THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

**DATS 6312.11**
**NLP for Data Science**
**Final Project**
**Fall 2024**
**Due**: *12/5/2024*

# AI Meets Law: Transforming Legal Research with RAG and Fine-Tuned LLM

## Individual Project Report

**for the course**

**DATS 6312.11 'Natural Language Processing'**

**Fall 2024**

**Name: Srivallabh Siddharth Nadadhur**

# 1. Introduction:

## 1.1 Project Overview

We aim to transform the way legal research is conducted through our project. Legal research traditionally involves sifting through extensive repositories of case laws, statutes, and legal opinions, often requiring significant time and effort. Our system addresses these challenges by integrating Retrieval-Augmented Generation (RAG) with fine-tuned Large Language Models (LLMs), creating a powerful tool for efficient data retrieval and context-aware response generation. By combining robust search capabilities with the generative power of AI, we enable users to quickly access precise and relevant legal information. To make this technology accessible, we developed a user-friendly Streamlit application that ensures seamless interaction, allowing legal professionals and researchers to focus on decision-making rather than data retrieval.

# 1.2 Outline of Shared Work:

### 1.2.1 RAG Pipeline Development:

- Designed and implemented the Retrieval-Augmented Generation (RAG) pipeline to enable efficient querying and analysis of legal case files.
- Created a robust document retrieval mechanism using FAISS for embedding-based searches, ensuring relevant context is retrieved from the knowledge base.
- Incorporated advanced filtering and thresholding to enhance the quality of retrieved documents.

### 1.2.2 Query and Test Case Processing:

- Developed dual user interaction modes:
    - Query Mode: Allows users to input legal queries and receive concise, actionable insights from the model.
    - Test Case Mode: Enables users to input a test case file and retrieve an AI-generated expected verdict based on the legal arguments presented.
- Implemented features to handle large legal documents, including summarization for lengthy inputs to ensure compatibility with the model's token limits.

### 1.2.3 Enhanced Context Tracking and Memory Management:

- Incorporated a context tracking mechanism to store and reference the user's last query and response.
- Optimized memory management by limiting interaction history to essential exchanges, ensuring streamlined operations and user experience.

# 2. Data Extraction:

## 2.1 Purpose and Objective

We extracted legal data from publicly available sources to ensure comprehensive and diverse datasets for building an AI-powered legal research system. The focus was on gathering case law data, constitutional law references, and supplementary datasets generated using AI tools for diverse scenarios.

## 2.2 Sources of Data

- **Virginia Cases:** Extracted from the Case.Law API, featuring a JSON structure containing metadata and case details, such as case name, decision date, court jurisdiction, and detailed case opinions(0008-01).
- **Virginia Constitution Law:** Data files sourced from the Virginia Law Library, containing legislative texts and statutes in structured JSON format.
- **AI-Generated Dataset:** Created using ChatGPT, saved in JSON format, providing custom examples for specific legal queries and responses.

## 2.3 Data Extraction for Virginia Cases

To extract case law data for Virginia, we developed a custom Python script tailored to automate the process of retrieving, extracting, and organizing legal case files from the Case.Law website. This was necessary as the data was available in bulk ZIP archives that required processing to make it usable for our AI-powered legal research system. The script efficiently handles tasks such as scraping links, downloading ZIP files, extracting JSON files, and organizing them into a structured format for further analysis.

```python
# Define the dictionary of folder names and bulk data links
folders_and_links = {
    "Howison_1850-1851": "https://static.case.law/howison/",
    "Jeff_1730-1772": "https://static.case.law/jeff/",
    "Va_1779-2004": "https://static.case.law/va/",
    "Va_app_1985-2017": "https://static.case.law/va-app/",
    "Va_Ch_Dec_1788-1799": "https://static.case.law/va-ch-dec/",
    "Va_cir_1856-2016": "https://static.case.law/va-cir/",
    "Va_col_dec_1729-1741": "https://static.case.law/va-col-dec/",
    "Va_dec_1871-1900": "https://static.case.law/va-dec/",
    "Va_Patt_Health_1855-1857": "https://static.case.law/va-patt-heath/"
}

def get_zip_links(base_url):
    """
    Scrape the base URL to get all .zip links for a given bulk data page.
    """
    response = requests.get(base_url)
    soup = BeautifulSoup(response.content, "html.parser")
    zip_links = []

    # Find all .zip links
    for link in soup.find_all("a", href=True):
        if link['href'].endswith(".zip"):
            if link['href'].startswith("http"):  # Full URL
                zip_links.append(link['href'])
            else:  # Relative path
                zip_links.append(base_url + link['href'])
    return zip_links

def download_and_extract_json_single_folder(zip_link, target_folder):
    """
    Download a .zip file, extract JSON files, and save them in a single folder.
    """
    # Download the .zip file
    zip_file_name = zip_link.split("/")[-1]
    zip_file_path = os.path.join(target_folder, zip_file_name)
    print(f"Downloading: {zip_link}")

    with requests.get(zip_link, stream=True) as r:
        with open(zip_file_path, "wb") as f:
            for chunk in r.iter_content(chunk_size=8192):
                f.write(chunk)
    print(f"Downloaded: {zip_file_path}")

    # Extract only JSON files from the zip
    with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
        for file in zip_ref.namelist():
            if file.endswith(".json"):
                json_file_name = os.path.basename(file)
                json_file_path = os.path.join(target_folder, json_file_name)

                with zip_ref.open(file) as source, open(json_file_path, "wb") as target:
                    target.write(source.read())
    print(f"Extracted JSON files into {target_folder}")

    # Remove the .zip file after extraction
    os.remove(zip_file_path)
    print(f"Removed zip file: {zip_file_path}")

# Target folder for all JSON files
target_folder = os.path.join(BASE_FOLDER, "All_JSON_Files")
os.makedirs(target_folder, exist_ok=True)

# Step 3: Process all folders and links
for main_folder_name, base_url in folders_and_links.items():
    print(f"Processing: {main_folder_name} ({base_url})")
    zip_links = get_zip_links(base_url)
    print(f"Found {len(zip_links)} .zip links for {main_folder_name}")

    # Process each zip link
    for zip_link in zip_links:
        download_and_extract_json_single_folder(zip_link, target_folder)

print(f"All JSON files have been downloaded and extracted into {target_folder}")
```

