НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету інформаційних технологій	**Завідувач кафедри комп'ютерних наук**
_____ ___Ігор Болбот____	_____ ___Белла Голуб___
(підпис) (ПІБ)	(підпис) (ПІБ)
"___" _____ 2025 р.	"___" _____ 2025 р.

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему ____Програмне забезпечення інтелектуальної системи пошуку інформації_____

Спеціальність_____121 Інженерія програмного забезпечення_____
(код і найменування)

Освітня програма_____Програмне забезпечення інформаційних систем_____
(назва)

Орієнтація освітньої програми ____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми

___доц.к.ф.-м.н_____ _____ _____Віктор Киричено__
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник магістерської кваліфікаційної роботи

____д.т.н, проф._____ _____ ____Цюцюра М. І._____
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ _____Шевчун Д. В._____
(підпис) (ПІБ студента)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) __інформаційних технологій_____

ЗАТВЕРДЖУЮ
Завідувач кафедри__комп'ютерних наук_
___доцент, к.т.н._____ _____ _Голуб Б. Л._
(науковий ступінь, вчене звання) (підпис) (ПІБ)
"_10_" ___листопада_____2024 року

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

_____Шевчун Денис Валентинович_____
(прізвище, ім'я, по батькові)

Спеціальність 121 «Інженерія програмного забезпечення»
(код і назва)

Освітня програма ____Програмне забезпечення інформаційних систем
(назва)

Орієнтація освітньої програми_____освітньо-професійна_____

Тема магістерської кваліфікаційної роботи Програмне забезпечення інтелектуальної системи пошуку інформації
затверджена наказом ректора НУБіП України від "_1_" _листопада____2024р. №1963 «С»
Термін подання завершеної роботи на кафедру __14.11.2025_____
(рік, місяць, число)

Вихідні дані до магістерської кваліфікаційної роботи: Класичні методичні джерела з інформаційного пошуку (BM25, nDCG/MRR/Recall), офіційна документація OpenSearch/FastAPI/FAISS.

Перелік питань, що підлягають дослідженню:

1.Які граничні значення продуктивності досяжні за різних конфігурацій і що дає кешування популярних запитів?

2. Яка чутливість системи до домішок мови та вибору моделі ембеддингів?

3. Які підходи до дедуплікації корпусу найбільше впливають на якість пошуку та розмір індексу?

4. Який внесок переранжування cross-encoder у топ-10 і яка його операційна ціна?

Перелік графічного матеріалу (за потреби)

Дата видачі завдання "__1__" __листопада_____2024_ р.

Керівник магістерської кваліфікаційної роботи _____ ___Цюцюра М. І.____
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання _____ _____Шевчун Д. В._____
(підпис) (прізвище та ініціали студента)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

IT - інформаційні технології

ОС - операційна система

API - прикладний програмний інтерфейс

IDE - інтегроване середовище розробки

ПЗ - програмне забезпечення

UI - користувацький інтерфейс

URL - уніфікований локатор ресурсів

ООП - об’єктно орієнтоване програмування

HTTP - протокол передачі даних

RSS - технологія для “Really Simple Syndication”

IR - пошук інформації

“Golden set” - спеціальний набір даних для подальшого оброблення ШІ

ШІ - штучний інтелект

CORS - політика доступу з браузера

BM25 - алгоритм ранжування релевантності запитів

kNN - метод k-найближчих сусідів

HTML - мова розмітки гіпертексту

SaaS - програмне забезпечення як послуга

UI - користувацький інтерфейс

QA - забезпечення якості

UML - уніфікована мова моделювання

JSON - запис об’єктів від JavaScript

nDCG@10 - нормалізований дисконтований кумулятивний прибуток

MRR@10 - середній взаємний ранг

Recall@10 - частка релевантних документів

ВСТУП

Зростання обсягів текстових даних, їх багатомовність, а також потреба у швидкому доступі до технічних знань під час інформаційної ери робить інтелектуальні системи пошуку необхідною складовою для сучасних рішень в сфері ІТ. Безліч рішень до проблем можливо знайти в мережі Інтернет без зайвої потреби створення нового і, можливо, не найкращого рішення.

Хоча й класичний лексичний пошук (BM25) чудово відпрацьовує для точних збігів, але поступається недоліком в ролі семантичних запитів. В той же час векторні підходи потребують значних ресурсів і можуть надати неточні, “розмиті” результати. Актуальним підходом є поєднання цих підходів у гібридну систему з прозорими метриками якості, контрольованою затримкою та дотриманням вимог приватності та авторського права. Практично, така гібридизація дозволить швидке розгортання на доступному стеку для навчальних цілей або ж навіть невеликих промислових стартапів.

