МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"

РОЗРОБКА ТА ОЦІНКА СИСТЕМ ПОШУКОВО-ДОПОВНЕНОЇ ГЕНЕРАЦІЇ (RAG)

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторної роботи № 6
з дисципліни «Штучний інтелект в ігровому дизайні»
для студентів бакалаврського рівня вищої освіти спеціальності 121
"Інженерія програмного забезпечення"

Львів -- 2025

Розробка та оцінка систем пошуково-доповненої генерації (RAG): методичні вказівки до виконання лабораторної роботи №6 з дисципліни "Штучний інтелект в ігровому дизайні" для студентів першого (бакалаврського) рівня вищої освіти спеціальності 121 "Інженерія програмного забезпечення" . Укл.: О.Є. Бауск. -- Львів: Видавництво Національного університету "Львівська політехніка", 2025. -- 20 с.

Укладач: Бауск О.Є., к.т.н., асистент кафедри ПЗ

Відповідальний за випуск: Федасюк Д.В., доктор техн. наук, професор

Рецензенти: Федасюк Д.В., доктор техн. наук, професор

Задорожний І.М., асистент кафедри ПЗ

Тема роботи: Розробка та оцінка систем пошуково-доповненої генерації (RAG).

Мета роботи: Набути практичних навичок побудови повного конвеєра Пошуково-Доповненої Генерації (Retrieval-Augmented Generation, RAG) з нуля, що працює локально. Дослідити процес обробки документів (PDF), створення текстових ембедингів, реалізації семантичного пошуку та використання великої мовної моделі (LLM) для генерації відповідей на запити на основі знайденого контексту.

Теоретичні відомості

Пошуково-Доповнена Генерація (RAG)

Пошуково-Доповнена Генерація (Retrieval-Augmented Generation, RAG) — це підхід, що покращує результати генерації великих мовних моделей (LLM), доповнюючи їх знаннями з зовнішніх джерел. Цей підхід був представлений у статті *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*.

Процес RAG складається з трьох основних етапів:

- **Пошук (Retrieval)**: Знаходження релевантної інформації з джерела даних (наприклад, бази документів, бази знань) на основі вхідного запиту користувача.
- **Доповнення (Augmented)**: Модифікація вхідного запиту до генеративної моделі (LLM) шляхом включення знайденої релевантної інформації як контексту.
- **Генерація (Generation)**: Генерація текстової відповіді LLM на основі доповненого запиту, що включає як оригінальний запит, так і знайдений контекст.

Переваги RAG

Основна мета RAG — покращити якість та достовірність відповідей LLM. Ключові переваги:

1. **Запобігання галюцинаціям**: LLM схильні генерувати правдоподібну, але невірну інформацію ("галюцинації"). RAG допомагає LLM генерувати більш фактично обґрунтовані відповіді, надаючи їм релевантний контекст з надійних джерел. Крім того, RAG дозволяє відстежувати джерела інформації, що підвищує прозорість та можливість перевірки.
2. **Робота зі специфічними даними**: Базові LLM навчаються на загальних даних з Інтернету і можуть не мати знань у вузьких областях або доступу до приватної інформації (наприклад, внутрішньої документації компанії). RAG дозволяє "підключати" до LLM специфічні набори даних без необхідності повного перенавчання чи доналаштування моделі.

RAG може бути значно швидшим та менш ресурсовитратним рішенням порівняно з доналаштуванням (fine-tuning) LLM на специфічних даних.

Сфери застосування RAG

RAG може бути корисним у будь-якій ситуації, де потрібно генерувати відповіді на основі конкретного набору інформації, якого може не бути у тренувальних даних LLM. Приклади:

- **Системи питань та відповідей для підтримки клієнтів**: Створення чат-ботів, які відповідають на запити клієнтів на основі документації продукту чи бази знань.
- **Аналіз електронної пошти або документів**: Вилучення структурованої інформації або відповіді на питання щодо великих обсягів тексту (наприклад, аналіз ланцюжків листів у страхових справах).

- **Системи доступу до внутрішньої документації компанії**: Допомога співробітникам у пошуку інформації та отриманні відповідей на основі внутрішніх політик, інструкцій тощо.
- **Навчальні системи**: Створення інтерактивних помічників для роботи з підручниками чи навчальними матеріалами.

Ключові терміни RAG

Термін	Опис
Токен	Підсловова одиниця тексту. Текст розбивається на токени перед подачею в LLM. Приблизно 1 токен ≈ 4 символи англійською мовою.
Ембединг (Embedding)	Навчене числове представлення фрагмента даних (наприклад, тексту). Схожі за змістом фрагменти тексту мають схожі вектори ембедингів.
Модель ембедингів	Модель, призначена для перетворення вхідних даних (наприклад, тексту) на числове представлення (вектор ембедингу). Часто відрізняється від моделі LLM.
Пошук за подібністю	Знаходження векторів, які є близькими один до одного у багатовимірному просторі. Використовується для пошуку релевантних фрагментів тексту на основі запиту. Поширені метрики: косинусна подібність, скалярний добуток.
Велика Мовна Модель (LLM)	Модель, навчена на великих обсягах тексту для розуміння та генерації людської мови. Генерує продовження тексту на основі вхідного запиту (промпту).
Контекстне вікно LLM	Максимальна кількість токенів, яку LLM може прийняти як вхід. Впливає на те, скільки контекстної інформації можна передати моделі в RAG-системі.
Промпт (Prompt)	Вхідний текст, що подається до генеративної LLM для отримання відповіді. **Інженерія промптів** — це мистецтво структурування промптів для отримання бажаного результату.
Чанкінг (Chunking)	Процес розбиття великого тексту на менші, керовані частини (чанки) перед створенням ембедингів. Це допомагає врахувати обмеження моделей ембедингів та контекстного вікна LLM.
Квантизація (Quantization)	Процес зменшення точності числових представлень (ваг) моделі для зменшення її розміру та вимог до пам'яті, часто з невеликою втратою продуктивності. Використовується для запуску великих моделей на обмежених ресурсах.

