Zainab Hasan 27127 and Ikhlas Ahmed Khan 27096

Assignment 4

ITA RAG Assignment

20th April, 2025

1. Platform Details

For this experimentation, Kaggle was used as the primary platform. All stages, including data processing, embedding generation, storage, and retrieval, were conducted within the Kaggle environment.

2. Data Details

The dataset consists of 221 pages from the book *Harry Potter and the Philosopher's Stone*. Each page was treated as a separate document. The book's PDF was processed, and the embedding of each document was generated and stored in a vector database to facilitate semantic retrieval.

3. Chunking Strategy

The experiment began with a chunk size of 250 tokens. During the process, both the chunk size and the overlap between chunks were gradually adjusted. This was done to preserve the context within the text and to ensure that no important information was lost when splitting the content into smaller parts. The overlap between chunks was especially important to maintain cotinuity and improve the quality of information being retrieved.

Through multiple trials, it was observed that increasing the chunk size led to improved performance in terms of both faithfulness (how true the answers were to the original text) and relevancy (how well the answers matched the questions). The performance kept improving until a chunk size of 1000 tokens with an overlap of 750 tokens. This specific combination gave the highest scores during testing.

However, increasing the chunk size or overlap beyond this point caused the scores to drop. This decline suggested that there is a global maximum—an optimal point for chunk size and overlap—after which adding more content in a single chunk may introduce too much information, causing the system to lose focus and reduce retrieval accuracy.

These observations helped identify the most effective chunking configuration for the given dataset and showed how important it is to balance chunk size and overlap for the best retrieval results.

4. Summarization Technique

The main summarization technique used in the experiment involved summarizing the response generated by the language model (LLM) by extracting the first 1–2 sentences of the generated answer. This aimed to simplify the content for quick evaluation and comparison while keeping it within the LLM's context limits.

However, the updated results using a different LLM revealed that this summarization step slightly decreased both the faithfulness and relevancy scores across all retrieval methods when compared to using the full response. For instance, the faithfulness score for Hybrid Search dropped from 0.9253 (without summarization) to 0.9059 (with summarization), and the relevancy score decreased from 0.8682 to 0.8694. Similar trends were observed for other retrieval strategies: for example, Reciprocal Rank Fusion scored 0.9206 faithfulness and 0.8721 relevancy with summarization, whereas without summarization, it improved to 0.9253 and 0.8671, respectively.

Overall, the non-summarized approach consistently produced better scores, indicating that providing the full, detailed response from the LLM—without truncation—results in more faithful and relevant answers. Based on these findings, it's evident that excluding summarization is the more effective strategy for this specific task.

5. Retrieval Techniques

In the retrieval system, five methods were employed to retrieve relevant documents for answering the query: Semantic Search, Keyword Search (BM25), Hybrid Search, MMR Search, and Reciprocal Rank Fusion. Each method operates differently and produces varying results based on how it retrieves information.