Об’єктом цього дослідження процесів та технологій інтелектуальних систем пошуку інформації, а саме їх якість збору й індексування даних і формування релевантної відповіді користувачу на запит. Предметом дослідження являються програмні компоненти та методи, що забезпечують цей пошук: архітектурні рішення ПЗ, методи та програмні компоненти, що забезпечують релевантний і ефективний пошук (лексичні та векторні моделі), структуризація індексів, а також метрики оцінювання якості і продуктивності пошуку.

Мета досліду полягає у розробці та експериментальному оцінюванні програмного забезпечення інтелектуальної системи пошуку, що повинна забезпечувати високу релевантність відповідей за прийнятної затримки з підтримкою багатомовності (українська/англійська мови) і прозорої подачі актуальних результатів. Програма повинна містити гібридний тип пошуку.

Зміст поставлених завдань полягає у:

- Проведенні аналізу вимог і формуванню критеріїв якості;
- Проектуванні архітектури;
- Реалізації обробки пошукових запитів та індексації корпусу технічних текстів з дедуплікацією, визначенням мови й нормалізацією метаданих;
- Реалізації пошукового пайплайну;
- Розробці веб-інтерфейсу з фасетами (сайт/мова/дата), історією запитів й підказками;
- Формуванням контрольних запитів і релевантних документів;
- Виконанні налаштування параметрів під “якість-затримка”;
- Забезпеченні дотримання вимог приватності/авторського права (зберігання оригінальних URL, відображення уривків), журналювання;
- Узагальненні результатів, проведенні аналізу загроз валідності, формулюванні рекомендації для масштабування.

У дослідженні застосовано комплекс методів побудови та оцінювання інтелектуальної системи пошуку інформації. Основу складатиме лексичний пошук (вищезазначений BM25) і, за потреби, мовні моделі зі згладжуванням. Для семантичної релевантності результатів пошуку використовується гібридний підхід, що поєднує лексичний та векторний пошуки. Якість системи буде оцінюватись офлайн-метриками nDCG@k, MRR@k та Recall@k з перевіркою статистичної значущості. Продуктивність же буде вимірюватись через латентність та навантажувальним тестуванням. Інженерні практики включатимуть експериментальний дизайн з чіткими базовими лініями, контроль версій даних і моделей, журналювання і спостережуваність, що дозволить забезпечити відтворюваність і прозорість отриманих результатів.

Апробація програмного додатку. Результати апробовані у вигляді тез доповідей та постеру на Науковій Інтернет-конференції Національного університету біоресурсів і природокористування України.

Структура записки. Магістерська робота складається зі вступу, чотирьох розділів, висновків, переліку використаних джерел і додатків. Загальний обсяг роботи становить 62 сторінки. Список використаних джерел налічує 15 найменувань.

РОЗДІЛ 1. СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Опис предметної області

Предметна область охоплює в себе процеси збирання, підготовки, індексації та пошуку текстових документів з використанням різнорідних джерел у відкритому доступі в мережі Інтернет. Основна ціль - надання користувачеві релевантних до його пошукового запиту результатів у зручному та інтуїтивному інтерфейсі. Однак, інтелектуальна система пошуку інформації має одну, ключову відмінність від звичайного “рядка пошуку”, а саме використання інтелектуальних методів, таких як лексичні та семантичні моделі, переранжування, тощо. За рахунок цих характеристик, якість відповіді підвищується кардинально, оскільки система має розуміння змісту запиту та документів, підтримує багатомовність і структурованість метаданих. Багатомовність в цьому випадку дає змогу користувачеві знайти більш бажані та точні результати на його пошуковий запит: за володінням англійської мови, користувач без проблем матиме змогу знайти оригінальні й найбільш релевантні до його пошукового запиту результати, що кардинально скорочує час на пошук необхідної інформації.

Однак без джерел пошукова система банально не є робочою, тому обговоримо джерела та типи даних як наступний пункт. Основними джерелами були взяті RSS/Atom-стрічки технічних блогів і сторінки з повними статтями. Документи представлені в різних форматах: HTML, Markdown, інколи навіть в класичному форматі PDF! Для кожного з цих документів важливі метадані для роботи з ними: назва, URL, сайт або видавець, автор(и), дата публікації, теми або мітки/теги, мова. Без цих метаданих пошуковий рушій не матиме змогу працювати з цими документами. Не менш важливими є службові атрибути, такі

як контрольна сума вмісту і час обробки запиту. Знаходження більш актуальних до пошукового запиту результатів за короткий проміжок часу є пріоритетом №1 для кожної пошукової системи, в іншому випадку ж користувач банально перейде на більш оптимізовану пошукову систему для пошуку необхідної йому інформації.

