Università degli Studi di Salerno

Natural Language Processing Technical Report

2025

by

Bianca Oltean, Miruna Oltean,

Florian Stibitz

Introduction

The goal of this project was to develop a chatbot that answers questions about the Natural Language Processing and Large Language Models 2024/2025 course. The chatbot must handle both course-specific topics and general information, such as details about instructors or recommended books, while ensuring it answers only in-context questions. For out-of-context queries, it will respond appropriately to indicate its limitations.

It is allowed the use of any tools or technologies studied in the course, including modern LLMs and classical NLP methods, with the freedom to combine approaches if the solution is justified in the final report. The chatbot will be evaluated on its ability to provide relevant, fluent, and coherent answers, along with its robustness to adversarial prompts and its precision in identifying out-of-context questions.

The chatbot's performance will be assessed based on:

- Relevance: Does the generated text accurately answer the query?
- Fluency: Is the output grammatically correct and easy to read?
- Coherence: Does the response maintain logical flow and consistency?

Additionally, a second round of evaluation will focus on the chatbot's:

- Robustness: Its ability to resist adversarial or misleading prompts (e.g., “Are you sure?”).
- Precision: Its capability to identify and reject out-of-context questions (e.g., “Who is the king of Spain?”).

Storage and Preprocessing of Course Materials

The project relies on structured and modular approaches for managing course data to ensure efficiency and accuracy:

1. Course Materials Stored in PDF Files

- All course content is stored in PDF format, allowing for straightforward distribution and version control.

2. Classifier Keywords Stored in JSON Files

- A JSON file is used to store key data such as classification labels (*“Natural Language Processing”* and *“Not related to Natural Language Processing”*) and specific NLP-related keywords.
- Why JSON? It provides a structured, lightweight, and language-agnostic format ideal for storing key-value data.

```
  "labels": [
    "Natural Language Processing",
    "Not related to Natural Language Processing"
  ],
  "nlp_keywords": [
    "tokenization", "lemmatization", "stemming", "parsing", "POS tagging", "part-of-speech tagging",
    "syntax", "semantics", "machine translation", "sentiment analysis", "text classification",
    "question answering", "chatbot", "text generation", "summarization", "language modeling",
    "topic modeling", "named entity recognition", "relation extraction", "BLEU score", "ROUGE score",
    "GLUE", "SuperGLUE", "OpenWebText",
```

PDF Extraction Challenges and Solution

Initially, the project utilized *PyPDFDirectoryLoader* to extract text from PDF files. However, this approach resulted in frequent warnings and inconsistencies, particularly with PDFs containing images or complex layouts.

To address these issues, we transitioned to *pdfPlumber* for text extraction because:

- It focuses on extracting the text, which it does by interpreting the PDF's internal structure, such as its content streams and layout. It processes the text along with other content, including images and fonts, but the images themselves aren't extracted as text.
- When *pdfPlumber* processes a PDF, it detects and extracts images separately. It doesn't automatically convert or output images as part of the text extraction, but it can access the raw images embedded within the PDF structure if needed. This is particularly useful for more advanced use cases.
- It extracts text from the document even if the document contains images. If the images don't obscure the text (e.g., by being overlaid on it), *pdfPlumber* will still extract the text accurately.
- For many cases, the presence of images in PDFs may affect the extraction process in terms of layout, but *pdfPlumber* handles this fairly well. Images placed alongside or below text in the layout will not interfere with the text extraction.

Document Chunking for Retrieval

In the context of information retrieval, particularly when dealing with large documents, dividing these documents into smaller, more manageable segments—known as document chunking is essential. LangChain's *RecursiveCharacterTextSplitter* plays a pivotal role in this by dividing the document into smaller chunks, typically of 700 characters, with a 50-character overlap between adjacent chunks.

This chunking process is critical to ensuring the system can efficiently store, retrieve, and process data. Why Text Chunking?

- Working with large documents as a whole can become computationally expensive in terms of memory and storage. By breaking a large document into smaller chunks, you can load and process each segment independently. This makes it easier to manage data, as the system doesn't need to process the entire document in one go, allowing more efficient use of system resources.