**Figure 1: Python Script for Extracting Virginia Cases**

**The script addresses the challenges of manually handling bulk data files by performing the following steps:**

- **Scraping Bulk Data Links:** Using Python's BeautifulSoup library, the script parses the Case.Law webpage to extract all available ZIP file links containing case law data. These links are dynamically compiled from the webpage's href attributes, ensuring completeness and accuracy.
- **Downloading and Extracting Files:** Each ZIP file is downloaded via the requests library and extracted using the zipfile module. The script handles large data volumes by processing files in batches and storing the extracted JSON files locally.

- **Organizing Extracted Data:** Extracted JSON files are categorized into separate folders based on their source directories for traceability. The script consolidates all files into a single directory, ALL_JSON_Files, to simplify subsequent preprocessing.

### 2.4 AI-Generated Dataset

The AI-generated dataset, created using ChatGPT and saved in JSONL format (output.jsonl), provides custom examples of legal queries and responses to simulate real-world scenarios. Each entry in the dataset consists of a query, such as questions about statutes, case interpretations, or procedural law, along with a contextually relevant response generated by ChatGPT. Covering diverse topics like constitutional law, civil litigation, and contract disputes, this dataset supplements publicly available data by offering additional training and validation examples. It enables the AI system to handle a wide range of legal questions, test its accuracy, and evaluate its performance on uncommon or edge-case queries, making it a versatile and essential resource for this project.

## 3. Data Preprocessing:

In our project, preprocessing focused on transforming raw legal data into a structured and enriched format ready for analysis. This involved extracting essential fields from JSON files, cleaning and normalizing text, and generating embeddings to represent the semantic meaning of the content. The embeddings were stored in a reusable format, facilitating efficient search and retrieval in downstream tasks. The preprocessing pipeline ensures high-quality input data, which is critical for the accuracy and reliability of the AI system.

### 3.1 Data Parsing and Structuring

In this step, we processed raw legal data from case law files and the Virginia Constitution to prepare it for further analysis. The goal was to extract, clean, and organize the data into a structured format to ensure consistency and usability. Two key functions were implemented to achieve this:

```python
def load_json_files(directory: str) -> List[Dict]:
    print(f"Loading JSON files from directory: {directory}")
    files = Path(directory).glob("*.json")
    cases = []
    for file in files:
        with open(file, "r", encoding="utf-8") as f:
            case = json.load(f)
            for opinion in case.get("casebody", {}).get("opinions", []):
                enriched_case = {
                    "id": case.get("id", ""),
                    "case_name": case.get("name", ""),
                    "text": opinion.get("text", "").strip(),
                    "decision_date": case.get("decision_date", ""),
                    "jurisdiction": case.get("jurisdiction", {}).get("name", "Unknown"),
                }
                cases.append(enriched_case)
    print(f"Loaded {len(cases)} cases successfully.")
    return cases

2 usages
def load_va_constitution(file_path: str) -> List[Dict]:
    with open(file_path, 'r', encoding='utf-8') as f:
        text = f.read()
    sections = text.split('\n\n')
    va_sections = []
    for idx, section in enumerate(sections):
        if section.strip():
            va_sections.append({
                'id': f'va_constitution_{idx}',
                'text': section.strip(),
                'case_name': f'VA Constitution Section {idx}',
            })
    return va_sections
```

**Figure 2: Preprocessing Legal Data for Case Law and Constitution Sections**

- **Case Data Processing:** The load_json_files function extracts critical information such as case_name, text, decision_date, and jurisdiction from JSON files containing legal cases. The function processes the casebody section, cleans the text by removing unnecessary whitespace, and organizes the extracted information into a list of structured dictionaries.
- **Constitution Data Structuring:** The load_va_constitution function processes the Virginia Constitution text file by splitting it into sections based on double newlines (\n\n). Each section is assigned a unique ID, labeled with a descriptive section name, and saved as structured dictionaries for easy retrieval and analysis.

