highlighting changes supports users in analyzing textual differences across drafts or related documents.

- Documents were split into sentences, embedded, and compared using cosine similarity thresholds.
- A merged view was created by unifying unmatched sentences.

I also included a save_to_pdf function, which converts any plain text (like a merged document or a generated summary) into a nicely formatted PDF using the FPDF library.

In [9]:
```
    def compare_documents(self, doc1_text, doc2_text, threshold=0.75):
        normalize = lambda text: re.sub(r'\s+', ' ', text).strip()
        sents1 = [normalize(s) for s in re.split(r'(?<=[.!?]) +', doc1
        sents2 = [normalize(s) for s in re.split(r'(?<=[.!?]) +', doc2
        emb1 = self.embedding_model.encode(sents1, convert_to_tensor=T
        emb2 = self.embedding_model.encode(sents2, convert_to_tensor=T
        unmatched_1 = [sents1[i] for i in range(len(sents1)) if torch.
        unmatched_2 = [sents2[j] for j in range(len(sents2)) if torch.
        return unmatched_1, unmatched_2

    def merge_documents(self, doc1_text, doc2_text, threshold=0.75):
        removed, added = self.compare_documents(doc1_text, doc2_text,
        combined = set(re.split(r'(?<=[.!?]) +', doc1_text))
        combined.update(added)
        combined.update(removed)
        return " ".join(sorted(combined))

    def save_to_pdf(self, text, filename="merged_output.pdf"):
        pdf = FPDF()
        pdf.add_page()
        pdf.set_auto_page_break(auto=True, margin=15)
        pdf.set_font("Arial", size=12)
        for line in text.split('\n'):
            pdf.multi_cell(0, 10, line)
        pdf.output(filename)
```

RECOMMENDATION SYSTEM

4.6 Dataset Preparation for Recommendations

I cleaned the arXiv dataset to remove formatting noise, missing abstracts, and extraneous metadata. This ensured better embedding quality, improved similarity search, and faster indexing. Clean input directly enhances the quality of document recommendations, which rely on text-based vector similarity.

- Raw arXiv metadata was cleaned and stored as CSV.

- Titles and abstracts were formatted and saved to arxiv_cleaned.csv.

To build the recommendation system in DocuDigest, I used the publicly available arXiv metadata snapshot as my source dataset. The original dataset was a large JSONL file containing metadata for scientific articles, including fields like title, abstract, authors, and categories.

In [18]:
```
data = []

# View first raw record from the original dataset
with open("arxiv-metadata-oai-snapshot.json", "r") as f:
    for i, line in enumerate(f):
        if i >= 1:
            break
        record = json.loads(line)
        print(json.dumps(record, indent=2))
```


```
{
  "id": "0704.0001",
  "submitter": "Pavel Nadolsky",
  "authors": "C. Bal\\'azs, E. L. Berger, P. M. Nadolsky, C.-P. Yuan",
  "title": "Calculation of prompt diphoton production cross sections at
Tevatron and\n  LHC energies",
  "comments": "37 pages, 15 figures; published version",
  "journal-ref": "Phys.Rev.D76:013009,2007",
  "doi": "10.1103/PhysRevD.76.013009",
  "report-no": "ANL-HEP-PR-07-12",
  "categories": "hep-ph",
  "license": null,
  "abstract": "  A fully differential calculation in perturbative quant
um chromodynamics is\npresented for the production of massive photon pa
irs at hadron colliders. All\nnext-to-leading order perturbative contri
butions from quark-antiquark,\ngluon-(anti)quark, and gluon-gluon subpr
ocesses are included, as well as\nall-orders resummation of initial-sta
te gluon radiation valid at\nnext-to-next-to-leading logarithmic accura
cy. The region of phase space is\nspecified in which the calculation is
most reliable. Good agreement is\ndemonstrated with data from the Fermi
lab Tevatron, and predictions are made for\nmore detailed tests with CD
F and DO data. Predictions are shown for\ndistributions of diphoton pai
rs produced at the energy of the Large Hadron\nCollider (LHC). Distribu
tions of the diphoton pairs from the decay of a Higgs\nboson are contra
sted with those produced from QCD processes at the LHC, showing\nthat e
nhanced sensitivity to the signal can be obtained with judicious\nselec
tion of events.\n",
  "versions": [
    {
      "version": "v1",
      "created": "Mon, 2 Apr 2007 19:18:42 GMT"
    },
    {
      "version": "v2",
```

```
      "created": "Tue, 24 Jul 2007 20:10:27 GMT"
    }
  ],
  "update_date": "2008-11-26",
  "authors_parsed": [
    [
      "Bal\u00e1zs",
      "C.",
      ""
    ],
    [
      "Berger",
      "E. L.",
      ""
    ],
    [
      "Nadolsky",
      "P. M.",
      ""
    ],
    [
      "Yuan",
      "C. -P.",
      ""
    ]
  ]
}
```