Хід роботи

УВАГА! В рамках даних методичних вказівок не розглядається використання локального інстанса Jupyter Notebook та використання локального GPU. Задача запустити RAG на локальному комп'ютері має багато нюансів і не може бути достатньо стандартизованою. Виконання даної работи не в Google Colab, а на локальному комп'ютері з демонстрацією при здачі і в звіті як це було досягнуто -- це підстава отримати максимальні бали по всім лабораторним роботам, незалежно від результатів інших лабораторних робіт. Виконання цієї вправи лишається на самостійне дослідження студента.

У цій роботі ми побудуємо RAG-конвеєр для взаємодії з PDF-документом за допомогою локально запущеної LLM.

Конвеєр складатиметься з двох основних етапів:

1. **Обробка документа та створення ембедингів**: Завантаження PDF, поділ тексту на частини (чанкінг), створення числових представлень (ембедингів) для кожної частини та їх збереження.
2. **Пошук та відповідь**: Прийняття запиту користувача, пошук релевантних частин тексту за допомогою ембедингів (семантичний пошук), доповнення запиту знайденим контекстом та генерація відповіді за допомогою LLM.

1. Обробка документа та створення ембедингів

1.1. Встановлення необхідних бібліотек та підготовка середовища

Переконайтеся, що у вас встановлені всі необхідні бібліотеки. Якщо ви працюєте в Google Colab, виконайте наступні команди в новому блокноті.

УВАГА! Занотуйте ключові результати роботи в Google Colab в вашому звіті.

```
# Виконати, якщо працюєте в Google Colab
import os
if "COLAB_GPU" in os.environ:
    print("[INFO] Running in Google Colab, installing requirements.")
    !pip install -U torch # Потрібен torch 2.1.1+
    !pip install PyMuPDF # Для читання PDF
    !pip install tqdm # Для індикаторів прогресу
    !pip install sentence-transformers # Для моделей ембедингів
    !pip install accelerate # Для завантаження квантизованих моделей
    !pip install bitsandbytes # Для квантизації моделей
    !pip install flash-attn --no-build-isolation # Для прискореного механізму уваги (якщо підтр
else:
    print("[INFO] Not running in Google Colab, skipping installs.")
    # Переконайтеся, що бібліотеки встановлені у вашому локальному середовищі:
    # pip install -U torch pymupdf tqdm sentence-transformers accelerate bitsandbytes flash-att
```

1.2. Завантаження та читання PDF-документа

В якості прикладу будемо використовувати відкритий підручник "Human Nutrition: 2020 Edition". Спочатку завантажимо його.

```
import os
import requests

# Шлях до PDF файлу
pdf_path = "ragfile.pdf"

# Завантаження PDF, якщо він ще не існує
if not os.path.exists(pdf_path):
  print("File doesn't exist, downloading...")
  # URL для завантаження PDF
  url = "https://pressbooks.oer.hawaii.edu/humannutrition2/open/download?type=pdf"
```

```
  filename = pdf_path
  response = requests.get(url)

  if response.status_code == 200:
      with open(filename, "wb") as file:
          file.write(response.content)
      print(f"The file has been downloaded and saved as {filename}")
  else:
      print(f"Failed to download the file. Status code: {response.status_code}")
else:
  print(f"File {pdf_path} exists.")
```

Тепер прочитаємо текст з PDF за допомогою бібліотеки `PyMuPDF` (`fitz`). Ми створимо список словників, де кожен словник представляє сторінку документа.

** Увага!** Якщо ваш PDF-документ має іншу нумерацію сторінок, наприклад починається з сторінки 30, вам треба буде змінити значення `page_number` на `page_number - 30` внизу в коді.

```
import fitz # PyMuPDF
from tqdm.auto import tqdm # Індикатори прогресу

def text_formatter(text: str) -> str:
    """Виконує базове форматування тексту."""
    cleaned_text = text.replace("\n", " ").strip()
    return cleaned_text

def open_and_read_pdf(pdf_path: str) -> list[dict]:
    """Відкриває PDF, читає текст по сторінках та збирає статистику."""
    doc = fitz.open(pdf_path)
    pages_and_texts = []
    for page_number, page in tqdm(enumerate(doc), total=len(doc)):
        text = page.get_text() # Отримати текст сторінки
        text = text_formatter(text)
        pages_and_texts.append({
            "page_number": page_number - 1 # Коригування номера сторінки (на випадок, якщо PDF
            "page_char_count": len(text),
            "page_word_count": len(text.split(" ")),
            "page_sentence_count_raw": len(text.split(". ")), # Приблизна кількість речень
            "page_token_count": len(text) / 4, # Приблизна кількість токенів (1 токен ~ 4 симво
            "text": text
        })
    doc.close()
    return pages_and_texts

pages_and_texts = open_and_read_pdf(pdf_path=pdf_path)

# Перегляд перших двох сторінок
print(pages_and_texts[:2])

# Перегляд статистики за сторінками
import pandas as pd
df = pd.DataFrame(pages_and_texts)
print(df.head())
print(df.describe().round(2))
```

Продивіться і занотуйте статистику. Ви маєте побачити, що середня кількість токенів на сторінку становить число в районі 287.