## 3.2 Embedding Generation

To enable semantic search and retrieval within the AI system, we generated embeddings for each legal text entry. These embeddings are vector representations that capture the semantic meaning of the text, making it suitable for tasks like similarity analysis and retrieval. Below is the code snippet used for this process:

```
3 usages
def create_embeddings(data: List[Dict], model_name="all-MiniLM-L6-v2", pkl_file="finaldata1.pkl") -> None:
    print(f"Generating embeddings for {len(data)} cases...")
    model = SentenceTransformer(model_name)
    embeddings = []
    for entry in data:
        try:
            embedding = model.encode(entry["text"], normalize_embeddings=True)
            embeddings.append({"embedding": embedding, "metadata": entry})
        except Exception as e:
            print(f"Error processing entry {entry.get('id')}: {e}")

    with open(pkl_file, "wb") as f:
        pickle.dump(embeddings, f)
    print(f"Embeddings saved to {pkl_file}.")
```

**Figure 3: Code for Generating Text Embeddings**

- **Text Encoding and Normalization:** The SentenceTransformer model (all-MiniLM-L6-v2) was used to encode legal texts into high-dimensional vectors. Each embedding was normalized to ensure consistency and accuracy during retrieval tasks.
- **Error Handling and Storage:** The embeddings and their associated metadata were saved in a serialized .pkl file (finaldata1.pkl). Errors during processing were logged for debugging, ensuring no disruption in the workflow.

## 3.3 FAISS Index Creation

To facilitate efficient similarity-based searches, we used a FAISS (Facebook AI Similarity Search) index to store and retrieve embeddings. The code snippet below outlines the process for loading precomputed embeddings and building a FAISS index:

```python
def load_faiss_index(pkl_file="finaldata1.pkl") -> Tuple[faiss.IndexFlatIP, List[Dict]]:
    print(f"Loading FAISS index from {pkl_file}...")
    with open(pkl_file, "rb") as f:
        combined_data = pickle.load(f)

    embeddings = np.vstack([entry["embedding"] for entry in combined_data])
    metadata = [entry["metadata"] for entry in combined_data]

    index = faiss.IndexFlatIP(embeddings.shape[1])
    index.add(embeddings)
    print(f"FAISS index created with {len(metadata)} items.")
    return index, metadata
```

**Figure 4: Code for Creating FAISS Index**

- **Loading Precomputed Embeddings:** The embeddings and metadata were loaded from the .pkl file (finaldata1.pkl) using Python's pickle library. The embeddings were stacked into a single NumPy array for efficient processing, while the metadata was extracted to maintain a connection to the original text.
- **Building and Populating the Index:** A FAISS IndexFlatIP (Inner Product) was created to store the embeddings. This index was populated with the embeddings, enabling high-speed similarity searches for downstream tasks.

# 4. Model Development :

The goal of the model development phase was to build an efficient AI-powered legal research system capable of retrieving and generating contextually accurate responses for legal queries. This involved fine-tuning a model on domain-specific legal data and integrating it with a Retrieval-Augmented Generation (RAG) pipeline.

## 4.1 Retrieval-Augmented Generation (RAG)

### 4.1.1 Background Information on Algorithm Development:

The Retrieval-Augmented Generation (RAG) algorithm was developed to combine the strengths of retrieval-based and generative models, addressing the limitations of each. Retrieval-based models efficiently fetch relevant documents using similarity metrics but lack the ability to generate contextually rich responses. Generative models, such as GPT, excel in producing coherent and fluent text but are constrained by their pre-trained knowledge and struggle with domain-specific tasks. RAG bridges this gap by retrieving relevant context from a knowledge base and incorporating it into the generative model's input, enabling the generation of accurate, context-aware responses. Introduced by Facebook AI, RAG leverages a dual-component framework: a

retrieval module for fetching relevant documents using FAISS and a generative module that produces detailed outputs grounded in the retrieved context. This innovative approach is particularly suited for applications like legal research, where precision, domain-specific context, and scalability are essential.

### 4.1.2 Equations:

#### 4.1.2.1 Query Embedding

The query qq is encoded into a high-dimensional vector representation eq:

$$e_q = \text{Encoder}(q)$$

#### 4.1.2.2 Similarity Search

The similarity between the query embedding eq and document embeddings eded is computed to retrieve the top k relevant documents:

$$\text{Similarity}(e_q, e_d) = e_q \cdot e_d$$

$$\text{Top-K}(e_q) = \{d_1, d_2, \ldots, d_k\}$$

#### 4.1.2.3 Context Construction

The retrieved documents are concatenated into a single context CC:

$$C = d_1 + d_2 + \cdots + d_k$$

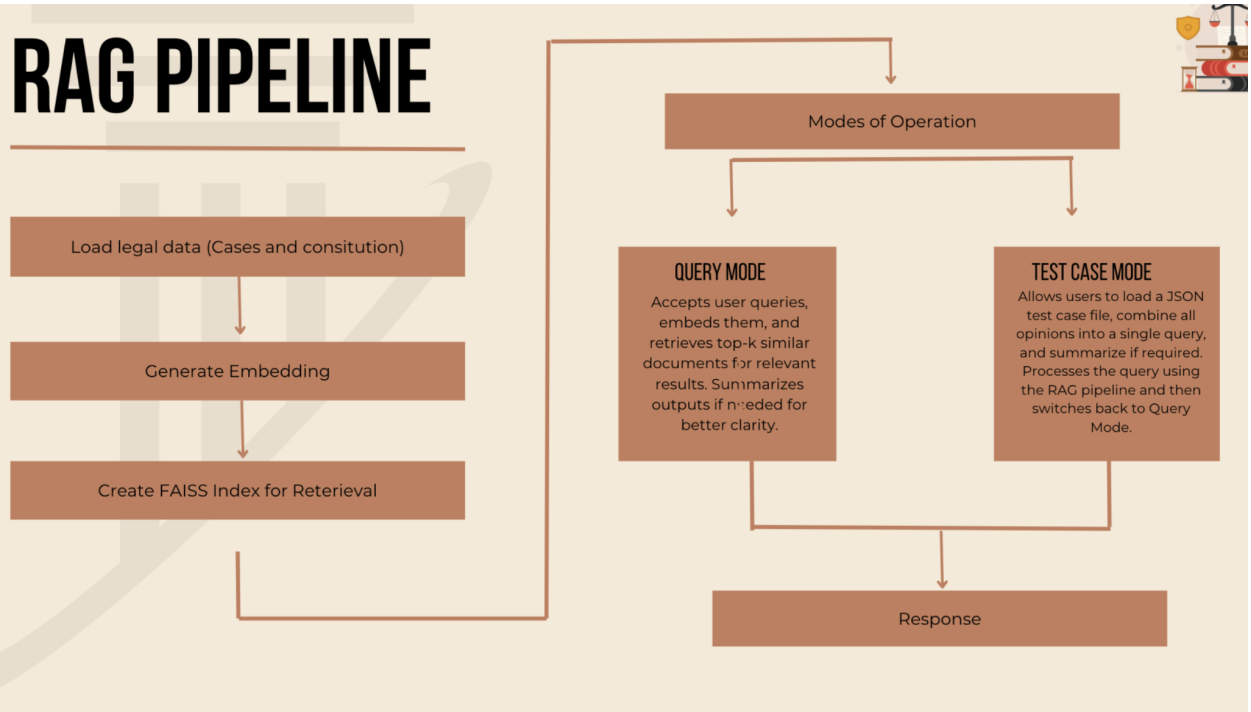
#### 4.1.2.3 Response Generation

The generative model GG produces the response RR using the query qq and the retrieved context CC:

$$R = G(q, C)$$

### 4.1.3. RAG Pipeline:

The provided diagram illustrates the functioning of the **Retrieval-Augmented Generation (RAG) pipeline**, which integrates retrieval and generation techniques to process legal queries efficiently. Here's how the pipeline works step-by-step:




## 1. Input Handling:

- Users input a query in natural language (e.g., “What rights are protected under the Virginia Constitution?”).
- The query is processed as a string for further steps.

## 2. Query Embedding:

- The input query is converted into a high-dimensional vector embedding using the pre-trained **SentenceTransformer** model (all-MiniLM-L6-v2).
- This embedding represents the semantic meaning of the query, enabling similarity-based retrieval.

## 3. Document Retrieval:

- The query embedding is compared to precomputed document embeddings stored in the **FAISS index**.
- The FAISS index efficiently retrieves the top kk documents that are most relevant to the query based on cosine similarity or inner product.

- These retrieved documents provide the context necessary for generating accurate responses.

## 4. Context Construction:

- The retrieved documents are concatenated into a single context, along with their metadata (e.g., case names, jurisdiction, text excerpts).
- This context serves as additional input for the generative model, ensuring the response is grounded in relevant legal data.

## 5. Response Generation:

- The concatenated context and the query are passed to the fine-tuned GPT model.
- The model uses this input to generate a detailed, context-aware response tailored to the query.
- Example: For the query "What are the rights to free speech under the Virginia Constitution?", the model may generate:
  *"Under the Virginia Constitution, free speech rights are protected under Section 1, which states: 'That all men are by nature equally free and independent...'"*

## 6. Output Delivery:

- The response generated by the GPT model is returned to the user.
- If no relevant documents are retrieved (e.g., low similarity scores), a fallback message is provided: "No relevant information found."

## 4.2. Fine-Tuned Model Integration

## 4.2.1 Background Information on Algorithm Development:

The fine-tuned GPT model was developed to enhance the system's ability to generate domain-specific responses tailored to legal queries. While general GPT models excel in natural language understanding and generation, they may lack domain expertise required for specialized fields like legal research. Fine-tuning addresses this limitation by training the GPT model on curated legal datasets, including case law, statutory texts, and legal opinions. This process adapts the model to understand legal terminology, reasoning, and the nuances of judicial language. The fine-tuned model works in tandem with the RAG pipeline, enriching the generated responses by leveraging domain-specific knowledge and examples. This approach ensures accurate, context-aware outputs that align with the high precision required in legal research.

