Лабораторна робота №5

RAG з генеративною моделлю (LLM Integration)

Мета роботи: інтегрувати механізм *semantic retrieval* з великою мовною моделлю (LLM) у межах архітектури *Retrieval-Augmented Generation (RAG)* та дослідити вплив контексту, отриманого з векторної бази даних, на якість, релевантність та повноту згенерованих відповідей.

Науково-теоретичне обґрунтування

1. Від retrieval до generation

До ЛР5 пайплайн виконував лише пошук, повертав список релевантних чанків із cosine score. LLM-інтеграція перетворює цей список на природномовну відповідь. Ключова ідея: LLM не 'знає' відповідь - вона синтезує її з наданих чанків (grounded generation), що мінімізує галюцинації.

2. Prompt template - центральний компонент ЛР5

Prompt template - це шаблон, що визначає як саме retrieved чанки та запит користувача подаються LLM. Якість відповіді критично залежить від структури промпту. Базовий шаблон складається з трьох частин:

- *System prompt - роль моделі, стиль відповіді, інструкції щодо використання контексту та поведінки при відсутності відповіді.*
- *Context block - відформатовані retrieved чанки, зазвичай пронумеровані для можливості citation.*
- *User question - запит у незмінному вигляді.*

3. Управління context window

LLM має фіксований ліміт токенів (context window): GPT-4o-mini - 128k, Gemini 1.5 Flash - 1M, Mistral 7B - 32k. Якщо сума розмірів чанків + системного промпту + відповіді перевищить ліміт - запит завершиться помилкою. Тому потрібно:

- *Контролювати сумарний розмір контексту перед відправкою (token counting).*
- *Вибирати Top-K таким чином, щоб загальний розмір не перевищував ліміт.*
- *При необхідності скорочувати або truncate чанки.*

4. Grounded generation та citation

Grounded generation - відповідь LLM обмежена наданим контекстом. Модель інструктується відповідати лише на основі переданих чанків і явно вказувати, якщо відповідь відсутня у контексті (відповідь 'I don't know' або 'Недостатньо інформації'). Citation grounding - прив'язка фрагментів відповіді до конкретних пронумерованих джерел [1], [2].

5. Типи prompt templates

Тип шаблону	**Стиль відповіді**	**Коли використовувати**	**Ключова інструкція**
Базовий	Коротко, без citation	Загальні запитання	Answer the question using the provided context.
З citation	Відповідь + посилання на чанк	Фактичні запитання	...Use [1], [2] to cite your sources.
Chain-of-thought	Розмірковування + відповідь	Складні запитання	Think step by step before answering.
Без відповіді	Відмова якщо немає контексту	Незнайомі теми	If context is insufficient, say 'I don't know'.

6. Вибір LLM для Colab

У навчальних проектах критично зручність та безкоштовний доступ. Рекомендований вибір:

Модель	Вартість	API ключ	Якість	Примітка
Google Gemini 1.5 Flash	Безкоштовно	Так (API key)	Висока	Рекомендовано для Colab
OpenAI GPT-4o-mini	~$0.15/1M	Так (API key)	Висока	Платний, але дешевий
Mistral 7B (HuggingFace)	Безкоштовно	Так (token)	Середня	Відкрита модель
Ollama llama3 (локально)	Безкоштовно	Ні (локально)	Середня	Потребує GPU/CPU
Anthropic Claude Haiku	~$0.25/1M	Так (API key)	Висока	Швидкий, ефективний

Ніколи не вставляйте API ключ напряму у код. Використовуйте: os.environ['API_KEY'] або Google Colab Secrets (ліва панель → key icon). Публічно розкритий ключ буде заблокований.