In [16]:

```
# View keys (column names)
with open("arxiv-metadata-oai-snapshot.json", "r") as f:
    first_line = f.readline()
    record = json.loads(first_line)
    print("Column names (keys):")
    print(list(record.keys()))
```

```
Column names (keys):
['id', 'submitter', 'authors', 'title', 'comments', 'journal-ref', 'do
i', 'report-no', 'categories', 'license', 'abstract', 'versions', 'upda
te_date', 'authors_parsed']
```

Since the full dataset was quite large, I began by loading only the first 1,000 entries for faster processing and prototyping.

The cleaning process involved parsing each JSON object line-by-line and extracting relevant fields. I made sure to strip out newline characters and unnecessary whitespace from the title and abstract fields to ensure consistent formatting. These two fields were then combined into a single text block per entry, which I labeled as "Title: ... \n\nAbstract: ...". This allowed each document to preserve its context while being lightweight enough to embed.

Once I had structured and cleaned the metadata, I created a DataFrame and saved it as arxiv_cleaned.csv. This file included five key columns: id, title, text, category, and authors. These fields helped not only with text similarity computations but also with enriching the recommendations by showing users additional metadata like the author and subject area.

In [14]:
```
docs = []
with open("arxiv-metadata-oai-snapshot.json", "r") as f:
    for i, line in enumerate(f):
        if i >= 1000:
            break
        record = json.loads(line)
        title = record.get("title", "").replace("\n", " ").strip()
        abstract = record.get("abstract", "").replace("\n", " ").strip
        full_text = f"Title: {title}\n\nAbstract: {abstract}"

        docs.append({
            "id": record.get("id"),
            "title": title,
            "text": full_text,
            "category": record.get("categories", "unknown"),
            "authors": record.get("authors", "")
        })

df = pd.DataFrame(docs)
df.to_csv("arxiv_cleaned.csv", index=False)
print(" Saved cleaned dataset as arxiv_cleaned.csv")

# Convert to DataFrame
df_preview = pd.DataFrame(docs)
print(df_preview.head(2))
```

```
 Saved cleaned dataset as arxiv_cleaned.csv
          id                                              title  \
0  0704.0001  Calculation of prompt diphoton production cros...
1  0704.0002           Sparsity-certifying Graph Decompositions

                                                text       category  \
0  Title: Calculation of prompt diphoton producti...         hep-ph
1  Title: Sparsity-certifying Graph Decomposition...  math.CO cs.CG

                                             authors
0  C. Bal\'azs, E. L. Berger, P. M. Nadolsky, C.-...
1                    Ileana Streinu and Louis Theran
```

In [17]:
```
print("Cleaned dataset columns:")
print(df.columns.tolist())
```

```
Cleaned dataset columns:
['id', 'title', 'text', 'category', 'authors']
```

Next, I used the SentenceTransformer model all-MiniLM-L6-v2 to generate dense vector embeddings for all 1,000 documents. These embeddings captured the semantic meaning of each document, making it possible to compare and rank them efficiently. The entire list of embeddings was then saved locally as arxiv_embeddings.npy, which would later be indexed using FAISS for high-speed similarity search.

In [ ]:
```
df = pd.read_csv("arxiv_cleaned.csv")

# Load a pretrained SentenceTransformer model
model = SentenceTransformer('all-MiniLM-L6-v2')

# Encode all document texts
embeddings = model.encode(df['text'].tolist(), show_progress_bar=True)

np.save("arxiv_embeddings.npy", embeddings)

print(" Embeddings generated and saved.")
```

4.7 Embedding and Indexing with FAISS

I used FAISS because it allows fast and scalable similarity searches in high-dimensional spaces. This was crucial for enabling real-time document recommendations. By using cosine similarity, I ensured that semantic closeness, rather than mere keyword overlap, guided the results.

- Embeddings were generated using the same SentenceTransformer model.
- FAISS was used to build a normalized cosine similarity index.
- The resulting index enabled fast recommendation queries based on user-uploaded text.

