AS.470.738 Final Project

John Morse

Introduction

The AI solution developed in this application is designed to support public inquiry into the context and actual policies implemented through executive orders enacted by the Trump Administration. The application acts as a simple Q&A retrieval tool, allowing questions about executive orders and providing two responses: an overall summary of what statements are made about the topic across the executive orders, and a three-bullet summary of the text in each of the three most relevant executive orders.

> NOTE: This Quarto document is implemented based on an LLM model accessed through Microsoft Azure. This has been used for faster response time during development. The primary deliverable has been implemented using Mistral 7B for LLM.

Libraries & Initialization

```
library(tidyverse)
library(reticulate)
library(glue)
library(httr)
library(jsonlite)
library(keyring)
```

The following variables control whether initial loading, chunking, and embedding the RAG corpus, based on the executive orders, is included during rendering. If set to FALSE, the code assumes these steps have been completed and stored in files for direct retrieval. After an initial rendering, subsequent efforts can be processed more quickly by skipping the effort.

```
# Init flag indicates if initialization should run.
# If FALSE, load needed files from CSV.
init_flag <- FALSE

# Embed flag indicates if chunks need creating and embedding.
# If FALSE, load needed files from CSV.
embed_flag <- FALSE
```

This next section loads the text from cleaned text files and stores them in a base file. This will serve as the base of our RAG corpus for the application. It can be considered a reference library of executive orders.

```
if (!init_flag) {
  # To skip initialization, load the corpus from CSV.
  eo_corpus <- read_csv("eo_corpus.csv")
} else {
  # continue with initialization

  # load the original meta data for reference
  fname <- paste0("documents_signed_in_2025_signed_by_",
                  "donald_trump_of_type_presidential_document_",
                  "and_of_presidential_document_type_executive_order-1.csv")
  eo_meta <- read_csv(fname)

  # Create a new data frame with the
  # executive_order_number, pub date, and title columns from eo_meta.
  # Add an empty column to store the text retrieved from each document.
  eo_base <- eo_meta %>%
    filter(!is.na(pdf_url), !is.na(executive_order_number)) %>%
    select(executive_order_number, publication_date, title) %>%
```

```
  mutate(text = NA_character_)

# Iterate through the executive order numbers in eo_base.
# The number is used to concatenate a file name for each
# of the text files generated in the scrape_eos script.
# The text is loaded and stored in a new column.
for (i in seq_len(nrow(eo_base))) {
  eo_number <- eo_base$executive_order_number[i]

  #filename <- glue("executive_orders_pdf/EO_{eo_number}.txt")
  filename <- glue("executive_orders_cleaned/EO_{eo_number}.txt")

  if (!file.exists(filename)) {
    message(glue("File for EO {eo_number} does not exist."))
    next
  }
  text <- read_file(filename)
  eo_base$text[i] <- text
}

# Copy the data to eo_corpus so that we retain this base file.
eo_corpus <- eo_base
# Clean the text data:
eo_corpus <- eo_corpus %>%
  rename(eo_id = executive_order_number) %>%  # Rename executive_order_number column
  mutate(clean_text = gsub("\\s+", " ", text),   # Remove excessive white space
         clean_text = trimws(clean_text)) %>%
  select(-text) %>%                           # Drop the text column
  mutate(clean_text = tolower(clean_text))   # Convert all text to lower case.

## eo_corpus is now a reference data set
## with full text of EOs. The clean_text
# column includes that text.
```

```
  # Save to CSV for future use.
  write_csv(eo_corpus, "eo_corpus.csv")
}

## When we exit this section, eo_corpus has the full text of
## executive orders in the clean_text column.
```

Chunk RAG Corpus

This section breaks the RAG corpus, the EO text, into sentence size chunks. FIrst we provide a function that accepts a long string of text and breaks it up based on the presence of periods.

```
# Create a function that will chunk the text in eo_corpus
# based on sentences. Use a period to identify the end of a sentence.
chunk_text_by_sentence <- function(text, min_chunk_size = 5) {
  # Split the text into sentences based on periods.
  sentences <- unlist(strsplit(text, "(?<=\\.)\\s+", perl = TRUE))

  # Drop sentences that have fewer than min_chunk_size characters.
  # We assume these do not have significant value.
  sentence_lengths <- nchar(sentences)
  sentences <- sentences[sentence_lengths >= min_chunk_size]

  # Return the chunks (sentences) as a character vector.
  return(sentences)
}
```

