Task 2: Optimising RAG Model

Two Innovative Techniques for Optimising the RAG Model

This section details two innovative techniques that can significantly enhance the performance and accuracy of a Retrieval Augmented Generation (RAG) model, moving beyond a basic implementation.

Technique 1: Contextual Compression using Relevance Filtering

Problem Statement: When retrieving multiple chunks (top-k) from the vector store, irrelevant or redundant content can reduce generation quality and increase token costs.

Proposed Innovative Technique: Relevance Filtering + Contextual Compression before generation:

- **Relevance Filtering:** Use cosine similarity thresholds to discard low-relevance chunks.
- **Contextual Compression:** Use a lightweight summarisation LLM (e.g., `text-davinci-003` or `gpt-3.5-turbo-instruct`) to compress similar or redundant context into fewer, information-rich tokens.

How it Works:

1. **Initial Retrieval:** Perform a standard vector similarity search (e.g., using Pinecone) to retrieve a larger set of top-N documents.
2. **Relevance Filtering:** Apply a cosine similarity threshold to these N documents. Documents with a score below this threshold are discarded, ensuring only sufficiently relevant chunks proceed.
3. **Contextual Compression:** The remaining relevant documents are then passed to a summarization LLM. This LLM is prompted to condense the information from these multiple chunks into a more concise, yet comprehensive, summary. If multiple chunks cover very similar ground, they are effectively merged into a single, denser piece of context.

Implementation Overview:

```
# Pseudo-code for Contextual Compression
# Assume 'retrieved_documents' is a list of Document objects from initial
retrieval
# and 'llm_for_summarization' is an initialized lightweight LLM

# 1. Relevance Filtering (example, assuming 'retrieved_documents' have
scores)
# filtered_docs = [doc for doc in retrieved_documents if doc.score >
RELEVANCE_THRESHOLD]

# 2. Contextual Compression using LangChain's summarize chain
from langchain.chains.summarize import load_summarize_chain
from langchain_openai import ChatOpenAI # Or other lightweight LLM

# Initialize a lightweight LLM for summarization
llm_for_summarization = ChatOpenAI(model_name="gpt-3.5-turbo-instruct",
temperature=0.3)

# Load the summarization chain. 'map_reduce' is suitable for multiple
documents.
summary_chain = load_summarize_chain(llm_for_summarization,
chain_type="map_reduce")

# 'context_docs' would be the list of Document objects after relevance
filtering
# For demonstration, let's assume `context_docs` is a list of Document
objects
# created from the relevant chunks.
# Example: context_docs = [Document(page_content="chunk 1 text"),
Document(page_content="chunk 2 text")]

# Run the summarization chain
# compressed_context = summary_chain.run(context_docs)
# Note: In a real implementation, ensure context_docs is a list of
LangChain Document objects.
```

Innovation and Benefits:

- **Reduced Prompt Size:** By compressing redundant or less critical information, the number of tokens sent to the main generative LLM is significantly reduced, leading to lower API costs and faster inference times.
- **Increased Contextual Relevance:** The summarization process focuses on extracting the most salient information, ensuring that the LLM receives a highly concentrated and relevant context, improving output precision and reducing the likelihood of hallucination.
- **Efficiency for Long Documents:** This technique is particularly beneficial when dealing with very long source documents that, when chunked, might still produce many relevant but verbose chunks. It makes the RAG pipeline more efficient for comprehensive knowledge bases.

Technique 2: Hybrid Retrieval: Dense + Sparse Fusion

Problem Statement: Relying solely on dense vector retrieval (embeddings) can sometimes miss semantically important but rare keywords or exact matches. While dense embeddings excel at capturing semantic meaning and conceptual similarity, they might struggle with highly specific names, codes, or terms that appear infrequently in the training data of the embedding model, or when an exact keyword match is critical for relevance.

Proposed Innovative Technique: Implement a **Hybrid Retrieval System** that combines the strengths of both dense and sparse retrieval methods. This approach ensures that the system captures both semantic meaning and keyword specificity.

How it Works:

1. **Dense Retrieval (Semantic Similarity):** The user query is embedded into a vector, and a similarity search is performed on the Pinecone vector database. This retrieves documents based on their semantic meaning. (e.g., top-K1 documents).
2. **Sparse Retrieval (Keyword Matching):** Simultaneously, a traditional keyword-based search is performed using a sparse retrieval model (e.g., BM25, TF-IDF) or a search engine like Elasticsearch. This retrieves documents based on the presence and frequency of exact keywords or n-grams from the query. (e.g., top-K2 documents).
3. **Result Fusion and Re-ranking:** The results from both dense and sparse retrieval are combined. A weighted scoring algorithm is then applied to normalize and merge the scores from both methods. For example, a linear combination `final_score = alpha * dense_score + (1 - alpha) * sparse_score` can be used, where `alpha` is a tunable weight. The combined list of documents is then re-ranked based on these fused scores.
4. **Final Context Selection:** The top-N documents from the fused and re-ranked list are selected to form the context for the LLM.

Implementation Overview (Pseudo-code):

```
# Pseudo-code for Hybrid Retrieval
# Assume 'pinecone_index' for dense retrieval and 'bm25_retriever' for
sparse retrieval
# and 'query_embedding' is the user query's embedding

# 1. Dense Retrieval
# dense_results = pinecone_index.query(vector=query_embedding, top_k=K1,
include_metadata=True)
# dense_docs = [(match.metadata['text'], match.score) for match in
dense_results.matches]

# 2. Sparse Retrieval (e.g., using a pre-built BM25 index or Elasticsearch)
# sparse_results = bm25_retriever.retrieve(user_query, top_k=K2)
# sparse_docs = [(doc.text, doc.score) for doc in sparse_results]

# 3. Normalize and Merge Scores
# Assign unique IDs to documents to handle overlaps
# Create a dictionary to store the max score for each unique document
# merged_docs = {}
# for text, score in dense_docs:
#     merged_docs[text] = merged_docs.get(text, 0) + alpha * score # Apply
weight
# for text, score in sparse_docs:
#     merged_docs[text] = merged_docs.get(text, 0) + (1 - alpha) * score #
Apply weight

# Convert back to list of (text, score) and sort
# final_ranked_docs = sorted(merged_docs.items(), key=lambda item: item[1],
reverse=True)

# Select top-N documents for context
# context_for_llm = [doc_text for doc_text, score in final_ranked_docs[:N]]
```

Innovation and Benefits:

- **Comprehensive Information Capture:** This hybrid approach ensures that the system captures both the semantic meaning (dense) and the exact keyword specificity (sparse), leading to a more comprehensive understanding of the query.
- **Improved Answer Accuracy:** Especially in business domains, where specific product names, codes, or technical terms are crucial, hybrid retrieval ensures that documents containing these exact terms are not missed, leading to more accurate and reliable answers.
- **Robustness for Diverse Query Types:** The system becomes more robust to various query styles, from conversational and semantic questions to precise keyword-driven searches.
- **Fallback Mechanism:** Provides a natural fallback. If dense retrieval struggles with a novel term, sparse retrieval can still find relevant documents based on keyword matches.

Conclusion

These two advanced techniques—Contextual Compression (with relevance filtering and summarization) and Hybrid Retrieval Fusion—significantly enhance the RAG pipeline by reducing token usage, increasing precision, and improving robustness for diverse query types. Implementing them makes the RAG QA Bot smarter, faster, and more business-friendly, delivering more accurate and cost-effective responses.

Intern Name: VINO K

Internship Project: Retrieval Augmented Generation (RAG) QA Bot

Date: July 2025