TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1
Introduction

This project presents a question-answering application tailored specifically for the Indian Penal Code. Its primary objective is to provide accurate and relevant responses to user queries by leveraging a comprehensive knowledge base. The application's response generation mechanism is designed to derive answers exclusively from the curated knowledge base, ensuring that the information provided adheres strictly to the established legal framework without incorporating any external or prior knowledge from language models.

A distinguishing feature of this endeavor is its utilization of open-source models, thereby eliminating the need for commercial, closed-source language model services and the associated costs. Furthermore, a significant emphasis is placed on achieving an optimal balance between response time and accuracy, a critical consideration in the development of an effective question-answering system.

Chapter 2
Approaches along the way

2.1 Fine Tuning

In the pursuit of developing an effective question-answering system tailored to the Indian Penal Code, the initial exploration centered around the fine-tuning technique. Fine-tuning involves taking a pre-trained language model and further training it on a custom dataset specific to the target domain or task. While this approach has proven successful in various applications, it presents a significant limitation when dealing with custom data: the tendency of the model to hallucinate, generating responses that deviate from the provided information.

Hallucination arises when the model relies excessively on its prior knowledge acquired during the pre-training phase, rather than strictly adhering to the given context. For instance, if we were to fine-tune a language model on the Indian Penal Code dataset and pose a question about a specific section, the model might incorporate external information from its training, potentially leading to inaccurate or incomplete answers, compromising the integrity and reliability of the system.

2.2 In Context Learning

To address this issue, the Instruction-Context Learning (ICL) method emerged as a promising alternative. The ICL approach involves providing the language model with a specific context, such as relevant sections from the Indian Penal Code, and instructing it to generate answers solely based on that context. This method effectively restricts the model's response to the given information, significantly reducing the risk of hallucination.

As an illustration, consider a scenario where we provide the relevant sections of the Indian Penal Code as the context and ask a question about a specific offense. By employing the

ICL method, the model will generate a response based exclusively on the provided context, ensuring the accuracy and consistency of the answer, as it does not rely on external or potentially irrelevant information.

However, implementing the ICL method in real-world scenarios posed a significant challenge, as the context can often be extensive, potentially spanning thousands of pages or even entire legal documents. Providing such a vast amount of data as context is impractical and computationally inefficient, as it would require substantial computational resources and processing time, hindering the system's performance and scalability.

2.3 Retrieval Augmented Generation

To overcome this hurdle, the Retrieval Augmented Generation (RAG) method emerged as an elegant solution. In the RAG approach, the dataset (in this case, the Indian Penal Code) is divided into manageable chunks and stored in a vector database. When a user submits a query, a retrieval mechanism identifies the relevant data chunks from the database based on their similarity to the question.

For example, if a user inquires about the punishment for a specific offense, the retrieval mechanism would search the vector database and retrieve the relevant sections or chunks that discuss that particular offense. These retrieved chunks serve as the context, which is then provided to the language model, enabling it to generate a final answer based on the extracted information.

The RAG method seamlessly combines the strengths of both retrieval and generation, allowing for efficient retrieval of relevant information while leveraging the language model's generation capabilities to produce coherent and comprehensive responses. By focusing solely on the retrieved context, the model is less likely to hallucinate, as it does not rely on external knowledge beyond the provided chunks, thereby ensuring the accuracy and reliability of the generated answers.

Chapter 3
Why Vector Representation

First of all, the big question is how to retrieve relevant data according to the user query. There are mainly two approaches for this. 1) Text search 2) Vector search

3.1 Text Search

The traditional text search approach, which relies on exact keyword or phrase matching, presents several limitations in the context of this project. While text search ensures precise matches, it lacks the ability to capture the underlying semantic meaning and context of the text. This can lead to the retrieval of irrelevant or incomplete information, especially when dealing with complex legal terminology and nuanced language found in the Indian Penal Code.

Furthermore, text search struggles to handle variations in language, such as synonyms, misspellings, or different word forms, which can result in missing relevant sections or provisions. Additionally, traditional text search methods often fail to provide an effective relevance ranking, presenting results based solely on the presence of keywords rather than their degree of relevance to the query.

3.2 Vector Search

To overcome these limitations, this project employs vector search, a technique that represents text documents and queries as high-dimensional vectors in a semantic space. In this semantic space, similar documents or queries are positioned closer together, enabling the retrieval of relevant information based on their underlying meaning and context.

Vector search offers several advantages over traditional text search methods in the context of this RAG application:

1. Semantic understanding: By capturing the underlying meaning and context of the legal text, vector search enables the retrieval of relevant sections or chunks of the Indian Penal Code, even if they do not contain the exact keywords or phrases from the user's query. This ensures that the generated answers are comprehensive and accurately reflect the legal provisions.

2. Relevance ranking: Vector search algorithms can rank the retrieved sections or chunks based on their semantic similarity to the query, presenting the most relevant information first. This feature is crucial in the legal domain, where the degree of relevance can significantly impact the accuracy and applicability of the generated answers.