7. Оцінювання якості генерації

Після отримання відповіді потрібно оцінити їх якість. На цьому етапі використовують три виміри:

- *Faithfulness - чи відповідь базується лише на наданому контексті (0–1). Порушення = галюцинація.*
- *Answer Relevance - наскільки відповідь відповідає запиту (0–1).*
- *Context Recall - чи retrieval знайшов усі необхідні для відповіді чанки (0–1).*

Повноцінна автоматична оцінка через RAGAS реалізується у ЛР6. У ЛР5 виконується ручна оцінка за п'ятибальною шкалою або бінарно.

Стек технологій

Sentence-Transformers *— створення embeddings*

ChromaDB / FAISS *— векторна база даних*

LangChain *— побудова RAG pipeline*

OpenAI / Llama / Mistral *— генеративна модель*

Python

NumPy / Pandas

Matplotlib *— візуалізація результатів*

Зміст роботи

Завдання 1. *Встановити залежності та ініціалізувати LLM API клієнт (рекомендовано Google Gemini або OpenAI).*

Методичні рекомендації

Встановлюємо всі необхідні бібліотеки:

- sentence-transformers – генерація щільних векторів (embeddings) з тексту;

– chromadb – локальна векторна база даних з HNSW-індексом;

– datasets – зручне завантаження публічних дата сетів;

– pandas – табличні операції та збереження результаті;

– google-generativeai – Python SDK для Google Gemini API (безкоштовно).

Альтернативи LLM (розкоментуйте потрібне):

```
#!pip install openai            # GPT-4o-mini (~$0.15 / 1M токенів)
#!pip install anthropic         # Claude Haiku (~$0.25 / 1M токенів)

!pip install sentence-transformers chromadb datasets pandas
!pip install google-generativeai
```

Підключаємо всі бібліотеки і визначаємо центральні константи системи.

Константи і для чоговони:

– BI_ENC_MODEL – модель bi-encoder для генерації векторів документів і запитів. *all-MiniLM-L6-v2* дає 384-вимірні вектори, є швидкою і точною для англомовних текстів

– CE_MODEL – *cross-encoder* для ранкінгу кандидатів. Оцінює пару (запит, документ) спільно – набагато точніший за cosine

– TOP_K_BIENC – скільки кандидатів забираємо з *ChromaDB* на першому етапі. Більше кандидатів = краще покриття, але повільніший cross-encoder.

– TOP_K_RERANK – скільки документів залишаємо після ранкінгу і передаємо в контекст LLM. 3 – оптимум між якістю та розміром *prompt'y*

– MAX_CONTEXT – ліміт символів контексту перед відправкою до LLM. Захищає від перевищення *context window* моделі.

```
import os
import time
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

BI_ENC_MODEL  = 'sentence-transformers/all-MiniLM-L6-v2'
CE_MODEL      = 'cross-encoder/ms-marco-MiniLM-L-6-v2'
```

```
COLLECTION_NAME = 'lab5_rag'

TOP_K_BIENC  = 10     # кандидати від bi-encoder
TOP_K_RERANK = 3      # фінальний Top-N після cross-encoder

# Ліміт символів у контексті для LLM
MAX_CONTEXT  = 3000
```

Підключаємо Google Gemini - рекомендована модель для Colab, оскільки вона безкоштовна і має великий контекстний window (1M токенів).

ВАЖЛИВО щодо API ключа:

Ніколи не вставляйте ключ прямо в код - він потрапить у git/GitHub!

Спосіб 1 (Colab): ліва панель → іконка ключа → додати GEMINI_API_KEY

from google.colab import userdata

GEMINI_KEY = userdata.get('GEMINI_API_KEY')

Спосіб 2 (локально): export GEMINI_API_KEY=ваш_ключ в терміналі

Отримати безкоштовний API ключ: https://aistudio.google.com/apikey

```
import google.generativeai as genai

GEMINI_KEY = os.environ.get('GEMINI_API_KEY', 'YOUR_KEY_HERE')

if GEMINI_KEY == 'YOUR_KEY_HERE':
    print("API ключ не встановлено!")
else:
    genai.configure(api_key=GEMINI_KEY)

    # Ініціалізуємо модель Gemini 1.5 Flash
    llm = genai.GenerativeModel('gemini-1.5-flash')
    print(f"LLM ініціалізовано: {llm.model_name}")
```

Завдання 2. *Повторно використати ChromaDB-колекцію з ЛР1 або перебудувати пайплайн retrieval.*

Завдання 3. *Реалізувати функцію build_prompt(query, chunks) - базовий і citation-style шаблони.*

Методичні рекомендації

Код, який потрібно написати повинен генерувати prompt для LLM у системі RAG: query + retrieved chunks → структурований prompt → LLM.