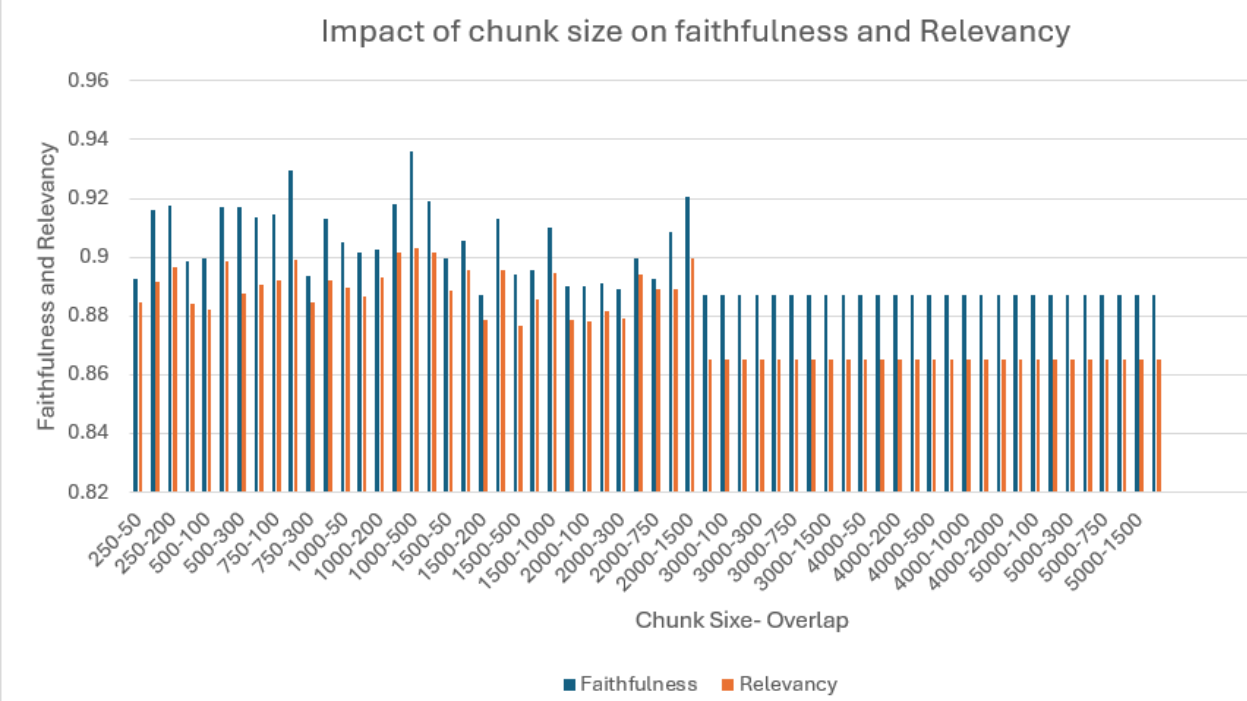

Figure 1: Semantic Search Performance Illustration

Semantic Search retrieves documents based on their semantic meaning, closely aligning the retrieved documents with the query and generally offering the highest performance. In contrast, Keyword Search (BM25) relies on exact keyword matching and, while effective, does not capture deeper semantic relationships between terms.

Hybrid Search combines both semantic and keyword-based retrievals using an ensemble approach, offering a good balance between the two retrieval techniques. MMR Search (Maximum Marginal Relevance) focuses on promoting diversity in the selection of documents, though this diversity can sometimes reduce the quality of retrieval.

Reciprocal Rank Fusion (RRF) merges results from both semantic and keyword-based searches, providing strong, balanced results by leveraging the strengths of each technique. Overall, Semantic Search remains the most effective standalone method, while Hybrid and RRF approaches offer robust alternatives by combining multiple retrieval strategies.

6. Model Evaluation

The evaluation shows a clear trend where models using powerful embedding architectures and instruction-tuned generative models consistently perform better in terms of both faithfulness and relevancy.

Among all the tested combinations, the best performing setup is the pair of `thenlper/gte-large` as the embedding model and `deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B` as the generative model. This combination achieves the highest faithfulness score of 0.9319 and relevancy score of 0.903, making it the most reliable and contextually accurate model overall.

The reason for this choice lies in the strengths of both components: `gte-large` is a state-of-the-art embedding model that captures nuanced semantic meaning effectively, and DeepSeek's instruction-tuned model is designed to generate coherent, fact-based, and highly relevant responses—making them together the most robust choice for retrieval-augmented generation tasks.