Також не менш важливою є коротка, однак зрозуміла подача інформації при видачі результату користувачеві. Як приклад, рекомендую розглянути структуру подачі результатів від найпопулярнішого в світі пошукового сервісу Google.

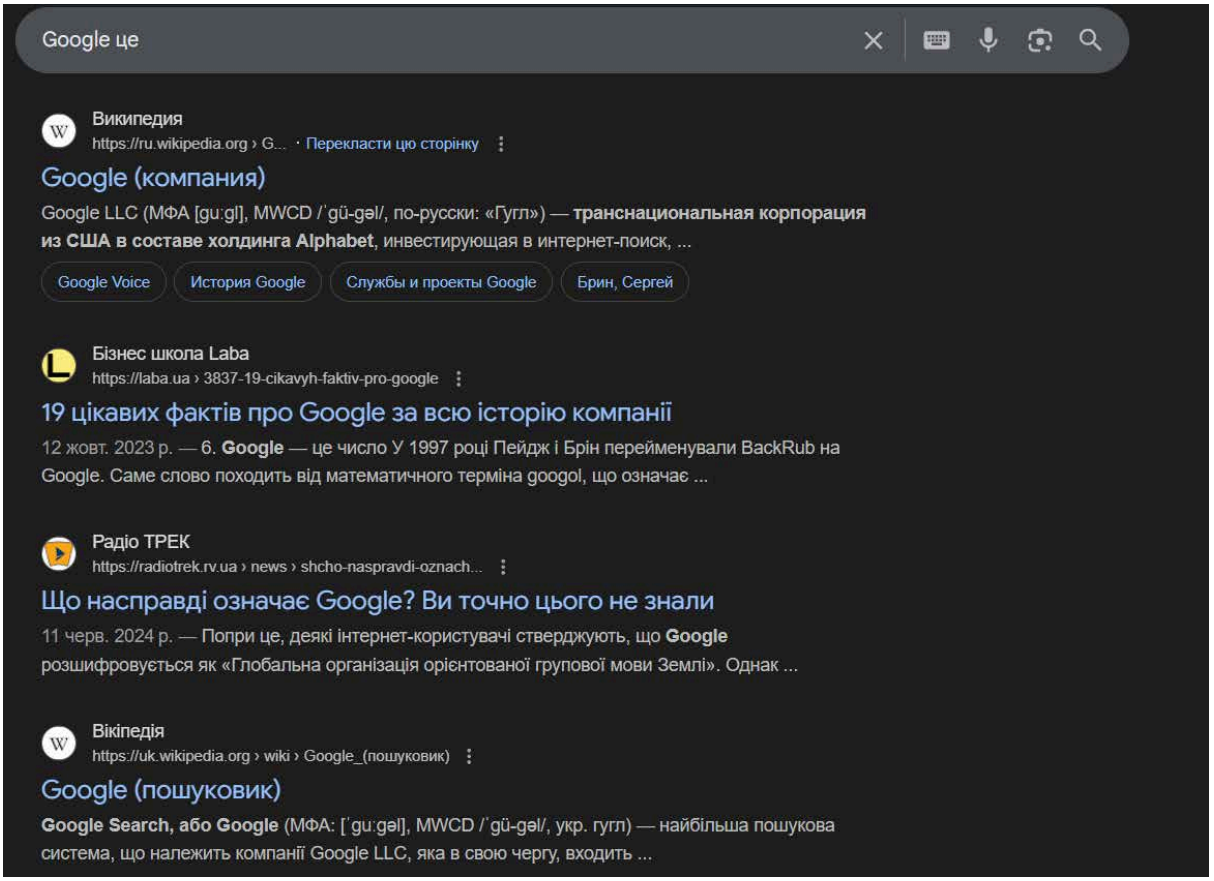

Рис. 1.1 Структура подачі результатів на запит пошукової системи Google

Подача інформації є короткою, однак користувачеві видно ключові пункти: сайт, URL, заголовок, і уривок з сайту, який пошукова система вважає найбільш актуальним до запиту. Це допомагає зрозуміти, що конкретно користувач потребує в першу чергу при пошуку інформації через пошукові сервіси - контекст.