Для цього конкретного випадку це означає, що ми можемо вбудувати середню цілу сторінку за допомогою моделі all-mpnet-base-v2 (ця модель має вхідну ємність 384).

Додайте в звіті статистику вашого PDF-документа і пояснення, чому ми будемо використовувати дану модель.

1.3. Поділ тексту на речення

Моделі ембедингів часто мають обмеження на довжину вхідного тексту (наприклад, 384 токени для `all-mpnet-base-v2`). Тому доцільно розбити текст на менші одиниці, наприклад, речення. Ми використаємо бібліотеку `spaCy` для більш надійного поділу на речення.

```
from spacy.lang.en import English

nlp = English()
# Додаємо компонент для поділу на речення
nlp.add_pipe("sentencizer")

# Створюємо документ як приклад
doc = nlp("This is a sentence. This another sentence.")
assert len(list(doc.sents)) == 2

# Отримуємо речення документа
list(doc.sents)
```

Ми не обов'язково повинні використовувати spaCy, однак це відкрита бібліотека, яка призначена для виконання NLP-завдань на масштабі.

Отже, давайте запустимо наш пайплайн поділу на речення на наших сторінках тексту.

```
for item in tqdm(pages_and_texts):
    item["sentences"] = list(nlp(item["text"]).sents)
    # Перетворюємо речення на рядки
    item["sentences"] = [str(sentence) for sentence in item["sentences"]]
    # Рахуємо кількість речень за допомогою spaCy
    item["page_sentence_count_spacy"] = len(item["sentences"])

# Оновлюємо DataFrame та переглядаємо статистику
df = pd.DataFrame(pages_and_texts)
print(df.describe().round(2))
```

1.4. Поділ на частини (Chunking)

Тепер згрупуємо речення у більші "чанки". Це робиться для того, щоб:

- Керувати фрагментами тексту однакового розміру.
- Не перевищувати ліміт токенів моделі ембедингів.
- Ефективно використовувати контекстне вікно LLM.

Що слід зазначити, так це те, що існує багато різних способів для створення чанків інформації/тексту.

Наразі ми будемо просто групувати речення у групи по 10 (це число є довільним і може бути змінено, просто добре підходить для нашої моделі ембедингів ємністю 384).

Середньо на сторінці маємо 10 речень.

Середньо на сторінці маємо 287 токенів.

Таким чином, наші групи по 10 речень будуть також мати близько 287 токенів.

Це дає нам достатньо місця для вбудування тексту нашою моделлю all-mpnet-base-v2 (вона має ємність 384 токенів).

Щоб розділити наші групи речень на частини по 10 або менше, створимо функцію, яка приймає список як вхід і рекурсивно ділить його на підсписки заданого розміру.

```
# Розмір чанку (кількість речень)
num_sentence_chunk_size = 10

def split_list(input_list: list, slice_size: int) -> list[list[str]]:
    """Рекурсивно ділить список на підсписки заданого розміру."""
    return [input_list[i:i + slice_size] for i in range(0, len(input_list), slice_size)]

# Ділимо речення на чанки для кожної сторінки
for item in tqdm(pages_and_texts):
    item["sentence_chunks"] = split_list(input_list=item["sentences"],
                                         slice_size=num_sentence_chunk_size)
    item["num_chunks"] = len(item["sentence_chunks"])

# Переглядаємо приклад
import random
print(random.sample(pages_and_texts, k=1))

# Статистика після чанкінгу
df = pd.DataFrame(pages_and_texts)
print(df.describe().round(2))
```

Зверніть увагу, що середня кількість чанків становить приблизно 1.5, що є очікуваним, оскільки багато наших сторінок містять лише в середньому 10 речень.

Ми хочемо вбудувати кожен чанк речень у власний числовий представлення.

Щоб все було чистим, створимо новий список словників, кожен з яких містить один чанк речень з відносною інформацією, такою як номер сторінки, а також статистику кожного чанку.

```
import re

# Створюємо список, де кожен елемент - окремий чанк
pages_and_chunks = []
for item in tqdm(pages_and_texts):
    for sentence_chunk in item["sentence_chunks"]:
        chunk_dict = {}
        chunk_dict["page_number"] = item["page_number"]
```

```
        # Об'єднуємо речення чанку в один рядок
        joined_sentence_chunk = "".join(sentence_chunk).replace("  ", " ").strip()
        # Додаємо пробіл після крапки перед великою літерою (для кращої читабельності)
        joined_sentence_chunk = re.sub(r'\.([A-Z])', r'. \1', joined_sentence_chunk)
        chunk_dict["sentence_chunk"] = joined_sentence_chunk

        # Статистика для чанку
        chunk_dict["chunk_char_count"] = len(joined_sentence_chunk)
        chunk_dict["chunk_word_count"] = len(joined_sentence_chunk.split(" "))
        chunk_dict["chunk_token_count"] = len(joined_sentence_chunk) / 4 # Приблизна кількість

        pages_and_chunks.append(chunk_dict)

print(f"Загальна кількість чанків: {len(pages_and_chunks)}")

# Перегляд випадкового чанку
print(random.sample(pages_and_chunks, k=1))

# Статистика за чанками
df = pd.DataFrame(pages_and_chunks)
print(df.describe().round(2))
```

Ми розбили весь наш підручник на чанки по 10 речень або менше, а також номер сторінки, з якої вони прийшли.

