Asistent AI pentru materia ASC

1 Introducere și scopul proiectului

Proiectul realizat are ca scop dezvoltarea unui asistent AI educațional capabil să răspundă la întrebări legate de materia Arhitectura Sistemelor de Calcul (ASC), utilizând conținutul cursurilor de la facultate. Aplicația se bazează pe un model de inteligență artificială pre-antrenat (LLaMA 3.1, accesat prin Ollama) și folosește o bază de date vectorială construită pe baza cursurilor în format PDF.

Scopul principal este de a sprijini studenții în învățarea individuală, permițându-le să pună întrebări în limbaj natural și să primească răspunsuri precise, extrase din materialele oficiale ale cursului.

2 Descrierea și analiza setului de date

Pentru antrenarea și testarea sistemului a fost utilizat un set de fișiere PDF reprezentând cursurile și materialele de laborator de la disciplina ASC.

Sursa datelor

Datele sunt împărțite în două directoare logice:

- `DOCS`: Conține fișiere PDF cu materia de curs generală.
- `DOCS2`: Conține fișiere PDF structurate special sub formă de Întrebare și Răspuns (Q&A).

Toate materialele provin din resurse oferite de facultate, destinate exclusiv uzului didactic.

Structura datelor

- **Format:** PDF (text și formule)
- **Domeniu:** Arhitectura Sistemelor de Calcul
- **Limbă:** Română

Preprocesare și împărțire

Procesul este gestionat de scriptul `Loader.py` și folosește strategii diferite în funcție de sursa datelor:

- **Pentru `DOCS` (Cursuri):** Documentele sunt încărcate cu `PyPDFLoader` și împărțite folosind `RecursiveCharacterTextSplitter`. S-a folosit o dimensiune a fragmentului (`CHUNK_SIZE`) de 1000 de caractere și o suprapunere (`CHUNK_OVERLAP`) de 200 de caractere.
- **Pentru `DOCS2` (Q&A):** Documentele sunt încărcate, iar textul este împărțit folosind o expresie regulată (RegEx) specifică (`QA_SPLIT_REGEX = r'(?=nQ:)'`). Această abordare păstrează unitatea logică a fiecărei perechi întrebare-răspuns.

Toate fragmentele rezultate (atât din cursuri, cât și din Q&A) sunt combinate într-o singură listă pentru a crea baza de date vectorială.

3 Descrierea algoritmului inteligent (AI)

Logica centrală a sistemului este definită în `rag_core.py`.

3.1 Tipul algoritmului

Proiectul folosește o abordare de tip RAG – Retrieval-Augmented Generation. Această metodă combină două componente:

- **Retrieval (Regăsire de informații)** – caută fragmente relevante în baza de date vectorială.
- **Generation (Generare)** – modelul AI (LLaMA 3.1) formulează un răspuns coerent folosind doar contextul extras.

3.2 Modelul de limbaj folosit

Sistemul folosește două modele distincte accesate prin Ollama:

- **Model de Generare (LLM):** `llama3.1`. Acesta este modelul principal care generează răspunsurile finale pentru utilizator.
- **Model de Embeddings:** `nomic-embed-text`. Acesta este un model specializat, folosit pentru a transforma fragmentele de text în vectori numerici (embeddings).

3.3 Embeddings și baza de date vectorială

Pentru reprezentarea semantică a textelor s-a folosit componenta `OllamaEmbeddings(model="nomic-em`
Acești vectori sunt stocați într-o bază de date `Chroma`, salvată local în directorul `chroma_db`.

3.4 Prompting și Filtrarea Contextului

Spre deosebire de o căutare simplă, sistemul folosește o metodă de filtrare avansată pentru a crește eficiența și acuratețea, menținând un timp de răspuns redus.

- **Prompt Template Strict:** S-a definit un *system prompt* (`PROMPT_TEMPLATE`) foarte strict, care instruiește modelul să acționeze ca un extractor de text. Modelul este forțat să răspundă *STRICT și DOAR* pe baza contextului și să nu-și folosească cunoștințele generale. Dacă informația lipsește, trebuie să răspundă cu un text specific.
- **Retriever cu Filtru:** S-a implementat un `ContextualCompressionRetriever`. Acesta folosește un retriever de bază care extrage un număr mai mare de documente ($k = 12$), urmat de un compresor `EmbeddingsFilter`. Acest filtru recalculează similaritatea local și păstrează doar documentele care depășesc un anumit prag (`EMBEDDING_FILTER_THRESHOLD = 0.60`).