Зазвичай тексти мають сталу структуру, таку як заголовки, підзаголовки, кодові блоки, медіафайли, тощо. Це спрощує виділення основного контенту та формування коротких уривків, що допоможе розробити структуру подачі результатів, аналогічну до нашого конкурента.

Обговоривши подачу контексту, повернімося до користувачів, а саме до цільової аудиторії розробленого програмного забезпечення. Цільовими користувачами в першу чергу вважаються студенти, інженери-практики, дослідники і техрайтери. Орієнтуючись на більш конкретну цільову аудиторію, є можливість зрозуміти, які запити та результати варто очікувати. Орієнтовно, можливі такі сценарії взаємодії з пошуковим сервісом:

- Інформаційний пошук “how-to” (як зробити) - зазвичай інтерпретується як пошук порад або гайдів на те, як користуватись функціоналом програмного додатку або реалізації компоненту;
- Навігаційний пошук за конкретним джерелом - напр. “статті Chrome Dev за 2024 рік”;
- Дослідницький пошук з неформальним запитом - напр. “Gherkin у тестуванні”;
- Порівняльні запити - порівняння двох програм або методик, для дослідження переваг, недоліків і актуальності кожної.

Враховуючи вищевказані сценарії, тепер є можливість в подальшому оцінити якість роботи пошуку точніше. Конкретно повинні бути враховані можливість підсвічування збігів, фільтрації за сайтом, датою, мовою, а також посилання на джерело. Таким чином, це забезпечить надання користувачеві актуальної інформації без потреби змінювати пошуковий запит або переглядати весь список наданих результатів для знаходження одного або декількох необхідних.

Враховуючи всі вищеописані нюанси, можна дійти до висновку, що якщо йде мова про пошуковий сервіс, то очікуваним кінцевим результатом повинна бути програма, здатна надавати користувачеві точну й актуальну інформацію за

короткий проміжок часу. Повертаючись до найпопулярнішого пошукового сервісу на сьогодення - Google - можна помітити, що компанія навіть на теперішній її стан все ще шукає найбільш ефективний і швидкий підхід надання інформації користувачеві до цих пір. Як приклад, ми маємо Google Gemini AI - нейромережа, розроблена в лютому 2024 року, що здатна аналізувати запит користувача й не тільки надати можливі джерела, але й одразу стислу та зазвичай актуальну відповідь на запит. До сьогодення компанія покращує якість роботи Gemini та актуальність наданих нейромережею запитів.

Ключовими викликами предметної області являються:

- Різнорідність форматів і якість розмітки HTML;
- Багатомовність і технічна лексика з абревіатурами;
- Дублікати результатів;
- Баланс між якістю і латентністю;
- Потреба у відтворюваних експериментах і прозорих порівняннях.

В цей же час вирішення цих викликів потребуватиме точне оброблювання пошукового запиту, чітку схему індексації, зважену комбінацію лексичних і семантичних методів і дисципліну в рамках інженерних практик.

Предметна область інтелектуального пошуку для технічних текстів поєднує керований збір відкритих даних, їх якісну підготовку в ролі очищення, дедуплікації та взаємодії з метаданими, поетапне індексування з акцентом на BM25 як надійну основу. Семантичні методи і переранжування дають помітний приріст якості для складних запитів, але додають витрати, тому доцільно застосовувати їх як надбудову над лексичним ядром із чітким контролем затримки. Для цільових користувачів (інженери, студенти, дослідники) ключові очікування - релевантність, швидка відповідь, прозорість (підсвічені уривки і метадані) та прості фасети за сайтом, датою і мовою. Система має спиратися на відтворювані експерименти, базові лінії й порівняння, що дозволять обґрунтовано балансувати між якістю і продуктивністю.

Практично це означає реалістичний шлях впровадження. Спершу - реалізація точної обробки пошукового запиту і BM25 з підсвіткою та фільтрами. Далі - точкове додавання гібридного пошуку й переранжера за потреби. Паралельно - вимірювання якості та швидкодії, логування й дотримання правових обмежень (цитування з посиланням). Такий підхід надасть змогу мінімізувати ризики, забезпечує прозорість і масштабованість, а також надасть чітку основу для подальших наукових експериментів і промислового розвитку.

1.2. Аналіз наявних рішень

Для аналізу рішень щодо розробки програмного забезпечення інтелектуальної системи пошуку інформації, перш за все варто оцінити вимоги, які потребуватиме це програмне забезпечення.