SYSTEM_* - системні інструкції (роль моделі), найважливіша частина RAG - керування поведінкою LLM. Системні інструкції для різних стилів (приклад):

– відповідає тільки на основі контексту. Використання: прості запити → швидки відповіді

```
SYSTEM_BASIC = """You are an AI assistant for a media catalog.
Answer the question based ONLY on the provided context.
If the answer is not found in the context, say exactly:
'Insufficient information in the provided documents.'
Be concise and accurate."""
```

– Додає citation [1], [2]. Використання для наукових задач, пояснювального AI, RAG систем.

```
SYSTEM_CITATION = """You are an AI assistant for a media catalog.
Answer the question based ONLY on the provided context.
For each fact you mention, cite the source using [N] format (e.g., [1],
[2]).
If the answer is not found in the context, say exactly:
'Insufficient information in the provided documents.'
Be concise and accurate."""
```

– Модель думає по кроках, потім дає відповідь. Використання: складні питання; логічні задачі

```
SYSTEM_COT = """You are an AI assistant for a media catalog.
Answer the question based ONLY on the provided context.
First, think step by step about what the context tells you.
Then provide your final answer.
If the answer is not found in the context, say exactly:
'Insufficient information in the provided documents.'"""
```

Функція build_prompt() - Core RAG логіка (Core Retrieval-Augmented Generation) описує базовий «скелет» або стандартний процес роботи системи, яка поєднує можливості штучного інтелекту (наприклад, LLM і ChatGPT) із власними даними (документами, архівами, базами даних).

Будує текстовий prompt для LLM. Параметри:

– query - запит користувача
– chunks - список документів від retrieve_and_rerank()
– style - 'basic' | 'citation' | 'chain_of_thought'
– max_chars - максимальна кількість символів контексту

Повертає: готовий рядок *prompt* для передачі в LLM.

```
def build_prompt(query: str,
                 chunks: list[dict],
                 style: str = 'basic',
                 max_chars: int = MAX_CONTEXT) -> str:

    # Вибираємо системну інструкцію залежно від стилю
    style_map = {
        'basic'           : SYSTEM_BASIC,
        'citation'        : SYSTEM_CITATION,
        'chain_of_thought': SYSTEM_COT,
    }
    system_prompt = style_map.get(style, SYSTEM_BASIC)
    # Формуємо нумерований блок контексту
    # Обрізаємо контекст, якщо сума символів перевищує max_chars.
    context_parts = []
    total_chars   = 0

    for i, chunk in enumerate(chunks, 1):
        title = chunk['metadata'].get('title', 'Unknown')
        year  = chunk['metadata'].get('year', '?')
        text  = chunk['text'].strip()

        # Формуємо рядок з метаданими для контексту
        chunk_line = f"[{i}] Title: {title} ({year})\n{text}"

        remaining = max_chars - total_chars
        if remaining <= 0:
            break

        if len(chunk_line) > remaining:
            # Обрізаємо останній чанк якщо він не влізає повністю
            chunk_line = chunk_line[:remaining] + "..."

        context_parts.append(chunk_line)
        total_chars += len(chunk_line)

    context_str = "\n\n".join(context_parts)
    # Фінальний prompt
    prompt = f"""{system_prompt}
CONTEXT
{context_str}
QUESTION
{query}
ANSWER"""
    return prompt
```