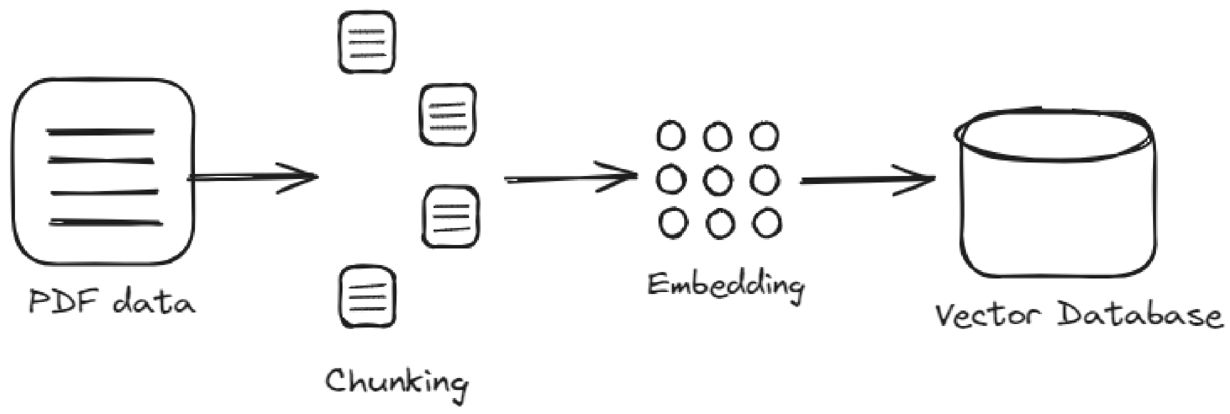
3. Handling variations: By representing the legal text in a semantic space, vector search can effectively handle variations in language, such as synonyms, misspellings, or different word forms. This capability is particularly valuable in the domain of law, where terminology can be nuanced and context-dependent, ensuring that relevant information is not overlooked due to variations in phrasing or terminology

4. Scalability: Vector search can efficiently handle large corpora of text documents, such as the extensive Indian Penal Code, as the computational complexity of similarity calculations is independent of the corpus size. This scalability is essential for maintaining the system's performance and responsiveness as the legal corpus grows or evolves.

By leveraging vector search, this RAG application can accurately retrieve the most relevant sections or chunks of the Indian Penal Code, enabling the language model to generate comprehensive and contextually appropriate answers to user queries. This approach ensures that the generated responses accurately reflect the legal provisions, while accounting for the complexities and nuances of legal language.

Chapter 4
Components of RAG

4.1 Data Ingestion



(Figure 4-1 Data Ingestion)

1. Chunking:

The initial step involves dividing the PDF data, which may contain extensive legal text or documents, into smaller, more manageable chunks. This chunking process is necessary because providing large amounts of data as context to the language model can be computationally inefficient and impractical. By breaking down the data into smaller chunks, the system can more effectively manage and retrieve relevant information.

2. Embedding:

After chunking the PDF data, each individual chunk is converted into a dense vector representation, also known as an embedding. This embedding process maps the textual content of each chunk into a high-dimensional vector space, where similar chunks or documents are positioned closer together based on their semantic similarity.

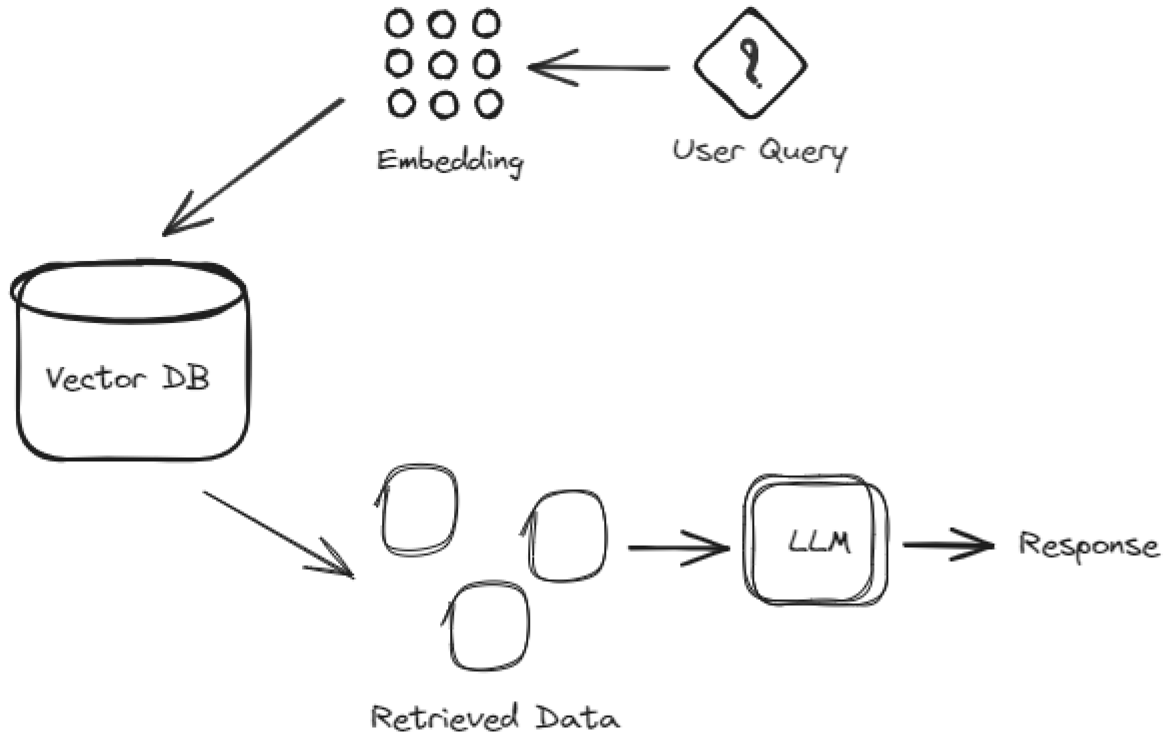
The embedding process captures the underlying meaning and context of the text, enabling the vector search to retrieve relevant chunks even if they do not contain the exact keywords or phrases from the user's query. This step is crucial for overcoming the limitations of traditional text search methods and enabling the RAG application to handle variations in language and legal terminology.