4 Arhitectura și metodologie experimentală

4.1 Structura generală a aplicației

Aplicația este compusă din patru module principale și un fișier de configurare:

- `config.py` – Fișier central care stochează toate constantele (căi, nume de modele, parametri RAG).
- `Loader.py` – Script rulat pentru a citi PDF-urile din `DOCS` și `DOCS2`, a le fragmenta conform strategiilor diferite și a construi/rescrie baza de date vectorială `Chroma`.
- `rag_core.py` – Modulul central care definește funcția `initialize_rag_system()`, construiește lanțul RAG (inclusiv filtrul și formatarea contextului), încarcă modelele și configurează retriever-ul.
- `Learner.py` – Aplicația principală. O interfață grafică (GUI) bazată pe `Tkinter` care permite interacțiunea utilizatorului cu modelul. Folosește `threading` pentru a nu bloca interfața.
- `eval.py` – Script separat pentru evaluarea cantitativă a performanței modelului pe un set de 25 de întrebări predefinite.

4.2 Fluxul de lucru (RAG Optimizat)

1. Se încarcă baza de date vectorială `Chroma` și modelul de embeddings (`nomic-embed-text`).
2. Se inițializează modelul LLM `llama3.1` (prin Ollama).
3. Se configurează retriever-ul de bază pentru a extrage $k = 12$ documente.
4. Se configurează `EmbeddingsFilter` cu un prag de similaritate de 0.60.
5. Se creează `ContextualCompressionRetriever` folosind filtrul de mai sus.
6. Se construiește lanțul RAG (LCEL pipe).
7. Utilizatorul introduce o întrebare în interfața `Learner.py`.
8. Retriever-ul de bază extrage 12 documente; `EmbeddingsFilter` le filtrează rapid, păstrând doar cele relevante (care trec de prag).

9. Documentele filtrate sunt trimise funcției `format_docs`, care le transformă într-un singur string de context curat.

10. Acest string este injectat în `PROMPT_TEMPLATE` (cel strict).

11. Modelul LLM (cu `temperature=0.4`) generează un răspuns pe baza contextului și îl afișează în fereastra de chat.

4.3 Parametri importanți

- **Model LLM:** `llama3.1`
- **Embedding model:** `nomic-embed-text`
- **Bază de date:** `Chroma` (din `chroma_db`)
- **Retriever:** `ContextualCompressionRetriever` + `EmbeddingsFilter`
- **Număr fragmente (bază):** $k = 12$
- **Prag similaritate (filtru):** 0.60
- **Temperatură LLM (răspuns):** 0.4
- **Dimensiune chunk (cursuri):** 1000 caractere, overlap 200

5 Evaluare și Metodologie Implementată

Evaluarea este implementată activ în scriptul `eval.py`.

5.1 Setul de date pentru evaluare

S-a definit o listă de **25 de întrebări-cheie** din materia ASC, numită `EVAL_SET`. Fiecare întrebare din set este însoțită de o listă de cuvinte cheie esențiale (`expected_keywords`) care ar trebui să apară într-un răspuns corect.

Exemplu de item din `EVAL_SET`:

```
{
    "question": "De ce instrucţiunea 'mov [v], 0' produce eroarea
    ↪  'operation size not specified'?",
    "expected_keywords": ["operation size not specified", "eroare",
                          "dimensiunea", "ambiguitate", "octet",
                          "cuvânt", "dublucuvânt"]
}
```

5.2 Metodologia de testare

Scriptul `eval.py` parcurge automat fiecare întrebare din `EVAL_SET`:

1. Trimite întrebarea către sistemul RAG (inițializat prin `rag_core.py`).

2. Primește răspunsul generat de AI.

3. Compară (case-insensitive) răspunsul AI cu lista `expected_keywords`.

4. Calculează un prag de succes: un răspuns este considerat "Corect" dacă conține cel puțin jumătate (rotunjit în sus) din numărul total de cuvinte cheie așteptate.

5. Afișează la consolă statusul (Corect/Partial/Gresit) și cuvintele cheie lipsă (dacă e cazul).

5.3 Metrica de acuratețe

La finalul rulării, scriptul calculează o metrică de acuratețe generală bazată pe numărul de răspunsuri care au îndeplinit pragul de cuvinte cheie.

$$\text{Acuratețea} = \frac{\text{Număr răspunsuri considerate corecte}}{\text{Număr total întrebări (25)}} \times 100\%$$

Această metodă, deși nu evaluează corectitudinea semantică perfectă, oferă o măsură cantitativă rapidă a relevanței contextului extras și a capacității modelului de a formula răspunsuri pe baza lui.

6 Comparație cu Stadiul Curent (SOTA)

Metodologia implementată în acest proiect reprezintă o abordare eficientă a ceea ce este cunoscut în literatura de specialitate drept "RAG Clasic" sau "Naive RAG". Această paradigmă, fundamentată de lucrări precum *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks* (Lewis et al., 2020), presupune un flux liniar: primirea întrebării, regăsirea documentelor relevante și generarea unui răspuns pe baza acestora. Sistemul nostru optimizează acest flux prin utilizarea unui filtru rapid (`EmbeddingsFilter`) în locul unor apeluri LLM multiple pentru rafinarea contextului.