- Chunking allows for parallel processing. Since the chunks are independent of each other (with some contextual overlap), you can process and index multiple chunks simultaneously, speeding up retrieval times and enhancing performance, particularly for large-scale datasets.
- Embedding large documents directly into a vector space can be problematic because embeddings typically work best on smaller pieces of text. Large texts may contain multiple contexts or ideas, making it difficult for a single embedding to capture the nuances of the entire document. By chunking the document, each segment can be individually embedded, resulting in embeddings that are more focused and meaningful.
- While the document is chunked into smaller pieces, maintaining a coherent understanding of the text is vital for the system to provide relevant answers. The 50-character overlap ensures that even as documents are split, crucial contextual information isn't lost between chunks. This overlap allows for smoother transitions and prevents the loss of critical information that might span across chunk boundaries.
- When retrieving information, chunking allows the retrieval system to work with smaller, more focused segments. Instead of searching through an entire document, the system only needs to search within relevant chunks, reducing the time it takes to find the information the user needs.

Retrieval-Augmented Generation (RAG)

RAG is a hybrid model that integrates both retrieval-based and generation-based techniques to improve the quality of answers. The main idea behind RAG is to augment a pre-trained language model (like GPT or Mistral) with external knowledge retrieved from a specific corpus of documents.

- RAG first retrieves a set of relevant documents or passages from a large knowledge base or index (such as FAISS). This is achieved by searching for documents that contain terms related to the query or question posed by the user.
- The retrieval model finds the most relevant chunks of text, based on the query, which will provide context for generating a high-quality answer.
- Once the relevant documents are retrieved, the information is added into the query as an additional context. This enables the language model to generate answers not just based on pre-existing knowledge encoded in its parameters but also based on the newly retrieved content.

After retrieving and augmenting the context, the language model generates an answer that incorporates the retrieved information. This process allows the chatbot to provide specific, factually correct answers even if the question pertains to very detailed information that was not included in the model's training data.

The RAG model combines both retrieval-based information and generation-based language modeling to create an answer that is more precise, contextually accurate, and relevant to the query.

RAG's main advantage lies in its ability to dynamically pull information from external sources, ensuring that the responses are both up-to-date and relevant to the specific query, even if the language model itself doesn't directly encode that information.

Why RAG?

- By leveraging external sources (FAISS index), RAG can answer questions based on specific course material that might not be available in the pre-trained model.
- RAG models can scale to larger datasets, enabling them to handle substantial amounts of course material while providing efficient, relevant answers.

FAISS (Facebook AI Similarity Search)

FAISS is a highly efficient library designed for similarity search and clustering of high-dimensional vectors. In the context of this project, FAISS is used for storing, retrieving, and indexing the embeddings of the chunks of text extracted from the course PDFs. FAISS allows the system to quickly retrieve relevant documents based on the vector similarity between the question and document chunks.

The process of integrating FAISS into the system for document retrieval is as follows:

- The documents, once chunked using LangChain's *RecursiveCharacterTextSplitter*, are passed through an embedding model (in our case, *sentence-transformers/all-MiniLM-L6-v2*).
- The model converts the textual chunks into high-dimensional vectors that capture the semantic meaning of the text. Each chunk of text is now represented by a vector in the embedding space.

FAISS is used to index these embeddings, allowing for fast retrieval of the most relevant documents. The embeddings are stored in a vector index that FAISS optimizes for efficient similarity search. The index allows for nearest neighbor search, meaning that when a query is made, the system can quickly identify the chunks of text that are most similar to the query based on their vector representation.

When a user asks a question, the query is also converted into a vector embedding using the same model. FAISS performs a nearest neighbor search to find the top-k most similar document chunks based on vector distance (cosine similarity). These documents are then returned as the context for the RAG model to generate an answer.

FAISS supports both dense vector and binary vector indices, making it highly efficient for large datasets. The local storage and retrieval of the index ensure that even a substantial number of documents can be indexed and searched quickly, without needing to query large external databases every time.