3. Vector Database:

The final step involves storing the embeddings of the text chunks in a specialized vector database. This vector database is designed to efficiently store and manage high-dimensional vector representations, enabling fast similarity searches and retrieval.

When a user submits a query to the RAG application, the query is also converted into an embedding vector. The vector database then performs a similarity search, identifying the most relevant chunks based on the proximity of their embeddings to the query embedding. These retrieved chunks serve as the context, which is provided to the language model for generating the final answer.

4.2 Retrieval Augmented Generation



(Figure 4-2 Retrieval Augmented Generation)

In this Retrieval Augmented Generation (RAG) application, the process of generating responses to user queries involves the following steps:

1. User Query Embedding:

When a user submits a query to the system, the query is first converted into a dense vector representation, also known as an embedding. This embedding process maps the textual content of the query into the same high-dimensional vector space as the previously embedded chunks of the Indian Penal Code. This step ensures that the user's query can be effectively compared to the existing embeddings in the vector database, enabling the retrieval of relevant information.

2. Similarity Search in Vector Database:

The embedded query vector is then used to perform a similarity search within the vector database, which contains the embeddings of all the chunked text from the Indian Penal Code. The vector database leverages efficient algorithms to identify the most similar embeddings to the query embedding, based on their proximity in the vector space.

3. Retrieval of Relevant Data Chunks:

The similarity search results in the retrieval of the most relevant data chunks from the Indian Penal Code, as represented by their corresponding embeddings. These retrieved chunks contain the sections or provisions that are most semantically related to the user's query, capturing the underlying meaning and context beyond mere keyword matching.

4. Context Provision to Language Model:

The retrieved data chunks serve as the context, which is then provided as input to the language model component of the RAG application. By focusing solely on the retrieved context, the language model can generate a response that is directly relevant to the user's query while adhering to the specific legal provisions and terminology found in the Indian Penal Code.

5. Response Generation:

The language model, armed with the relevant context from the retrieved data chunks, generates a comprehensive and contextually appropriate response to the user's query. This response is based on the information contained within the retrieved chunks, minimizing the risk of hallucination or the incorporation of external, potentially irrelevant information.

Chapter 5
Implementation

5.1 Insertion into vector database

Step 1: converting pdf into text chunks

Simple Directory Reader used here to convert the pdf data into text. Simply pass in a input directory or a list of files. It will select the best file reader based on the file extensions.

It returns list of strings.

```
from llama_index import SimpleDirectoryReader

documents = SimpleDirectoryReader(
    input_files=["./penal_code.pdf"]
).load_data()
```

Step 2: Preprocessing on text nodes

The clean_up_text function removes the unnecessary spaces and patterns from the documents, and returns a new list of documents named cleaned_docs.

```
import re

def clean_up_text(content: str) -> str:

    # Fix hyphenated words broken by newline
    content = re.sub(r'(\w+)-\n(\w+)', r'\1\2', content)

    # Remove specific unwanted patterns and characters
    unwanted_patterns = [
        "\\n", "  -", "__________", "_________", "_____",
        r'\\u[\dA-Fa-f]{4}', r'\uf075', r'\uf0b7'
    ]
    for pattern in unwanted_patterns:
        content = re.sub(pattern, "", content)
    # Fix improperly spaced hyphenated words and normalize
whitespace
    content = re.sub(r'(\w)\s*-\s*(\w)', r'\1-\2', content)
    content = re.sub(r'\s+', ' ', content)
    return content

# Call function
cleaned_docs = []
for d in documents:
    cleaned_text = clean_up_text(d.text)
    d.text = cleaned_text
    cleaned_docs.append(d)
```

Step 3: Defining Embedding

```
embed_model=LangchainEmbedding(
    HuggingFaceEmbeddings(model_name="BAAI/bge-base-en-v1.5"))
```

BGE is short for BAAI general embedding. This model is specifically created to achieve tasks like feature extraction and sentence similarity.

There are many models available of BAAI family. The one we are using "**BAAI/bge-base-en-v1.5**" trained only on English language. This is the base version in which vector dimension is 768.

Step 4: Defining Storage Context

This code defines the vector database to store the embeddings.

```
vector_store = PineconeVectorStore(pinecone_index=pinecone_index)
storage_context =
StorageContext.from_defaults(vector_store=vector_store)
```

Step 5: Defining Sentence window parser

Sentence Window Node Parser:

The SentenceWindowNodeParser is a parser provided by LlamaIndex that is designed to split text into "windows" of sentences.

The SentenceWindowNodeParser works by sliding a window of a specified size (in terms of the number of sentences) across the input text. For each window position, it creates a new chunk or "node" containing the sentences within that window. The windows overlap with each other, ensuring that neighboring sentences are included in multiple nodes, providing additional context.