Now we can use that function and apply it to every row of our base RAG data frame. The new data frame has a separate row for every sentence of every document. We also assign chunk IDs for each row of each document.

```r
# This is controlled by the embed_flag
if (!embed_flag) {
  # Load the already chunked data from CSV.
  chunked_corpus <- read_csv("chunked_corpus.csv")
} else {
  # continue with chunking eo_corpus

  chunked_corpus <- eo_corpus %>%
    # Apply the chunk_text_by_sentence function to
    # each row in the clean_text column.
    mutate(chunks = map(clean_text, ~ chunk_text_by_sentence(.x,
                                      min_chunk_size = 12))) %>%

    # Unnest the list of chunks into separate rows.
    # Each chunk gets its own row tied to the original
    # document id.
    unnest_longer(chunks, values_to = "chunk_text") %>%
    select(eo_id, chunk_text)

  # Now that we have the sentences, we can
  # remove punctuation from the chunked text.
  chunked_corpus <- chunked_corpus %>%
    mutate(chunk_text = gsub("[[:punct:]]", "", chunk_text))

  # Add a column with the number of characters in each chunk.
  chunked_corpus <- chunked_corpus %>%
    mutate(chunk_length = nchar(chunk_text))

  # By each eo_id, create a chunk ID for
  # each chunk within that document, and store in chunk_num.
  chunked_corpus <- chunked_corpus %>%
    group_by(eo_id) %>%
    mutate(chunk_id = row_number()) %>%
```

```
    ungroup() %>%
    select(eo_id, chunk_id, chunk_length, chunk_text)

  # write the chunked corpus to CSV for later use
  write_csv(chunked_corpus, "chunked_corpus.csv")
}

## When we exit this section, chunked_corpus has the
## chunked text in the chunk_text column.
```

Embed RAG Corpus

To embed the RAG corpus, as well as the user query that will be compared to the the corpus, we need an LLM model that is deigned for embedding text. This code loads the Pyuthon `sentence-transformers` library, then uses that to load a model from MiniLM.

```
# Import the sentence_transformers Python library
sentence_transformers <- import("sentence_transformers")
# Load a specific pre-trained model: all-MiniLM-L6-v2
model <- sentence_transformers$SentenceTransformer("all-MiniLM-L6-v2")
```

With the MiniLM model, we can embed each chunk of text previously created from the RAG corpus.

```
if (!embed_flag) {
  # To skip embedding, load the embedded chunks from a file.
  embed_corpus <- readRDS("embed_corpus.rds")
} else {
  # continue with embedding

  # Copy the chunked data frame to preserve the original data
  embed_corpus <- chunked_corpus
```

```
  # Add new column to data frame with the embeddings.
  embed_corpus$embedding <- map(embed_corpus$chunk_text,
                                function(text) model$encode(text))

  # The embed_corpus data frame now includes
  # the chunk_text and the associated embeddings.
  # Save to CSV for future use.
  saveRDS(embed_corpus, "embed_corpus.rds")
}

## When we exit this section, embed_corpus has the vector
## embeddings for each chunk in the embedding column.
```

Fetch Top K Chunks

Cosine similarity does the important work of comparing one set of vector embeddigns to another set of vector embeddings, then returning a value indicating similarity. Here we wrap this mathematical process into a function for easy reference.

```
cosine_sim <- function(a, b) {
  sum(a * b) / (sqrt(sum(a^2)) * sqrt(sum(b^2)))
}
```

Here we have a function that allows us to compare a user query or question to each sentence in the chunked and embedded RAG corpus. This let's us find elements of text that are most similar to the user query. To do this, we have to:

- Embed the user query using the same MiniLM model that the RAG was embedded with, and
- Call the Cosine similarity function for each row in the chunked corpus to compare it to the user query.

The function returns the three most similar chunks in the RAG corpus.

```r
# Fetch Top K chunks for query based on similarity scores.
fetch_top_k <- function(query_text, corpus_df, model, k = 3) {
  # encode the query based on the LLM model provided.
  query_embed <- model$encode(query_text)

  # Calculate similarity scores using our cosine_sim function.
  corpus_df <- corpus_df %>%
    mutate(similarity = map_dbl(embedding,
                                ~ cosine_sim(., query_embed))) %>%
    # sort the data frame by similarity scores
    arrange(desc(similarity)) %>%
    # return just the Top K results
    slice_head(n = k)

  # Return the data frame with the Top K results.
  return(corpus_df)
}
```

Get Overview

In this version of the application, I am not using a Hugging Face library locally. Instead, I am using Azure OpenAI from Microsoft. This requires deploying an LLM model inside of the Azure portal and Azure Foundry. I chose `gpt-4o-mini` (i.e. GPT 4o).

Through Hugging Face, we would load a model like Mistral 7B using the `transformers` Python library. With an LLM pipeline based on the model, we can make LLM requests. Azure Open AI uses a REST API. The call is sent to the LLM model via https, offloading the processing onto Microsoft Azure. With the local Mistral 7B library, the LLM calls in this application take over an hour. With Azure Open AI, they take only seconds.