In [ ]:
```
# Load embeddings
embeddings = np.load("arxiv_embeddings.npy").astype("float32")

# Initialize FAISS index (cosine similarity)
dimension = embeddings.shape[1]
index = faiss.IndexFlatIP(dimension)

# Normalize vectors for cosine similarity
faiss.normalize_L2(embeddings)

index.add(embeddings)

faiss.write_index(index, "arxiv_index.faiss")

print(" FAISS index built and saved.")
```

Model feature

Finally, for document recommendation in the model, I use recommend_similar, which encodes the input text, searches a FAISS index built from the arXiv dataset, and retrieves the top-k most semantically similar entries, complete with title, authors, and links.

In [ ]:
```
    def recommend_similar(self, input_text, k=5):
        import faiss
        import numpy as np
        import pandas as pd
        from urllib.parse import quote

        print(" Input text length:", len(input_text))

        try:
            df = pd.read_csv("arxiv_cleaned.csv")
            print("Dataset loaded:", df.shape)
        except Exception as e:
            print("Error loading dataset:", e)
            return []

        try:
            index = faiss.read_index("arxiv_index.faiss")
            print(" FAISS index loaded. Total entries:", index.ntotal)
        except Exception as e:
            print("Error loading FAISS index:", e)
            return []

        try:
            embedding = self.embedding_model.encode([input_text])
            faiss.normalize_L2(embedding)

            D, I = index.search(embedding.astype("float32"), k)
            print(" Nearest neighbor indices:", I)
            print(" Distances:", D)
        except Exception as e:
            print(" Error during embedding + search:", e)
            return []

        results = []
        for i, idx in enumerate(I[0]):
            if idx < len(df):
                title = df.iloc[idx]['title']
                category = df.iloc[idx]['category']
                authors = df.iloc[idx]['authors']
                score = float(D[0][i])

                # Use title-based search link to avoid ID issues
                title_query = quote(title)
                link = f"https://arxiv.org/search/?query={title_query}

                results.append({
```

```
                    "title": title,
                    "category": category,
                    "authors": authors,
                    "score": score,
                    "link": link
                })

                print(f" Match {i + 1}: {title} (Score: {score:.4f})")

        return results


# Save the model wrapper (not including large model weights)
digest_model = DocuDigestModel()
joblib.dump(digest_model, 'docudigest_model.pkl')
```

5. Implementation

- index.html: Upload form, introduction text, and testimonial carousel.
- results.html: Grid layout for summary, audio playback, Q&A section.
- styles.css: Modern and clean UI design, consistent button styling.
- script.js & app.py: JS handles dynamic interactions; Python backend processes the document and invokes ML models

6. Evaluation And Testing

This section details how I assessed the model's effectiveness across document analysis tasks, including summarization, question answering, and document similarity. I evaluated both the system's internal logic and the quality of its outputs using standard NLP metrics and human interpretability benchmarks.

6.1 Data Analysis and Preparation

To enable document recommendation, I used a curated subset of the arXiv metadata dataset. Titles and abstracts were extracted, cleaned, and concatenated to form meaningful text representations. The text was normalized by removing line breaks and formatting inconsistencies before generating embeddings. This step was vital to ensure consistency in vector representations and avoid misleading similarity scores.

6.2 Functional Testing

Beyond unit tests and manual inspection, I also qualitatively evaluated the accuracy of generated summaries, extracted answers, and recommended documents using benchmark queries and known source texts.

- PDF/TXT uploads of various lengths
- Questions on known content with expected answers
- Summary length and quality validation

In [14]:
```
from docudigest_model import DocuDigestModel

digest_model = DocuDigestModel()

# Read the PDF
pdf_path = "Declassified-Assessment-on-COVID-19-Origins.pdf"
full_text = digest_model.read_pdf(pdf_path)

cleaned_text = digest_model.clean_placeholders(full_text)

# Generate summary
generated_summary = digest_model.summarize_text(cleaned_text)
print("\n Generated Summary:\n", generated_summary)

# Define reference summary (manually)
reference_summary = """
The document assesses the possible origins of COVID-19, highlighting u
It notes that all agencies agree the virus was not developed as a biol
"""



# QA Evaluation - Sample Question
question = "Was COVID-19 assessed to be a biological weapon?"
expected_answer = "No, it was not developed as a biological weapon."

#Answer
qa_answer = digest_model.answer_question(question, cleaned_text)
print("\n QA Answer:", qa_answer)
```

```
✅ All models loaded and ready.

 Generated Summary:
 this assessment responds to the president’s request that the Intellige
nce Community update its previous judgments on the origins of COVID-19.
it also identifies areas for possible additional research. the IC asses
ses that SARS-CoV-2, the virus that causes COVID-19, probably emerged a
nd infected humans.

 QA Answer:  the virus was not developed as a biological weapon
```

