МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"

ЛЕКЦІЯ 11. Налаштування виводів моделей ШІ: fine-tuning та RAG

Львів -- 2025

Лекція зі штучного інтелекту 2025-11

Вступ

На цьому занятті ми розглянемо методи удосконалення та налаштування виводів моделей штучного інтелекту після їх початкового навчання. В умовах стрімкого розвитку великих мовних моделей (LLM) та генеративного ШІ, здатність адаптувати попередньо навчені моделі до конкретних завдань та доменів стає критично важливою. Ми дослідимо два основні підходи: fine-tuning (тонке налаштування) моделей та Retrieval-Augmented Generation (RAG, генерація з доповненням через пошук). Особливу увагу приділимо технології RAG, яка дозволяє покращити точність, актуальність та надійність відповідей моделей ШІ, поєднуючи їхні внутрішні знання з зовнішніми джерелами інформації.

Теми, що розглядаються

1. Обмеження попередньо навчених моделей
2. Fine-tuning: налаштування моделей під конкретні задачі
3. Retrieval-Augmented Generation (RAG): основні концепції
4. Архітектура та компоненти RAG-систем
5. Індексація та пошук у великих корпусах даних
6. Вбудовування (Embeddings) та їх роль у RAG
7. Методи покращення пошуку та релевантності
8. Інтеграція результатів пошуку в промпти
9. Оцінка та вимірювання ефективності RAG-систем
10. Практичні застосування RAG у розробці програмного забезпечення

Обмеження попередньо навчених моделей

Незважаючи на вражаючі здібності сучасних великих мовних моделей (LLM), вони мають ряд суттєвих обмежень, які впливають на їхню ефективність та надійність у реальних застосуваннях:

1. Статичність знань

Великі мовні моделі, такі як GPT, LLaMA, Claude та інші, навчаються на масивних корпусах текстових даних, доступних до певної дати ("cutoff date"). Це призводить до кількох проблем:

- **Неактуальність інформації**: Моделі не мають доступу до новин, подій та знань, які з'явилися після дати навчання.
- **Неможливість оновлення**: Повне перенавчання масштабних моделей вимагає значних обчислювальних ресурсів та часу, що робить часті оновлення неможливими.
- **"Галюцинації"**: Моделі можуть впевнено генерувати неправильну інформацію, особливо щодо подій після дати навчання.

2. Обмежений контекстний простір

Хоча сучасні моделі мають значно більші контекстні вікна порівняно з попередніми поколіннями, вони все ще обмежені:

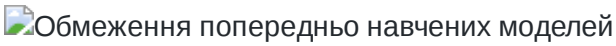
- **Фіксований розмір контексту**: Більшість моделей може обробляти обмежену кількість токенів у одному запиті (від 4K до 128K токенів).
- **Неможливість обробки великих документів**: Цілі бази знань, книги або технічні документації не можуть бути повністю завантажені в контекст.
- **Проблеми з довгостроковою пам'яттю**: Моделі можуть забувати інформацію з початку довгого контексту при генерації відповідей.

3. Узагальнені знання vs. спеціалізація

- **Брак експертних знань**: Хоча LLM мають широкі загальні знання, вони часто не володіють глибокими експертними знаннями у вузькоспеціалізованих доменах.
- **Недостатня представленість нішевої інформації**: Рідкісні теми, які мало представлені в навчальних даних, не будуть добре відтворені в моделі.
- **Компроміс між широтою та глибиною**: Збільшення об'єму загальних знань не обов'язково покращує експертизу в конкретних областях.

4. Проблеми достовірності та надійності

- **Відсутність належних посилань**: Моделі генерують відповіді без підтвердження з авторитетних джерел.
- **Неможливість перевірки фактів**: Моделі не мають механізмів для самостійної перевірки точності інформації.
- **Впевненість у неправильних відповідях**: Моделі можуть надавати неправильну інформацію з високою впевненістю.



Fine-tuning: налаштування моделей під конкретні задачі

Fine-tuning (тонке налаштування) — це процес додаткового навчання попередньо навченої моделі на меншому, спеціалізованому наборі даних для адаптації її до конкретних завдань або доменів. Цей підхід дозволяє зберегти загальні знання моделі, одночасно покращуючи її здатність вирішувати конкретні проблеми.

Основні методи fine-tuning

1. Повний Fine-tuning

- **Суть методу**: Оновлення всіх параметрів моделі на основі нових даних.
- **Переваги**: Максимальна адаптація до нового домену або завдання.
- **Недоліки**: Великі обчислювальні вимоги, ризик перенавчання, необхідність значного обсягу даних.
- **Застосування**: Коли доступно достатньо обчислювальних ресурсів та даних для навчання, а завдання суттєво відрізняється від оригінального домену моделі.

2. Parameter-Efficient Fine-tuning (PEFT)

PEFT-методи дозволяють адаптувати моделі, навчаючи лише невелику підмножину параметрів, що значно зменшує обчислювальні вимоги:

- **LoRA (Low-Rank Adaptation)**:
  - Додає низькорангові адаптери до ваг моделі.
  - Навчає лише ці додаткові ваги, зберігаючи оригінальні параметри незмінними.
  - Дозволяє зменшити кількість навчуваних параметрів до <1% від загальної кількості.
- **Prefix/Prompt Tuning**:
  - Додає навчувані вектори (префікси) до вхідних даних або прихованих станів моделі.
  - Навчає лише ці вектори, залишаючи модель недоторканою.
  - Особливо ефективний для задач з невеликою кількістю навчальних прикладів.
- **Adapters**:
  - Вставляє невеликі навчувані модулі між шарами моделі.
  - Дозволяє використовувати різні адаптери для різних завдань з однією базовою моделлю.

3. Instruction Tuning та RLHF

- **Instruction Tuning**:
  - Налаштування на наборах даних, де завдання сформульовані як інструкції природною мовою.
  - Покращує здатність моделі розуміти та виконувати різноманітні інструкції користувачів.
- **RLHF (Reinforcement Learning from Human Feedback)**:
  - Використовує зворотний зв'язок від людей для подальшого вдосконалення моделі.
  - Включає навчання моделі винагороди, яка оцінює якість відповідей.
  - Оптимізує модель методами навчання з підкріпленням, максимізуючи очікувану винагороду.

Переваги та недоліки fine-tuning

Переваги:

- **Покращена продуктивність**: Значне підвищення ефективності у вузькоспеціалізованих доменах.
- **Адаптація до стилю та жаргону**: Модель краще розуміє специфічну термінологію та способи вираження в конкретній галузі.
- **Редукція токенів**: Спеціалізовані моделі можуть надавати релевантні відповіді з використанням менших промптів.
- **Персоналізація**: Можливість адаптувати модель під конкретні потреби організації чи користувача.

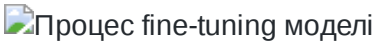
Недоліки:

- **Забування загальних знань**: Ризик "катастрофічного забування", коли модель втрачає раніше набуті знання.

- **Обчислювальна вартість**: Навіть з PEFT-методами, fine-tuning вимагає значних обчислювальних ресурсів.
- **Потреба в якісних даних**: Для ефективного налаштування потрібні репрезентативні та якісні дані.
- **Обмежена адаптивність**: Fine-tuned моделі все одно мають статичні знання та не вирішують проблему актуальності інформації.