By using the SentenceWindowNodeParser, the RAG application can retrieve not only the most relevant sentences or chunks but also the additional context needed to accurately interpret and understand the retrieved information. This additional context is crucial for the language model to generate coherent and contextually appropriate responses to user queries.

```
# Set up sentence windowing
node_parser = SentenceWindowNodeParser.from_defaults(
    window_size=3,
    window_metadata_key="window",
    original_text_metadata_key="original_text",
)

# Create the ServiceContext with sentence windowing
sentence_context = ServiceContext.from_defaults(
    llm=llm,
    embed_model=embed_model,
    node_parser=node_parser,)
```

Step 6: Storing chunks into vector database

Below code takes cleaned_docs as an argument, generates vector embeddings defined in the service_context, and stores the embeddings into vector store defined in the storage_context.

```
# Create the index with the ServiceContext
sentence_index = VectorStoreIndex.from_documents(
        cleaned_docs, service_context=sentence_context,
storage_context=storage_context
)
```

By default VectorStoreIndex stores in a In-Memory database. Here, I provided the storage context created before with pinecone vector database configuration.

5.2 Retrieving the relevant data and construct answer using llm

Step 1: Connect with the pinecone database

```
pc = Pinecone(api_key=api_key)
pinecone_index = pc.Index("test")
vector_store = PineconeVectorStore(pinecone_index)
```

Step 2: Defining the same embedding used in the insertion

```
embed_model=LangchainEmbedding(
    HuggingFaceEmbeddings(model_name="BAAI/bge-base-en-v1.5"))
```

Step 3: Using Inference API

Below code uses Mistral model through the Hugging Face Inference API which is completely free to use. The main advantage of using inference API is that it uses its own computer resources for LLM.

```
HF_TOKEN = "hf_XXXXXXXXXXXXXXXXXXXXXXX"
llm = HuggingFaceInferenceAPI(
    model_name="mistralai/Mistral-7B-Instruct-v0.2",
token=HF_TOKEN
)
```

Step 4: Making index instance from the vector database

Below code used to load the vector indexes from the vector database. “index” has all the embeddings which are stored in the vector database.

```
from llama_index import VectorStoreIndex, ServiceContext

service_context = ServiceContext.from_defaults(
    llm=llm,
    embed_model=embed_model,
)
```

```
index =
VectorStoreIndex.from_vector_store(vector_store=vector_store,
service_context=service_context)
```

Step 5: Create an engine to perform question answering task

This function is responsible for creating a `RetrieverQueryEngine` instance, which is used for question answering tasks using a vector index and a language model.

Here's what the function does:

1. It defines two post-processors: `MetadataReplacementPostProcessor` and `SentenceTransformerRerank`. The `MetadataReplacementPostProcessor` is used to replace placeholders in the output with metadata from the input, and the `SentenceTransformerRerank` is used to rerank the retrieved sentences based on their relevance to the query.

2. It creates a `ResponseSynthesizer` instance using the `get_response_synthesizer` function. This synthesizer is responsible for generating the final answer to the query based on the retrieved sentences.

3. It creates a `VectorIndexRetriever` instance, which is used to retrieve relevant sentences from the vector index based on the query.

4. It returns a `RetrieverQueryEngine` instance, which combines the retriever, the response synthesizer, and the post-processors. This engine is responsible for executing the end-to-end question answering process.

```
def get_sentence_window_query_engine(
    qa_prompt_tmpl,
    service_context,
    index,
    similarity_top_k=6,
    rerank_top_n=3,
):
    # define postprocessors
    postproc =
MetadataReplacementPostProcessor(target_metadata_key="window")
```

```
    rerank = SentenceTransformerRerank(
        top_n=rerank_top_n, model="BAAI/bge-reranker-base"
    )
    response_synthesizer_sentence = get_response_synthesizer(
      service_context=service_context,
      response_mode=ResponseMode.COMPACT,
      text_qa_template=Prompt(qa_prompt_tmpl,
prompt_type=PromptType.QUESTION_ANSWER),
      refine_template=Prompt(DEFAULT_REFINE_PROMPT_TMPL,
prompt_type=PromptType.REFINE),
    )

    # build retriever
    retriever = VectorIndexRetriever(
        index=index,
        similarity_top_k=6,
    )
    sentence_window_engine = RetrieverQueryEngine(
        retriever=retriever,
response_synthesizer=response_synthesizer_sentence,
node_postprocessors=[postproc, rerank]
    )

    return sentence_window_engine
```

Use of Re-Ranker:

The ReRanker is a component that aims to refine and reorder the initial set of retrieved results based on their relevance to the user's query. While the initial retrieval process may return a list of potentially relevant chunks or sentences, the ReRanker helps to further optimize this list by reranking the results based on a more sophisticated relevance scoring mechanism.

The SentenceTransformerRerank class is used to create a ReRanker instance. This class leverages pre-trained sentence transformer models, such as the "BAAI/bge-reranker-base" model, to compute semantic similarity scores between the user's query and the retrieved chunks or sentences.

The "BAAI/bge-reranker-base" model is a fine-tuned version of the Sentence-BERT model, specifically optimized for reranking tasks. It has been trained on a diverse set of question-answering and information retrieval datasets, enabling it to capture the semantic relationships between queries and passages more accurately.

By incorporating this ReRanker into the RAG application, the following benefits are achieved:

1. Improved Relevance Ranking:

   The ReRanker employs advanced language understanding techniques to assess the semantic similarity between the user's query and the retrieved chunks or sentences. This allows for a more accurate ranking of the results, ensuring that the most relevant information is presented at the top of the list.