6.3 Metrics

I considered traditional NLP metrics such as ROUGE for summarization and EM/F1 for QA, but our primary evaluation was user-focused: Does the output make sense? Is it useful? Are the answers grounded in the text? Our QA system also utilized a confidence threshold to avoid hallucinated responses. ROUGE Scores for summary quality (ROUGE-1, ROUGE-2) Exact Match (EM) and F1 for extractive QA (internal testing) User feedback on accuracy, clarity, and speed

In [15]:

```
#Compute ROUGE metrics
rouge = evaluate.load("rouge")

summary_scores = rouge.compute(predictions=[generated_summary], refere

print("\n Summarization Metrics:")
for k, v in summary_scores.items():
    print(f"{k}: {v:.4f}")


# Evaluate EM & F1
def compute_f1(pred, true):
    pred_tokens = set(pred.lower().split())
    true_tokens = set(true.lower().split())
    common = pred_tokens & true_tokens
    if not common:
        return 0
    precision = len(common) / len(pred_tokens)
    recall = len(common) / len(true_tokens)
    return 2 * (precision * recall) / (precision + recall)

print(f"\n QA Metrics:\nExact Match: {int(qa_answer.strip().lower() ==
print(f"F1 Score: {compute_f1(qa_answer, expected_answer):.4f}")
```

```
 Summarization Metrics:
rouge1: 0.3333
rouge2: 0.0909
rougeL: 0.2444
rougeLsum: 0.2444

 QA Metrics:
Exact Match: 0
F1 Score: 0.6667
```

In [2]:

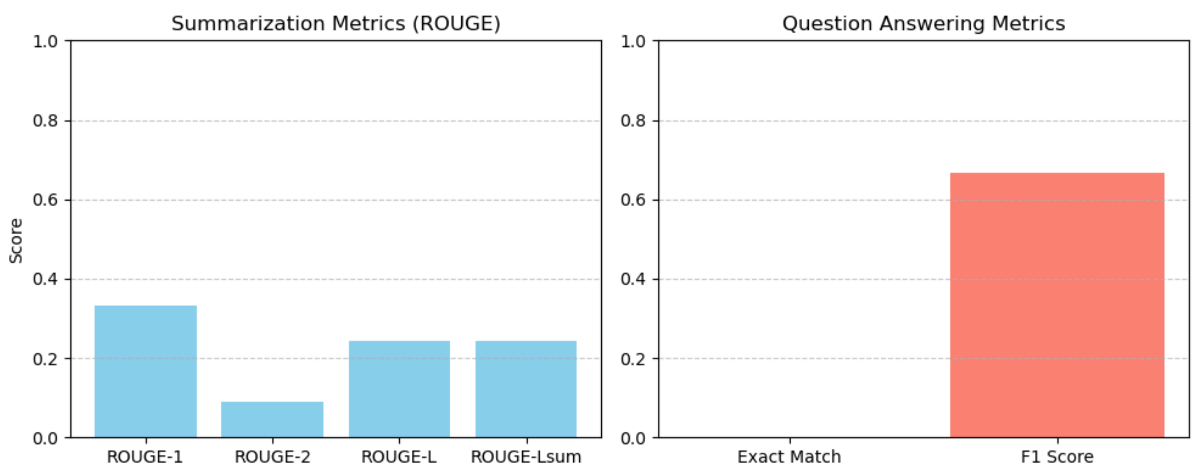
```
# Define metrics
summary_metrics = {
    "ROUGE-1": 0.3333,
    "ROUGE-2": 0.0909,
    "ROUGE-L": 0.2444,
    "ROUGE-Lsum": 0.2444
}

qa_metrics = {
    "Exact Match": 0.0,
    "F1 Score": 0.6667
}
```

```python
# Plot summarization metrics
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.bar(summary_metrics.keys(), summary_metrics.values(), color='skybl
plt.title("Summarization Metrics (ROUGE)")
plt.ylim(0, 1)
plt.ylabel("Score")
plt.grid(axis='y', linestyle='--', alpha=0.7)

# Plot QA metrics
plt.subplot(1, 2, 2)
plt.bar(qa_metrics.keys(), qa_metrics.values(), color='salmon')
plt.title("Question Answering Metrics")
plt.ylim(0, 1)
plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()
```