Приклад процесу fine-tuning з використанням LoRA

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, TrainingArguments
from peft import LoraConfig, TaskType, get_peft_model

# Завантаження базової моделі та токенізатора
model_name = "meta-llama/Llama-2-7b-hf"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

# Конфігурація LoRA
lora_config = LoraConfig(
    task_type=TaskType.CAUSAL_LM,
    r=8,  # Ранг адаптації
    lora_alpha=32,  # Параметр масштабування
    lora_dropout=0.1,
    target_modules=["q_proj", "v_proj"]  # Цільові модулі для адаптації
)

# Створення PEFT-моделі
peft_model = get_peft_model(model, lora_config)

# Відображення навчуваних параметрів
print(f"Навчуваних параметрів: {peft_model.num_parameters(True)}")
print(f"Всього параметрів: {peft_model.num_parameters()}")
print(f"Співвідношення: {peft_model.num_parameters(True) / peft_model.num_parameters():.2%}")

# Конфігурація навчання
training_args = TrainingArguments(
    output_dir="./lora_model",
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,
    learning_rate=2e-4,
    num_train_epochs=3,
    save_strategy="epoch",
)

# Навчання моделі (потрібно імпортувати та підготувати набір даних)
# trainer = Trainer(
#     model=peft_model,
#     args=training_args,
#     train_dataset=train_dataset,
#     data_collator=data_collator,
# )
#
# trainer.train()
```



Хоча fine-tuning значно покращує здатність моделей вирішувати конкретні завдання, воно не вирішує фундаментальні обмеження, пов'язані з актуальністю інформації та валідацією фактів. Для подолання цих обмежень був розроблений підхід Retrieval-Augmented Generation (RAG), який ми розглянемо в наступних розділах.

Retrieval-Augmented Generation (RAG): основні концепції

Retrieval-Augmented Generation (RAG) — це гібридний підхід до генерації контенту, який поєднує потужність великих мовних моделей із можливістю отримувати та використовувати зовнішню інформацію. На відміну від стандартних LLM, які покладаються виключно на внутрішні параметри для генерації відповідей, RAG-системи використовують зовнішні бази знань для пошуку релевантної інформації, яка потім включається в процес генерації.

Суть та принципи роботи RAG

Основний принцип

RAG поєднує дві ключові операції:

1. **Retrieval (Пошук)**: Пошук релевантних фрагментів інформації з бази знань на основі запиту користувача.
2. **Generation (Генерація)**: Використання знайденої інформації разом із запитом для генерації відповіді LLM.

Цей підхід дозволяє моделі "читати" зовнішні джерела перед формуванням відповіді, подібно до того, як людина-експерт може звернутися до документації або інших матеріалів.

Історія розвитку

Термін "Retrieval-Augmented Generation" був формально введений дослідниками Meta AI (тоді Facebook AI) у 2020 році в роботі "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". З того часу концепція значно розвинулась і стала одним з основних підходів до покращення якості генеративних ШІ-систем.

Порівняння зі стандартними LLM

Аспект	Стандартні LLM	RAG-системи
Джерело знань	Виключно ваги моделі	Ваги моделі + зовнішні бази даних
Актуальність	Обмежена датою навчання	Може включати найсвіжішу інформацію
Прозорість	"Чорна скринька"	Можливість надати джерела інформації
Точність	Схильність до "галюцинацій"	Підвищена фактична точність
Адаптивність	Потребує перенавчання	Легко оновлюється через оновлення бази знань

Ключові переваги RAG

1. Покращена точність та надійність

- **Зниження "галюцинацій"**: Доступ до точних джерел інформації зменшує ймовірність генерації неправильних фактів.
- **Верифіковані відповіді**: Можливість надавати посилання на джерела, що підвищує довіру до відповідей.
- **Підвищена релевантність**: Відповіді точніше відповідають контексту запиту та домену.

2. Актуальність інформації

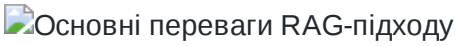
- **Динамічність знань**: База знань може постійно оновлюватися без необхідності перенавчання моделі.
- **Доступ до нової інформації**: Можливість включати найсвіжіші дані, документи та новини.
- **Подолання часового обмеження**: Усунення проблеми "cutoff date" стандартних LLM.

3. Спеціалізація та адаптивність

- **Доменна адаптація**: Легко адаптується до спеціалізованих доменів через включення відповідних документів.
- **Персоналізація**: Можливість інтегрувати особисті або організаційні знання.
- **Масштабованість знань**: Здатність працювати з великими обсягами інформації, що не поміщається в контекст моделі.

4. Прозорість та контрольованість

- **Обґрунтованість відповідей**: Можливість бачити, на основі яких даних було сформовано відповідь.
- **Контроль джерел**: Можливість вибирати та фільтрувати джерела інформації.
- **Відстеження походження інформації**: Здатність аудитувати джерела та забезпечувати відповідність регуляторним вимогам.



Типові сценарії використання RAG

- **Чат-боти з доступом до корпоративних знань**: Надання точних відповідей на основі внутрішньої документації компанії.
- **Пошукові системи нового покоління**: Генерація структурованих відповідей замість списку посилань.
- **Системи підтримки прийняття рішень**: Допомога експертам у аналізі великих обсягів даних та формуванні аргументованих висновків.
- **Персональні асистенти**: Доступ до особистих документів, нотаток та історії взаємодії для надання контекстуальних порад.
- **Освітні платформи**: Генерація навчальних матеріалів та відповідей на запитання з посиланнями на відповідні джерела.
- **Дослідницькі інструменти**: Допомога в аналізі наукової літератури та формуванні гіпотез.

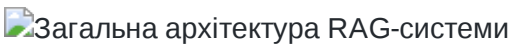
Архітектура та компоненти RAG-систем

Архітектура типової RAG-системи складається з кількох ключових компонентів, які працюють разом для забезпечення ефективного пошуку та генерації відповідей.

Загальна архітектура RAG

Стандартна архітектура RAG-системи включає такі основні компоненти:

1. **Corpus (Корпус документів)**: База знань, яка містить документи, статті, записи та інші джерела інформації.
2. **Indexing Pipeline (Конвеєр індексації)**:
    - Document Loader: Завантаження документів з різних джерел.
    - Chunker: Розбиття документів на менші частини (чанки).
    - Embedding Model: Перетворення текстових чанків у векторні представлення.
    - Vector Database: Зберігання та індексація векторних представлень для ефективного пошуку.
3. **Retrieval Pipeline (Конвеєр пошуку)**:
    - Query Understanding: Аналіз та обробка запиту користувача.
    - Query Embedding: Перетворення запиту у векторне представлення.
    - Similarity Search: Пошук найбільш релевантних документів у векторній базі даних.
    - Reranking (опціонально): Додаткове впорядкування результатів для підвищення релевантності.
4. **Generation Pipeline (Конвеєр генерації)**:
    - Context Builder: Формування контексту з вхідного запиту та знайдених документів.
    - LLM: Генерація відповіді на основі запиту та наданого контексту.
    - Response Processor: Додаткова обробка відповіді, включаючи форматування та цитування джерел.



Основні компоненти RAG-системи в деталях

1. Підготовка та індексація документів

Процес підготовки документів для використання в RAG-системі:

- **Збір документів**: Інтеграція з різними джерелами (файлові системи, бази даних, API, веб-скрапінг).
- **Завантаження та парсинг**: Обробка різних форматів документів (PDF, HTML, DOCX, Markdown тощо).
- **Розділення на чанки**:

    - Чанки повинні бути достатньо великими, щоб містити змістовну інформацію, але достатньо малими для ефективного пошуку.
    - Типові розміри чанків: 256-1024 токенів.
    - Стратегії розділення: за абзацами, фіксованою кількістю токенів, семантичними розділами.
- **Збагачення метаданими**: Додавання інформації про джерело, дату, автора, тематику тощо.

2. Векторні бази даних

Векторні бази даних спеціально оптимізовані для зберігання та швидкого пошуку векторних представлень (ембедингів) документів:

- **Принцип роботи**: Пошук найближчих сусідів (Approximate Nearest Neighbor, ANN) у багатовимірному просторі.
- **Ключові метрики**:
    - Швидкість пошуку: час, необхідний для знаходження релевантних документів.
    - Точність: здатність знаходити справді релевантні документи.
    - Масштабованість: ефективність при роботі з великими обсягами даних.
- **Популярні векторні бази даних**:
    - **Pinecone**: Хмарна векторна база даних з високою масштабованістю.
    - **Weaviate**: Повнофункціональна векторна база даних з відкритим кодом.
    - **Milvus**: Високопродуктивна векторна база даних для великих колекцій.
    - **FAISS**: Бібліотека для ефективного пошуку схожих векторів від Meta.
    - **Chroma**: Легка векторна база даних, оптимізована для додатків RAG.
    - **Qdrant**: Векторна база даних з фокусом на фільтрацію та балансування навантаження.

3. Процес пошуку (Retrieval)

Пошук релевантних документів на основі запиту користувача:

- **Векторизація запиту**: Перетворення запиту користувача у векторне представлення за допомогою тієї ж моделі вбудовування, що й для документів.
- **Семантичний пошук**: Пошук документів за семантичною подібністю, а не за ключовими словами.
    - Метрики подібності: косинусна схожість, евклідова відстань.
    - k-NN пошук: знаходження k найближчих сусідів у векторному просторі.
- **Гібридний пошук**: Комбінування семантичного пошуку з традиційним пошуком за ключовими словами.
- **Параметри пошуку**:
    - Top-k: кількість документів, які повертаються з бази даних.
    - Порогові значення подібності: мінімальна релевантність для включення документа.
    - Стратегії фільтрації: обмеження пошуку за метаданими (дата, автор, тема тощо).

4. Ранжування та фільтрація (Reranking)

Додаткова обробка результатів пошуку для підвищення їхньої релевантності:

- **Cross-encoders**: Модели, які оцінюють релевантність пари "запит-документ".
  - Більш точні, але обчислювально дорожчі порівняно з би-енкодерами (bi-encoders).
  - Застосовуються до обмеженої кількості кандидатів, попередньо знайдених за допомогою векторного пошуку.
- **Множинні критерії ранжування**:
  - Релевантність до запиту.
  - Свіжість інформації.
  - Авторитетність джерела.
  - Різноманітність інформації.
- **Фільтрація результатів**:
  - Видалення дублікатів та надлишкової інформації.
  - Застосування бізнес-правил та обмежень.
  - Перевірка на відповідність політикам безпеки та доступу.

5. Генерація відповіді (Generation)

Формування відповіді на основі знайденої інформації:

- **Формування промпту**:
  - Включення запиту користувача.
  - Включення релевантних документів як контексту.
  - Додавання інструкцій для моделі (система-промпт).
- **Методи включення знайденої інформації**:
  - Пряме включення в промпт.
  - Порівняльний аналіз між запитом та документами.
  - Складні схеми промптів з покроковим міркуванням.
- **Пост-обробка відповіді**:
  - Форматування та структурування.
  - Додавання посилань на джерела.
  - Перевірка узгодженості та повноти.

Типи RAG-архітектур

1. Базовий RAG

Найпростіша форма RAG, де знайдені документи безпосередньо включаються в промпт:

```
Система: Ти асистент, який надає відповіді на основі наданих документів.

Контекст:
[Документ 1]: ... текст документа 1 ...
[Документ 2]: ... текст документа 2 ...
[Документ 3]: ... текст документа 3 ...

Запитання користувача: ... запитання ...

Відповідь:
```

2. Розширений RAG (Advanced RAG)

Вдосконалені підходи, що покращують стандартну RAG-архітектуру:

- **Multi-query RAG**: Генерація декількох версій запиту для розширення пошуку.
- **Hypothetical Document Embeddings (HyDE)**: Створення гіпотетичного документа-відповіді перед пошуком.
- **Recursive RAG**: Ітеративний процес пошуку та генерації для складних запитів.
- **Multi-step RAG**: Розбиття процесу на покрокові етапи (декомпозиція запиту, пошук, синтез тощо).

3. Модульна RAG-архітектура

Гнучка архітектура, що дозволяє комбінувати різні компоненти та підходи:

- **Спеціалізовані індексатори**: Окремі індекси для різних типів контенту або доменів.
- **Множинні LLM**: Використання різних моделей для різних етапів процесу.
- **Адаптивні стратегії пошуку**: Динамічний вибір стратегії пошуку залежно від типу запиту.
- **Інтегровані інструменти**: Поєднання RAG з викликом API, виконанням коду та іншими інструментами.

Індексація та пошук у великих корпусах даних

Ефективність RAG-систем значною мірою залежить від якості індексації та пошуку у великих масивах документів. Розглянемо ключові аспекти цих процесів.

Підходи до індексації документів

1. Традиційна індексація на основі ключових слів

- **Інвертований індекс**: Класичний підхід, що відображає слова на документи, які їх містять.
- **BM25/TF-IDF**: Методи ранжування, що враховують частоту слів у документі та корпусі.
- **Переваги**: Швидкий пошук за точними словами та фразами.
- **Недоліки**: Не враховує семантичне значення, проблеми з синонімами та пов'язаними концепціями.

2. Семантична індексація на основі вбудовувань (embeddings)

- **Векторні представлення**: Кожен документ представлений як вектор у багатовимірному просторі.
- **Щільні вектори**: Типові розмірності від 384 до 1536, залежно від моделі вбудовування.
- **Переваги**: Здатність знаходити семантично пов'язані документи, навіть якщо вони не містять точних ключових слів.
- **Недоліки**: Вищі обчислювальні вимоги, складніша інфраструктура.

3. Гібридні підходи

- **Комбінування пошуку за ключовими словами та семантичного пошуку**:
  - Колаборативна фільтрація результатів обох методів.
  - Використання лексичного пошуку для первинного відбору, з подальшим семантичним ранжуванням.
- **Багаторівнева індексація**:
  - Різні індекси для різних рівнів деталізації документів.
  - Спеціалізовані індекси для різних типів інформації (текст, числа, дати, сутності).

Стратегії розділення документів на чанки

Ефективне розділення документів на чанки є критично важливим для RAG-систем, оскільки впливає на точність пошуку та якість контексту для генерації.

1. Методи розділення

- **Фіксований розмір**: Розділення за певною кількістю токенів, слів або символів.
  - Простий у реалізації, але може розділяти логічно пов'язані частини.
  - Типовий розмір: 256-1024 токенів.
- **Розділення за структурою документа**:
  - За абзацами, розділами, заголовками.
  - Зберігає логічну структуру, але може призводити до чанків різного розміру.
- **Семантичне розділення**:
  - На основі тематичної єдності тексту.
  - Складніше реалізувати, але дає більш змістовні чанки.

2. Стратегії перекриття (overlap)

- **Перекриття чанків**: Включення частини попереднього чанку в наступний.
  - Допомагає зберегти контекст на границях чанків.
  - Типовий відсоток перекриття: 10-30%.
- **Ковзне вікно**:
  - Поступове переміщення "вікна" заданого розміру через документ.
  - Гнучкий контроль над розміром перекриття.

3. Метадані та структурна інформація

- **Збагачення чанків метаданими**:
  - Джерело документа, автор, дата.
  - Позиція в оригінальному документі.
  - Ієрархічна інформація (розділ, підрозділ).
- **Підтримка структурних зв'язків**:
  - Збереження відношень між чанками.
  - Можливість відтворення оригінального контексту.

Приклад процесу індексації документів

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Chroma
from langchain.document_loaders import TextLoader, PyPDFLoader

# Завантаження документів
pdf_loader = PyPDFLoader("document.pdf")
text_loader = TextLoader("document.txt")

documents = pdf_loader.load() + text_loader.load()

# Розділення документів на чанки
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200,
    length_function=len,
    separators=["\n\n", "\n", " ", ""]
)

chunks = text_splitter.split_documents(documents)

# Створення вбудовувань та індексація
embeddings = OpenAIEmbeddings()
vectorstore = Chroma.from_documents(
    documents=chunks,
    embedding=embeddings,
    persist_directory="./chroma_db"
)

# Збереження індексу
vectorstore.persist()
```

Оптимізація пошуку

1. Алгоритми наближених найближчих сусідів (ANN)

Для ефективного пошуку в великих колекціях векторів застосовуються спеціальні алгоритми:

- **Locality-Sensitive Hashing (LSH)**: Хешування, де схожі вектори мають високу ймовірність потрапити в один хеш-бакет.
- **Hierarchical Navigable Small World (HNSW)**: Графова структура, що дозволяє швидко знаходити приблизних найближчих сусідів.
- **Inverted File Index (IVF)**: Розбиття простору на кластери для прискорення пошуку.