2. Context-Aware Ranking:

   The "BAAI/bge-reranker-base" model is designed to consider the broader context of the retrieved information, rather than relying solely on keyword matching. This context-awareness is crucial in the legal domain, where the meaning and interpretation of a section or clause often depend on the surrounding context.

3. Increased Accuracy and Comprehensiveness:

   By refining and reranking the retrieved results, the ReRanker helps to improve the overall accuracy and comprehensiveness of the generated responses. The language model can rely on the most relevant and contextually appropriate information, leading to more accurate and insightful answers to user queries.

Step 6: Fire the query

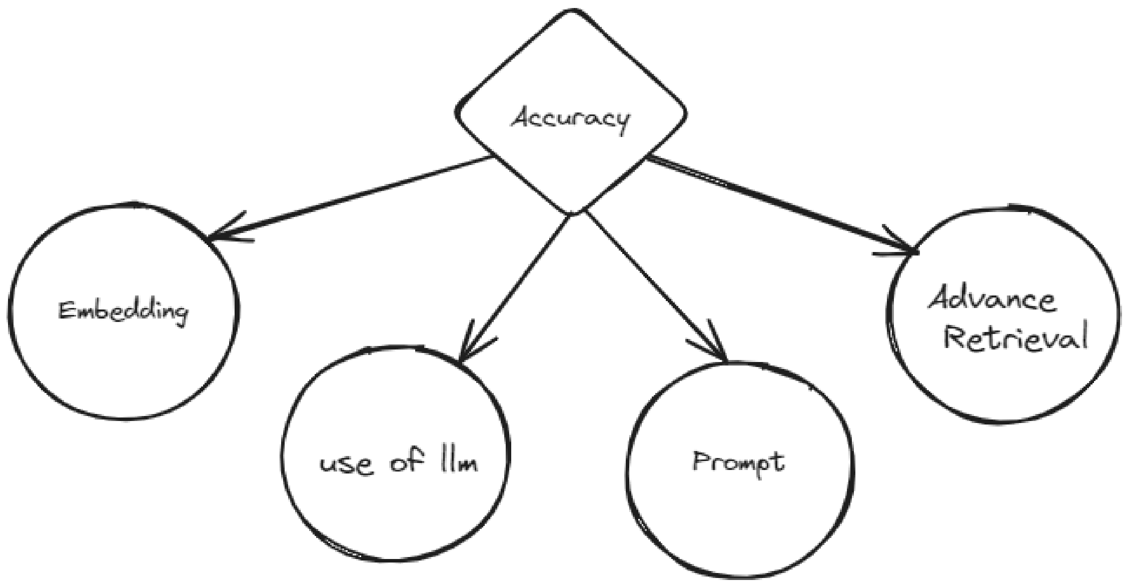
```
sentence_window_engine =
get_sentence_window_query_engine(qa_prompt_tmpl, service_context,
index)

window_response = sentence_window_engine.query(
    "what is the punishment for making false claim in court?"
)
```

```
print(str(window_response))
```

```
The punishment for making a false claim in court is imprisonment
for a term which may extend to two years, and the person is also
liable to a fine as per section 209 of the penal code.
```

Chapter 6
Evaluation and Observation



(Figure 6-1 Methods to Improve Accuracy)

6.1 Evaluation and Observation

Evaluation and Observation

In this project, the evaluation process involves assessing the performance of the system using two main criteria: faithfulness and relevancy. The evaluation is carried out through the implementation of the `FaithfulnessEvaluator` and `RelevancyEvaluator` classes from the LlamaIndex library.

Faithfulness Evaluation

The `FaithfulnessEvaluator` is designed to evaluate the faithfulness of the system's responses to the source data. It checks whether the generated output accurately reflects the information present in the underlying data source. The faithfulness evaluation process involves comparing the system's responses with the original data to ensure that no contradictory or unsupported statements are made.

Relevancy Evaluation

The `RelevancyEvaluator` assesses the relevancy of the system's responses to the given queries. It determines whether the generated output is relevant and pertinent to the user's query. The relevancy evaluation process considers factors such as the context of the query, the information needs of the user, and the appropriateness of the response to the specific query.

Batch Evaluation

To streamline the evaluation process, the `BatchEvalRunner` class is employed. This class allows for the simultaneous evaluation of multiple queries using the `FaithfulnessEvaluator` and `RelevancyEvaluator` instances. In the provided code snippet, the top 10 queries from the `queries` list are selected for evaluation.

The evaluation results are stored in the `eval_results` dictionary, which contains two keys: `'faithfulness'` and `'relevancy'`. Each key corresponds to a list of evaluation results for the respective criterion.

Evaluation Scores

The final faithfulness and relevancy scores are calculated by summing the number of passing results and dividing by the total number of results. In the provided code, the faithfulness score and relevancy score are both reported as 0.9, indicating that 90% of the evaluated responses were considered faithful and relevant, respectively.

These evaluation scores provide quantitative measures of the system's performance, allowing for the assessment of its accuracy, reliability, and suitability for the intended task.

