**CSE665: Large Language Models**

**Assignment 1**

# Exploring and Probing Large Language Models

## Part 1 : Analyzing the  hallucinations in two Large Language Models (LLMs)

### 1. Environment Setup:
*Library Installation:*
Install necessary libraries such as transformers, accelerate, and bitsandbytes to ensure compatibility and access to the latest model configurations.

*Import Libraries:*
Import essential components like torch, transformers, warnings, and model-specific classes.
Suppress warnings for a clean output focused on relevant information.

### 2. Model Loading:
*LLAMA-2 Model:*
Load the LLAMA-2 model (NousResearch/Llama-2-7b-chat-hf) using AutoModelForCausalLM with load_in_4bit=True to optimize memory usage.
Configure device_map='auto' to automatically assign the model to the available GPU for efficient processing.

*OpenHathi Model:*
Load the LlamaForCausalLM model (sarvamai/OpenHathi-7B-Hi-v0.1-Base) with torch.bfloat16 precision for enhanced computation on the GPU.

### 3. Tokenizer Setup:
*LLAMA-2 Tokenizer:*
Use AutoTokenizer.from_pretrained() to load the LLAMA-2 tokenizer.
Set pad_token to eos_token to handle padding correctly during tokenization.

*OpenHathi Tokenizer:*
Load the LlamaTokenizer corresponding to the OpenHathi model.

### 4. Generation Configuration:
*LLAMA-2 Configuration:*
Define a GenerationConfig object with:
do_sample=True: Enables diverse and creative text generation.
temperature=0.8: Balances randomness in the output.
repetition_penalty=1.5: Prevents repetitive outputs.
max_new_tokens=256: Limits the length of the generated text.

*OpenHathi Configuration:*
Use the generate() function with max_length=128 to control the response length.

### 5. Generating Responses:
*Response Function for LLAMA-2 (generateTokens()):*
Tokenize the input prompt and convert it into input tensors for processing.
Generate responses using the configured generation settings.
Decode and print the generated output, ensuring it’s readable and skips unnecessary tokens.

*Response Function for OpenHathi (genTokens()):*
Tokenize the input prompt and transfer the input IDs to the GPU.
Use the model's generate() function to produce a response, then decode and print it.

**6. Evaluation of Model Outputs:**
*Factual Examples:*
Use factual questions (e.g., "Who killed Gandhi?", "What is the law of large language numbers?") to test the models' accuracy in recalling correct information.

*Self-Consistency Testing:*
Repeatedly ask the same questions to assess if the models maintain consistent responses or if they deviate/hallucinate over multiple iterations.

**7. Summary of Findings:**
*Observations from LLAMA-2:*
LLAMA-2 demonstrates diverse responses with creative variations, but may suffer from hallucinations or incorrect factual outputs.

*Observations from OpenHathi:*
OpenHathi shows potential in generating responses but struggles with maintaining accuracy and consistency, especially with factual data.

***Overall Insights:***
This combined pipeline illustrates a comprehensive approach to setting up, configuring, and evaluating the LLAMA-2 and OpenHathi models. It highlights key areas like response generation, configuration tuning, and evaluation criteria to assess model performance in terms of accuracy and consistency.

## Part 2 : Applying RAG to reduce the hallucination issues.

### Methodology and Steps Undertaken

**Environment and Model Setup:**
*Libraries Installation and Import:*
- Essential libraries like torch, transformers, chromadb, and langchain components were imported to facilitate model loading, text processing, and setting up the retrieval-augmented generation pipeline.

*Model Configuration and Loading:*
- Two large language models, NousResearch/Llama-2-7b-chat-hf and sarvamai/OpenHathi-7B-Hi-v0.1-Base, were loaded from Hugging Face.
- For both models, the BitsAndBytesConfig was set up to load the models in 4-bit precision using the bitsandbytes library, which helps reduce GPU memory usage, allowing the models to run efficiently on available resources. This included parameters like load_in_4bit=True and specific configurations (bnb_4bit_quant_type='nf4', etc.).
- The models were loaded using the AutoModelForCausalLM class with device mapping set to auto for optimized execution on available GPUs.