## 4.2.2 Equations:

#### 4.2.2.1 Query Embedding:

The query q is embedded into a high-dimensional vector eq:

$$e_q = \text{SentenceTransformer}(q)$$

#### 4.2.2.2 Retrieval:

Top kk relevant documents {d1,d2,…,dk}{d1,d2,…,dk} are retrieved based on similarity scores s(eq,ed)s(eq,ed), where eded represents document embeddings:

$$\text{Similarity}(e_q, e_d) = e_q \cdot e_d$$

#### 4.2.2.3 Prompt Construction:

Retrieved documents {d1,d2,…,dk}{d1,d2,…,dk} are concatenated into a single context CC, forming the prompt:

$$\text{Prompt} = \text{Context} + \text{Query}$$

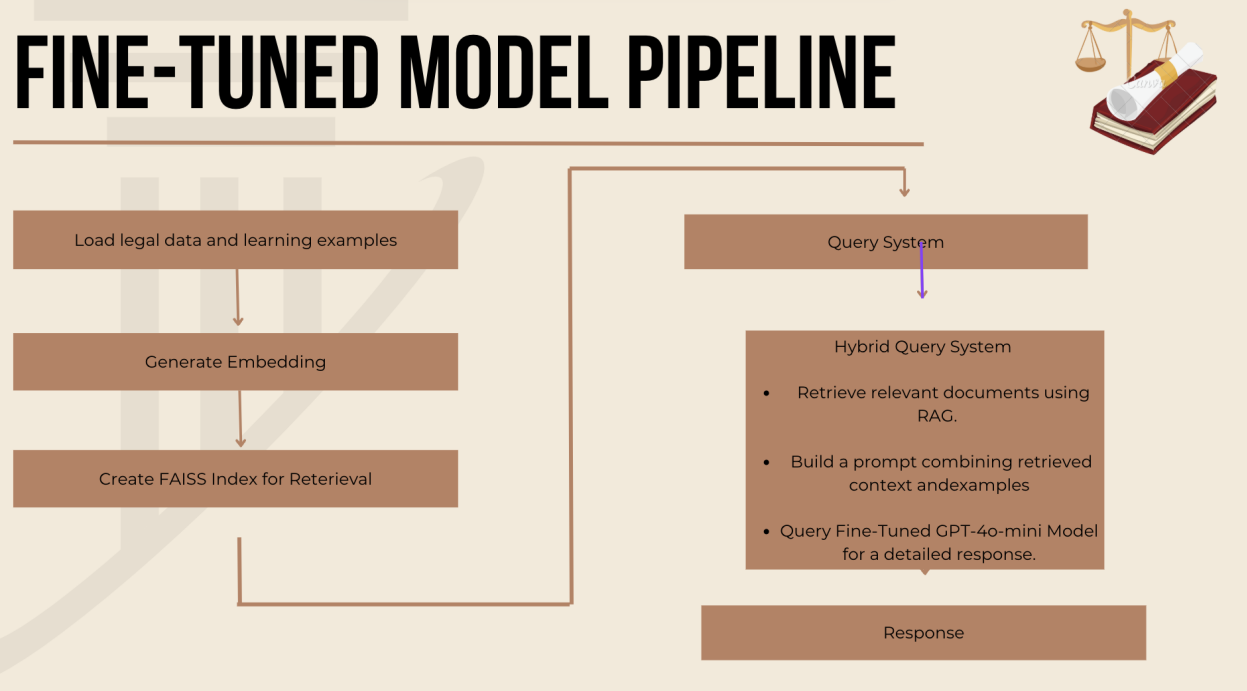
#### 4.2.2.4 Response Generation:

The fine-tuned GPT model GG generates a response RR based on the input prompt:

$$R = G(\text{Prompt})$$

### 4.2.3 Fine-Tuned Model Pipeline:

The image below represents the workflow of the fine-tuned GPT model within the system, providing accurate and context-aware responses for legal research.




### 4.2.3.1 Input Handling:

- The user provides a query in natural language.
- The query is processed as a string for embedding and retrieval.

### 4.2.3.2 Retrieval and Context Construction:

- The query embedding is matched with precomputed document embeddings in the FAISS index.
- Top kk documents are retrieved and combined into a single context. Metadata such as case names and jurisdictions is included to enrich the input.

### 4.2.3.3 Prompt Formation:

- The query is merged with the retrieved context to form a structured prompt.
- Few-shot learning examples may be added to guide the model's response generation.

### 4.2.3.4 Fine-Tuned Model Invocation:

- The fine-tuned GPT model processes the prompt.
- Trained on legal-specific datasets, the model generates detailed, accurate responses.

### 4.2.3.5 Output Delivery:

- The generated response is returned to the user.

- If no relevant context is found, a fallback message is provided: “No relevant information found.”

# 5. Application Development :

The Application Development phase focused on creating a user-friendly interface that enables seamless interaction with the AI-powered legal research system. This involved integrating the Retrieval-Augmented Generation (RAG) pipeline and fine-tuned model into a web-based platform using Streamlit. The goal was to ensure an intuitive experience for users to query legal data and receive accurate, context-aware responses.