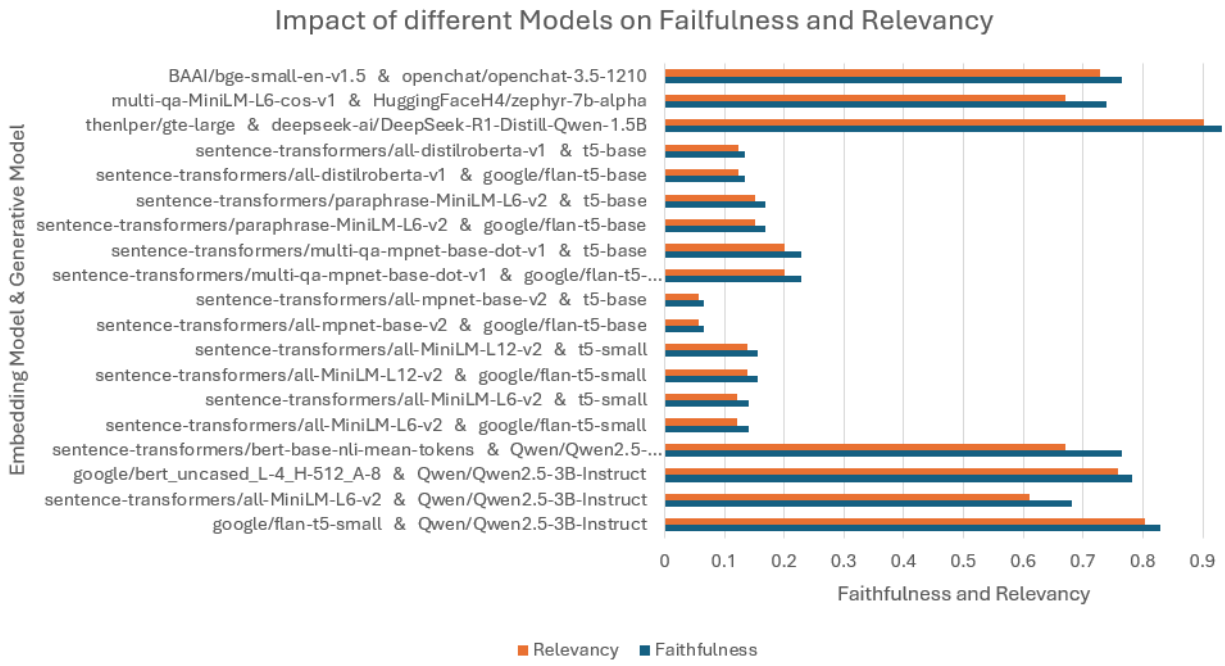

Figure 2: Impact of different models on Faithfulness and Relevance Scores

7. Performance

Semantic Search

When comparing all the methods, Semantic Search stands out as the most effective. It gives the most accurate (faithful) and relevant answer to the question "Who is Hagrid?" with a faithfulness score of 0.9329 and a relevancy score of 0.8895, both of which are the highest among all methods. It's also fast, taking only 0.03 seconds to retrieve and 30.54 seconds to generate the answer. The final response was clear, context-aware, and avoided unnecessary repetition.

Keyword Search (BM25)

On the other hand, Keyword Search (BM25) was the fastest overall with zero retrieval time, but it performed the worst in accuracy and relevancy, scoring the lowest on both metrics. It simply matched exact words and couldn't understand the deeper context of the question. The result was a bit generic and less meaningful.

Hybrid Search

Hybrid Search, which mixes both keyword and semantic search, gave decent results. It had a good balance of accuracy and relevancy, but it suffered from repetitive phrasing, such as constantly describing Hagrid as "a very kind person." It was fast like semantic search but not as clear or relevant.

MMR Search

MMR Search provided fairly accurate content with scores close to Hybrid, but it took the longest time to retrieve results—over 92 seconds, making it the slowest method overall with a total time of **122.08 seconds**. Although it introduced a bit of variety in the answer, it repeated itself and didn't add enough value to justify the wait.

Reciprocal Rank Fusion

Reciprocal Rank Fusion, which combines the results from different ranking methods, also performed well in terms of accuracy and relevancy, but again, it included unnecessary repetition and wasn't as clear or concise as Semantic Search. It was fast but didn't offer any unique advantages over the other methods.

Conclusion

In conclusion, **Semantic Search is the best method** among all. It offers the most faithful, most relevant, and quickest well-rounded answer with no confusing repetition. It understands the meaning behind the words and provides a cleaner, smarter response, making it the top choice for answering questions like "Who is Hagrid?"