Це означає, що ми можемо звернутися до частини тексту і знати його джерело.

Відфільтруємо дуже короткі чанки (менше 30 токенів), оскільки вони часто є заголовками/колонтитулами і несуть мало інформації.

```
min_token_length = 30
pages_and_chunks_over_min_token_len = df[df["chunk_token_count"] > min_token_length].to_dict(or
print(f"Кількість чанків після фільтрації: {len(pages_and_chunks_over_min_token_len)}")
```

1.5. Створення ембедингів

Перетворимо кожен текстовий чанк на вектор чисел (ембединг) за допомогою моделі ембедингів. Ми будемо використовувати модель `all-mpnet-base-v2` з бібліотеки `sentence-transformers` (це бібліотека моделей на сервісі Hugging Face).

```
from sentence_transformers import SentenceTransformer
import torch

# Визначаємо пристрій (GPU, якщо доступний)
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using device: {device}")

# Завантажуємо модель ембедингів
embedding_model = SentenceTransformer(model_name_or_path="all-mpnet-base-v2", device=device)

# Приклад створення ембедингу для одного речення
single_sentence = "Це приклад речення для ембедингу."
single_embedding = embedding_model.encode(single_sentence)
```

```
print(f"Речення: {single_sentence}")
print(f"Розмірність ембедингу: {single_embedding.shape}")
# print(f"Ембединг (перші 10 значень): {single_embedding[:10]}") # Розкоментуйте, щоб побачити

# Створюємо ембединги для всіх чанків (використовуємо батчінг для швидкості)
# Перетворюємо чанки на список рядків
text_chunks = [item["sentence_chunk"] for item in pages_and_chunks_over_min_token_len]

# Створюємо ембединги батчами
text_chunk_embeddings = embedding_model.encode(
    text_chunks,
    batch_size=32, # Розмір батчу (можна експериментувати)
    convert_to_tensor=True, # Повертати як torch.Tensor
    show_progress_bar=True # Показувати індикатор прогресу
)

print(f"Розмірність тензора ембедингів: {text_chunk_embeddings.shape}")

# Додаємо ембединги до наших даних
for i, item in enumerate(pages_and_chunks_over_min_token_len):
    item["embedding"] = text_chunk_embeddings[i].cpu().numpy() # Зберігаємо як numpy array на C
```

1.6. Збереження ембедингів

Збережемо оброблені чанки разом з їх ембедингами у файл CSV для подальшого використання.

```
# Зберігаємо дані у CSV
text_chunks_and_embeddings_df = pd.DataFrame(pages_and_chunks_over_min_token_len)
embeddings_df_save_path = "text_chunks_and_embeddings_df.csv"
text_chunks_and_embeddings_df.to_csv(embeddings_df_save_path, index=False)

# Перевірка завантаження
text_chunks_and_embedding_df_load = pd.read_csv(embeddings_df_save_path)
print(text_chunks_and_embedding_df_load.head())
```

2. RAG - Пошук та Відповідь

2.1. Семантичний пошук

На цьому етапі ми будемо використовувати створені ембединги для пошуку частин тексту, найбільш релевантних до запиту користувача. Це називається семантичним (або векторним) пошуком.

Спочатку завантажимо збережені дані та підготуємо ембединги як тензор PyTorch.

```
import numpy as np

# Перевірка наявності файлу
if not os.path.exists(embeddings_df_save_path):
    print(f"Error: File not found at {embeddings_df_save_path}. Please run the embedding creati
else:
    # Завантажуємо дані
    text_chunks_and_embedding_df = pd.read_csv(embeddings_df_save_path)
```

```
    # Конвертуємо стовпець ембедингів назад у numpy array
    text_chunks_and_embedding_df["embedding"] = text_chunks_and_embedding_df["embedding"].apply
    # Конвертуємо DataFrame у список словників
    pages_and_chunks = text_chunks_and_embedding_df.to_dict(orient="records")
    # Конвертуємо ембединги у тензор PyTorch та переміщуємо на GPU (якщо доступний)
    embeddings = torch.tensor(np.array(text_chunks_and_embedding_df["embedding"].tolist()), dty
    print(f"Завантажено {len(pages_and_chunks)} чанків.")
    print(f"Розмір тензора ембедингів: {embeddings.shape}")
```

Тепер виконаємо пошук. Алгоритм:

1. Визначити запит.
2. Створити ембединг для запиту, використовуючи ту ж модель, що й для чанків.
3. Обчислити подібність (скалярний добуток або косинусну подібність) між ембедингом запиту та всіма ембедингами чанків.
4. Відсортувати результати за спаданням подібності та вибрати топ-k найбільш релевантних чанків.