В якості функціональних вимог, система повинна мати:

- Можливість обробляти нові документи з фідів та сторінок для забезпечення актуальності інформації;
- Очищувати HTML і виділяти основний текст для подачі користувачеві ключових слів;
- Нормалізувати і зберігати метадані;
- Будувати індекси для лексичного і векторного пошуків;
- Приймати запити через API;
- Формувати список кандидатів на запит і виконувати переранжування;
- Подавати користувачеві результати зі сніпетами (частинками тексту), підсвічуванням і фасетними фільтрами.

За можливості, варто реалізувати механізми підказок, у разі якщо користувач зробив потенційну помилку в запиті, історії запитів для можливості

користувачеві повертатись до його минулих запитів без зайвого витрачання часу і логування взаємодій для поліпшення якості.

Обговоривши функціональні вимоги, не менш важливим пунктом являються нефункціональні вимоги. До них входять:

- Релевантність результатів;
- Низька затримка;
- Масштабованість;
- Відмовостійкість - резерви індексів;
- Прозорість результатів - уривки тексту з посиланням;
- Багатомовність - враховуватиметься підтримка української та англійської мов у запитах і документах.

Говорячи про мови, варто також враховувати мовні та семантичні аспекти. Запити можуть і будуть короткими, міститимуть абревіатури, назви продуктів або їх версій. Деякі з документів можуть містити код або ж псевдокод (нефункціональний скрипт, який описує структуру або роботу функціонального), або ж банально відрізняються лексикою та стилем оформлення. Для коректного пошуку будуть необхідні токенізація без агресивних стоп-списків, лематизація, виявлення мови, а також можливість семантичного співставлення. Це допоможе системі не “втрачати” тех-терміни, не надавати не актуальні результати через просту неможливість відмінювання слова, або якщо ключові слова у запиті й документі не збігаються дослівно.

На етапі виконання запиту система повинна нормалізувати запит, формувати кандидатів з лексичного індексу і, за наявності з векторного, зливати списки. Необхідно також, щоб система переранжовувала верхню частину списку, генерувала сніпети з підсвічуванням ключових уривків, застосовувала фасетні фільтри й повертала результат на запит. Таким чином, розміри топ-k, параметри зливання й переранжування створюватимуть компроміс між якістю і часом відповіді на запит.

Індексація повинна включати інвертований індекс з підтримкою для заголовків і фільтрів, а також рекомендується підтримка векторного індексу для ембеддинґів фрагментів. Вміст буде зберігатись у сховищі документів з унікальними ідентифікаторами, контрольними сумами та часовими мітками обробки пошукови запитів. Дедуплікація за контент-хешами і приблизними методами буде знижувати шум і повтори.

Стосовно джерел, звідки будуть обиратися результати на запитів, то використовуватимуться виключно відкриті джерела з повагою до ліцензій. У видачі результату, завжди повинні бути короткі уривки з посиланням на оригінал. Персональні дані включатись не будуть, журнали запитів анонімізуються. Система загалом повинна бути прозорою: користувач повинен легко бачити й розуміти, чому документ показано завдяки підсвіченим збігами і видимим метаданим.

Орієнтуючись на вимоги, розглянемо тепер аналіз наявних рішень в плані розробки програмного забезпечення інтелектуальної системи пошуку інформації.

В якості класичних лексичних рушіїв маємо стандарти для повнотекстового пошуку, такі як ElasticSearch або OpenSearch, що включають в себе інвертований індекс, BM25, підсвічування, фільтри, агрегації, доволі розширене API і масштабування. Elastic і OpenSearch підтримують векторні поля, однак OpenSearch також має k-NN плагін - плагін, що використовує метод k-найближчих сусідів (k-Nearest Neighbor). Цей метод досліджує мітки обраної кількості точок даних, що оточують цільову точку, щоб зробити прогноз щодо класу, до якого належить ця точка даних. Концептуально алгоритм вважається простим на думку експертів, але незважаючи на це є дуже потужним. З цих причин він є одним із найпопулярніших алгоритмів машинного навчання. Apache Solr являється також непоганим варіантом в якості лексичного рушію, однак має менш кращу підтримку “хмарного” збереження на відміну від ElasticSearch та OpenSearch. Більш легковаговими альтернативами являються

Meilisearch і Typesense. Ці рушії є дуже простими в розгортанні, швидкі для автопідказок, однак є обмеженими в складних запитах.