Stadiul actual (SOTA) în RAG (2023-2024) a evoluat însă către sisteme "adaptive" și "corective". Noile arhitecturi nu mai tratează contextul regăsit ca fiind implicit corect, ci introduc o etapă de validare.

- **Self-RAG (Auto-Reflecție):** Modele precum *SELF-RAG* (Asai et al., 2023) sunt antrenate să genereze "jetoane de reflecție" (reflection tokens). Aceste jetoane permit modelului să evalueze calitatea contextului regăsit *înainte* de a formula răspunsul. Modelul poate decide activ dacă informația este relevantă, dacă are nevoie de mai mult context sau dacă ar trebui să refuze să răspundă.

- **Corrective RAG (Auto-Corecție):** Alte abordări, precum *ReSeek* (2024), implementează un mecanism de auto-corecție. Dacă agentul AI (LLM-ul) judecă informația regăsită ca fiind irelevantă sau incorectă, el poate declanșa o nouă căutare ("re-planificare"), corectându-și dinamic strategia de regăsire pentru a găsi informații mai bune.

Comparație Directă

Sistemul nostru (bazat pe `EmbeddingsFilter`) este "pasiv". Dacă un document este semantic similar (trece pragul de 0.60), dar totuși incorect sau irelevant pentru nuanța

întrebării, LLM-ul va fi forțat să îl folosească, putând duce la erori pe care le-am observat (halucinații).

Sistemele SOTA sunt "active". Un model *Self-RAG* sau *ReSeek*, în aceeași situație, ar folosi mecanismul său intern de reflecție/judecată pentru a identifica contextul ca fiind slab și ar declanșa o nouă căutare, crescând substanțial acuratețea răspunsului final.

În concluzie, proiectul nostru stabilește o bază de referință (baseline) solidă și eficientă local, iar metricile din `eval.py` sunt esențiale pentru a măsura performanța acestei implementări "clasice". Trecerea la un model de auto-corecție reprezintă pasul următor evident pentru îmbunătățirea performanței, așa cum este menționat în secțiunea de "Avantaje" și "Viitor".

7 Rezultate și discuții

Scriptul de evaluare `eval.py` permite testarea rapidă a modificărilor aduse sistemului (ex. schimbarea pragului de filtrare, a modelului, a prompt-ului).

Limitări

- Evaluarea bazată pe cuvinte cheie este simplistă. Un răspuns poate fi semantic corect, dar poate folosi sinonime neincluse în lista `expected_keywords`.
- Calitatea depinde de conținutul cursurilor și de acuratețea textului extras din PDF.

Avantaje

- Sistemul este local, fără dependență de cloud (folosind Ollama).
- `ContextualCompressionRetriever` cu `EmbeddingsFilter` oferă un echilibru excelent între viteză și relevanță, eliminând apelurile LLM multiple din timpul regăsirii.
- Evaluarea automată (`eval.py`) permite *regression testing* și optimizare iterativă.

În viitor, se pot adăuga:

- O interfață web modernă (ex. FastAPI + React/Streamlit).
- Salvarea istoricului conversațiilor.
- Un mecanism de evaluare mai complex, posibil bazat pe un LLM ca judecător (LLM-as-a-judge).

8 Biblioteci folosite

- `tkinter` – interfața grafică
- `threading`, `sys`, `os`, `re`, `shutil` – utilitare de sistem
- `langchain_classic.retrievers` – (pentru `ContextualCompressionRetriever`)
- `langchain_classic.retrievers.document_compressors` – (pentru `EmbeddingsFilter`)

- `langchain_core` – (pentru `StrOutputParser`, `RunnablePassthrough`, `Document`)
- `langchain_ollama` – (pentru `OllamaEmbeddings`, `OllamaLLM`)
- `langchain_community` – (pentru `PyPDFLoader`, `DirectoryLoader`)
- `langchain_chroma` – (pentru `Chroma` DB)
- `langchain_text_splitters` – (pentru `RecursiveCharacterTextSplitter`)
- Lewis, P. et al. (2020). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. Advances in Neural Information Processing Systems (NeurIPS).
- Asai, A. et al. (2023). *SELF-RAG: LEARNING TO RETRIEVE, GENERATE, AND CRITIQUE THROUGH SELF-REFLECTION*. arXiv.
- (Sursă SOTA) Gao, Y. et al. (2023). *Retrieval-Augmented Generation for Large Language Models: A Survey*. arXiv.
- (Sursă SOTA) *ReSeek: A Self-Correcting Framework for Search Agents with Instructive Rewards*. (2024). arXiv.

9 Referințe

- Meta AI – LLaMA 3.1 Model Card
- LangChain Documentation – https://python.langchain.com/
- Ollama – https://ollama.ai
- ChromaDB – https://docs.trychroma.com