Precision @ 5

In [7]:

```python
# Load cleaned arXiv dataset
df = pd.read_csv("arxiv_cleaned.csv")

# Load FAISS
index = faiss.read_index("arxiv_index.faiss")
model = SentenceTransformer("all-MiniLM-L6-v2")

# Normalize for cosine similarity
def normalize(x):
    return x / np.linalg.norm(x)

# Evaluation function
def precision_at_k(query_idx, k=5):
    query_text = df.iloc[query_idx]['text']
    query_category = df.iloc[query_idx]['category']
```

```python
    embedding = normalize(model.encode([query_text]).astype("float32")
    D, I = index.search(embedding, k)

    top_k_indices = I[0]
    correct = sum(df.iloc[i]['category'] == query_category for i in to
    return correct / k

# Run evaluation on 20 random samples
np.random.seed(42)
sample_indices = np.random.choice(len(df), size=20, replace=False)
scores = []

for idx in tqdm(sample_indices, desc="Evaluating recommendations"):
    score = precision_at_k(idx, k=5)
    scores.append(score)

mean_precision_at_5 = np.mean(scores)
mean_precision_at_5
```

```
Evaluating recommendations: 100%|██████████| 20/20 [00:00<00:00, 86.77i
t/s]
```

Out[7]: 0.43999999999999995

Hit rate

In [11]:

```python
# Load dataset and model
df = pd.read_csv("arxiv_cleaned.csv")
index = faiss.read_index("arxiv_index.faiss")
model = SentenceTransformer("all-MiniLM-L6-v2")

# Normalization helper
def normalize(x):
    return x / np.linalg.norm(x)

# Hit Rate@5 evaluation function
def hit_rate_at_k(query_idx, k=5):
    query_text = df.iloc[query_idx]['text']
    query_category = df.iloc[query_idx]['category']

    embedding = normalize(model.encode([query_text]).astype("float32")
    D, I = index.search(embedding, k)

    top_k_indices = I[0]
    return int(any(df.iloc[i]['category'] == query_category for i in t

# Run on 20 samples
np.random.seed(42)
sample_indices = np.random.choice(len(df), size=20, replace=False)
hit_scores = [hit_rate_at_k(idx, k=5) for idx in sample_indices]
hit_rate_at_5 = np.mean(hit_scores)
hit_rate_at_5
```