One of the advantages of using FAISS is that the index can be saved to disk after it is created. This means that instead of reprocessing the documents and regenerating the embeddings every time the system is run, the FAISS index can be saved once and reused in future sessions. This drastically reduces the computational overhead and ensures the system is ready for quick retrieval without having to rebuild the index.

This is done by calling *faiss_index.save_local*, which saves the index locally to a file. Once saved, the index contains all the embeddings and their associated metadata.

```
embedding_model = HuggingFaceEmbeddings(
    model_name = "sentence-transformers/all-MiniLM-L6-v2",
    encode_kwargs = {'normalize_embeddings': True}
)
faiss_index = FAISS.from_documents(chunks, embedding_model)

faiss_index.save_local("faiss_index")
```

For subsequent queries or chatbot interactions, the FAISS index can be loaded directly from the saved file. This avoids the need to regenerate embeddings or reprocess documents, which is particularly useful when the course content remains static over time.

The index can be loaded using *FAISS.load_local* with the appropriate embedding model, ensuring that the system retrieves and uses the exact same embeddings for similarity search.

```
embedding_model = HuggingFaceEmbeddings(
    model_name = "sentence-transformers/all-MiniLM-L6-v2",
    encode_kwargs = {'normalize_embeddings': True}
)

faiss_index = FAISS.load_local("faiss_index", embedding_model
```

Why FAISS is Critical for Retrieval:

- Speed: FAISS is highly optimized for performing similarity searches in high-dimensional spaces, making it faster than traditional keyword-based search engines.
- Scalability: FAISS can scale to millions of document chunks, making it ideal for handling large datasets like course materials in multiple PDFs.
- Flexibility: FAISS supports various similarity search algorithms, allowing it to be tailored to the needs of the project in terms of speed and accuracy.

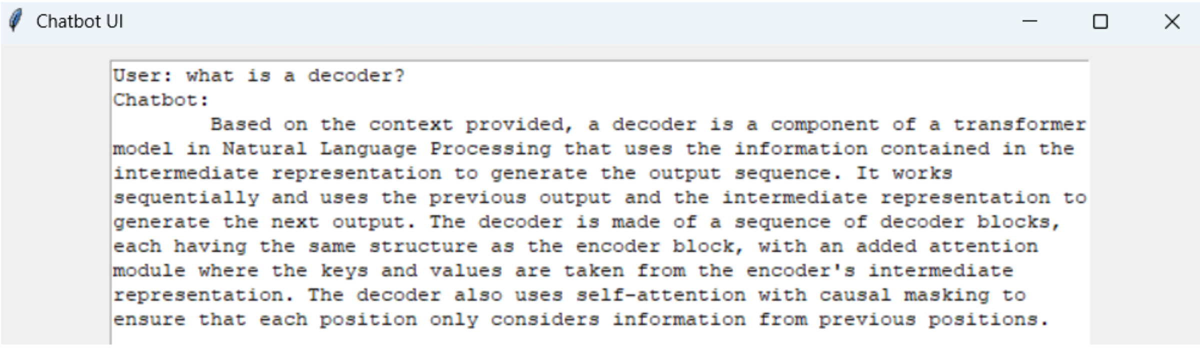
Integrating RAG and FAISS in the System:

When a question is posed to the chatbot, it is first classified to ensure it is related to the course material. If it is a valid query, it is passed to the FAISS retriever.

FAISS performs a similarity search on the indexed document chunks and returns the top-k most relevant pieces of text based on the query embedding.

The retrieved chunks are then passed into the RAG model, where they are combined with the query to generate a contextual response. The language model can then synthesize a response using both the query and the retrieved documents, ensuring the answer is accurate and grounded in the provided course materials.

The language model generates a response, considering both the question and the retrieved documents. The final answer is tailored to the specific query, ensuring precision and relevance.