***Pipeline Setup:***
- The Llama-2 model's generation pipeline was prepared using the transformers.pipeline function. This setup allowed the model to generate responses using specified settings

(torch_dtype=torch.float16, max_new_tokens=50), ensuring efficient and accurate response generation.
- The model and tokenizer setup time was logged to understand performance overhead.

**Retrieval Setup Using ChromaDB:**

*ChromaDB Client Initialization:*
- A ChromaDB client was created to set up the vector database for document retrieval.

*Collection Creation and Document Insertion:*
- A collection named my_collection was created, and simulated knowledge base documents were added to it, each having an ID and a corresponding text.
- The documents included various factual data, such as historical events, definitions, and general knowledge items.

*Retriever Setup:*
- A retriever was set up using the Chroma class, connecting it to the ChromaDB collection, which enabled the retrieval of relevant context based on query embeddings.

**Retrieval-Augmented Generation (RAG) Pipeline Setup:**

*Prompt Template Creation:*
- A prompt template was defined to instruct the assistant on how to generate responses based on the retrieved context and the given question. The template structure was simple, focusing on combining context with questions.

*RAG Chain Initialization:*
- The RetrievalQA class from LangChain was used to create the retrieval-augmented generation pipeline. This chain combines the language model with the retriever, enabling it to generate responses based on the most relevant context fetched from ChromaDB.

**Testing the RAG Pipeline:**
- Several test queries were run through the pipeline to evaluate its performance. The questions tested factual recall, self-consistency, and the models' ability to integrate retrieved context into their answers. For instance:
- Questions like "Who was the rail minister at the time of the coal scam?" and "What is the national tree of India?" were asked to evaluate the pipeline's accuracy and consistency.

**Pipeline Replication for OpenHathi Model:**
- The entire process was replicated using the OpenHathi model, setting up a separate collection (collection2) in ChromaDB and testing with similar queries to compare performance and accuracy.

**Summary**

This approach effectively demonstrates setting up a retrieval-augmented generation system combining large language models with a retrieval mechanism (ChromaDB). By loading models in an optimized configuration, retrieving context efficiently, and generating responses, the task aimed to highlight how these models can be used together to answer complex questions accurately while managing resources efficiently.

## Part 3 : Probing

### Process/Methodology Followed in the Task

**Setup and Data Preparation:**
- Installed necessary libraries (transformers, torch, bitsandbytes, etc.) for model loading and data processing.

- Loaded a custom dataset using the datasets library, focusing on specific columns like Name, Address, Salary, DOJ, Age, Sex, etc.
- Cleaned the data by removing rows with missing values in the Age column.
- Selected a random 2% sample of the dataset to create a smaller working subset.

**Model Loading and Embedding Extraction:**

- Loaded the LLAMA model (Meta-Llama-3-8B-Instruct) with 4-bit quantization to efficiently handle large model sizes with limited GPU memory.
- Tokenized input text and extracted hidden state embeddings from three layers (first, middle, last) of the model.
- For each data sample, extracted embeddings at specified layers to understand how the model encodes the information.

**Classification Task:**

- Created feature sets using the embeddings from each target layer.
- Trained Random Forest classifiers on these embeddings to predict the Sex label.
- Evaluated the models using accuracy metrics.

**Regression Task:**

- Used the same embeddings to predict the numerical value of Age using Linear Regression models.
- Assessed the performance of each layer's embeddings by calculating the Mean Squared Error (MSE) of predictions.

**Findings and Reflections on Results**

*Accuracy Observations:*

- The classifier trained on the first layer's embeddings achieved only 36% accuracy, suggesting that early layers do not encode information relevant to the classification task effectively.
- Classifiers using embeddings from the 16th (middle) and 31st (last) layers achieved 100% accuracy, indicating that deeper layers capture more complex and relevant features for the task.

*Regression Observations:*

- The MSE for the first layer was high, indicating poor performance in predicting the Age.
- The MSE dramatically improved for the 16th layer and reached zero for the last layer, showing a clear pattern of deeper layers having better numerical prediction capabilities.

**Conclusion and Patterns Noted**

- The results clearly show that embeddings from deeper layers of the model are more effective for both classification and regression tasks.
- This suggests that the LLAMA model progressively refines and encodes richer, task-relevant information as you move deeper into the layers.
- The difference in performance across layers reflects the model’s hierarchical learning, where early layers capture general patterns, and deeper layers specialize in encoding complex, contextual information.