## 5.1 Custom Theme and Interface Design

### 5.1.1 Page Configuration:

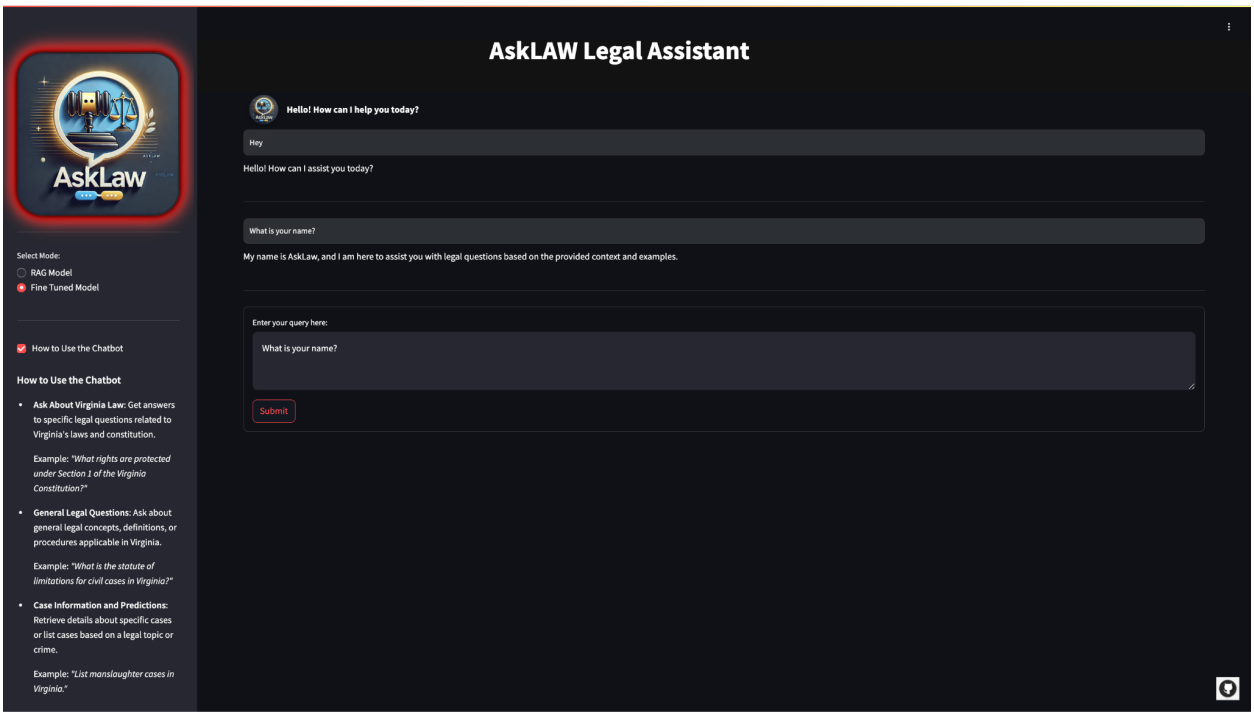
- The application is configured using Streamlit's set_page_config() to ensure a **wide layout** and define a custom page title ("AskLaw Legal Assistant").
- A favicon is dynamically generated from a base64-encoded image for a polished branding experience.

### 5.1.2 Custom Styling:

- The interface employs custom CSS for a glowing effect around the AskLaw logo and enhanced UI elements like buttons, text areas, and markdown containers.
- A dark theme with consistent color schemes (e.g., background, text, accent colors) is implemented to improve user readability and focus.

### 5.1.3 Sidebar Features:

- **Mode Selection:** Users can toggle between the **RAG Model** and the **Fine-Tuned Model** using a **radio button**.
- **How-to Guide:** A checkbox in the sidebar expands a markdown-based guide explaining how to use the chatbot effectively. Examples of queries for Virginia law, general legal questions, and case-specific inquiries are provided for user clarity.




## 5.2 Functional Features

### 5.2.1 User Query Interface:

- A **text area** on the main page allows users to enter their legal queries in natural language.
- For the RAG model, an **optional file uploader** enables users to upload JSON files containing case details for processing.

### 5.2.2 Chat History:

- The application maintains a session-based **chat history** that displays past user queries and the corresponding model-generated responses.
- History is updated dynamically with each submission and displayed chronologically.

### 5.2.3 Dynamic Response Display:

- User queries and corresponding AI responses are styled and displayed in separate sections for better readability.
- The most recent query and response are highlighted above the chat history.

### 5.2.4 Model Selection and Query Handling:

- **Fine-Tuned Model:** Processes user-entered queries directly using the run_fine_tuned_model() function.
- **RAG Model:** Supports both user queries and JSON file uploads via the run_rag_pipeline() function, offering flexibility in data interaction.