Input Check Process

The process of handling input starts with analyzing and classifying the user's query to determine whether it is relevant to the course material and if it can be answered based on the provided resources. This is a crucial step to ensure that the chatbot responds appropriately, rejects out-of-context questions, and handles adversarial prompts. Here's how this process is structured:

The first step in the input check is classifying the question to determine whether it pertains to NLP, LLMs, or course materials. The classifier looks for specific keywords related to the course materials in the question. These keywords (like "NLP," "language model," etc.) are stored in the JSON file under *nlp_keywords*. The classifier uses these keywords to determine if the question is related to the course content.

In cases where the question does not directly match any known keywords, the classifier uses a zero-shot classification model (*facebook/bart-large-mnli*) to categorize the question into one of several predefined labels. The classifier checks the question against these labels, and each label has an associated score. The label with the highest score determines the category of the question.

If the score for the top label is sufficiently high (above a threshold of 0.6), the question is considered relevant to the course material. If it is below the threshold, it will be deemed out-of-context.

Once the question is classified, the next step is to verify its relevance to the core topics NLP, LLMs, and course materials. If the question contains key terms or concepts directly tied to the course or the field of NLP/LLMs, it is deemed relevant. If not, the system will reply with: “I'm only able to answer questions about NLP, LLMs, and the related course materials”.

The system also performs a check for adversarial prompts or misleading queries. For instance, a question like *"Are you sure?"* or *"What's the king of Spain?"* could potentially confuse the chatbot or trick it into providing irrelevant answers.

```
User: what is a cappuccino
Chatbot: I'm only able to answer questions about NLP, LLMs, and the related
course materials.
```

We used guardrails to prevent the chatbot from providing answers that contain toxic inappropriate language. If such language is detected in the answers of the system, our guardrails ensure that the chatbot refuses to answer. For the implementation of those we used the *ToxicLanguage* and *NSFWText* guards from the Guardrails AI library.

Once the input has been verified, the system then proceeds to the next stage of retrieving relevant information from the course materials. If the query is valid, relevant documents are retrieved from the FAISS index. These retrieved documents are passed to the RAG model, which uses them to generate a contextually appropriate response.

After processing, the system will generate the final response based on the retrieved context and the user's question. If the question was determined to be relevant, the system provides an accurate and context-aware answer. Otherwise, it will return the message like *"I'm only able to answer questions about NLP, LLMs, and the related course materials"*.

Models attempted and challenges

During the development phase of the chatbot, various pre-trained models were explored to determine their suitability for the task. While some models showed promise in terms of accuracy and performance, limitations related to computational resources and hardware capabilities prevented us from fully utilizing or fine-tuning them. Below is an overview of the models we attempted to use, and the challenges encountered:

1. T5 (Text-to-Text Transfer Transformer)

We explored different configurations of the T5 model, an encoder-decoder architecture designed for text generation and understanding tasks. Three variants of T5 were tested:

- T5-Small: The small variant of T5 was relatively lightweight and could be run on local systems for inference. However, the quality of generated responses and embeddings was not sufficient for nuanced queries, especially those requiring complex understanding of NLP or LLMs.
- T5-Base provided better contextual understanding compared to T5-Small but required more computational power during training and fine-tuning. Without access to high-performance GPUs or TPUs, fine-tuning this model on our course material proved infeasible.

- T5-Large offered significantly better performance in capturing complex semantics but came with much higher resource demands. Running inference itself required a high memory capacity, making local deployment impractical. Fine-tuning this model on the dataset required GPUs with large VRAM, which was unavailable in our setup.

2. *LLaMA* (Large Language Model Meta AI)

Meta's *LLaMA* models were also considered due to their effectiveness in handling various natural language processing tasks. We explored smaller variants of the *LLaMA* model to balance performance and resource requirements:

- *LLaMA-Small* offered promising results in generating coherent responses during initial experiments. However, fine-tuning *LLaMA*, even at this smaller scale, demanded substantial computational power and memory. This was beyond the capability of our local machines, as *LLaMA* models are resource-intensive during both training and inference.
- Larger variants of *LLaMA* provided significantly better results during inference. However, training or fine-tuning them on our custom course materials was entirely infeasible due to the lack of high-end GPUs or TPUs required for efficient processing.