Ми будемо використовувати скалярний добуток, оскільки модель `all-mpnet-base-v2` видає нормалізовані вектори, і скалярний добуток в цьому випадку еквівалентний косинусній подібності, але обчислюється швидше.

```
from sentence_transformers import util
from time import perf_counter as timer
import textwrap

def print_wrapped(text, wrap_length=80):
    """Друкує текст з перенесенням рядків."""
    wrapped_text = textwrap.fill(text, wrap_length)
    print(wrapped_text)

def retrieve_relevant_resources(query: str,
                                embeddings: torch.tensor,
                                model: SentenceTransformer=embedding_model,
                                n_resources_to_return: int=5,
                                print_time: bool=True,
                                device: str = device):
    """Створює ембединг запиту та повертає топ-k оцінок подібності та індексів."""
    # Створюємо ембединг запиту
    query_embedding = model.encode(query, convert_to_tensor=True).to(device)

    # Обчислюємо скалярний добуток
    start_time = timer()
    dot_scores = util.dot_score(query_embedding, embeddings)[0]
    end_time = timer()

    if print_time:
        print(f"[INFO] Час обчислення подібності для {len(embeddings)} ембедингів: {end_time-st

    scores, indices = torch.topk(input=dot_scores, k=n_resources_to_return)
    return scores, indices

def print_top_results_and_scores(query: str,
                                 embeddings: torch.tensor,
                                 pages_and_chunks: list[dict]=pages_and_chunks,
```

```
                                    n_resources_to_return: int=5):
    """Знаходить релевантні ресурси та друкує їх."""
    scores, indices = retrieve_relevant_resources(query=query,
                                                  embeddings=embeddings,
                                                  n_resources_to_return=n_resources_to_return)

    print(f"Запит: {query}\n")
    print("Результати:")
    for score, index in zip(scores, indices):
        print(f"Оцінка подібності: {score:.4f}")
        print("Текст:")
        print_wrapped(pages_and_chunks[index]["sentence_chunk"])
        print(f"Номер сторінки: {pages_and_chunks[index]['page_number']}")
        print("\n")

# Приклад запиту
query = "macronutrients functions"
print_top_results_and_scores(query=query, embeddings=embeddings)

# Інший приклад
query = "symptoms of pellagra"
print_top_results_and_scores(query=query, embeddings=embeddings, n_resources_to_return=3)
```

2.2. Завантаження LLM для генерації

Тепер нам потрібна велика мовна модель (LLM) для генерації відповідей. Ми будемо використовувати модель Gemma від Google, завантажену локально за допомогою бібліотеки `transformers`. Вибір конкретної версії (2B чи 7B) та точності (float16 чи 4-bit) залежить від доступної відеопам'яті (VRAM) вашого GPU.

Спочатку перевіримо доступну пам'ять GPU.

```
# Перевірка доступної пам'яті GPU
try:
    gpu_memory_bytes = torch.cuda.get_device_properties(0).total_memory
    gpu_memory_gb = round(gpu_memory_bytes / (2**30))
    print(f"Доступно відеопам'яті: {gpu_memory_gb} GB")
except Exception as e:
    print(f"Не вдалося отримати інформацію про GPU: {e}")
    gpu_memory_gb = 0 # Припускаємо, що GPU немає або недоступний

# Вибір моделі та конфігурації квантизації на основі пам'яті
# (Зверніть увагу: ці пороги є орієнтовними)
use_quantization_config = False
model_id = "google/gemma-2b-it" # Модель за замовчуванням

if gpu_memory_gb == 0:
     print("GPU не знайдено або недостатньо пам'яті. Робота LLM буде дуже повільною на CPU.")
     # Можливо, варто зупинити виконання або використати меншу модель/API
elif gpu_memory_gb < 5.1:
    print(f"Пам'ять GPU ({gpu_memory_gb}GB) може бути недостатньою для Gemma без квантизації.")
    print("Спробуємо Gemma 2B з 4-бітною квантизацією.")
    use_quantization_config = True
    model_id = "google/gemma-2b-it"
elif gpu_memory_gb < 8.1:
```

```
    print(f"Пам'ять GPU: {gpu_memory_gb}GB | Рекомендована модель: Gemma 2B (4-bit)")
    use_quantization_config = True
    model_id = "google/gemma-2b-it"
elif gpu_memory_gb < 19.0:
    print(f"Пам'ять GPU: {gpu_memory_gb}GB | Рекомендована модель: Gemma 2B (float16) або Gemma
    # Вибираємо Gemma 7B 4-bit для кращої якості, якщо пам'ять дозволяє
    if gpu_memory_gb >= 8.1: # Достатньо для 7B 4-bit
         print("Вибираємо Gemma 7B (4-bit).")
         use_quantization_config = True
         model_id = "google/gemma-7b-it"
    else:
         print("Вибираємо Gemma 2B (float16).")
         use_quantization_config = False
         model_id = "google/gemma-2b-it"
elif gpu_memory_gb >= 19.0:
    print(f"Пам'ять GPU: {gpu_memory_gb}GB | Рекомендована модель: Gemma 7B (4-bit або float16)
    print("Вибираємо Gemma 7B (float16).") # Найвища якість
    use_quantization_config = False
    model_id = "google/gemma-7b-it"

print(f"Обрано model_id: {model_id}")
print(f"Використовувати квантизацію (4-bit): {use_quantization_config}")
```

Тепер завантажимо токенізатор та саму модель. Для моделей Gemma потрібна автентифікація в Hugging Face Hub.

УВАГА! Після ПЕРШОГО запуску цього коду, вам потрібно буде ввести свій токен доступу до Hugging Face Hub.