В ролі локальних бібліотек можна використовувати такі бібліотеки як FAISS від Facebook Meta, або ж HNSWlib чи Annoy. FAISS, або ж facebooksearch має високу продуктивність, однак потребує ручне управління збереження сховища. У випадку ж з Annoy або HNSWlib, то вони є дуже простими бібліотеками у використанні, однак є менш гнучкими, у порівнянні з facebooksearch.

В якості векторних баз даних можна використовувати Milvus, Qdrant, Weavitate. Всі сервіси мають зручне API. Найпростішим сервісом є pgvector, який являється доповненням до PostgreSQL для векторів. Також хорошими альтернативами є керований сервіс Pinecone, або ж розширені платформи, такі як Chroma або ж Vespa. Серед них Vespa має лексичні й векторні індекси та складну ранжувальну логіку

При потребі моделей переранжування при розробці програмного забезпечення, то наявні такі моделі:

- SBERT - підтримка багатомовності і простота у використанні;
- Instructor - краще узгодження з інформаційними запитами;
- MiniLM, monoT5, ColBERT - підвищують якість перших у списку результатів, але додають затримку при обробці запиту.

Для покращення якості пошуку, варто покращити якість інгесту, або ж оброблення пошукового запиту. В цьому випадку найкращими варіантами вважаються feedparser, який є простим та зрозумілим скриптом для Python, trafilatura в якості текстового екстрактору, що дозволить краще обробляти пошуковий запит і SimHash - алгоритм для швидкого оцінювання схожості між двома наборами даних шляхом створення хеш-значень, що дозволить швидко і точно знаходити дублікати і, відповідно, надавати точні пошукові результати для користувача.

Варто враховувати також не менш важливий пункт при створенні пошукового сервісу - а саме якість пошуку. Для цього були створенні інструменти для оцінювання метрик якості пошуку. Серед них найпопулярніші:

- BEIR, MS MARCO, TREC - стандартні колекції для валідації методик;
- ir_measures, trec_eval - пакети для оцінення метрик якості IR;
- PyTerrier - дозволяє проводити оцінювання пайплайнів й метрик для IR.

Оцінивши наявні бібліотеки, пакети та інструменти для створення пошукового сервісу, маємо змогу обрати найбільш ефективний інструментарій для ПЗ, таким чином надаючи керований компроміс між якістю та складністю розробки. Це дозволить створити навіть доволі простий пошуковий сервіс, який при цьому має надійну лексичну базу, опціональні семантичні поліпшення і при цьому не маючи громіздкої інфраструктури, що дозволить в гнучке покращення програмного продукту в майбутньому, зважаючи на зворотний зв’язок від користувачів.

Для оптимальної основи при розробці програмного забезпечення інтелектуальної системи пошуку інформації вистачить зрілого лексичного пошуку, підсвітку збігів, фасети і просте масштабування - усе, що потрібно для якісного продукту з мінімальними затратами в рамках ресурсів без зайвої інфраструктурної ваги. Легковагові рушії на кшталт Meilisearch/Typesense зручні для демонстрацій, але швидко впираються в обмеження гнучкості запитів і аналітики. Семантичний шар доречно додавати виключно як надбудову: в цьому випадку ідеально підходять pgvector у Postgres або локальний FAISS з простим ф’южном поверх алгоритму ранжування релевантності запитів.

Також є можливість додати керовані пошукові сервіси типу Software as a Service (SaaS), такі як Algolia, Azure Cognitive Search або ж Elastic Cloud. Ці сервіси надають швидкий запуск пошукового рушія, авто-підказки, аналітику і

масштабування. Однак єдине їх головне обмеження - вартість і менша контрольованість дослідження ефективності пошуку, що не робить їх чудовою опцією для розробки простого, але ефективного програмного продукту.

1.3. Постанова завдання щодо проведення дослідження

Стрімке зростання масивів технічної документації, блогів та статей ускладнює швидкий доступ до релевантних знань для користувачів мережі Інтернет. Класичні інвертовані індекси (BM25) забезпечують високу точність за наявності лексичних збігів, але є обмеженими у випадку синонімії або ж перефразувань. В той же час векторні підходи (ембеддинґи + kNN) підсилюють семантичну релевантність, однак потребують доволі ретельного налаштування й контролю латентності. Звідси випливає основна ціль роботи: створити інженерно-практичний прототип інтелектуального пошуку, який збалансовує якість видачі та швидкодію і спирається на відтворювані експерименти.