Code

```
from llama_index.evaluation import FaithfulnessEvaluator
faithfulness =
FaithfulnessEvaluator(service_context=service_context_llm)

from llama_index.evaluation import RelevancyEvaluator
relevancy =
RelevancyEvaluator(service_context=service_context_llm)

# Batch Evaluation
from llama_index.evaluation import BatchEvalRunner

# Let's pick top 10 queries to do evaluation
batch_eval_queries = queries[:10]

# Initiate BatchEvalRunner to compute FaithFulness and Relevancy
Evaluation.
runner = BatchEvalRunner(
    {"faithfulness": faithfulness, "relevancy": relevancy},
    workers=8,
)

# Compute evaluation
eval_results = await runner.aevaluate_queries(
    sentence_window_engine, queries=batch_eval_queries
)

# Let's get faithfulness score
faithfulness_score = sum(result.passing for result in
eval_results['faithfulness']) / len(eval_results['faithfulness'])
faithfulness_score

0.9

relevancy_score = sum(result.passing for result in
eval_results['relevancy']) / len(eval_results['relevancy'])
relevancy_score

0.9
```

6.2 Methods to improve accuracy

6.2.1 Embedding:

One of the key factors contributing to the accuracy of the RAG application is the quality of the embeddings used to represent the text data. While a longer embedding vector dimension can potentially capture more nuanced information, leading to improved accuracy, it is not the sole determinant of embedding effectiveness. The most crucial aspect is the nature of the data used to train the embedding model.

In the context of this RAG application tailored to the Indian Penal Code, the accuracy of the retrieved information and the subsequent generation of responses can be significantly enhanced by employing an embedding model that has been specifically trained on legal documents, particularly those related to Indian law. By leveraging an embedding model that has been exposed to and optimized for the intricate language and domain-specific terminology of the legal domain, the application can better capture the semantic nuances and contextual dependencies present in the Indian Penal Code.

Furthermore, an embedding model specifically tailored for sentence similarity tasks in the legal domain would enhance the application's ability to identify and retrieve the most relevant chunks or sections from the Indian Penal Code. By accurately understanding the contextual similarities between the user's query and the available data, the application can retrieve the most pertinent information, resulting in more accurate and comprehensive responses.

By focusing on using a domain-specific embedding model optimized for the legal domain and complementing it with additional accuracy-enhancing techniques, the RAG application can achieve higher levels of precision and reliability in retrieving relevant information from the Indian Penal Code and generating accurate and insightful responses to user queries.

6.2.2 Use of LLM:

In the context of this Retrieval Augmented Generation (RAG) application, the inclusion of a large language model (LLM) plays a crucial role in enhancing the accuracy and effectiveness of the retrieval process. The need for providing an LLM arises in two distinct scenarios, each serving a specific purpose within the application's workflow. These scenarios are as follows:

Case 1: Providing an LLM during the data ingestion and indexing phase

In the first case, an LLM is provided during the creation of the index from the source documents, as demonstrated in the following code snippet:

```
sentence_context = ServiceContext.from_defaults(
   llm=llm,
   embed_model=embed_model,
   node_parser=node_parser,)
# Create the index with the ServiceContext
sentence_index = VectorStoreIndex.from_documents(
   cleaned_docs,
   service_context=sentence_context,
   storage_context=storage_context
)
```

The inclusion of an LLM in the `ServiceContext` during this phase serves two primary purposes:

1. Text Summarization and Compression:

When ingesting large documents or datasets, such as the Indian Penal Code, the LLM can be leveraged to summarize and compress the text, ensuring that the most relevant and concise information is preserved in the indexed data chunks. This summarization process can improve the overall quality and relevance of the retrieved information, as well as reduce the computational overhead associated with storing and processing unnecessary or redundant data.

2. Context-Aware Chunking:

By providing an LLM during the indexing phase, the chunking process can be enhanced to consider the contextual relationships and dependencies within the text. The LLM can assist in identifying logical boundaries for chunking, ensuring that related information is kept together within the same chunk, while separating unrelated or disconnected content. This context-aware chunking can improve the coherence and completeness of the retrieved information, ultimately leading to more accurate and comprehensive responses.

Case 2: Providing an LLM during the vector store index initialization

In the second case, an LLM is provided during the initialization of the `VectorStoreIndex` from an existing vector store, as shown in the following code snippet:

```
from llama_index import VectorStoreIndex, ServiceContext
service_context = ServiceContext.from_defaults(
    llm=llm,
    embed_model=embed_model,
)
index = VectorStoreIndex.from_vector_store(
    vector_store=vector_store,
    service_context=service_context
)
```

In this scenario, the data has already been ingested, chunked, and indexed into a vector store. However, the inclusion of an LLM in the `ServiceContext` is still beneficial for the following reasons:

1. Query Expansion and Reformulation:

   During the retrieval process, the LLM can be utilized to expand and reformulate the user's query, enhancing its coverage and increasing the likelihood of retrieving relevant information. This is particularly valuable in the legal domain, where queries may involve complex terminology, contextual dependencies, or ambiguities that can be better understood and addressed by an LLM.

2. Response Generation and Refinement:

   The LLM plays a crucial role in generating coherent and informative responses based on the retrieved information. By leveraging the LLM's natural language understanding and generation capabilities, the application can produce responses that accurately convey the relevant legal provisions while accounting for the nuances and complexities of the domain.

It is important to note that while an LLM is not strictly required during the vector store index initialization if the data has already been ingested and indexed, its inclusion can significantly enhance the application's ability to interpret and understand user queries, retrieve contextually relevant information, and generate accurate and insightful responses.

6.2.3 Prompt

In the context of a Retrieval Augmented Generation (RAG) application, crafting an effective prompt for the language model is crucial in guiding it to generate accurate and relevant responses. A well-designed prompt can significantly enhance the quality of the generated answers by providing clear instructions and context to the language model.