Зайдіть на цей сайт і створіть свій токен з рівнем доступу "Read".

(Створіть акаунт Hugging Face, якщо його у вас ще немає.)

Вставте свій токен у поле `login()` залогінтесь на Hugging Face і запустіть цей блок коду ще раз.

Занотуйте результати логіну і виконання цього блоку коду.

```
from huggingface_hub import login
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig
from transformers.utils import is_flash_attn_2_available

# Автентифікація (якщо ще не виконано) - може знадобитися токен доступу
try:
    login()
except Exception as e:
    print(f"Помилка автентифікації Hugging Face Hub: {e}")
    print("Переконайтеся, що ви увійшли через `huggingface-cli login` або налаштували токен.")

# 1. Створення конфігурації квантизації (якщо потрібно)
quantization_config = None
if use_quantization_config:
    quantization_config = BitsAndBytesConfig(load_in_4bit=True,
                                             bnb_4bit_compute_dtype=torch.float16)
    print("[INFO] Створено конфігурацію 4-бітної квантизації.")
```

```
# Перевірка та налаштування Flash Attention 2 (опціонально, для прискорення)
attn_implementation = "sdpa" # Scaled Dot Product Attention - стандартний варіант
if device == "cuda" and (is_flash_attn_2_available()) and (torch.cuda.get_device_capability(0)[
  print("[INFO] Flash Attention 2 доступний, вмикаємо.")
  attn_implementation = "flash_attention_2"
else:
  print(f"[INFO] Flash Attention 2 недоступний або не підтримується GPU. Використовується: {att

# 2. Завантаження токенізатора
print(f"[INFO] Завантаження токенізатора для {model_id}...")
tokenizer = AutoTokenizer.from_pretrained(pretrained_model_name_or_path=model_id)
print("[INFO] Токенізатор завантажено.")

# 3. Завантаження моделі LLM
print(f"[INFO] Завантаження моделі {model_id} (це може зайняти час)...")
llm_model = AutoModelForCausalLM.from_pretrained(
    pretrained_model_name_or_path=model_id,
    torch_dtype=torch.float16, # Використовуємо float16 для економії пам'яті
    quantization_config=quantization_config, # Застосовуємо квантизацію, якщо налаштовано
    low_cpu_mem_usage=False, # Використовуємо повну пам'ять CPU при завантаженні
    attn_implementation=attn_implementation, # Використовуємо обраний механізм уваги
    device_map="auto" # Автоматично розміщуємо модель на доступних пристроях (GPU/CPU)
)
print("[INFO] Модель LLM завантажено.")

# Якщо не використовували квантизацію, переконуємося, що модель на GPU (якщо він є)
# if device == "cuda" and not use_quantization_config:
#     llm_model.to(device)
#     print("[INFO] Модель переміщено на GPU.") # device_map="auto" має це зробити автоматично

# Функції для отримання інформації про модель (опціонально)
def get_model_num_params(model: torch.nn.Module):
    return sum(param.numel() for param in model.parameters())

def get_model_mem_size(model: torch.nn.Module):
    mem_params = sum(param.nelement() * param.element_size() for param in model.parameters())
    mem_buffers = sum(buf.nelement() * buf.element_size() for buf in model.buffers())
    model_mem_bytes = mem_params + mem_buffers
    model_mem_gb = model_mem_bytes / (1024**3)
    return {"model_mem_gb": round(model_mem_gb, 2)}

print(f"[INFO] Кількість параметрів моделі: {get_model_num_params(llm_model)}")
print(f"[INFO] Розмір моделі в пам'яті: {get_model_mem_size(llm_model)['model_mem_gb']} GB")
```

2.3. Генерація тексту з LLM (без RAG)

Спробуємо згенерувати відповідь на запит без використання знайденого контексту. Моделі Gemma, налаштовані на інструкції (`-it`), очікують спеціального форматування промпту, яке можна застосувати за допомогою `tokenizer.apply_chat_template()`.

```
input_text = "What are the macronutrients, and what roles do they play in the human body?"
print(f"Вхідний текст:\n{input_text}")

# Створюємо шаблон діалогу для моделі
```

```
dialogue_template = [
    {"role": "user", "content": input_text}
]

# Застосовуємо шаблон чату
prompt = tokenizer.apply_chat_template(
    conversation=dialogue_template,
    tokenize=False, # Не токенізувати зараз
    add_generation_prompt=True # Додати спеціальні токени для генерації
)
print(f"\nПромпт (форматований):\n{prompt}")

# Токенізуємо форматований промпт
input_ids = tokenizer(prompt, return_tensors="pt").to(device) # Переміщуємо на пристрій моделі

# Генеруємо відповідь
print("\n[INFO] Генерація відповіді LLM (без RAG)...")
outputs = llm_model.generate(
    **input_ids,
    max_new_tokens=256 # Максимальна кількість нових токенів
)
print("[INFO] Генерація завершена.")

# Декодуємо токени у текст
output_text = tokenizer.decode(outputs[0])
print(f"\nВідповідь моделі (декодована):\n{output_text}")

# Очищуємо відповідь від промпту та спеціальних токенів
cleaned_output = output_text.replace(prompt, "").replace("<bos>", "").replace("<eos>", "").stri
print(f"\nВідповідь моделі (очищена):\n{cleaned_output}")
```

2.4. Формування промпту з контекстом (Augmentation)