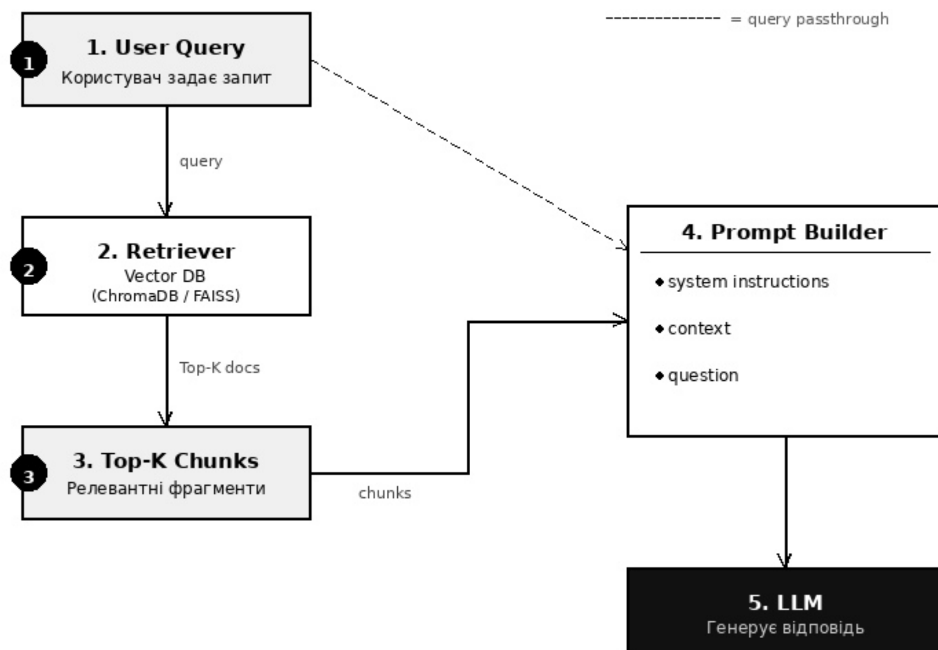
Проводимо перевірку чи правильно формується prompt, чи не перевищений розмір

```
sample_prompt = build_prompt(test_q, test_r, style='citation')
print(f"\nПриклад prompt (перші 600 символів):")
```

```
print("—" * 60)
print(sample_prompt[:600])
print("—" * 60)
print(f"Загальна довжина prompt: {len(sample_prompt)} символів")
```

Функція *build_prompt* реалізує механізм формування структурованого запиту до LLM у межах RAG-архітектури. Вона забезпечує інтеграцію *retrieved* контексту, контроль обмеження контекстного вікна та керування стилем генерації через системні інструкції.

RAG пай план:



Завдання 4. *Реалізувати функцію generate_answer(query, prompt) - виклик LLM API зі streaming або sync.*

Методичні рекомендації

Надсилає prompt до Gemini і повертає (відповідь, час генерації, мс). Виклик Gemini з *temperature=0.1*, обгорнутий у *try/except*. Повертає кортеж (відповідь, *latency_мс*).

```
def generate_answer(prompt: str, max_tokens: int = 512) -> tuple[str,
float]:
    t_start = time.perf_counter()
    try:
        response = llm.generate_content(
            prompt,
            generation_config=genai.types.GenerationConfig(
                max_output_tokens=max_tokens,
                temperature=0.1,       # майже детермінована відповідь
                top_p=0.9,
```

```
                  # модель обирає слова з топ 90% ймовірностей
            )
        )
        answer  = response.text.strip()
        latency = (time.perf_counter() - t_start) * 1000

    except Exception as e:
        answer  = f"[API ERROR]: {e}"
        latency = (time.perf_counter() - t_start) * 1000

    return answer, latency

# Тест generate_answer()
print("\n Тест LLM генерації...")
test_answer, test_latency = generate_answer(sample_prompt)
print(f"LLM latency: {test_latency:.0f},мс")
print(f"\nВідповідь LLM:")
print("—" * 60)
print(test_answer)
print("—" * 60)
```