The `azure_chat` function here bundles the user prompt into a REST API call. The function can then be used in much the same way as the call to the local LLM model.

```r
# Function to call Azure OpenAI chat completions. This takes
# the place of a transformer pipeline call in Hugging Face.
# This uses a REST API call to Azure OpenAI.
azure_chat <- function(prompt,
                       deploy = "jhu-summarizer",   # AOAI Deployment on gpt-4o-mini
                       api_ver = "2023-05-15",      # Deployment API Version
                       max_tokens = 200L,
                       temperature = 0) {

  api_key  <- keyring::key_get("aoai_api_key")       # stored securely
  endpoint <- keyring::key_get("aoai_endpoint")
  url <- glue("{endpoint}/openai/deployments/{deploy}/chat/completions?api-version={api_ver}")

  body <- list(
    messages = list(
      list(role = "system", content = "You are a concise, evidence-based assistant. Answer only within <ANSWER>."),
      list(role = "user", content = prompt)
    ),
    max_tokens = max_tokens,
    temperature = temperature
  )

  resp <- POST(
    url,
    add_headers(`api-key` = api_key, `Content-Type` = "application/json"),
    body = toJSON(body, auto_unbox = TRUE)
  )
  stop_for_status(resp)
  out <- content(resp, as = "parsed", type = "application/json")
  out$choices[[1]]$message$content
}
```

Now we can define a function that accepts the user's request, identifies the Top 3 chunks in our RAG corpus based on the

query, then generate a summary using our LLM model. Note how the `azure_chat` function is passed as the LLMgenerator parameter to this function. With a Hugging Face LLM like Mistral 7B, we would pass the model pipeline returned by the `transformers` library.

```
# Function to return overview of EO content based on question submitted.
get_overview <- function(aquery,
                         ragdf = embed_corpus,
                         embed_model = model,
                         top_k = 3,
                         LLMgenerator = azure_chat) {

  # Fetch top-k relevant chunks (your existing similarity search function)
  top_results <- fetch_top_k(aquery, ragdf, embed_model, top_k)
  context <- paste(top_results$chunk_text, collapse = "\n")

  # Build the structured prompt
  myprompt <- glue(
    "<CONTEXT>\n{context}\n</CONTEXT>\n\n",
    "<QUESTION>\n{aquery}\n</QUESTION>\n\n<ANSWER>"
  )

  # Call Azure OpenAI
  output <- LLMgenerator(myprompt, max_tokens = 200L, temperature = 0)

  # Strip everything before <ANSWER>
  output <- sub("(?s).*<ANSWER>\\s*", "", output, perl = TRUE)
  return(output)
}
```

Get Document Summaries

This function is designed to get summaries of specific executive orders. Namely, it fetches the top 3 executive orders again, identifies the order IDs, then goes back to the base RAG reference library (in `eo_corpus`), and generates a summary of

each in the form of three bullet points. Note that I am using the `azure_chat` function again as the LLMgenerator.

Another key element to notice in this function is how I am adding text to the `request` variable emphasizing that these documents are executive orders signed in 2025 by Trump. This was mitigation for hallucinations I encountered with Mistral 7B. The Mistral 7B LLM was published in 2023 when President Biden was still in office. I frequently encountered document summaries that wrongly attributed these executive orders to Biden instead of Trump.

This became a lesson in prompt engineering. From the text assigned to the `request`, you can see that I am forcing the response to attribute the executive orders correctly through my prompt.

```
## Function to get summaries of the most relevant
## executive orders from the corpus based on the query.
get_doc_summaries <- function(aquery,
                              ragdf = embed_corpus,
                              full_corpus = eo_corpus,
                              embed_model = model,
                              top_k = 3,
                              LLMgenerator = azure_chat) {

  # Get the Top K chunks
  top_chunks <- fetch_top_k(aquery, ragdf, embed_model, top_k)
  # Get the unique eo_id values from those chunks
  eo_ids <- unique(top_chunks$eo_id)

  # Get the full text for those eo_id values
  relevant_docs <- full_corpus %>%
    dplyr::filter(eo_id %in% eo_ids)

  # Iterate through those documents and generate a summary for each one.
  summaries <- list()

  for (i in seq_len(nrow(relevant_docs))) {
    doc_text <- relevant_docs$clean_text[i]
    doc_id <- relevant_docs$eo_id[i]
```

```
  request <- glue(
    "Provide a summary of the document above in three bullet points. ",
    "You are analyzing executive orders signed by President Donald Trump in 2025. ",
    "Do not attribute them to any other president."
  )

  summary_prompt <- glue(
    "<DOCUMENT>\n{doc_text}\n</DOCUMENT>\n\n",
    "<REQUEST>\n{request}\n</REQUEST><SUMMARY>"
  )

  # Call the AOAI generator
  summary_response <- LLMgenerator(summary_prompt,
                                   max_tokens = 300L,
                                   temperature = 0)

  # AOAI returns plain text content
  summary_output <- summary_response

  # Strip off everything before the <SUMMARY> tag
  summary_output <- sub("(?s).*<SUMMARY>\\s*", "", summary_output, perl=TRUE)
  # Replace the <SUMMARY> tag with the eo_id for clarity
  summary_output <- gsub("<SUMMARY>", paste0("<", doc_id, ">"), summary_output)
  summaries[[as.character(doc_id)]] <- summary_output

  # Remove the <ANSWER> and </ANSWER> tags if present
  summaries[[as.character(doc_id)]] <- gsub("<ANSWER>", "", summaries[[as.character(doc_id)]])
  summaries[[as.character(doc_id)]] <- gsub("</ANSWER>", "", summaries[[as.character(doc_id)]])

}

# Return the list of summaries
return(summaries)
```

```
}
```

