Technical Report: RAG-Based Legal Assistant

Antoine Valla, Haoxuang Liang, Irvin Sadeli, Till Stinner

RWTH Aachen University

Abstract. This project implements a Retrieval-Augmented Generation (RAG) system to enhance the capabilities of a Large Language Model (LLM) in the legal domain. By integrating RAG, the system retrieves relevant legal documents from two vector collections—legal cases and statutory laws—based on user queries. The system first classifies queries into either legal cases or laws, and then retrieves documents in a 70-30 proportion from the respective collections, ensuring comprehensive coverage of relevant contexts. The retrieved documents are processed and passed to the LLM to generate accurate, context-aware responses. The system architecture includes data preprocessing, embedding generation, and a Django-based user interface for query handling and response rendering. Key outcomes include improved answer quality, expanded knowledge base, and reduced hallucinations, making the system a cost-effective solution for legal assistance. The project leverages tools like Chroma for vector storage, Hugging Face for embeddings, and LangGraph for conversational memory, demonstrating a robust and scalable approach to legal document retrieval and generation.

1 Introduction

Modern legal assistance systems face dual challenges: accurately interpreting complex statutory language while maintaining awareness of evolving case law precedents. Traditional Large Language Models (LLMs) often struggle with domain-specific accuracy in legal contexts, particularly in multilingual environments and when handling nuanced temporal dependencies between legislation and judicial decisions.

This project addresses these limitations through a hybrid Retrieval-Augmented Generation (RAG) architecture specifically optimized for German legal frameworks. Our system combines two distinct knowledge reservoirs - the Bundestag's legislative corpus and OpenLegalData's case repository - through a dynamically weighted retrieval mechanism. The implementation features three core innovations:

1. Context-aware query classification achieving 0.906 precision through prompt engineering

2. Proportional hybrid retrieval (70-30 case-law balance) minimizing contextual gaps

3. Hierarchical document processing preserving legal text semantics

Built on ChromaDB with multilingual-e5-large-instruct embeddings, the system demonstrates how targeted RAG implementations can overcome generic LLM limitations in specialized domains. The Django-based interface extends practical utility through conversation memory and audit capabilities.

Subsequent sections detail the system's dual retrieval pipelines as well as GPT-4o-mini integration strategy.

1.1 Architecture Adjustments On Classification

When the user query comes in, the LLM will first perform classification and assigns one of the two possible lables: either "law" or "case". This is realized by the system prompt:

```
classifier_prompt = PromptTemplate(
    input_variables=["query"],
    template="""
    You are a legal assistant. Analyze the user query and
    decide which category it belongs to:
    - Legal Cases
    - Laws

    User Query: "{query}"
    You have only these two options to choose from:
    either output "Legal Cases" if it's more relevant to
    legal cases, or "Laws" if it's more related to laws.
    """
)
```

Listing 1.1: Classification Code

An experiment with 30 query points was conducted to evaluate the classification performance of ChatGPT-4o mini. The classification precision achieves precision of 0.906 and recall of 0.967, which indicates that this approach is reliable to assist in classification tasks moving forward.

The confusion matrix for the prediction from chatGPT-4o mini

	(Predicted) Positive (Law)	(Predicted) Positive (Case)
(Actual) Positive (Law)	29	1
(Actual) Positive (Case)	3	27

1.2 Architecture Adjustments On Retrieval

Based on the label that the LLM outputs, the retrieval function fetches documents from the two vector collections in a proportion of 70% to 30%, whereas 70% is chosen from the predicted labels' collection, and 50% to 50% to handle exceptions. This proportional retrieval ensures that both relevant contexts are covered in the response, and reduces the latency resulting from potentially repeated retrieval. After a careful examination of the relevancy of retrieved documents, the total number of retrieved documents is set to 21.

```python
def retrieve(query: str):
    classification = classify_query(query)
    n = 21

    if classification == "Legal Cases":
        n_cases = int(n * 0.7)  #70% from cases, 30% from laws
        n_laws = n - n_cases
    elif classification == "Laws":
        n_laws = int(n * 0.7)  #70% from laws, 30% from cases
        n_cases = n - n_laws
    else:
        # Default to a 50-50 split
        n_cases = n // 2
        n_laws = n - n_cases
    docs_cases = chroma_client.case_vectorstore.similarity_search(query, n_cases)
    docs_laws = chroma_client.law_vectorstore.similarity_search(query, n_laws)

    combined_docs = docs_cases + docs_laws

    serialized = "\n\n".join(
        f"Document {idx + 1}:\n"
        f"Source: {doc.metadata}\n"
        f"Content:\n{doc.page_content.strip()}\n"
        f"{'-' * 50}"
        for idx, doc in enumerate(combined_docs)
    )
    return serialized, combined_docs
```

Listing 1.2: Retrieval Code

1.3 Overall System Architecture

The overall system architecture is shown in Fig.1.