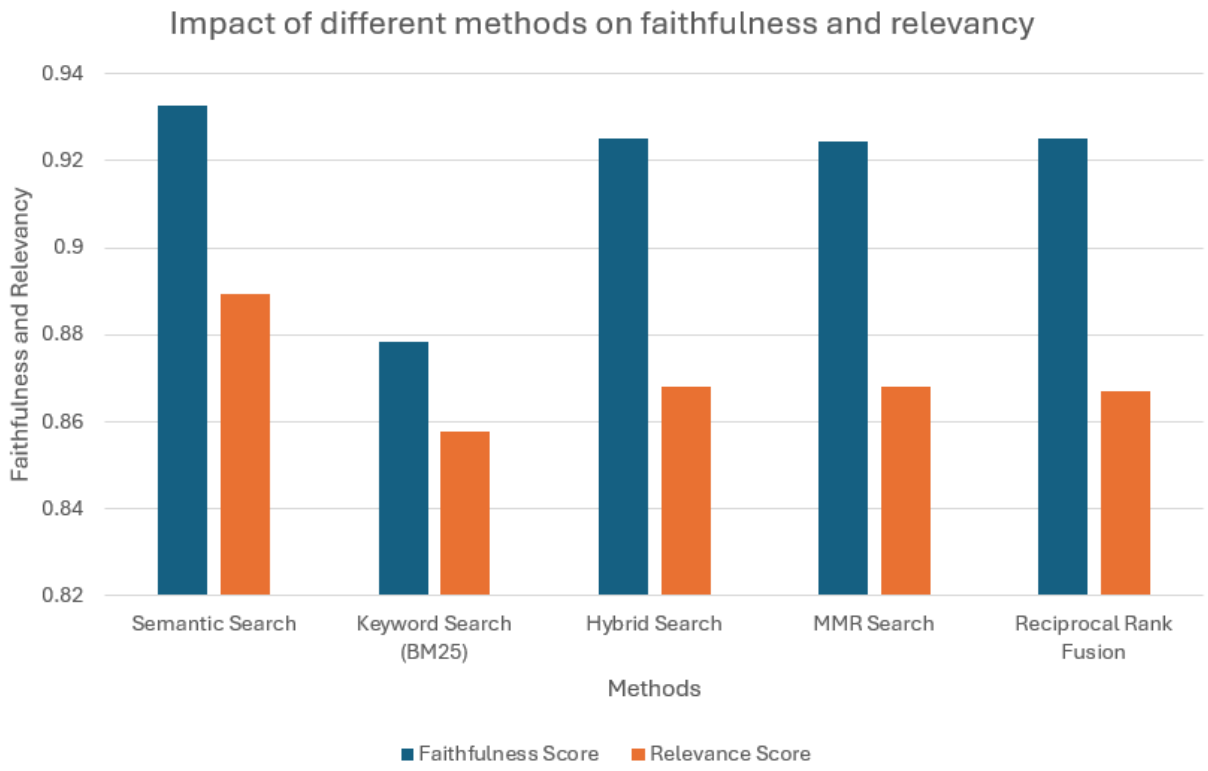

Figure 3: Comparison of Faithfulness and Relevance Scores for Different Search Methods

Variation in Value of K

Among all values of k, k = 7 performed the best with the highest faithfulness score (0.9426) and a strong relevancy score (0.9079). This shows that retrieving 7 documents gave the most accurate and relevant answer overall, making it the optimal choice.