In the case of your RAG application for the Indian Penal Code, you have tweaked the default prompt to the following:

```
qa_prompt_tmpl = (
    "Context information is below.\n"
    "---------------------\n"
    "<context>\n"
    "{context_str}\n"
    "</context>\n"
    "---------------------\n"
    "Instructions are given below.\n"
    "answer the query based on given Context. \n"
    "If you do not find the answer from Context then write
response like below. \n"
    "Do not able to answer this question.\n"
    "Query is given below.\n"
    "<query>\n"
    "{query_str}\n"
    "</query>\n"
    "Take a deep breath and provide the answer from the context
and not from the prior knowledge.\n")
```

This customized prompt includes several key elements that can contribute to the generation of accurate and relevant answers:

1. Context Identification:

   The prompt clearly separates the context information from the query, making it easier for the language model to identify and focus on the relevant sections or chunks retrieved from the Indian Penal Code.

2. Explicit Instructions:

   The prompt provides explicit instructions to the language model, guiding it to answer the query based solely on the given context. This instruction helps mitigate the risk of hallucination, where the model generates responses based on its prior knowledge rather than the provided information.

3. Fallback Response:

   The prompt includes a fallback response instruction, which directs the language model to indicate when it cannot find an answer within the given context. This feature can enhance the transparency and reliability of the system by acknowledging the limitations of the available information.

4. Query Identification:

   The prompt clearly delineates the user's query, making it easier for the language model to understand and focus on the specific information being requested.

5. Reinforcement of Context-Based Answering:

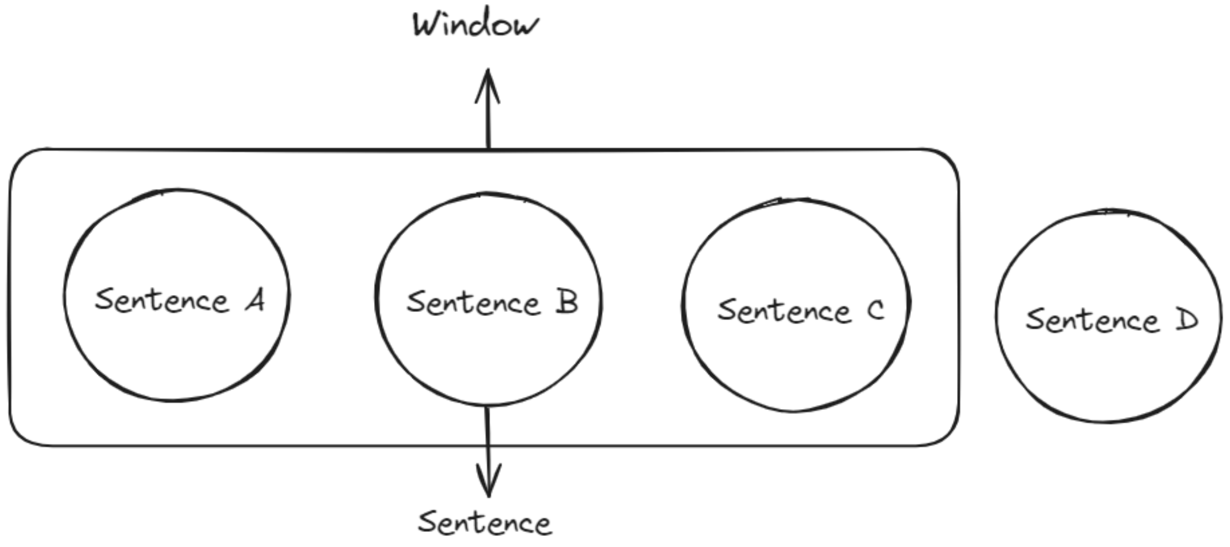
   The prompt includes a reinforcement instruction, reminding the language model to "provide the answer from the context and not from the prior knowledge." This reiteration helps to reinforce the desired behavior of the model, reducing the likelihood of generating responses based on external or potentially irrelevant information.

6.2.4 Advance Retrieving Techniques:

Sentence Window Retrieval:

Sentence window retrieval is a technique that aims to improve the relevance of the retrieved text by considering not only the individual sentences but also their surrounding context. Instead of treating each sentence as an isolated unit, this method retrieves a window of sentences around the most relevant sentence. The idea is that the meaning and context of a sentence can be better understood when viewed in conjunction with the sentences before and after it.

For example, consider the following document snippet:



(Figure 6-2 Sentence Window Retrieval)

If Sentence B is identified as the most relevant sentence, sentence window retrieval would retrieve not only Sentence B but also the surrounding sentences (Sentence A and Sentence C) to provide additional context. By retrieving a window of sentences, the system can capture more relevant information and provide more accurate and comprehensive answers.

Chapter 7
Tools and Technology

The main Technology used to do all this steps from chunking to retrieval is LlamaIndex.

LlamaIndex:

LlamaIndex, formerly GPT Index, is a Python data framework designed to manage and structure LLM-based applications, with a particular emphasis on storage, indexing and retrieval of data.

LlamaIndex provides a complete set of tools to automate tasks such as data ingestion from heterogeneous sources (PDF files, Web pages, ...) and retrieval-augmented generation (RAG); it also features a rich ecosystem of plugins that make it possible to connect with third-party components, from vector stores to data readers.