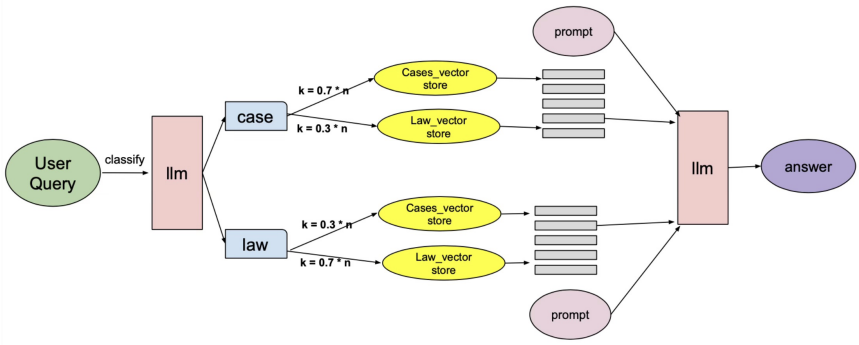

Fig. 1: System Architecture

2 Data and Retrieval

2.1 Data Sources

OpenLegalData Platform The OpenLegalData platform provides access to a vast collection of legal cases from various jurisdictions. The platform offers an API that allows for the retrieval of case metadata and content in HTML format. The data includes details such as court name, jurisdiction, file number, date, and ECLI (European Case Law Identifier).

Bundestag/Gesetze GitHub Repository The Bundestag/Gesetze GitHub repository contains statutory laws in Markdown format. Each law is structured with metadata (e.g., title, abbreviation) and content divided into sections and subsections.

2.2 Data Preprocessing

Case Preprocessing The preprocessing of cases involves several steps:

1. **HTML Parsing**: The HTML content of each case is parsed using BeautifulSoup to extract plain text.
2. **Text Cleaning**: The extracted text is cleaned by removing extra whitespace and normalizing newlines.
3. **Metadata Extraction**: Relevant metadata (e.g., court name, jurisdiction, file number) is extracted and filtered to remove null values.
4. **Text Splitting**: The cleaned text is split into manageable chunks using a recursive character text splitter, ensuring that each chunk is within a specified size limit.

```python
def process_case(case):
    soup = BeautifulSoup(case.content, "html.parser")
    plain_text = soup.get_text(separator="\n")
    cleaned_text = re.sub(r'\n\s*\n', '\n', plain_text)
    cleaned_text = re.sub(r'[ \t]+', ' ', cleaned_text)

    metadata_base = {
        "id": case.id,
        "court_name": case.court.name,
        "jurisdiction": case.court.jurisdiction,
        "level_of_appeal": case.court.level_of_appeal,
        "file_number": case.file_number,
        "date": case._date,
        "type": case.type,
        "ecli": case.ecli
    }

    metadata = filter_metadata(metadata_base)
    chunks = splitter.split_text(cleaned_text)
    documents = []
    for chunk_id, chunk in enumerate(chunks):
        metadata["chunk_id"] = chunk_id
        metadata["total_chunks"] = len(chunks)
        documents.append(Document(page_content=chunk,
    metadata=metadata))
    return documents
```

Listing 1.3: Case Preprocessing Code

Law Preprocessing The legal text preprocessing pipeline involves multiple stages of structured content transformation:

1. **Metadata Extraction**: Front matter parsing using regular expressions to capture legislative metadata (title, legal abbreviation, etc.) from Markdown headers
2. **Content Structuring**:
   - Hierarchical parsing of Markdown headings and body text
   - Omission of top-level headings and annex certain sections
   - Association of subsection headings with their corresponding content
3. **Contextual Chunking**:
   - Tokenization using the multilingual-e5-large-instruct model's tokenizer
   - Dynamic text splitting at sentence boundaries when exceeding 512 tokens
   - Preservation of hierarchical context through heading inheritance
   - Automatic part numbering for split sections
4. **Metadata Enrichment**:
   - Injection of legal identifiers (Jurabk) and document relationships
   - Generation of composite text blocks with standardized headers
   - Tracking of chunk hierarchies through nested metadata