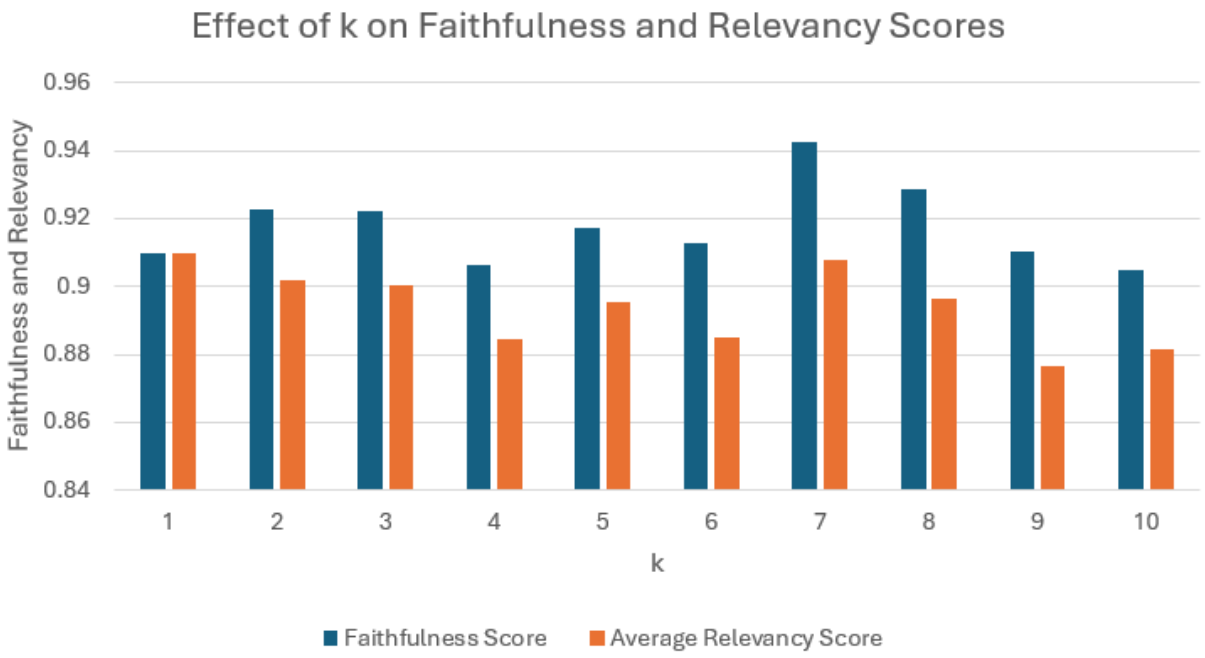

Figure 4: Effect of k on Faithfulness and Relevance Scores

8. Best Model Selection

Our best RAG setup used semantic search with **k = 7**, combining the embedding model **thenlper/gte-large** and the generative model **deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B**. This combination gave the most accurate and relevant answers by retrieving meaningful context and generating coherent responses. The **semantic search** technique helped match the question with the most related content, making this setup the most effective. The code for the best model is given below.

