DATA236: Distributed Systems for Data Engineering

Objective:
Create a Node.js application with Express that implements a simple user authentication system for the Department of Applied Data Science at SJSU. The application should include:

- User login and logout functionality.
- Session management to keep users logged in.
- Protected routes that only logged-in users can access.
- Styling using Bootstrap to make the application visually appealing.

Requirements:

1. Routes- Handled Separately in a router:

Home Page (/): Display a welcome message for ADS-SJSU. Show a link to the login page if the user is not logged in. Show a link to the dashboard and log out if the user is logged in.

Login Page (/login): Display a login form with fields for username and password. Validate the credentials and log the user in if they are correct. Redirect to the dashboard on successful login.

Dashboard Page (/dashboard): Display a welcome message with the user's name. Show a logout link. Protect this route so only logged-in users can access it.

Logout (/logout): Destroy the session and redirect the user to the home page.

2. Session Management: Use express-session to manage user sessions. Store the logged-in user's information in the session. Ensure that the session cookie is secure

3. Styling with Bootstrap: Explore and use Bootstrap to style all pages. Make the application responsive and visually appealing. Use Bootstrap components, such as the Navbar for navigation, Cards for forms and content, Buttons for actions, and Alerts for messages.

Include one screenshot per route page. Include only the app.js code at the bottom and one screenshot of your views directory

Part 2: Compare Three LlamaIndex Chunking Techniques (Retrieval-Only RAG)

Implement **three chunking techniques** in LlamaIndex on the *Tiny Shakespeare*, build in-memory vector indexes, and **compare retrieval quality**. You'll print the **embeddings and retrieval outputs** for a shared query, then argue which technique is best and why.

Techniques to implement:

1. **Token-based chunking** — TokenTextSplitter (LlamaIndex)
2. **Semantic chunking** — SemanticSplitterNodeParser (LlamaIndex)
3. **Sentence-window chunking** — SentenceWindowNodeParser (LlamaIndex)

Dataset

Use the same file as in class:

- Tiny Shakespeare (raw text):

  https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt

What you must build

A. Environment & Setup Install: llama-index, llama-index-embeddings-huggingface, sentence-transformers, faiss-cpu, numpy, pandas.

- Use a **public** sentence embedding model (e.g., sentence-transformers/all-MiniLM-L6-v2). (pull from huggingface)

B. One retrieval-only pipeline per technique

For each chunker (Token / Semantic / Sentence-window):

1. **Chunking**
    - Token: set a token chunk_size and chunk_overlap (choose sensible values). (LlamaIndex)
    - Semantic: pick a buffer_size and use your embed model for the splitter to find semantically coherent boundaries. (LlamaIndex)
    - Sentence-window: split to single sentences and attach a *window* (neighbor sentences) in metadata to keep surrounding context available. (LlamaIndex)
2. **Indexing (in memory)**
    - Build a VectorStoreIndex over your nodes with an **in-memory** vector store (e.g., SimpleVectorStore) to keep everything local and fast. (LlamaIndex)
3. **Retrieval-only function**
   Write a helper that, given a query and k, does the following:
    - Compute the **query embedding** (show its **dimension** and the **first 8 values**).
    - Retrieve **top-k** nodes; for each, compute and print:
        - Store similarity score (if available from retriever).
        - **Cosine similarity** between the query embedding and the **document embedding** (compute embeddings of the returned chunks explicitly).
        - Chunk length and a short text preview (first ~160 chars).
    - Print the **shapes** of the query vector and the stacked doc vectors.

Your printed output should clearly identify the technique used and list a **table** with: rank, store_score, cosine_sim, chunk_len, preview.

Query to use

Use this **one** query to print outputs for all three techniques:

- **Query:** Who are the two feuding houses?

You may optionally add 1–2 more queries (like, *"Who is Romeo in love with?"*, *"Which play contains the line 'To be, or not to be'?"*) to strengthen your comparison

What to compare (report section)

After you run the three pipelines:

1. **Retrieval Quality:**
    - **top-1 cosine** (highest similarity among the top-k for that technique)
    - **mean@k cosine** (average of top-k cosines)
    - **#chunks** produced by the chunker and the **avg chunk length** (characters or tokens)
    - **retrieval latency** in milliseconds (time the similarity search took; simple timer is fine)
2. **Observations (1–2 short paragraphs):**
    - Discuss **why** one technique performed better on this query (e.g., sentence coherence, semantic boundary detection, token-budget alignment, context carried via sentence window).
    - If the best technique differs across your optional extra queries, mention it.
3. **Your conclusion (2–5 sentences):**
    - State which technique you judge **best** for this corpus and **why**

Make a pdf with your code and the report as explained above.