Generate Full Response

This final function puts everything together. WIth just a user query, it will call the prior functions to identify the most relevant documents in the RAG corpus, summarize the overall policy stance based on the executive orders, and provided three-bullet summaries for each of the Top 3 documents in the RAG corpus.

```
# In one request, respond to a query with both
# an overview and document summaries.
generate_response <- function(query) {
  overview <- get_overview(query)        # Get overview of the documents
  summaries <- get_doc_summaries(query)  # Get summaries of individual docs

  # Combine into a final response
  answer <- paste0("Overview:\n", overview, "\n\nDocument Summaries:\n")
  for (eo_id in names(summaries)) {
    answer <- paste0(answer, "EO ID ", eo_id, "", summaries[[eo_id]], "\n\n")
  }
  answer <- gsub("</SUMMARY>", "", answer)
  return(answer)
}
```

So now we can pose a question. This has a selection of questions that can be commented or uncommented as desired.

```
# Provide a query and test the full function
#my_question <- "What do the documents provided here say about gun control?"
#my_question <- "What did Trump say about immigration?"
#my_question <- "What do the documents provided here say regarding climate change?"
#my_question <- "What executive orders address healthcare reform?"
my_question <- "What are Trump's policies on education as outlined in these executive orders?"
```

And we can call the function above and generate an LLM response based on our query.

```
the_response <- generate_response(my_question)
```

Finally, we can show the response from the user query:

```
# Some extra code to get the text to appear correctly
# when rendering to PDF.
parts <- strsplit(the_response, "\n", fixed = TRUE)[[1]]
wrapped_parts <- lapply(parts, strwrap, width = 90)
cat(unlist(wrapped_parts), sep = "\n")
```

```
Overview:
Trump's policies on education, as outlined in Executive Order 14242, focus on empowering
parents, teachers, and communities to ensure student success. Key points include:

1. **Compliance with Federal Law**: Ensuring that federal education funds are allocated
in compliance with federal laws and policies, particularly against illegal
discrimination.
2. **Opposition to Certain Programs**: Prohibiting funding for programs labeled as
promoting diversity, equity, and inclusion, or gender ideology.
3. **Regulatory Enforcement**: Developing a plan to advance these policies through
regulatory enforcement, litigation, and collaboration with Congress and state
governments.

Overall, the order emphasizes a shift towards parental and community empowerment while
restricting certain educational programs.

Document Summaries:
EO ID 14322
- Executive Order 14322 aims to protect and expand college sports, particularly
non-revenue and women's sports, by establishing policies to ensure scholarship
opportunities and roster spots are maintained or increased in collegiate athletic
```

departments based on their revenue.
- The order prohibits third-party pay-for-play payments to collegiate athletes, emphasizing that any compensation should reflect fair market value for endorsements, while also directing federal agencies to develop plans to enforce these policies.
- It includes provisions for clarifying the status of collegiate athletes, protecting college athletics from legal challenges, and consulting with the U.S. Olympic and Paralympic Committee to safeguard the role of collegiate athletics in developing athletes for international competition.

EO ID 14242
- Executive Order 14242 aims to improve education outcomes by closing the Department of Education and returning authority over education to states and local communities, citing failures of federal control and bureaucracy.
- The order highlights the significant federal spending on education, the poor performance of students in reading and math, and the inefficiency of the Department of Education in managing student loans.
- It mandates the Secretary of Education to facilitate the department's closure while ensuring compliance with federal laws and prohibits funding for programs promoting illegal discrimination under the guise of diversity and inclusion.