Temperature - це параметр стохастичності генерації, який контролює ступінь випадковості вибору наступних токенів у мовній моделі. Temperature = наскільки "сміливо" модель обирає слова: низька → обирає найімовірніші слова; висока → дозволяє ризиковані / рідкісні слова.

temperature ↓ → точність ↑

temperature ↑ → креатив ↑

Завдання 5. *Реалізувати повний end-to-end найплайн: query → retrieval → reranking → prompt → LLM → answer.*

Методичні рекомендації

1. Реалізуйте функцію *rag_pipeline(query, style, min_ce_score)* як єдину точку входу, що послідовно викликає re*trieve_and_rerank()* → *build_prompt()* → *generate_answer()* та повертає структурований словник із усіма проміжними результатами.

2. Виміряйте *latency* кожного етапу окремо через *time.perf_counter()* - це дозволить виявити вузьке місце (*bottleneck*): зазвичай найдовшим є етап *LLM*, а не *retrieval*.

3. Зберігайте у результаті не лише фінальну відповідь, а й повний *prompt*, список *chunk_ids* і *scores* - це необхідно для *debugging* та аналізу у завданнях 7–8.

4. Переконайтеся, що пайплайн є ідемпотентним: однаковий запит при однакових параметрах має повертати стабільний результат. Для цього встановіть *temperature=0.1* у *LLM* і зафіксуйте *random_seed* у *cross-encoder*, якщо це підтримується.

5. Протестуйте пайплайн мінімум на 3 запитах вручну перед запуском масового тесту у завданні 6. Переконайтеся, що *scores* мають сенс, *prompt* коректно сформований, а відповідь не є порожньою.

6. Додайте логування кожного виклику: *print(f"[{idx}] BI={bi_ms:.0f}мс CE={ce_ms:.0f}мс LLM={llm_ms:.0f}мс")* - це допомагає відстежувати прогрес і виявляти зависання при API rate limits.

Завдання 6. *Виконати генерацію для 10 тестових запитів та зберегти результати у таблицю.*

Методичні рекомендації

1. Сформуйте 10 тематично різноманітних запитів: включіть конкретні (жанр, актор, рік), абстрактні (емоція, тема), та навмисно нерелевантний запит поза тематикою - це перевірить *fallback*-механізм із завдання 7.

2. Реалізуйте обхід у циклі з обробкою винятків на рівні кожного запиту (*try/except* всередині циклу), щоб одна помилка API не зупиняла весь процес.

3. Збережіть результати у *pandas.DataFrame* зі стовпцями: *Query, Fallback, Retrieved_IDs, Top_BI_Score, Top_CE_Score, Top_Title, Answer, BI_Latency_ms, CE_Latency_ms, LLM_Latency_ms, Total_Latency_ms.* Це структура, яку використовуватимете в завданні 8 для оцінки.

4. *Збережіть таблицю у два формати: df.to_csv('lab5_results.csv', encoding='utf-8-sig') - для Excel, і df.to_json('lab5_results.json', force_ascii=False) - для подальшої обробки у ЛР6.*

5. Додайте паузу між запитами *time.sleep*(1) для уникнення *API rate limit* (*Gemini Free Tier*: 15 запитів/хвилину). При помилці *429 Too Many Requests* збільшіть паузу до 5–10 секунд.

6. Після завершення виведіть зведену статистику: середній *Total_Latency_ms*, кількість *fallback*-спрацювань, розподіл *CE scores* - це дасть загальне розуміння якості *retrieval* до ручної оцінки.