Тепер створимо функцію, яка буде формувати промпт для LLM, включаючи в нього контекст, знайдений на етапі пошуку. Ми додамо знайдені чанки до основного запиту, щоб LLM могла використовувати їх для генерації відповіді.

```
def prompt_formatter(query: str, context_items: list[dict]) -> str:
    """Формує промпт для LLM, доповнюючи запит контекстом."""
    # Об'єднуємо контекстні елементи в один рядок
    context = "- " + "\n- ".join([item["sentence_chunk"] for item in context_items])

    # Базовий шаблон промпту з інструкціями та прикладами
    base_prompt = """На основі наведених нижче контекстних елементів, будь ласка, дайте відпові
Спочатку виділіть релевантні уривки з контексту, а потім сформулюйте відповідь.
Повертайте лише остаточну відповідь, без етапу виділення уривків.
Намагайтеся, щоб відповіді були максимально пояснювальними.
Використовуйте наступні приклади як зразок ідеального стилю відповіді.

Приклад 1:
Запит: Які є жиророзчинні вітаміни?
Відповідь: Жиророзчинні вітаміни включають Вітамін A, Вітамін D, Вітамін E та Вітамін K. Ці віт

Приклад 2:
Запит: Які причини діабету 2 типу?
```

```
Відповідь: Діабет 2 типу часто пов'язаний з надмірним харчуванням, зокрема надмірним споживання

Приклад 3:
Запит: Яке значення гідратації для фізичної працездатності?
Відповідь: Гідратація є надзвичайно важливою для фізичної працездатності, оскільки вода відігра

Тепер використайте наступні контекстні елементи, щоб відповісти на запит користувача:
{context}

Запит користувача: {query}
Відповідь:""" # LLM почне генерувати звідси

    # Вставляємо контекст та запит у базовий шаблон
    base_prompt = base_prompt.format(context=context, query=query)

    # Створюємо шаблон діалогу
    dialogue_template = [
        {"role": "user", "content": base_prompt}
    ]

    # Застосовуємо шаблон чату Gemma
    prompt = tokenizer.apply_chat_template(
        conversation=dialogue_template,
        tokenize=False,
        add_generation_prompt=True
    )
    return prompt

# Тестуємо форматувальник промпту
test_query = "What is the RDI for protein per day?"
test_scores, test_indices = retrieve_relevant_resources(query=test_query, embeddings=embeddings
test_context_items = [pages_and_chunks[i] for i in test_indices]
test_prompt = prompt_formatter(query=test_query, context_items=test_context_items)
print(f"Тестовий RAG промпт:\n{test_prompt}")
```

2.5. Генерація відповіді з контекстом (RAG)

Тепер об'єднаємо всі кроки: пошук релевантних ресурсів, форматування промпту з контекстом та генерація відповіді за допомогою LLM.

```
def ask(query: str,
        temperature: float = 0.7,
        max_new_tokens: int = 256,
        format_answer_text: bool = True,
        return_answer_only: bool = True) -> str | tuple[str, list[dict]]:
    """
    Виконує повний RAG цикл: пошук, доповнення, генерація.
    """
    # 1. Пошук (Retrieval)
    scores, indices = retrieve_relevant_resources(
        query=query,
        embeddings=embeddings,
        n_resources_to_return=5 # Кількість контекстних елементів
    )
```

```
    # Створюємо список контекстних елементів
    context_items = [pages_and_chunks[i] for i in indices]
    # Додаємо оцінки подібності до контекстних елементів
    for i, item in enumerate(context_items):
        item["score"] = scores[i].cpu().item() # Переміщуємо на CPU

    # 2. Доповнення (Augmentation)
    prompt = prompt_formatter(
        query=query,
        context_items=context_items
    )

    # Токенізуємо промпт
    input_ids = tokenizer(prompt, return_tensors="pt").to(device)

    # 3. Генерація (Generation)
    outputs = llm_model.generate(
        **input_ids,
        temperature=temperature, # Контролює випадковість генерації
        do_sample=True, # Використовувати семплінг
        max_new_tokens=max_new_tokens # Максимальна довжина відповіді
    )

    # Декодуємо відповідь
    output_text = tokenizer.decode(outputs[0], skip_special_tokens=False) # Не пропускати спец.

    # Форматуємо/очищуємо відповідь (опціонально)
    if format_answer_text:
        # Видаляємо промпт, <bos>, <eos> та можливі стандартні фрази моделі
        output_text = output_text.replace(prompt, "").replace("<bos>", "").replace("<eos>", "")
        # Потенційно можна додати ще очищення тут, якщо модель додає щось зайве
        output_text = output_text.strip()


    # Повертаємо результат
    if return_answer_only:
        return output_text
    else:
        return output_text, context_items

# Приклад використання функції ask
final_query = "What are the health benefits of dietary fiber?"
print(f"Запит: {final_query}")

# Запускаємо RAG конвеєр
rag_answer, rag_context = ask(query=final_query, temperature=0.7, max_new_tokens=512, return_ar

print("\nВідповідь RAG:")
print_wrapped(rag_answer)

print("\nВикористаний контекст (перші 2 елементи):")
for item in rag_context[:2]:
    print(f"Оцінка подібності: {item['score']:.4f}")
    print("Текст:")
    print_wrapped(item['sentence_chunk'])
    print(f"Номер сторінки: {item['page_number']}")
    print("-" * 10)
```

```
# Ще один приклад
final_query_2 = "How does saliva help with digestion?"
print(f"\nЗапит: {final_query_2}")
rag_answer_2 = ask(query=final_query_2, temperature=0.2, max_new_tokens=256, return_answer_only
print("\nВідповідь RAG:")
print_wrapped(rag_answer_2)
```

2.6. Індивідуальне завдання

Придумайте свій запит на тему харчування, травлення, або нутріціології, виконайте цей запит як у коді нижче.