В рамках цього дослідження, необхідно розробити й експериментально оцінити програмне забезпечення інтелектуальної системи пошуку інформації. Програмне забезпечення буде поєднувати в собі лексичні методи, такі як BM25, з опційною семантичною надбудовою, наприклад ембеддинги, k-NN і переранжування cross-encoder. Це забезпечить прозорий пошук та прийнятну латентність.

В рамках дослідження також буде проведений аналіз процесів й програмних засобів інтелектуального пошуку текстової інформації: обробка пошукового запиту, індексація, ранжування, візуалізація результатів на стороні UI для подальшого оцінювання якості та продуктивності програмного забезпечення.

Буде реалізовано поєднання лексичних і семантичних методів у єдиному пайплайні: архітектурні рішення ПЗ, моделі, стратегії злиття, метрики якості та продуктивності.

Граничними межами в рамках дослідження були поставлені використання публічних технічних блогів або статтей**.** В якості розширення скоупу програмного забезпечення, програмою повинні підтримуватись робота з англійською і українською мовами. Обсяг документів для роботи орієнтовно повинен складати від десятків до сотень тисяч. В кінцевому результаті програмне забезпечення повинно мати інтерактивний пошук типу top-N ($N \leq 50$) з фасетами (сайт, мова, дата).

В ролі вхідних даних буде використовуватись 15-30 технічних джерел з публічним доступом.

В якості ризиків варто враховувати:

- “Шум” або ж дублікацію інформації в технічних джерелах;
- Нестабільність джерел або ж фідів;
- Латентність при оновленні актуальних результатів;
- Перенавчання на валідації;
- Правові обмеження в рамках ліцензування інформації.

Для усунення або ж мінімізування цих ризиків, будуть впроваджені такі шляхи:

- Ретельна дедуплікація через такі алгоритми як SimHash, фільтри за довжиною і якістю тексту;
- Кешування, список альтернативних джерел, ручне оновлення посилань;
- Зменшення максимуму виведення результатів до top-100, попереднє фільтрування;
- Показ коротких уривків або ж цитування джерел.

В якості очікуваного результату - працездатний прототип в ролі програмного забезпечення інтелектуальної системи пошуку інформації з BM25-основою та опційною семантичною надбудовою і наявність експериментальних докази впливу гібридизації та переранжування на якість при виведенні топ-10 або топ-20 результатів.

Були поставлені критерії приймання програмного забезпечення, що дозволить розробити програму, яка відповідає кінцевому результату.

Функціональні критерії:

- Обробка пошукових запитів;
- Індексація;
- Пошук з підсвічуванням та фасетами.

Вимірювальні критерії:

- Наявність golden set;
- Наявність вітів nDCG/MRR/Recall.

Критерії продуктивності:

- Підтверджені метрики p50/p95 при навантажувальному тестуванні.

Етичні критерії:

- Коректне цитування;
- Анонімізовані журнали.

Під час розробки програмного забезпечення і особливо при отриманні кінцевого продукту варто враховувати такі обмеження:

- Відсутність приватних/закритих джерел;
- Потенційні неточності у визначенні мови;
- Обмежений об’єм вхідних даних;
- Відсутність повноцінного QA-модуля.

РОЗДІЛ 2. МОДЕЛЮВАННЯ СИСТЕМИ

2.1. Діаграма прецедентів

В рамках цього розділу була поставлена задача описати процес роботи програмного забезпечення інтелектуальної системи пошуку інформації за допомогою моделювання процесів системи. Конкретно в цьому розділі, буде розглянуто діаграму прецедентів.

Діаграма прецедентів - в UML, діаграма, в якій зображено відношення між акторами і прецедентами в системі. Діаграма прецедентів повинна показувати різні варіанти використання з різними типами користувачів системи, зазвичай також супроводжується іншими типами діаграм (які будуть розглянуті в інших підрозділах).

Діаграма прецедентів складається з:

- Акторів - зовнішні ролі, які описуються в діаграмі як користувачі системи, підсистеми або ж служби;
- Прецедентів - сценарії взаємодії з системою;
- Зв’язків.

Мета розробки і демонстрації діаграми прецедентів - узгодження обсягів і меж системи, формуванням вимог у вигляді зрозумілих всім сценаріїв і служить як умовна карта для подальших специфікацій.

Була розроблена діаграма прецедентів для програмного забезпечення інтелектуальної системи пошуку інформації. В цьому підрозділі будуть обговорені актори, прецеденти і зв’язки між ними в детальному обсязі для розуміння природи системи і взаємодій в її рамках.