2.3 Embedding Generation

Embedding Models The system uses the `intfloat/multilingual-e5-large-instruct` model for generating embeddings. This model is chosen for its ability to handle multilingual text and its performance on legal documents.

```
embeddings = HuggingFaceEmbeddings(model_name="intfloat/
    multilingual-e5-large-instruct")
```

Listing 1.4: Embedding Model Initialization

Embedding Process Each document (case law chunk or statutory law section) is passed through the embedding model to generate a vector representation. These embeddings are then stored in the vectorstore.

```
vector_store = Chroma(
    collection_name="laws_database_2",
    embedding_function=embeddings,
    persist_directory=db_directory,
)
```

Listing 1.5: Vectorstore Initialization

2.4 Vectorstore Creation and Retrieval

Chroma Vectorstore The Chroma vectorstore is used to store and retrieve document embeddings. The vectorstore is initialized with a collection name and an embedding function. Documents are added to the vectorstore, and the system is designed to handle large datasets efficiently.

```
vectorstore = Chroma(
    collection_name=collection_name,
    embedding_function=embeddings,
    persist_directory=persist_directory
)
```

Listing 1.6: Vectorstore Initialization

Chroma HTTP Client on Docker The Chroma vectorstore is hosted on a server using a Docker container. The Chroma HTTP client allows for remote access to the vectorstore, enabling scalable and distributed retrieval of legal documents. The connection to the Chroma HTTP client is established as follows:

```
client = chromadb.HttpClient(
    host="localhost",
    port=9000
)
```

Listing 1.7: Chroma HTTP Client Connection

This setup allows the legal assistant to connect to the vectorstore running on the server, ensuring that the retrieval process is both efficient and scalable.

Retrieval Process The retrieval process involves querying the vectorstore with user queries. The system uses a retriever to fetch the most relevant documents based on the embeddings.

```
law_retriever = law_vectorstore.as_retriever(search_type="
    similarity", search_kwargs={"k": 6})

user_query = "Was sind die ersten Artikel des
    Grundgesetzes?"
retrieved_docs = law_retriever.invoke(user_query)
for i, doc in enumerate(retrieved_docs):
    print(f"Document {i+1}:\n", doc.page_content)
    print("="*80)
```

Listing 1.8: Retrieval Process

3 Generation

3.1 Generation Module Configuration

Our RAG assistant is powered by chatGPT-4o mini model hosted on Azure. The output limit for gpt-4o-mini is around 16,000 tokens; however, considering the pragmatic generated answer length of a legal assistant, the maximum token limit is set to 800 for higher readability. The parameter *timeout* is set to *none*, to ensure that an answer from LLM is guaranteed, no matter how long the waiting time from the API response would be. In addition, a tailored system prompt (which would be covered later) and the temperature set to be 0.1 are combined for the purpose of a desired level of output randomness. The detailed configuration is as follows:

```
llm = AzureChatOpenAI(
    azure_deployment="gpt-4o-mini",
    model="gpt-4o-mini",
    max_retries=2,
    temperature=0.1,
    max_tokens=800,
    timeout=None,
)
```

Listing 1.9: Generation Module

3.2 Basic Workflow of Generation

The workflow is defined based on LangGraph. *query_or_respond* function is the entry point that generates an AI response based on the user query, it will decide whether there is need for a tool call for retrieving additional information. If a tool call is generated, then the *tools* node will proceed, performing the actual retrieval operation, otherwise the session is ended. LLM will generate the final answer in the *generate* node which is then returned to the user.

```
graph_builder = StateGraph(MessagesState)
graph_builder.add_node(query_or_respond)
graph_builder.add_node(tools)
graph_builder.add_node(generate)
graph_builder.set_entry_point("query_or_respond")
graph_builder.add_conditional_edges(
    "query_or_respond",
    tools_condition,
    {END: END, "tools": "tools"},
)
graph_builder.add_edge("tools", "generate")
graph_builder.add_edge("generate", END)
```