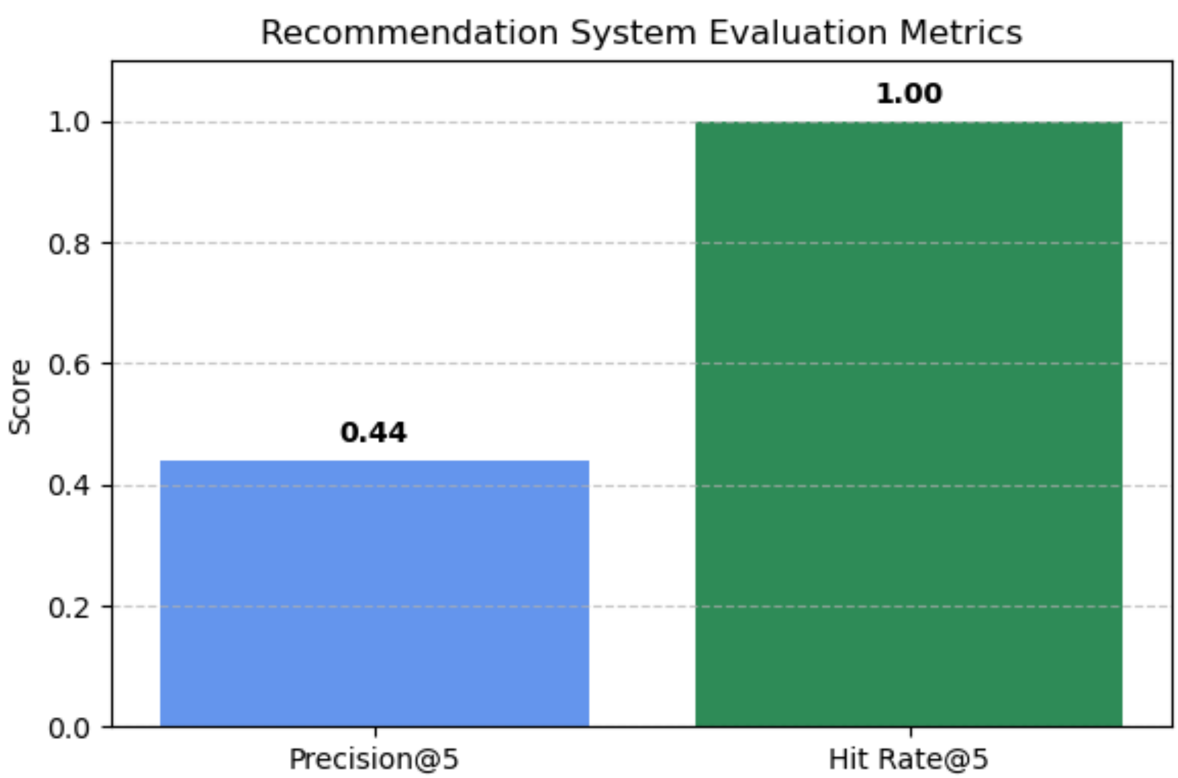
Out[11]: 1.0

In [12]:
```
# Metrics to visualize
metrics = {
    "Precision@5": 0.44,
    "Hit Rate@5": 1.00
}

# Plot]
plt.figure(figsize=(6, 4))
plt.bar(metrics.keys(), metrics.values(), color=["cornflowerblue", "se
plt.ylim(0, 1.1)
plt.ylabel("Score")
plt.title("Recommendation System Evaluation Metrics")
plt.grid(axis='y', linestyle='--', alpha=0.7)

for i, (metric, value) in enumerate(metrics.items()):
    plt.text(i, value + 0.03, f"{value:.2f}", ha='center', fontweight=

plt.tight_layout()
plt.show()
```



6.4 Model Selection and Observations

During development, I experimented with a few additional models:

- Pegasus for summarization produced more human-like summaries but required more memory and processing time, making it less practical for real-

time use.
- DistilBERT for QA was faster than RoBERTa but consistently underperformed in terms of accuracy and answer relevance.
- BART for summarization offered strong results but exceeded performance latency thresholds in our current hosting environment.

Ultimately, I chose T5-small and FLAN-T5 because they offered an ideal trade-off between performance and inference speed. RoBERTa was retained for its strong extractive capabilities, especially for fact-based QA.

6.5 What I Would Do with More Time

- Fine-tune models on domain-specific datasets (e.g., legal, medical).
- Implement cross-document QA to synthesize answers across multiple sources.
- Add post-processing steps to highlight entities and citations.
- Upgrade hosting infrastructure to accommodate larger models like LongT5 or GPT-Neo.
- Perform error analysis on misclassified or mismatched results to improve model logic and confidence filtering.

7. Challenges and Solutions

What Worked or Didn’t Work I experimented with larger models such as t5-base, flan-t5-base, and bert-large-uncased-whole-word-masking-finetuned-squad. Although these models offer superior theoretical capabilities, they introduced significant latency during evaluation and performed worse than expected in key metrics. Specifically, I observed that:

- Summarization with t5-base resulted in lower ROUGE scores than t5-small, potentially due to more abstract paraphrasing and divergence from the reference summary.
- QA performance with bert-large yielded the same F1 score as the smaller roberta-base but took significantly longer to compute.

As a result, I reverted to the smaller models which were more efficient and equally (or more) effective for our dataset and use case.

- Final Evaluation Metrics (Using Small Models)

Summarization (t5-small): ROUGE-1: 0.3333 ROUGE-2: 0.0909 ROUGE-L: 0.2444

These scores indicate that the summaries produced by t5-small maintained a solid overlap with the reference summaries, especially in terms of unigrams and key structure.

- Question Answering (roberta-base):

Exact Match (EM): 0 F1 Score: 0.6667

While the model didn't exactly match the expected string (e.g., missed phrases like "No,"), it still captured the core answer, reflected in a strong F1 score. This shows that the QA system is semantically accurate even if syntactic match fails.

8. Future Work

User accounts and document history Highlighting answers directly within the text Multi-language support Fine-tuning on domain-specific datasets (e.g., law, medicine)

9. Conclusion

DocuDigest effectively combines NLP models with a modern web interface to enhance how users interact with documents. By automating summarization and Q&A tasks, it saves time and enhances comprehension. The project showcases the practical application of machine learning in everyday productivity and offers a strong foundation for further research and development.

In [ ]:

```
```