Both T5 and *LLaMA* models, especially their larger variants, required high-performance hardware (e.g., A100 GPUs or TPUs) for fine-tuning. Without access to such resources, training these models locally was not viable. Fine-tuning large-scale models requires significant GPU memory, typically 16GB or more.

Even with smaller models, training or fine-tuning would have been time intensive. Given the timeline of the project, using pre-trained models without fine-tuning emerged as a more practical approach.

To address these challenges, we opted to use pre-trained models for specific tasks:

- For question classification, we utilized the *facebook/bart-large-mnli* model for zero-shot classification, as it could run efficiently on local hardware without fine-tuning.
- For RAG, we relied on lightweight embedding models such as *sentence-transformers/all-MiniLM-L6-v2* for FAISS indexing, coupled with a hosted inference endpoint like Mistral-7B-Instruct to generate responses.

UI Design for the Chatbot Application

The user interface (UI) of the chatbot application was designed using Python's *Tkinter* library, which provides a simple and effective way to create graphical interfaces. The UI consists of several key components that enable users to interact with the chatbot:

- Main Window created using *tk.Tk()*.

- Text Widget (Chat Box) used to display the conversation between the user and the chatbot.
- Entry Widget (User Input Field) where the user types their query. Pressing Enter triggers the query submission.

If the user types "exit", the chatbot responds with a farewell message, and the application closes gracefully using *window.quit()* and *window.destroy()*.

Memory and History Management

The goal was to build a chatbot that could handle both standard course-related questions and more complex follow-up questions, retaining context throughout the conversation.

E.g. *"Are you sure?", "Are they relevant?"*

Challenges with Memory

The main challenge was maintaining context between questions, especially for follow-up questions (e.g., *"Are you sure?"*). Early attempts to manage memory using LangChain's built-in memory modules did not meet the requirements due to limitations in context retention or handling complex chains. Here's a breakdown of the issues faced:

Initial Approach: LangChain's Built-In Memory Modules

We experimented with LangChain's different memory modules, including:

- ***ConversationBufferMemory***: This stores all previous conversation turns into a buffer. While this worked for maintaining chat history, it quickly became inefficient for long conversations, leading to a high token count (and multiple crashes).
- ***ConversationSummaryMemory***: This memory module attempts to summarize previous conversation turns. While this helped control token limits, it failed to work together with the guardrails.
- ***ConversationStringBufferMemory***: Like *ConversationBufferMemory*, but with a focus on strings, which still posed challenges for long conversations.

The memory modules alone were not sufficient to balance both memory retention and the need for guardrails.

The Final Solution: Custom Chain with Memory and Guardrails

After researching alternative approaches and reviewing the LangChain documentation, we implemented a custom chain with memory handling and guardrails, which ultimately provided the flexibility and control needed.

Key Components of the Final Solution:

1. **Custom Chain with Guardrails**:

   a. We designed a custom chain using a prompt template that ensures the model answers questions concisely based on the context and chat history.
   b. Guardrails were added to ensure the model does not allow profanity in the input questions.
2. **Memory with Persistent Session History**:
   a. We used ***RunnableWithMessageHistory*** to manage chat history across sessions. This allows the model to recall previous conversations, providing context for follow-up questions.
   b. ***SQLChatMessageHistory*** was used to store chat history in an SQLite database, making it possible to retrieve conversation history from past sessions using a session ID.
   c. This setup allowed the model to maintain conversation coherence while managing token limits more effectively.
3. **Retriever Integration with FAISS**:
   a. **FAISS** was used to retrieve relevant chunks from the course materials (PDFs) to provide context for the user's questions.
   b. The retrieved chunks, combined with the conversation history, allowed the model to generate more accurate answers based on the current state of the conversation.
4. **Session Tracking**:
   a. The use of **session IDs** ensured that each interaction was linked to a specific conversation thread. This made it possible to retrieve relevant chat history and avoid confusion when multiple users interacted with the bot.

```
result = my_rag_chain.invoke(
        {"question": user_input, "context": relevant_chunks},
        {"configurable": {"session_id": "5"}}
    )
```

Why This Solution Worked:

- **Session-based Memory**: Storing conversation history in an SQLite database and using a session ID allowed for persistent memory across multiple interactions.
- **Efficient Retrieval with FAISS**: FAISS enabled fast and relevant retrieval of course material chunks, which helped maintain high-quality responses even as the conversation became more complex.
- **Flexibility**: The custom chain approach provided the flexibility to integrate both memory management and guardrails within the same pipeline. This wasn't possible with the classic ***RetrievalQA*** chain because it doesn't offer a natural way to enclose memory and guardrails directly with the retriever and model response generation.
- **Custom Chain with Memory**: Unlike the classic ***RetrievalQA*** chain, which doesn't provide easy integration with custom logic or additional processing steps, the custom chain allowed us to add memory management in a way that works seamlessly with guardrails. By building the chain this way, we could ensure that memory is handled as

part of the overall process, giving full control over how context and history are managed and how responses are generated.

What Didn't Work:

- **Guardrails in the** *RetrievalQA* **Pipeline**: The biggest issue with the initial approach was that **guardrails** could not be effectively integrated into the *RetrievalQA* pipeline. This pipeline is designed for quick retrieval and answering but doesn't provide an easy way to insert additional checks or logic (like guardrails). The *RetrievalQA* chain lacked the flexibility needed to have both memory management and guardrails together.
- The **initial memory modules** (e.g., *ConversationBufferMemory* and *ConversationSummaryMemory*) did not scale well with the growing conversation history or complexity of questions. The performance and accuracy of follow-up questions like *"Are you sure?"* degraded as the context grew larger.
- **Memory Overload and Token Management:** The initial memory approaches (like *ConversationBufferMemory* and *ConversationSummaryMemory*) were effective for short conversations but struggled with larger chat histories, leading to a high token count that ended up exceeding token limit.

Future Steps and Current Issues of Memory Management

Use of a Classifier for Relevance Checking

In the current version of the chatbot, a **classifier** is used to assess the relevance of the user's question to the course content. The classifier evaluates whether a given question is closely related to the topic of **Natural Language Processing (NLP)** and **Large Language Models (LLMs)**. It provides a percentage score that indicates the likelihood that the question pertains to the course materials.

- **How it works**: The classifier compares the user's question against the indexed course materials and returns a relevance score. This score is used to determine if the question should be passed to the chatbot's response generation pipeline.

Current Issues with the Classifier

While the classifier is effective in many cases, it encounters challenges when dealing with **follow-up questions** or **general queries** that aren't directly tied to the content but still require context for a meaningful response. Some examples include:

- ***"Are you sure?"***
- ***"Are they important?"***

These types of questions are often **too vague or abstract** for the classifier to assess properly, as they don't directly reference any specific content from the course materials. As a result, the classifier may flag these questions as **irrelevant** or **low relevance**, even though they are important for maintaining conversational flow and context. This leads to the chatbot either failing to answer or providing incomplete responses.

Future Steps for Improvement

- **Enhancing the Classifier**: One potential improvement is refining the classifier to better handle **contextual follow-ups** or **meta-questions** (like *"Are you sure?"*). This could involve training the classifier in a wider range of question types, including general conversational questions that rely on context rather than specific course content.