Повторіть запит з іншим параметром `temperature` (наприклад, 0.5, 1.0, 1.5). Порівняйте результати.

Запишіть і проаналізуйте результати. Занотуйте запит і результати в звіт.

Придумайте запит, який не має відповіді у документі по нутріціології. Зробіть також цей запит і запишіть результати в звіт.

```
final_query_3 = "" # TODO: Вставте свій запит
print(f"\nЗапит: {final_query_3}")
rag_answer_3 = ask(query=final_query_3, temperature=0.2, max_new_tokens=256, return_answer_only
print("\nВідповідь RAG:")
print_wrapped(rag_answer_3)
```

УМОВА ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ

1. Відтворити RAG-конвеєр, описаний у розділі "Хід роботи", використовуючи PDF-файл підручника з нутріціології, що є у вільному доступі.
2. Протестувати роботу конвеєра, поставивши різні запитів на тему харчування (можна використати приклади з коду або придумати власні). Проаналізувати якість відповідей та релевантність знайденого контексту.
3. Провести експерименти з параметрами RAG-конвеєра, використовуючи один з документів (оригінальний або власний):
   - Змінити кількість ресурсів, що повертаються для контексту (`n_resources_to_return` у функції `retrieve_relevant_resources` або `ask` , наприклад, 3, 7, 10). Проаналізувати вплив на повноту та точність відповіді для 2-3 запитів.
   - Змінити параметр `temperature` у функції `ask` (наприклад, 0.2, 1.0). Описати, як змінюється стиль та детермінованість відповіді.

ІНДІВІДУАЛЬНІ ВАРІАНТИ ЗАВДАННЯ

Для індивідуальних варіантів завдання виконайте вимоги останнього завдання в розділі "Хід роботи".

ЗМІСТ ЗВІТУ

1. Тема та мета роботи.

2. Теоретичні відомості (коротко про RAG, його компоненти та переваги).
3. Постановка завдання (загальне та індивідуальне, якщо є).
4. Хід виконання роботи:
    - Опис середовища виконання (локально/Colab, параметри GPU, якщо використовувався).
    - Ключові етапи реалізації RAG-конвеєра (обробка тексту, створення ембедингів, завантаження LLM, семантичний пошук, формування промпту, генерація). Можна навести фрагменти коду для ілюстрації.
    - Результати тестування конвеєра на різних запитах (приклади запитів, отриманих відповідей та релевантності контексту).
    - Результати експериментів зі зміною параметрів (температура запиту).
    - Детальний опис та результати виконання індивідуального завдання.
5. Аналіз результатів:
    - Оцінка загальної ефективності створеного RAG-конвеєра.
    - Аналіз результатів індивідуального завдання.
    - Обговорення можливих проблем та обмежень підходу (наприклад, якість ембедингів, обмеження LLM, складність обробки різних форматів документів).
6. Висновки.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке Пошуково-Доповнена Генерація (RAG)? Назвіть три основні етапи.
2. Які основні переваги використання RAG порівняно зі звичайним використанням LLM?
3. Що таке текстовий ембединг і яка його роль у RAG?
4. Поясніть принцип роботи семантичного пошуку. Чим він відрізняється від пошуку за ключовими словами?
5. Що таке косинусна подібність та скалярний добуток? Коли їх використовують для порівняння ембедингів?
6. Навіщо потрібен етап "чанкінгу" (chunking) при обробці документів для RAG? Які є підходи до чанкінгу?
7. Що таке контекстне вікно LLM і як воно впливає на дизайн RAG-системи?
8. Що таке квантизація моделі? Навіщо її використовують при роботі з LLM?
9. Поясніть роль етапу "доповнення" (augmentation) в RAG. Як формується промпт для LLM?
10. Які бібліотеки Python використовувалися в цій лабораторній роботі для: а) читання PDF, б) створення ембедингів, в) завантаження та використання LLM?
11. Як параметр `temperature` впливає на процес генерації тексту LLM?
12. Які потенційні проблеми можуть виникнути при реалізації RAG-системи? (наприклад, з якістю даних, пошуком, генерацією)

СПИСОК ЛІТЕРАТУРИ

1. Lewis, P., et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv preprint arXiv:2005.11401.

2. Google. (2024). Gemma: Open Models Based on Gemini Research and Technology. https://blog.google/technology/developers/gemma-open-models/
3. Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. arXiv preprint arXiv:1908.10084. (Sentence Transformers Library)
4. Hugging Face Transformers Documentation. https://huggingface.co/docs/transformers
5. Hugging Face Hub (for models and datasets). https://huggingface.co/
6. PyMuPDF Documentation. https://pymupdf.readthedocs.io/en/latest/
7. spaCy Documentation. https://spacy.io/
8. Pinecone. Chunking Strategies. https://www.pinecone.io/learn/chunking-strategies/
9. Prompt Engineering Guide. https://www.promptingguide.ai/
10. Vaswani, A., et al. (2017). Attention Is All You Need. NeurIPS.