9. Best Model CODE

```
import os
from langchain_community.document_loaders import PyMuPDFLoader
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.schema.document import Document

from transformers import AutoModelForCausalLM, AutoTokenizer
from transformers import pipeline
import warnings
warnings.filterwarnings("ignore")
import textwrap

from ragas.metrics import faithfulness, answer_relevancy
```

```
from ragas.evaluation import evaluate

# Load and split PDF document
loader = PyMuPDFLoader("/kaggle/input/booooook/Harry Potter Book.pdf")
pages = loader.load_and_split()
print(len(pages))

# Initialize embedding model
embedding_model = HuggingFaceEmbeddings(model_name="thenlper/gte-large")

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,  # Adjust chunk size as needed
    chunk_overlap=750  # Ensures context continuity
)

# Apply chunking to pages
chunks = text_splitter.split_documents(pages)
len(chunks)

# Create FAISS vector database
#vectordb = FAISS.from_documents(pages, embedding_model)
vectordb = FAISS.from_documents(chunks, embedding_model)


# Save FAISS index to disk for later use
vectordb.save_local("faiss_index")

# Check the number of stored documents
print(f"Number of documents in the vector store: {vectordb.index.ntotal}")

# Query processing
question = "Who was Hagrid?"
retriever = vectordb.as_retriever(search_kwargs={"k": 8})
docs = retriever.get_relevant_documents(question)

# Print results
for i, doc in enumerate(docs, 1):
    page_number = doc.metadata.get('page', 'Unknown')
    print(f"Document {i} - Page {page_number} - Score: {doc.metadata.get('score', 'N/A')}")
    print(doc.page_content[:500])  # Print first 500 characters of each result
    print("-" * 80)

model_name = "deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B"
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    device_map="auto", #device_map='cuda'
    torch_dtype="auto",
    trust_remote_code=True,
)
tokenizer = AutoTokenizer.from_pretrained(model_name)

print(model.dtype)
total_params = sum(p.numel() for p in model.parameters())
print(f"Total Parameters: {total_params / 1e6} million")
memory_footprint = total_params * 2 / (1024 ** 2)  # Convert to MB
print(f"Estimated Memory Footprint: {memory_footprint:.2f} MB")

# Create a pipeline
generator = pipeline(
"text-generation",
model=model,
tokenizer=tokenizer,
return_full_text=False,
max_new_tokens=5000,
do_sample=False
)

# Extract page content and metadata properly
```

```
def format_response(doc):
    return f"Page {doc.metadata.get('page', 'Unknown')}: {doc.page_content.strip()}"

# Handle cases where fewer than 3 results are returned
retrieved_responses = [format_response(doc) for doc in docs[:7]]
while len(retrieved_responses) < 7:
    retrieved_responses.append("No relevant information.")  # Fill missing slots

# Construct the RAG prompt
prompt = f"""
You are an AI assistant tasked with answering questions based on retrieved knowledge.

### **Retrieved Information**:
{retrieved_responses[0]}
{retrieved_responses[1]}
{retrieved_responses[2]}
{retrieved_responses[3]}
{retrieved_responses[4]}
{retrieved_responses[5]}
{retrieved_responses[6]}

### **Question**:
{question}

### **Instructions**:
- Integrate the key points from all retrieved responses into a **cohesive, well-structured
    answer**.
- If the responses are **contradictory**, mention the different perspectives.
- If none of the retrieved responses contain relevant information, reply:
  **"I couldn't find a good response to your query in the database."**
"""


# Use Qwen2.5 3B with the correct message format
messages = [
    {"role": "user", "content": prompt}
]

# Generate output using the model
output = generator(messages)

# Print formatted response
print(textwrap.fill(output[0]["generated_text"], width=80))
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# Function to get embeddings for a list of texts (documents)
def get_embeddings_for_documents(documents, model):
    # Use embed_documents to get embeddings for a list of documents
    return model.embed_documents(documents)

# Function to get embedding for a single text (generated text)
def get_embedding_for_query(query, model):
    # Use embed_query to get embedding for a single query
    return model.embed_query(query)

# Get embeddings for the retrieved documents and the generated text
retrieved_documents = [doc.page_content for doc in docs[:8]]
retrieved_embeddings = get_embeddings_for_documents(retrieved_documents, embedding_model)
generated_text_embedding = get_embedding_for_query(output[0]["generated_text"],
    embedding_model)

# Calculate cosine similarity between each retrieved document and the generated text
similarities = []
for doc_embedding in retrieved_embeddings:
    similarity = cosine_similarity([doc_embedding], [generated_text_embedding])
    similarities.append(similarity[0][0])

```

```
# Print out cosine similarity scores for faithfulness and relevancy
print("Cosine Similarities (Relevancy Scores):")
for i, similarity in enumerate(similarities, 1):
    print(f"Document {i}: Similarity = {similarity:.4f}")

# Calculate the Faithfulness Score
# Faithfulness Score: This is the maximum cosine similarity between the generated response and any document
faithfulness_score = max(similarities)
print(f"\nFaithfulness Score: {faithfulness_score:.4f}")

# Calculate the average relevancy score (mean cosine similarity)
average_relevancy_score = np.mean(similarities)
print(f"Average Relevancy Score: {average_relevancy_score:.4f}")

print(f"Average Faithfulness Score")
```

Listing 1: Best Model Code