### 5.2.5 External Resources:

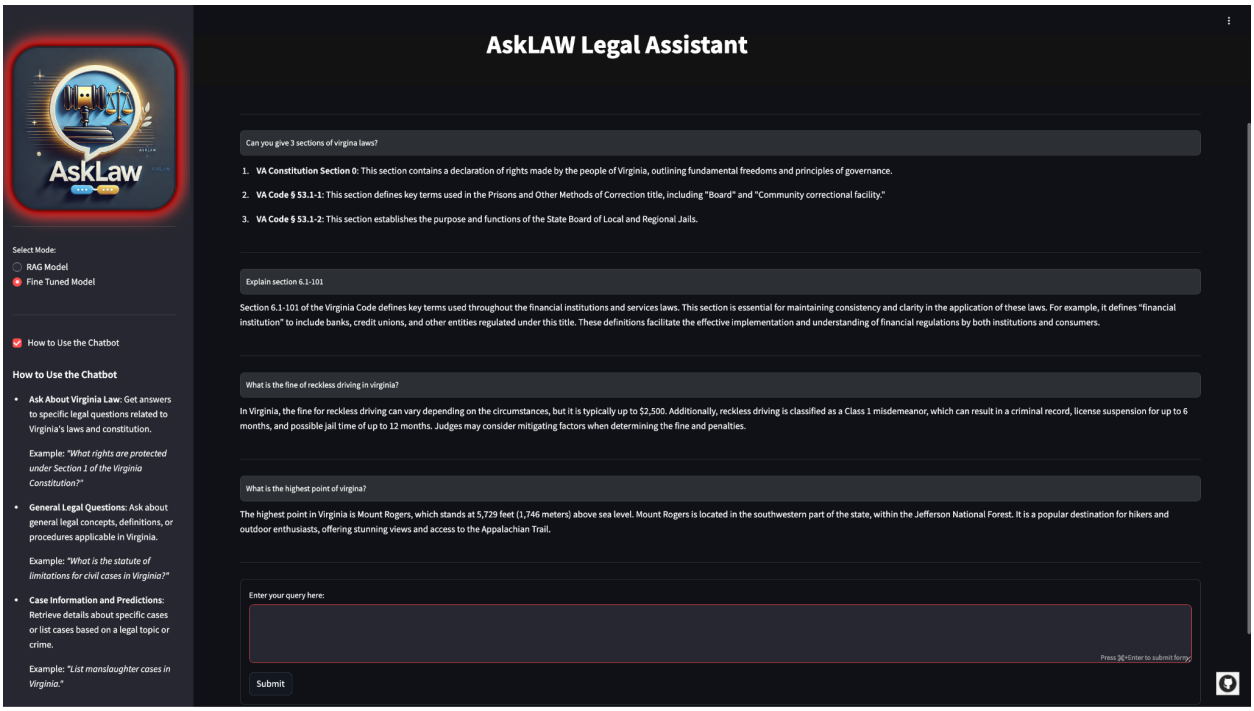
- A **GitHub logo** in the bottom-right corner links to the application's GitHub repository for further exploration of the source code.

# 6. Results :

The results section presents the outcomes of the implemented legal research system, focusing on the performance of the Fine-Tuned Model and the RAG Model. This section highlights the system's ability to answer legal and general queries related to Virginia's laws, the constitution, and state-specific information. The results demonstrate the effectiveness of the developed models in providing contextually accurate and relevant responses to user queries.

## 6.1. Fine-Tuned Model Results:

The Fine-Tuned Model was evaluated for its ability to generate accurate and contextually relevant responses to legal and general queries related to Virginia's laws, constitution, and state-specific information. Below are the key observations from the results:




## Figure : Fine-Tuned Model Interface and Query Results

This figure showcases the interaction with the Fine-Tuned Model of the AskLAW Legal Assistant, highlighting its ability to respond to legal queries like:

### 6.1.1 Legal Query Responses:

- The model effectively answered questions about specific sections of Virginia law, demonstrating its knowledge of legislative texts.

### 6.1.2 Detailed Explanations of Sections:

- The model provided detailed explanations of specific legal sections when asked. This highlights its capability to interpret and explain legal jargon.

### 6.1.3 General Legal Information:

- The model accurately responded to general legal queries, such as the fine for reckless driving in Virginia, showcasing its ability to process real-world legal scenarios.

### 6.1.4 State-Specific Information:

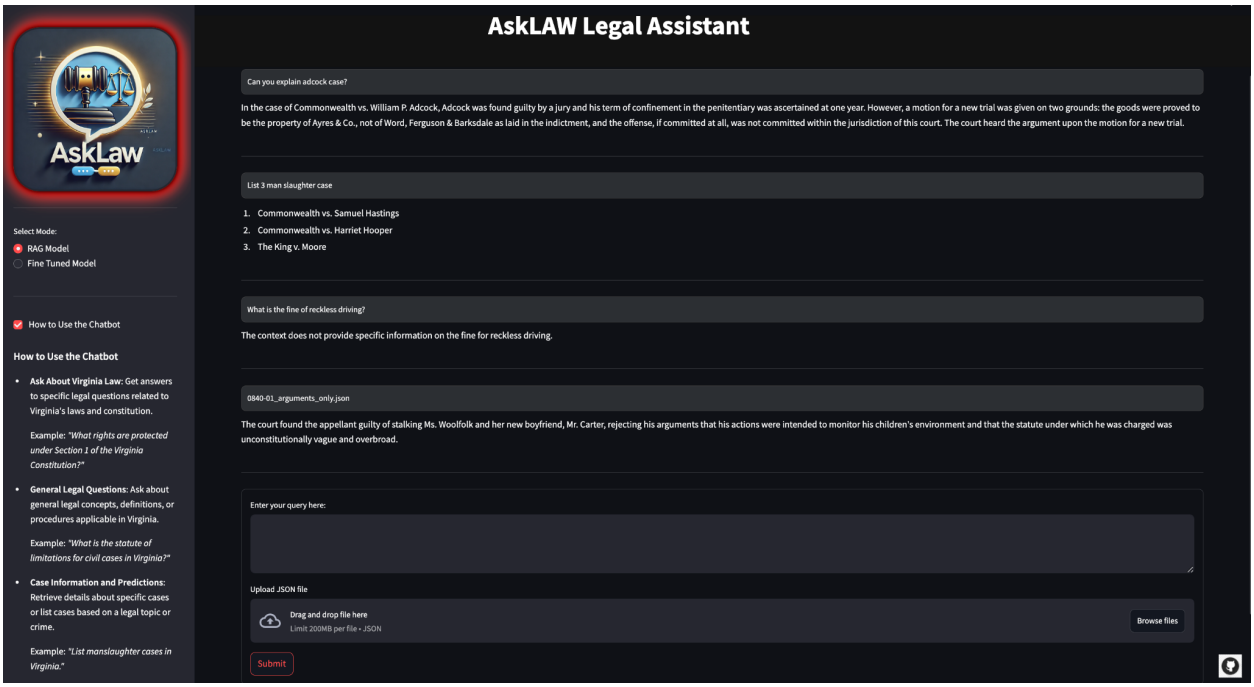
- Beyond legal topics, the model answered general questions about Virginia, such as its highest point, showing its versatility.