2. Параметри оптимізації пошуку

- **Індекс-тайм vs. запит-тайм оптимізації**:
  - Компроміс між швидкістю індексації та швидкістю пошуку.
  - Оптимізація структур даних під конкретні сценарії використання.
- **Кількість результатів (Top-k)**:
  - Вибір оптимальної кількості документів для повернення.
  - Балансування між повнотою інформації та розміром контексту.
- **Фільтрація та обмеження**:
  - Метаданні фільтри для звуження пошуку.
  - Часові та категоріальні обмеження.

3. Кешування та оптимізація запитів

- **Кешування популярних запитів**:
  - Зберігання результатів частих запитів.
  - Інкрементальне оновлення кешу.
- **Препроцесинг запитів**:
  - Нормалізація та очищення запиту.
  - Розширення запиту синонімами або пов'язаними термінами.

Вбудовування (Embeddings) та їх роль у RAG

Вбудовування (embeddings) — це щільні векторні представлення текстів, які кодують їх семантичне значення у багатовимірному просторі. Вони є критично важливим компонентом сучасних RAG-систем, оскільки забезпечують семантичний пошук та порівняння документів.

Основи текстових вбудовувань

1. Принцип роботи вбудовувань

- **Векторне представлення**: Тексти перетворюються на числові вектори фіксованої розмірності (наприклад, 768, 1024 або 1536).

- **Семантична близькість**: Тексти з подібним значенням мають близькі вектори у просторі вбудовувань.
- **Дистрибутивна семантика**: Значення слів та фраз визначається їхнім контекстом та вживанням.

2. Типи моделей вбудовувань

- **Універсальні моделі**:
  - OpenAI text-embedding-ada-002, text-embedding-3
  - Sentence-BERT (SBERT)
  - E5, GTEBase, Instructor
- **Спеціалізовані доменні моделі**:
  - Моделі, специфічні для медицини, юриспруденції, фінансів тощо.
  - Адаптовані моделі для конкретних мов.
- **Мультимодальні вбудовування**:
  - CLIP (текст та зображення)
  - Вбудовування для коду, аудіо, відео та комбінованого контенту.

3. Характеристики якісних вбудовувань

- **Висока розмірність**: Зазвичай від 384 до 1536 розмірностей для балансу між виразністю та ефективністю.
- **Щільність**: Кожен елемент вектора містить значущу інформацію.
- **Інваріантність**: Нечутливість до незначних змін формулювання, синонімів.
- **Збереження семантики**: Здатність відображати тонкі семантичні відмінності.

Моделі вбудовувань для RAG

1. Порівняння популярних моделей вбудовувань

Модель	Розмірність	Переваги	Недоліки	Типові застосування
OpenAI text-embedding-ada-002	1536	Висока якість, розвинені семантичні можливості	Платна, закрита	Загальні RAG-системи
OpenAI text-embedding-3	1536/3072	Покращена продуктивність, різні розміри	Платна, закрита	Сучасні RAG-системи з високими вимогами

Модель	Розмірність	Переваги	Недоліки	Типові застосування
BERT/Sentence-BERT	768	Відкрита, налаштовувана	Менша ефективність порівняно з сучасними моделями	Бюджетні та локальні RAG-системи
E5/GTEBase	768/1024	Висока продуктивність, відкриті моделі	Вищі обчислювальні вимоги	Відкриті RAG-системи з високою якістю
Instructor	768	Налаштування через інструкції	Складніша в використанні	Спеціалізовані RAG-системи з особливими вимогами

2. Тонке налаштування моделей вбудовувань

- **Доменна адаптація**:
  - Налаштування на специфічних галузевих даних.
  - Покращення релевантності для конкретного домену.
- **Контрастивне навчання**:
  - Використання пар "запит-документ" для покращення пошукових можливостей.
  - Оптимізація для конкретних задач пошуку.
- **Дистиляція знань**:
  - Перенесення знань з великих моделей у менші, ефективніші моделі.
  - Збереження продуктивності при зменшенні обчислювальних вимог.

3. Вбудовування для різних типів контенту

- **Вбудовування довгих документів**:
  - Стратегії для документів, які перевищують контекстне вікно моделі.
  - Ієрархічні вбудовування для збереження структури.
- **Мультимодальні вбудовування**:
  - Об'єднані вбудовування для тексту та зображень.
  - Представлення структурованих даних (таблиці, графіки).
- **Спеціалізовані типи контенту**:
  - Вбудовування коду для технічної документації.
  - Вбудовування математичних формул та наукових даних.

Метрики подібності та пошук

1. Основні метрики подібності

- **Косинусна подібність**:
  - Найпопулярніша метрика для вбудовувань.
  - Вимірює косинус кута між векторами.
  - Значення від -1 (протилежні) до 1 (ідентичні).
- **Евклідова відстань**:
  - Пряма відстань між векторами у просторі.
  - Корисна для моделей, спеціально навчених для евклідового простору.
- **Dot-product (скалярний добуток)**:
  - Простіший у обчисленні, але чутливий до розміру вектора.
  - Використовується в деяких системах для оптимізації.

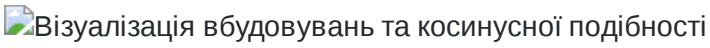
2. Приклад обчислення подібності вбудовувань

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from sentence_transformers import SentenceTransformer

# Ініціалізація моделі вбудовувань
model = SentenceTransformer('all-MiniLM-L6-v2')

# Запит та потенційні документи
query = "Які основні переваги RAG-систем?"
documents = [
    "RAG-системи поєднують пошук інформації з генеративними моделями.",
    "Retrieval-Augmented Generation значно зменшує проблему галюцинацій у LLM.",
    "Сучасні методи машинного навчання включають нейронні мережі та дерева рішень."
]

# Створення вбудовувань
query_embedding = model.encode(query)
document_embeddings = model.encode(documents)

# Обчислення косинусної подібності
similarities = cosine_similarity(
    [query_embedding],
    document_embeddings
)[0]

# Виведення результатів
for i, similarity in enumerate(similarities):
    print(f"Документ {i+1}: {similarity:.4f} - {documents[i]}")

# Сортування документів за подібністю
ranked_results = sorted(
    zip(documents, similarities),
```

```
    key=lambda x: x[1],
    reverse=True
)
```

3. Стратегії покращення пошуку через вбудовування

- **Ансамблі вбудовувань**:
  - Комбінування результатів різних моделей вбудовувань.
  - Використання спеціалізованих моделей для різних аспектів запиту.
- **Контекстуалізовані вбудовування**:
  - Адаптація вбудовувань відповідно до контексту запиту.
  - Динамічне налаштування вбудовувань для конкретної сесії.
- **Розширені представлення документів**:
  - Додаткові вбудовування для різних аспектів документа (заголовок, резюме, повний текст).
  - Багаторівневі вбудовування для ієрархічних документів.


Візуалізація вбудовувань та косинусної подібності

Методи покращення пошуку та релевантності

Ефективність RAG-системи значною мірою залежить від якості пошуку та релевантності знайдених документів. Розглянемо методи покращення цих аспектів.

Розширені техніки пошуку

1. Мульти-запитний RAG (Multi-query RAG)

Традиційний RAG використовує оригінальний запит користувача для пошуку документів. Мульти-запитний підхід генерує декілька варіацій або аспектів запиту для розширення пошуку:

- **Принцип роботи**:
  - Використання LLM для генерації альтернативних формулювань запиту.
  - Проведення пошуку з кожним із сформованих запитів.
  - Агрегація та дедуплікація результатів.
- **Приклад імплементації**:

```
from langchain.retrievers.multi_query import MultiQueryRetriever
from langchain.chat_models import ChatOpenAI

# Ініціалізація LLM для переформулювання запитів
llm = ChatOpenAI(temperature=0)

# Створення мульти-запитного ретривера
retriever = MultiQueryRetriever.from_llm(
```

```
    retriever=vector_db.as_retriever(),
    llm=llm
)