Reasons for using LlamaIndex:

When working with LLMs, one often needs to augment the power of a model (which comes equipped with general knowledge already) by supplying domain-specific, possibly proprietary data, such as internal reports, a corpus of PDF presentations, and so on.

Effective, reproducible management of this domain-specific data needed to augment the LLMs requires a fair amount of machinery, which is precisely what LlamaIndex offers: the framework supports several indexing and retrieval techniques, ranging from simple to quite sophisticated, and offers plugin for seamless integration with a variety of third-party products.

Hugging Face:

Hugging Face is a prominent open-source library and platform for natural language processing (NLP) models and tools. In the context of this Retrieval Augmented Generation (RAG) application for the Indian Penal Code, Hugging Face played a crucial role in two key aspects:

Large Language Model Inference: Hugging Face provided an interface to leverage the powerful "Mistral-7B-Instruct-v0.2" language model from mistralAI for generating accurate and contextual responses based on the retrieved information from the vector database.

Text Embedding: The library facilitated the integration of the "BAAI/bge-base-en-v1.5" model, a state-of-the-art text embedding model, which was utilized to convert the textual data into dense vector representations, enabling efficient similarity searches and retrieval from the vector database.

Pinecone:

Pinecone is a managed vector database service that offers efficient storage and querying capabilities for high-dimensional vectors. Some key features and advantages of using Pinecone as the vector store in this RAG application include:

1. Fast retrieval: Pinecone leverages advanced indexing and querying techniques, ensuring fast and accurate similarity searches, which are crucial for retrieving relevant context from the vector database during query processing.

2. Ease of integration: Pinecone provides client libraries and APIs for seamless integration with various programming languages and frameworks, simplifying the development process.

Chapter 8
Conclusion

The development of this Retrieval Augmented Generation (RAG) application tailored to the Indian Penal Code represents a significant stride towards enhancing access to legal information and promoting efficient query resolution. By leveraging state-of-the-art natural language processing techniques and cutting-edge technologies, this application addresses the challenges of retrieving relevant information from extensive legal corpora and generating accurate and contextually appropriate responses.

The implementation of the RAG approach, which combines retrieval and generation components, has proven to be an effective solution for mitigating the risk of hallucination inherent in traditional language models. By retrieving and providing only the most pertinent chunks of information from the Indian Penal Code as context, the application ensures that the generated responses are grounded in the specific legal provisions, minimizing the potential for erroneous or irrelevant information.

The integration of vector search and embedding techniques has played a pivotal role in enhancing the accuracy and relevance of the retrieved information. By representing the textual data in a high-dimensional semantic space, the application can effectively capture the nuances and contextual dependencies present in legal language, enabling precise retrieval of relevant sections and clauses.

The modular and extensible design of the application, facilitated by the adoption of robust frameworks and libraries like LlamaIndex, and Hugging Face, has fostered a seamless development process and facilitated the integration of various components, including language models, embedding models, and vector databases.

Chapter 9
Future Extension

While the current implementation of the Retrieval Augmented Generation (RAG) application for the Indian Penal Code has demonstrated promising results, there are several avenues for further enhancement and expansion:

1. Including other Indian Legal documents:

   Extending the application's scope to incorporate other crucial Indian legal documents, such as the Indian Constitution, Civil Law, Company Law, Property Law, and more, would significantly broaden its utility and applicability. By integrating these diverse legal corpora, the application could serve as a comprehensive legal knowledge hub, catering to a wider range of queries and use cases.

2. Providing an interface to easily chat with the application:

   Developing an intuitive and user-friendly interface that allows users to engage in natural language conversations with the application could greatly enhance its accessibility and usability. By enabling a conversational interaction mode, users could pose queries, receive responses, and follow up with additional questions or clarifications in a seamless and efficient manner, fostering a more natural and engaging experience.

By incorporating these future extensions, the RAG application would evolve into a comprehensive legal knowledge repository and intelligent assistant, capable of addressing a broader range of legal queries and catering to diverse user needs across various domains of Indian law.

Bibliography

References:

[1] Leonie Monigatti · Follow (2023). _A Guide on 12 Tuning Strategies for Production-Ready RAG Applications_. Towards Data Science. https://towardsdatascience.com/a-guide-on-12-tuning-strategies-for-production-ready-rag-applications-7ca646833439

[2] Plaban Nayak · Follow (2024). _Advanced RAG using Llama Index_. AI Planet. https://medium.aiplanet.com/advanced-rag-using-llama-index-e06b00dc0ed8

[3] Pinecone Team (n.d.). LlamaIndex Integration. Pinecone Docs. https://docs.pinecone.io/integrations/llamaindex

[4] LlamaIndex Developers (2023). _"Optimization by Prompting" for RAG_. LlamaIndex Documentation. https://docs.llamaindex.ai/en/stable/examples/prompts/prompt_optimization.html

[5] Cameron R. Wolfe, Ph.D. · Follow (2023). _Advanced Prompt Engineering: What to do when few-shot learning isn't enough..._ Towards Data Science. https://towardsdatascience.com/advanced-prompt-engineering-f07f9e55fe01

[6] LlamaIndex Developers (2023). _Chat Prompts Customization_. LlamaIndex Documentation. https://docs.llamaindex.ai/en/stable/examples/customization/prompts/chat_prompts/

Dataset:

https://lddashboard.legislative.gov.in/sites/default/files/A1860-45.pdf