Завдання 7. *Додати обробку помилок: немає контексту → відповідь 'Недостатньо інформації'.*

Методичні рекомендації

1. Реалізуйте перевірку порогу *cross-encoder score* перед викликом *LLM*: якщо *chunks[0]['ce_score'] < min_ce_score*, повертайте стандартне повідомлення без звернення до API. Порогове значення *min_ce_score = -2.0* є стартовим - підберіть його емпірично на своєму датасеті, проаналізувавши розподіл CE scores для релевантних і нерелевантних запитів.

2. Відрізняйте два типи *fallback*: *hard fallback* (*score* нижче порогу - LLM не викликається взагалі) і *soft fallback* (LLM викликається, але відповідає «Недостатньо інформації» згідно з системною інструкцією). *Hard fallback* економить API-квоту, *soft* - більш граційна поведінка при граничних випадках.

3. Протестуйте *fallback* щонайменше на 3 сценаріях: запит іншою мовою без мультимовної моделі, запит з предметної галузі поза тематикою (наприклад, математика), і навмисно безглуздий запит (набір символів).

4. Зафіксуйте *fallback=True/False* як окрему колонку у таблиці результатів - це для оцінки у завданні 8, оскільки *fallback*-відповіді не підлягають оцінці за *Faithfulness* та *Answer Relevance*.

5. Реалізуйте також обробку мережевих помилок: *google.api_core.exceptions.ResourceExhausted (rate limit), google.api_core.exceptions.ServiceUnavailable* (недоступність сервісу), *ValueError* (некоректна відповідь моделі). Для кожного типу помилки визначте окрему стратегію: *retry, fallback* або *propagate*.

Завдання 8. *Провести ручну оцінку якості відповідей (faithfulness, answer relevance) для кожного запиту.*

Методичні рекомендації

1. Оцініть кожну не-*fallback* відповідь за двома незалежними шкалами від 0 до 5: *Faithfulness* (вірність контексту): 5 = усі факти підтверджені наданими чанками, 0 = відповідь повністю вигадана або суперечить контексту. *Answer Relevance* (релевантність відповіді): 5 = відповідь повністю і точно стосується запиту, 0 = відповідь не має відношення до запиту.
2. При оцінці *Faithfulness* відкрийте поруч повний *prompt* (збережений у таблиці) і звіряйте кожне твердження відповіді з текстами чанків - тільки так можна достовірно виявити галюцинації.
3. Проведіть оцінку двічі з перервою між сесіями (або залучіть другого оцінювача) і порахуйте міжоцінювачну узгодженість через *Cohen's Kappa* або просте відсоткове співпадіння - це демонструє надійність ручної оцінки.
4. Побудуйте *scatter plot* «*CE Score vs Answer Relevance*» для виявлення кореляції між автоматичним скором *retrieval* і суб'єктивною якістю відповіді. Якщо кореляція слабка - це сигнал, що *cross-encoder score* не є достатнім предиктором якості кінцевої відповіді.
5. Обчисліть і зафіксуйте: середні значення *Faithfulness* і *Answer Relevance*, частку відповідей з *Faithfulness* < 3 (потенційні галюцинації), частку *fallback* серед усіх запитів. Ці числа стануть *baseline* для автоматичної оцінки через RAGAS у ЛР6.
6. Проаналізуйте: який стиль промпту (*basic, citation, chain_of_thought*) дав вищу середню *Faithfulness* і чому. Зазвичай *citation*-стиль підвищує *Faithfulness*, оскільки явно вимагає прив'язки до джерел, і модель рідше вигадує факти.

Контрольні запитання

1. Що таке grounded generation і як RAG зменшує галюцинації LLM?

2. Поясніть різницю між temperature=0 та temperature=1 у генерації відповідей.

3. Навіщо в prompt template є окремий system prompt? Що буде якщо його прибрати?

4. Як впливає розмір TOP_K_RERANK на якість та вартість генерації?

5. Яка проблема виникне якщо context window LLM менший за сумарний розмір чанків?

6. У чому різниця між citation-style та chain-of-thought промптом? Коли використовувати кожен?

7. Як реалізувати fallback коли жоден чанк не є релевантним для запиту?