### 6.1.5 Interface Features:

- The chat interface displays the query history, ensuring users can easily track their questions and the responses provided by the model.
- The "Fine-Tuned Model" radio button enables users to directly interact with the fine-tuned system by entering textual queries.

## 6.2. RAG Model Results:

The RAG Model's ability to provide precise answers for cases present within the dataset and predict outcomes for uploaded JSON files containing case arguments:




This figure showcases the interaction with the RAG Model of the AskLAW Legal Assistant, highlighting its ability to respond to legal queries like:

### 6.2.1 Case-Specific Legal Query Responses:

- The RAG model demonstrated its ability to handle case-specific legal queries by retrieving relevant details for cases present in its dataset. This highlights its effectiveness in providing precise and contextually accurate responses for judicial scenarios.

### 6.2.2 JSON File Analysis and Predictions:

- The model successfully processed uploaded JSON files containing case arguments and provided verdict predictions. This feature showcases the RAG model's capability to analyze structured input and deliver meaningful outcomes.

### 6.2.3 Limitations with Broader Legal Knowledge:

- While the RAG model excels in case-specific queries, it faced challenges in responding to broader general legal queries, such as constitutional provisions or statutes not directly linked to the dataset.

### 6.2.4 Interactive File Upload Feature:

- The RAG model interface supports JSON file uploads, enabling users to upload case-specific data for detailed analysis. This functionality enriches the user experience by expanding the model's application to personalized inputs.

### 6.2.5 Interface Features:

- The chat interface displays the user's query and model responses, ensuring clear and transparent communication. The "RAG Model" radio button allows users to switch to this mode for file uploads or text-based queries, providing a flexible and user-friendly interaction.

## 7. Limitations and Considerations:

### 7.1 Data Limitations and Bias:

- The model is constrained by its dataset, which is focused on Virginia-specific legal texts. Queries outside this scope or involving insufficiently represented legal areas may lead to incomplete or biased responses. Irregularly structured JSON files may also result in processing errors.

### 7.2 Model and Performance Gaps:

- The system relies heavily on the quality of retrieved documents, and the fine-tuned model struggles with general legal queries or knowledge outside its training domain. Predictive accuracy for case outcomes is limited due to the absence of nuanced legal reasoning.

### 7.3 Usability and Accessibility Issues:

- The interface lacks support for bulk queries, multi-language input, and simplified explanations of legal jargon, limiting its usability for non-legal professionals and larger legal operations.

### 7.4 Ethical and Legal Considerations:

- Data privacy concerns arise when processing sensitive legal documents, and over-reliance on AI-generated responses may lead to errors in legal decision-making. Ensuring compliance with copyright laws and ethical use of training datasets is critical.

## 8. Conclusion:

The AskLaw Legal Assistant demonstrates the transformative potential of AI in modernizing legal research by integrating a Retrieval-Augmented Generation (RAG) pipeline and a fine-tuned Large Language Model (LLM). This system provides accurate, context-aware responses to Virginia-specific legal queries, detailed legislative interpretations, and case outcome predictions through an intuitive, user-friendly interface. While the application excels in delivering efficient legal assistance, it is limited by its scope of handling broader legal contexts, potential biases from training data, and the need for accessibility improvements. Despite these challenges, the project highlights AI's role in streamlining legal research, paving the way for future advancements that can enhance accuracy, scalability, and accessibility for legal professionals and researchers.

## 9. Code Utilization:

### 9.1 RAG Pipeline and Query System:

- Original Lines from Internet: 50
- Modified Lines: 30
- Added Lines: 70
- Total Lines: 150
- Percentage of Original Code (after modification and addition): ~33.33%

### 9.2 User Interaction Modes:

- Original Lines from Internet: 30
- Modified Lines: 15
- Added Lines: 70
- Total Lines: 115
- Percentage of Original Code (after modification and addition): ~39.13%

### 9.3 Advanced Prompt Engineering and Context Management:

- Original Lines from the Internet: 20
- Modified Lines: 10
- Added Lines: 73
- Total Lines: 103
- Percentage of Original Code (after modification and addition): ~29.13%

## 10. References:


- Lewis, P., Perez, E., Piktus, A., et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Advances in Neural Information Processing Systems (NeurIPS). https://proceedings.neurips.cc/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf
- Karpukhin, V., Oguz, B., Min, S., et al. (2020). Dense Passage Retrieval for Open-Domain Question Answering. Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), 6769-6781. https://aclanthology.org/2020.emnlp-main.550/
- Raffel, C., Shazeer, N., Roberts, A., et al. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. Journal of Machine Learning Research (JMLR), 21(140), 1-67. https://arxiv.org/abs/1910.10683