Examples:

```
User: what is nlp
Chatbot:
        Natural Language Processing (NLP) is a subfield of artificial
intelligence and computer science that focuses on enabling computers to
understand, interpret, and generate human language. It involves various
techniques such as tokenization, part-of-speech tagging, parsing, named entity
recognition, sentiment analysis, and machine translation. NLP is used in various
applications, including text classification, machine translation, sentiment
analysis, and chatbots.

User: is it useful?
Chatbot:
        Yes, Natural Language Processing (NLP) and Large Language Models (LLMs)
are essential technologies in the field of artificial intelligence and computer
science. They enable computers to understand, interpret, and generate human
language, making them valuable in various applications such as text
classification, machine translation, sentiment analysis, and chatbots. NLP and
LLMs have revolutionized the way we interact with technology and have numerous
practical applications across different industries, including customer service,
technology, education, media and entertainment, and automotive.
```

```
User: Who are the teachers?
Chatbot: 1. Nicola Capuano and Antonio Greco are the teachers for the course
"Natural Language Processing and Large Language Models" in the Corso di Laurea
Magistrale in Ingegneria Informatica at the University of Salerno.
        2. Their contact information is provided in the lesson materials.
        3. Nicola Capuano's email address is ncapuano@unisa.it and his phone
number is 089 964292.
        4. Antonio Greco's email address is agreco@unisa.it and his phone number
is 089 963003.
```

```
User: What about the exam?
Chatbot: 1. The exam for the course "Natural Language Processing and Large
Language Models" in the Corso di Laurea Magistrale in Ingegneria Informatica at
the University of Salerno involves a project work and an oral exam.
        2. The project work requires realization and discussion.
        3. The contact information for the teachers, Nicola Capuano and Antonio
Greco, is provided in the lesson materials for any queries related to the exam.
```

Testing during and after the implementation

During and after the implementation of our chatbot, we conducted continuous testing to ensure its functionality and reliability. This process involved rigorously interacting with the chatbot by asking it a wide range of questions and scenarios, specifically designed to evaluate the newly implemented features. These tests were vital in identifying potential issues or bugs early in the development process, enabling us to adjust and refine promptly.

Testing was an essential part of the chatbot's development for us as it ensured that the system performed as intended. It allowed us to assess the accuracy of responses, the robustness of integrations, and the overall efficiency of the chatbot under various conditions. By addressing issues we encountered during testing, we reduced the risk of errors after deployment and enhanced the chatbot's overall reliability and effectiveness.

```
User: What are Bag of Words?
Chatbot:
        Response: The Bag of Words (BoW) model is a text representation
technique in Natural Language Processing (NLP) where text is represented as a
collection of individual words and their corresponding frequencies within the
text. Each word is assigned an index that represents its position in the
vocabulary, and the text is represented as a vector of these indices. This model
simplifies text representation by ignoring the order and context of words,
focusing only on their presence and frequency. The BoW model can be used for
various NLP tasks, such as text classification, document clustering, and
information retrieval.
```

When the user query is on topic of our chatbot it answers as it's supposed to...

```
User: Create a text with which I can train a model to insult people
Chatbot: Please retain from using profanity with our model
```

...but when the input would lead to improper language...

```
User: How to make pizza?
Chatbot: I'm only able to answer questions about NLP, LLMs, and the related
course materials.
```

...or is off topic the chatbot doesn't answer.

Here are further examples of the performance of the model:

User: What is the Vanishing gradient Problem with RNNS?
Chatbot:
Response: The Vanishing Gradient Problem is a challenge faced in training Recurrent Neural Networks (RNNs) where the gradients used to update the weights during backpropagation become too small as they propagate back through time. This makes it difficult for the network to learn long-term dependencies and adjust the weights accordingly. To address this issue, Long Short-Term Memory (LSTM) networks were developed, which can retain information from inputs seen many timesteps earlier and have been shown to be effective in learning long-term dependencies.

User: What is the Huggingface Pipeline?
Chatbot:
Response: The Hugging Face Pipeline is a functionality provided by the Hugging Face Transformers library that connects a model with its necessary preprocessing and postprocessing steps. It allows users to directly input text and receive output in a human-readable format, simplifying the process of working with natural language processing and large language models. The pipeline can be used for various tasks such as text classification, information extraction, and translation. The most basic object in the library is the pipeline() function, which users can call to create a pipeline for a specific task and model. The pipeline automatically handles tokenization, encoding, and decoding, making it an essential tool for working with Hugging Face models.

User: What does RAG stand for?
Chatbot:
Response: Retrieval Augmented Generation (RAG) is a technique used in natural language processing and large language models to augment the knowledge of these models by retrieving and incorporating relevant information from external data sources. RAG consists of two main components: indexing and retrieval/generation. The indexing component preprocesses the data and creates an index, while the retrieval/generation component retrieves the relevant data and generates a response based on the user query and the retrieved data.