# Використання ретривера
original_query = "Як покращити точність RAG-системи?"
documents = retriever.get_relevant_documents(original_query)
```

- **Переваги**:
  - Розширює охоплення пошуку, включаючи різні аспекти запиту.
  - Зменшує ймовірність пропуску релевантної інформації через специфічне формулювання.
  - Особливо ефективний для складних запитів з багатьма аспектами.

2. Гіпотетичні вбудовування документів (HyDE)

HyDE (Hypothetical Document Embeddings) — це підхід, який використовує LLM для генерації гіпотетичної відповіді перед пошуком:

- **Принцип роботи**:
  - Генерація гіпотетичного документа-відповіді на запит користувача.
  - Створення вбудовування для цього гіпотетичного документа.
  - Використання цього вбудовування для пошуку релевантних реальних документів.
- **Обґрунтування**:
  - Гіпотетичний документ може містити термінологію та структуру, схожу на реальні документи в корпусі.
  - Це допомагає подолати семантичний розрив між запитом та документами.
- **Приклад імплементації**:

```
def hyde_retrieval(query, llm, embedding_model, vector_db):
    # Генерація гіпотетичного документа
    prompt = f"""
    Питання: {query}
    Створи документ, який міг би містити відповідь на це питання.
    """
    hypothetical_doc = llm(prompt)

    # Створення вбудовування для гіпотетичного документа
    hyde_embedding = embedding_model.embed_query(hypothetical_doc)

    # Пошук документів за вбудовуванням гіпотетичного документа
    documents = vector_db.similarity_search_by_vector(hyde_embedding, k=5)

    return documents
```

3. Ітеративний RAG (Recursive RAG)

Цей підхід передбачає багатокроковий процес пошуку та уточнення інформації:

- **Принцип роботи**:
  - Початковий пошук за запитом користувача.
  - Аналіз знайдених документів та визначення "інформаційних прогалин".
  - Формування додаткових запитів для заповнення прогалин.
  - Повторення процесу до досягнення достатньої інформаційної повноти.
- **Застосування**:
  - Складні аналітичні запити, що вимагають синтезу інформації з різних джерел.
  - Дослідницькі запити, де користувач не знає заздалегідь повну структуру інформації.
  - Питання, що потребують врахування різних точок зору або аспектів.

Методи ранжування та фільтрації

1. Крос-енкодерне ранжування (Cross-encoder Reranking)

Крос-енкодери — це моделі, які оцінюють релевантність пари "запит-документ" з високою точністю:

- **Відмінності від би-енкодерів**:
  - Би-енкодери (стандартні моделі вбудовувань) кодують запит і документ окремо.
  - Крос-енкодери обробляють запит і документ разом, враховуючи їх взаємодію.
  - Крос-енкодери точніші, але обчислювально дорожчі.
- **Двоетапний пошук**:
  - Початковий пошук з би-енкодером для швидкого відбору кандидатів.
  - Повторне ранжування з крос-енкодером для вибору найбільш релевантних документів.
- **Приклад з використанням бібліотеки SBERT**:

```
from sentence_transformers import CrossEncoder

# Ініціалізація крос-енкодера
cross_encoder = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-v2')

# Припустимо, що у нас є запит та попередньо знайдені документи
query = "Як RAG покращує точність відповідей LLM?"
candidate_docs = retriever.get_relevant_documents(query, k=20)

# Створення пар для ранжування
pairs = [(query, doc.page_content) for doc in candidate_docs]

# Ранжування з використанням крос-енкодера
scores = cross_encoder.predict(pairs)

# Поєднання документів зі скорами та сортування
ranked_results = sorted(
    zip(candidate_docs, scores),
    key=lambda x: x[1],
    reverse=True
```

```
)

# Вибір топ-N результатів
top_docs = [doc for doc, _ in ranked_results[:5]]
```

2. Контекстуальна релевантність

Оцінка релевантності з урахуванням конкретного контексту взаємодії:

- **Персоналізована релевантність**:
  - Врахування історії запитів користувача.
  - Адаптація до рівня експертизи користувача.
  - Врахування попередніх взаємодій з RAG-системою.
- **Врахування мети запиту**:
  - Класифікація запитів за типом (інформаційний, навігаційний, транзакційний).
  - Адаптація критеріїв релевантності відповідно до наміру користувача.
  - Динамічне налаштування параметрів пошуку.

3. Семантичне фільтрування та кластеризація

- **Фільтрування надлишкової інформації**:
  - Виявлення та видалення дублікатів та майже дублікатів.
  - Виявлення семантично надлишкових фрагментів.
  - Оптимізація розміру контексту для генерації.
- **Тематична кластеризація**:
  - Групування знайдених документів за тематичними кластерами.
  - Забезпечення тематичного балансу та різноманітності.
  - Структурування інформації для більш когерентної генерації.
- **Приклад імплементації кластеризації**:

```
from sklearn.cluster import KMeans
import numpy as np

def cluster_documents(docs, embeddings, n_clusters=3):
    # Кластеризація документів на основі їхніх вбудовувань
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    clusters = kmeans.fit_predict(embeddings)

    # Групування документів за кластерами
    clustered_docs = {i: [] for i in range(n_clusters)}
    for i, cluster in enumerate(clusters):
        clustered_docs[cluster].append((docs[i], embeddings[i]))

    # Вибір найцентральніших документів з кожного кластера
    selected_docs = []
```

```
    for cluster, docs_in_cluster in clustered_docs.items():
        if docs_in_cluster:
            centroid = kmeans.cluster_centers_[cluster]
            distances = [np.linalg.norm(emb - centroid) for _, emb in docs_in_cluster]
            # Вибір документа, найближчого до центроїда кластера
            selected_docs.append(docs_in_cluster[np.argmin(distances)][0])

    return selected_docs
```

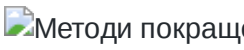
Адаптивний пошук

1. Динамічне налаштування параметрів пошуку

- **Адаптація до складності запиту**:
  - Прості запити: прямий пошук з стандартними параметрами.
  - Складні запити: розширений пошук з генерацією підзапитів.
- **Моніторинг та зворотний зв'язок**:
  - Аналіз взаємодії користувача з відповідями системи.
  - Уточнення параметрів пошуку на основі зворотного зв'язку.
  - A/B тестування різних стратегій пошуку.

2. Обізнаний про запитання пошук (Query-aware Retrieval)

- **Аналіз типу запиту**:
  - Класифікація запитів за типом: фактичний, процедурний, концептуальний тощо.
  - Налаштування стратегії пошуку відповідно до типу запиту.
- **Декомпозиція складних запитів**:
  - Розбиття складних запитів на простіші підзапити.
  - Агрегація результатів окремих підзапитів.
  - Переформулювання запитів для покращення релевантності.


Методи покращення релевантності пошуку

Інтеграція результатів пошуку в промпти

Ефективне включення контексту, отриманого через пошук, у промпти для LLM є ключовим аспектом RAG-систем. Розглянемо структуру промптів та стратегії інтеграції контексту.

Структура промптів для RAG

1. Компоненти RAG-промпту

Типовий промпт для RAG-системи складається з кількох ключових компонентів:

- **Системна інструкція (System Prompt)**:

- Визначає роль та обмеження моделі.
- Встановлює загальні правила формування відповіді.
- Задає основні параметри взаємодії.

- **Контекст з пошуку (Retrieved Context)**:
  - Включає знайдені документи або їх частини.
  - Може містити метадані про джерела.
  - Структурується для легкого сприйняття моделлю.

- **Запит користувача (User Query)**:
  - Оригінальний запит або його модифікована версія.
  - Може включати уточнення або додаткові інструкції.

- **Інструкція для генерації (Generation Instruction)**:
  - Вказівки щодо формату та змісту відповіді.
  - Вимоги до цитування та посилання на джерела.
  - Інші специфічні вимоги до відповіді.

2. Шаблони RAG-промптів

Базовий шаблон RAG-промпту:

```
Ти — корисний асистент, який надає достовірну інформацію.
Якщо ти не знаєш відповіді або вона не міститься в контексті, так і скажи замість припущень.
Завжди посилайся на джерела інформації у своїй відповіді.