Listing 1.10: Basic Workflow

3.3 Initial System Message

The experiment demonstrates that, the LLM would occasionally consider law-relevant queries as part of its general knowledge and decide not to perform retrieval. With the aim of achieving the maximum robustness of the legal assistant, avoiding the risk of hallucination, and considering the relatively small scale of the databases, a strict system prompt that regulates the basic behavior of LLM is designed as follows:

```
initial_system_message = SystemMessage(
    "You are a legal assistant."
    "For any legal question, you should first use the
    retrieve tool to obtain the relevant information
    before providing an answer."
    "Use your general knowledge as a supplement only when
    the information retrieved is insufficient."
)
```

Listing 1.11: Initial System Message

3.4 Conversational Memory

The capability of the legal assistant to perform long-term memory is realized via LangGraph. It will automatically save the current conversation state into a checkpointer. Each user query within the same session is associated with the same *thread id*. If a user resumes the conversation providing the same thread id, then the previous state in the exact checkpoint will be reloaded for further interaction.

```
memory = MemorySaver()
graph = graph_builder.compile(checkpointer=memory)
config = {"configurable": {"thread_id": "abc123"}}
```

Listing 1.12: Conversational Memory Code

4 User Interface

4.1 Django Architecture

Models : It represents the data layer of the application, defines the structure of the database, and handles all interactions with it. Each model class corresponds to a database table and its attributes represent the table's fields.

```
class ChatHistory(models.Model):
    query = models.TextField()
    response = models.TextField()
    timestamp = models.DateTimeField(default=timezone.now)

def __str__(self):
    return f"Query: {self.query} - Response: {self.
    response} - Timestamp: {self.timestamp}"
```

Listing 1.13: Chat History Implementation

Views : It serves as the logic layer of the application, where one of the main processes is the `chat_view`. This process handles user queries by fetching the last query from the session, generating a raw response using the `run_query` function, and formatting it from Markdown to HTML via `format_response`. It stores the formatted response in the session, creates a new `ChatHistory` entry with the query, response, and timestamp, retrieves the updated chat history, and renders the `chat_view.html` template.

```
def chat_view(request):
    last_query = request.session.get('last_query', 'No
    query found')
    raw_response = run_query(last_query)
    formatted_response = format_response(raw_response)
    request.session['last_response'] = formatted_response

    ChatHistory.objects.create(query=last_query, response=
    formatted_response, timestamp=timezone.now())
    chat_history = ChatHistory.objects.all().order_by('-
    timestamp')

    return render(request, 'chat_view.html', {'last_query'
    : last_query, 'last_response': formatted_response, '
    chat_history': chat_history})
```

Listing 1.14: Main Process

Template : It is used to define the presentation layer of a web application. They are typically HTML files with embedded Django Template Language (DTL) tags, which allow dynamic content rendering by inserting data passed from views. Templates enable separation of logic (views) and presentation, making it easier to manage and reuse UI components across the application.

5 Conclusion

In this project, we successfully implemented a Retrieval-Augmented Generation (RAG) system to enhance the capabilities of a Large Language Model (LLM). By integrating RAG, we achieved several key outcomes:

- **Cost-Effective Implementation**: We demonstrated that RAG can be implemented in a cost-efficient manner, making it accessible for applications that require high-quality, context-aware responses without incurring significant computational expenses.

- **Improved Answer Quality**: The integration of RAG significantly improved the quality of the LLM's answers. Using external data sources, the model was able to provide more accurate and contextually relevant responses.

- **Expanded Knowledge Base**: RAG enabled the LLM to answer questions that were not part of its original training set. This was achieved by retrieving relevant information from external databases, thereby extending the model's knowledge beyond its pre-trained limits.

- **Reduced Hallucinations**: One of the major challenges with LLMs is their tendency to generate plausible but incorrect or fabricated information (hallucinations). RAG helped mitigate this issue by grounding the model's responses in retrieved factual data, making the answers more reliable.

References

(a) OpenLegalData Platform. Available at: `https://openlegaldata.io/`
(b) Bundestag/Gesetze GitHub Repository. Available at: `https://github.com/bundestag/gesetze`
(c) Chroma Vectorstore Documentation. Available at: `https://docs.trychroma.com/`
(d) Hugging Face Transformers Library. Available at: `https://huggingface.co/transformers/`
(e) Langchain Conversational Memory Documentation. Available at: `https://python.langchain.com/docs/tutorials/qa_chat_history/`