Контекст:
{context}

Запитання: {query}

Відповідь:
```

Розширений шаблон з метаданими:

```
Ти — дослідницький асистент, який допомагає знаходити точну інформацію.
Використовуй лише факти з наданих джерел. Цитуй джерела, використовуючи номери в квадратних дуж

Контекст:
[1] Джерело: {source_1}
Дата: {date_1}
Текст: {context_1}

[2] Джерело: {source_2}
Дата: {date_2}
Текст: {context_2}

...
```

```
Запитання: {query}

Надай всебічну та детальну відповідь, цитуючи відповідні джерела.
```

Шаблон з покроковими міркуваннями:

```
Ти — аналітичний асистент, який дає обґрунтовані відповіді на складні запитання.
Відповідаючи на запитання, спочатку проаналізуй наданий контекст,
визнач ключові релевантні факти, а потім сформулюй відповідь на основі цього аналізу.

Контекст:
{context}

Запитання: {query}

Міркуй покроково:
1. Спочатку визнач, чи містить контекст достатньо інформації для відповіді.
2. Виділи ключові факти з контексту, що стосуються запитання.
3. Проаналізуй ці факти та їх взаємозв'язки.
4. Сформулюй чітку відповідь на основі цього аналізу.
```

Стратегії інтеграції контексту

1. Упорядкування та форматування контексту

- **Ранжування за релевантністю**:
  - Розміщення найбільш релевантних документів на початку контексту.
  - Врахування позиційних упереджень LLM (схильність краще запам'ятовувати початок і кінець контексту).
- **Структурування контексту**:
  - Чітке розмежування між різними документами або фрагментами.
  - Використання маркерів, нумерації або спеціальних розділювачів.
  - Ієрархічна організація інформації (загальне → специфічне).
- **Спрощення та сумаризація**:
  - Попередня обробка довгих текстів для зменшення розміру контексту.
  - Видалення надлишкової або нерелевантної інформації.
  - Використання LLM для створення стислих резюме перед включенням у промпт.

2. Управління контекстним вікном

- **Оптимізація розміру контексту**:
  - Балансування між повнотою інформації та обмеженнями токенів.
  - Адаптивний підхід залежно від складності запиту.
  - Використання різних стратегій для моделей з різним контекстним вікном.

- **Стратегії при перевищенні контекстного ліміту**:
  - Фільтрація та пріоритезація найбільш релевантних документів.
  - Розділення на послідовні запити з передачею проміжних результатів.
  - Використання сумаризації або вилучення ключової інформації.
- **Приклад багатоетапного підходу**:

```
def multi_stage_rag(query, documents, llm, max_tokens=4000):
    if estimate_tokens(documents) <= max_tokens:
        # Якщо всі документи поміщаються у контекст
        return standard_rag_query(query, documents, llm)

    # Якщо не поміщаються, спочатку сумаризуємо
    summaries = []
    for doc in documents:
        summary_prompt = f"Сумаризуй ключову інформацію з наступного тексту, що стосується '{qu
        summary = llm(summary_prompt, max_tokens=200)
        summaries.append(summary)

    # Використовуємо сумаризовані версії для основного запиту
    return standard_rag_query(query, summaries, llm)
```

3. Метапромпти та мета-RAG

- **Метапромпти**:
  - Використання LLM для створення оптимальних промптів для іншого LLM.
  - Динамічне формування системних інструкцій залежно від запиту.
  - Адаптація стратегії промптів залежно від типу запиту.
- **Мета-RAG**:
  - Використання LLM для вибору оптимальної стратегії RAG.
  - Динамічне визначення параметрів пошуку та формування промпту.
  - Багаторівневий підхід для складних запитів.

Техніки покращення відповідей

1. Цитування та атрибуція

- **Inline-цитування**:
  - Включення номерів джерел безпосередньо в текст відповіді.
  - Формати: [1], (Джерело 1), тощо.
- **Автоматична генерація посилань**:
  - Включення метаданих про джерела у промпт.
  - Інструкції LLM щодо форматування цитувань.
  - Валідація та перевірка атрибуції.

- **Приклад промпту з інструкціями щодо цитування**:

```
Ти — асистент з дослідження, який надає достовірну інформацію.
При відповіді на запитання:
1. Використовуй ТІЛЬКИ факти з наданих джерел.
2. Цитуй джерела в кінці кожного твердження, використовуючи номери в квадратних дужках [1], [2]
3. Якщо в джерелах недостатньо інформації, чесно визнай це.
4. В кінці відповіді додай розділ "Джерела" з повним списком використаних джерел.

Контекст:
[1] Джерело: "Основи RAG", 2023, Автор: А. Іванов
Текст: {context_1}

[2] Джерело: "Дослідження ефективності RAG", 2024, Автор: Б. Петров
Текст: {context_2}

Запитання: {query}
```

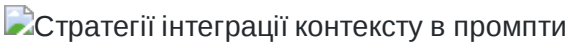
2. Саморефлексія та самокорекція

- **Техніка самоперевірки**:
  - Інструкція LLM критично оцінити власну відповідь.
  - Перевірка узгодженості з наданим контекстом.
  - Виявлення та корекція потенційних неточностей.
- **Приклад промпту з самоперевіркою**:

```
Запитання: {query}
Контекст: {context}

Відповідь на запитання в три етапи:
1. Спочатку сформулюй попередню відповідь на основі контексту.
2. Потім критично проаналізуй цю відповідь: перевір чи всі твердження підтверджуються контекстo
3. Нарешті, надай покращену фінальну відповідь з врахуванням результатів аналізу.
```

3. Керування невизначеністю

- **Явне визнання обмежень**:
  - Чітке зазначення, коли інформації недостатньо.
  - Розмежування між фактами з контексту та можливими припущеннями.
  - Вказування на протиріччя або невизначеності у джерелах.
- **Гранульованість упевненості**:
  - Використання кваліфікаторів рівня впевненості.
  - Диференціація між твердо встановленими фактами та ймовірною інформацією.
  - Надання альтернативних точок зору, коли це доречно.



Оцінка та вимірювання ефективності RAG-систем

Для розробки ефективних RAG-систем необхідно мати надійні методи оцінки їх продуктивності. Розглянемо основні підходи до тестування та вимірювання якості RAG-систем.

Багатовимірна оцінка RAG-систем

1. Ключові аспекти оцінки

- **Точність пошуку (Retrieval Accuracy)**:
  - Наскільки точно система знаходить релевантні документи.
  - Визначає верхню межу продуктивності всієї системи.
- **Якість генерації (Generation Quality)**:
  - Чіткість, послідовність та грамотність відповідей.
  - Здатність синтезувати інформацію з різних документів.
- **Фактична точність (Factual Accuracy)**:
  - Відсутність фактичних помилок та "галюцинацій".
  - Узгодженість відповідей із наданими джерелами.
- **Релевантність відповідей (Answer Relevance)**:
  - Відповідність відповіді поставленому запитанню.
  - Повнота охоплення всіх аспектів запиту.

2. Загальні метрики оцінки

- **Стандартні IR-метрики**:
  - **Точність (Precision)**: Частка релевантних документів серед знайдених.
  - **Повнота (Recall)**: Частка знайдених релевантних документів серед усіх релевантних.
  - **F1-score**: Гармонічне середнє між точністю та повнотою.
  - **MAP (Mean Average Precision)**: Середня точність на різних рівнях повноти.
  - **NDCG (Normalized Discounted Cumulative Gain)**: Оцінює не лише релевантність, але й порядок результатів.
- **Текстові метрики**:
  - **BLEU, ROUGE**: Оцінка схожості згенерованого тексту з еталонними відповідями.
  - **BERTScore**: Семантична оцінка, що використовує вбудовування.
  - **METEOR**: Враховує синоніми та перефразування.
- **Метрики для факт-чекінгу**:
  - **Точність атрибуції**: Частка правильно атрибутованих тверджень.
  - **Фактична консистентність**: Узгодженість фактів у відповіді з джерелами.

- **Кількість "галюцинацій"**: Частка тверджень, які не підтверджуються контекстом.

Методи оцінки RAG-компонентів

1. Оцінка компонента пошуку

- **Автоматичні методи**:
  - **Оцінка через золотий стандарт**: Використання розмічених пар "запит-документ".
  - **Проксі-оцінка**: Використання LLM для оцінки релевантності знайдених документів.
  - **Контрфактуальна оцінка**: Порівняння з результатами альтернативних стратегій пошуку.
- **Приклад метрики контекстуальної точності**:

```
def context_precision(retrieved_chunks, relevant_chunks):
    """
    Обчислення точності пошуку на основі співпадіння знайдених та релевантних чанків

    Args:
        retrieved_chunks: Список знайдених чанків
        relevant_chunks: Список дійсно релевантних чанків

    Returns:
        Точність пошуку (0-1)
    """
    retrieved_set = set(retrieved_chunks)
    relevant_set = set(relevant_chunks)

    if len(retrieved_set) == 0:
        return 0.0

    return len(retrieved_set.intersection(relevant_set)) / len(retrieved_set)
```

2. Оцінка якості генерації

- **Автоматичні методики**:
  - **Натреновані оцінювачі**: Моделі, спеціально навчені оцінювати якість відповідей.
  - **Аналіз на основі LLM**: Використання великих моделей для оцінки різних аспектів відповідей.
  - **Структуровані рубрики**: Систематична оцінка за кількома вимірами якості.
- **Приклад оцінки за допомогою LLM**:

```
def evaluate_rag_response(query, context, response, evaluator_llm):
    """
    Оцінка відповіді RAG-системи за допомогою LLM-оцінювача
    """
    evaluation_prompt = f"""
    Оціни відповідь на питання за наступними критеріями від 1 до 10:

    Питання: {query}
```

```
    Наданий контекст:
    {context}

    Відповідь:
    {response}

    Критерії оцінки:
    1. Фактична точність: Наскільки відповідь відповідає фактам з контексту? (1-10)
    2. Повнота: Наскільки повно відповідь охоплює всі аспекти питання? (1-10)
    3. Релевантність: Наскільки відповідь відповідає заданому питанню? (1-10)
    4. Зв'язність: Наскільки добре структурована та логічно організована відповідь? (1-10)

    Для кожного критерію надай обґрунтування оцінки та загальну оцінку.
    Також вкажи, чи містить відповідь інформацію, якої немає в контексті (галюцинації).

    Формат відповіді:
    Фактична точність: [оцінка] - [обґрунтування]
    Повнота: [оцінка] - [обґрунтування]
    Релевантність: [оцінка] - [обґрунтування]
    Зв'язність: [оцінка] - [обґрунтування]
    Наявність галюцинацій: [Так/Ні] - [приклади, якщо є]
    Загальна оцінка: [середня оцінка]
    """

    return evaluator_llm(evaluation_prompt)
```

3. E2E (наскрізна) оцінка

- **Людська оцінка**:
  - **Експертна оцінка**: Перевірка фахівцями предметної області.
  - **Сліпе порівняння**: Порівняння відповідей різних систем без знання їх джерела.
  - **А/Б тестування**: Порівняння різних версій RAG-системи з реальними користувачами.
- **Автоматизована E2E оцінка**:
  - **Pipeline-метрики**: Комбіновані метрики, що враховують всі етапи RAG.
  - **Оцінка кінцевого результату**: Фокус на якості фінальної відповіді.
  - **Decision-based metrics**: Оцінка здатності системи допомогти прийняти правильне рішення.

Тестові набори даних та середовища

1. Стандартні бенчмарки для RAG

- **KILT (Knowledge-Intensive Language Tasks)**: Набір завдань, що вимагають зовнішніх знань.
- **BEIR (Benchmarking IR)**: Колекція наборів даних для оцінки пошуку.
- **LIAR**: Датасет для перевірки фактів та виявлення недостовірної інформації.
- **NQ (Natural Questions)**: Набір реальних пошукових запитів та відповідей.
- **HotpotQA**: Мультистепові запитання, що вимагають міркувань на основі кількох документів.

2. Побудова власних тестових наборів

- **Синтетичні набори даних**:
  - Генерація запитань на основі корпусу документів.
  - Створення складних сценаріїв для тестування конкретних аспектів системи.
  - Використання LLM для автоматичного створення Q&A пар.
- **Процес створення тестового набору**:
  - **Вибір репрезентативної вибірки документів**.
  - **Створення різноманітних запитів**, що охоплюють різні типи запитань та складність.
  - **Розмітка очікуваних відповідей** та релевантних фрагментів.
  - **Валідація експертами** для забезпечення якості.
- **Приклад скрипта для генерації синтетичних запитань**:

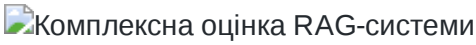
```
def generate_test_questions(document, llm, num_questions=5):
    """
    Генерація тестових запитань на основі документа
    """
    prompt = f"""
    На основі наступного документа згенеруй {num_questions} різноманітних запитань.
    Створи запитання різних типів складності:
    - Прості фактичні запитання (відповідь прямо в тексті)
    - Запитання, що вимагають поєднання кількох частин документа
    - Запитання, що вимагають виведення або узагальнення

    Для кожного запитання також вкажи:
    1. Очікувану відповідь
    2. Тип запитання (фактичне, логічне виведення, узагальнення тощо)
    3. Складність (легке, середнє, складне)

    Документ:
    {document}

    Формат відповіді:
    Q1: [запитання]
    Відповідь: [очікувана відповідь]
    Тип: [тип запитання]
    Складність: [складність]

    Q2: ...
    """

    return llm(prompt)
```

3. Безперервне тестування та моніторинг

- **A/Б тестування різних конфігурацій**:
  - Порівняння різних моделей вбудовувань, стратегій чанкінгу, промптів.
  - Вимірювання ефекту від змін на реальних користувацьких запитах.
- **Моніторинг продуктивності**:

    - Відстеження метрик релевантності та задоволеності користувачів.
    - Виявлення патернів помилок та проблемних типів запитів.
    - Аналіз відгуків користувачів та їхньої взаємодії з системою.

- **Адаптивне покращення**:
    - Використання зібраних даних для постійного вдосконалення системи.
    - Створення контрольних прикладів для регресійного тестування.
    - Документування успішних та невдалих оптимізацій.



Практичні застосування RAG у розробці програмного забезпечення

Технологія RAG знаходить широке застосування в різних сферах розробки програмного забезпечення. Розглянемо ключові практичні сценарії використання.

RAG у розробці чат-ботів та асистентів

1. Асистенти з доступом до корпоративних знань

- **Корпоративні бази знань**:
    - Інтеграція з внутрішньою документацією, вікі, базами знань.
    - Відповіді на запитання на основі корпоративних політик та процедур.
    - Доступ до історичних даних та інституційної пам'яті.

- **Технічна підтримка**:
    - Автоматизовані відповіді на часті запитання.
    - Направлення до відповідних ресурсів та документації.
    - Пріоритизація запитів для людського втручання.

- **Архітектурні особливості**:
    - Багаторівнева безпека та контроль доступу.
    - Можливість аудиту та відстеження джерел інформації.
    - Інтеграція з існуючими системами управління знаннями.

2. Персоналізовані асистенти

- **Інтеграція з особистими даними**:
    - Доступ до особистих документів, нотаток, електронних листів.
    - Аналіз історії взаємодій для контекстуалізації відповідей.
    - Адаптація до індивідуальних потреб та переваг.

- **Асистенти для професійних завдань**:

- Допомога в дослідженні та аналізі даних.
- Підготовка звітів та презентацій.
- Керування особистими проєктами та завданнями.

RAG у документно-орієнтованих системах

1. Розумний пошук та взаємодія з документами

- **Розширений пошук у документах**:
  - Семантичний пошук з розумінням контексту запиту.
  - Прямі відповіді замість списку документів.
  - Багатомовний пошук та крос-лінгвістичне розуміння.
- **Інтерактивна взаємодія з документами**:
  - Генерація резюме та виділення ключових моментів.
  - Відповіді на запитання щодо конкретних частин документів.
  - Пояснення складних концепцій на основі документації.

2. Аналіз та обробка юридичних документів

- **Аналіз контрактів та правових документів**:
  - Вилучення ключових умов та зобов'язань.
  - Порівняння з попередніми версіями або стандартами.
  - Виявлення потенційних ризиків та невідповідностей.
- **Правові дослідження**:
  - Пошук релевантних прецедентів та законодавчих актів.
  - Аналіз судової практики та правових тенденцій.
  - Підготовка юридичних аргументів та обґрунтувань.

RAG у розробці ігор

1. Інтелектуальні NPC та діалогові системи

- **Динамічні діалоги в іграх**:
  - Генерація контекстуально релевантних діалогів на основі ігрової бази знань.
  - Запам'ятовування та використання історії взаємодій з гравцем.
  - Адаптація діалогів до дій та виборів гравця.
- **Просунуті NPC з базами знань**:
  - NPC з доступом до всього лору гри.
  - Персонажі, що відповідають у відповідності зі своїм характером та знаннями.
  - Інтелектуальні противники, що адаптуються до стратегії гравця.

- **Приклад архітектури для ігрових діалогів**:

```
class GameDialogueRAG:
    def __init__(self, lore_database, character_profiles, dialogue_history):
        self.lore_db = self._index_lore(lore_database)
        self.character_profiles = character_profiles
        self.dialogue_history = dialogue_history
        self.llm = initialize_game_llm()

    def _index_lore(self, lore_database):
        # Індексує ігрову базу знань та лор
        # ...
        return indexed_lore

    def generate_response(self, character_id, player_input, current_game_state):
        # Отримання профілю персонажа
        character = self.character_profiles[character_id]

        # Пошук релевантної інформації у базі лору
        relevant_lore = self.lore_db.search(
            query=player_input,
            filters={"relevant_to": character_id, "game_stage": current_game_state["stage"]}
        )

        # Отримання релевантної історії діалогів
        dialogue_context = self.dialogue_history.get_recent(
            character_id=character_id,
            limit=5
        )

        # Формування промпту для LLM
        prompt = f"""
        Ти - персонаж {character['name']} в грі.

        Твій профіль:
        {character['description']}
        {character['personality']}
        {character['motivation']}

        Твої знання про гру:
        {relevant_lore}

        Попередні діалоги з гравцем:
        {dialogue_context}

        Поточний стан гри:
        {current_game_state['description']}

        Гравець говорить: {player_input}

        Відповідай в характері, використовуючи відповідний тон та знання, доступні твоєму персо
        Не розкривай інформацію, яку твій персонаж не повинен знати.
        """

        # Генерація відповіді
        response = self.llm(prompt)
```

```
        # Оновлення історії діалогів
        self.dialogue_history.add(
            character_id=character_id,
            player_input=player_input,
            character_response=response
        )

        return response
```

2. Автоматична генерація та адаптація ігрового контенту

- **Динамічне створення квестів та завдань**:
  - Генерація завдань, узгоджених з ігровим світом та історією гравця.
  - Адаптація складності та типу завдань до стилю гри та переваг гравця.
  - Забезпечення наративної цілісності та зв'язності з основним сюжетом.
- **Процедурна генерація ігрового контенту**:
  - Створення описів локацій та об'єктів на основі ігрової бази знань.
  - Генерація когерентних історій та передісторій персонажів.
  - Адаптивні наративні елементи, що розвиваються з часом.
- **Підтримка користувацького контенту**:
  - Інтеграція користувацьких модів та контенту в існуючий світ гри.
  - Перевірка узгодженості та балансу користувацьких доповнень.
  - Допомога гравцям у створенні контенту, що відповідає лору гри.

Інтеграційні та технічні аспекти

1. Архітектурні патерни для RAG у продуктах

- **Мікросервісна архітектура**:
  - Окремі сервіси для індексації, пошуку та генерації.
  - Масштабування окремих компонентів залежно від навантаження.
  - Гнучкість у виборі технологій для різних компонентів.
- **Асинхронна обробка**:
  - Фонова індексація та оновлення бази знань.
  - Черги повідомлень для обробки запитів.
  - Реактивні інтерфейси для взаємодії з користувачем.
- **Багаторівнева кешування**:
  - Кешування популярних запитів та відповідей.
  - Кешування результатів пошуку та вбудовувань.

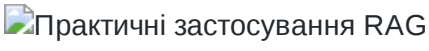
- Гарячі та холодні шляхи обробки запитів.

2. Практичні міркування щодо впровадження

- **Вибір між хмарними та локальними рішеннями**:
  - Готові хмарні сервіси vs. власна інфраструктура.
  - Міркування щодо конфіденційності та безпеки даних.
  - Оцінка загальної вартості володіння (TCO).
- **Поетапне впровадження**:
  - Початок з простих сценаріїв використання та поступове розширення.
  - А/Б тестування для оцінки впливу на користувачів.
  - Збір зворотного зв'язку та ітеративне покращення.
- **Обчислювальні вимоги та оптимізація**:
  - Баланс між якістю результатів та обчислювальними витратами.
  - Оптимізація конвеєрів обробки для зменшення затримки.
  - Стратегії для обробки пікових навантажень.

Майбутні тенденції та розвиток RAG

- **Мультимодальні RAG-системи**:
  - Інтеграція текстової, візуальної та аудіо інформації.
  - Пошук за різними модальностями.
  - Генерація мультимодального контенту на основі різнорідних джерел.
- **Автономні агенти на основі RAG**:
  - RAG як основа для довгострокової пам'яті агентів.
  - Проактивне оновлення та організація бази знань.
  - Здатність до самонавчання та адаптації.
- **Гібридні підходи та нові архітектури**:
  - Комбінація RAG з fine-tuning та іншими методами адаптації.
  - Інтеграція з символьними системами міркувань.
  - Нові архітектури для більш ефективного пошуку та синтезу інформації.



Оцінка ефективності RAG-систем

Оцінка ефективності RAG-систем є критично важливою для їх розробки та вдосконалення. Розглянемо основні підходи та метрики для оцінки різних аспектів RAG.

Метрики оцінки RAG-відповідей

1. Метрики якості відповідей

- **Точність відповіді (Answer Correctness)**:
  - Відповідність згенерованої відповіді фактичній інформації.
  - Оцінка експертами або порівняння з еталонними відповідями.
- **Фактична точність (Factual Accuracy)**:
  - Відсутність фактичних помилок та "галюцинацій".
  - Узгодженість відповідей із наданими джерелами.
- **Релевантність відповідей (Answer Relevance)**:
  - Відповідність відповіді поставленому запитанню.
  - Повнота охоплення всіх аспектів запиту.

2. Загальні метрики оцінки

- **Стандартні IR-метрики**:
  - **Точність (Precision)**: Частка релевантних документів серед знайдених.
  - **Повнота (Recall)**: Частка знайдених релевантних документів серед усіх релевантних.
  - **F1-score**: Гармонічне середнє між точністю та повнотою.
  - **MAP (Mean Average Precision)**: Середня точність на різних рівнях повноти.
  - **NDCG (Normalized Discounted Cumulative Gain)**: Оцінює не лише релевантність, але й порядок результатів.
- **Текстові метрики**:
  - **BLEU, ROUGE**: Оцінка схожості згенерованого тексту з еталонними відповідями.
  - **BERTScore**: Семантична оцінка, що використовує вбудовування.
  - **METEOR**: Враховує синоніми та перефразування.
- **Метрики для факт-чекінгу**:
  - **Точність атрибуції**: Частка правильно атрибутованих тверджень.
  - **Фактична консистентність**: Узгодженість фактів у відповіді з джерелами.
  - **Кількість "галюцинацій"**: Частка тверджень, які не підтверджуються контекстом.

Висновки

- RAG-системи пропонують ефективні рішення для пошуку та генерації інформації.
- Оцінка ефективності RAG-систем є важливою для їх вдосконалення.
- Метрики оцінки дозволяють порівнювати різні підходи та визначати найкращі рішення.