На рисунку 2.1. зображена діаграма прецедентів:

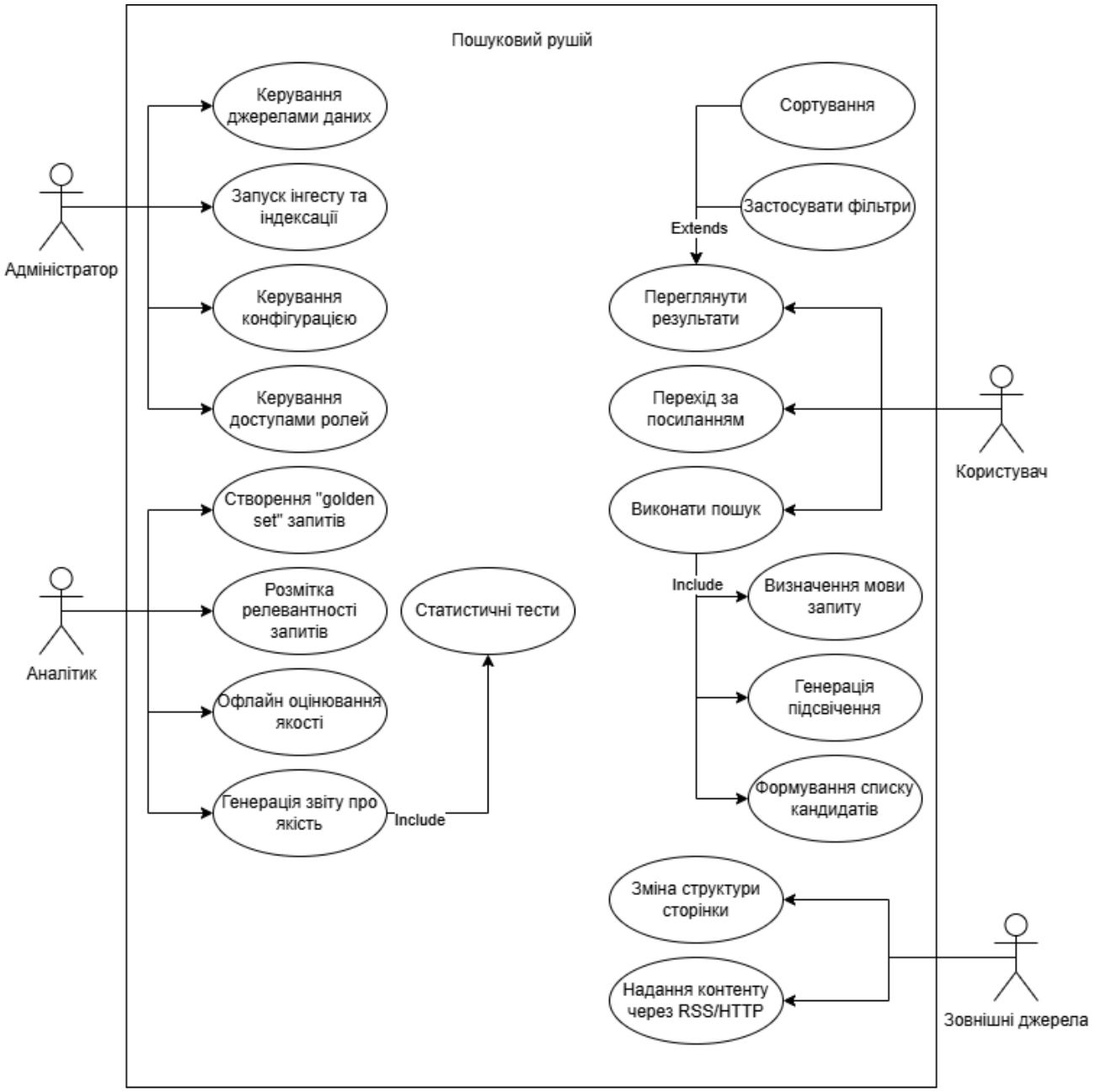

Рис. 2.1 Діаграма прецедентів програмного забезпечення інтелектуальної системи пошуку інформації

Розглянемо ж більш детальніше всі нюанси цієї діаграми для кращого розуміння всієї системи.

Почнімо ж з акторів та прецедентів, пов’язаних з ними. Акторами діаграми є:

- Користувач системи - використовує ПЗ для пошуку інформації, активно взаємодіє з UI;

- Адміністратор системи - керує джерелами, індексами, конфігурацією, правами ролей в системі;
- Аналітик - формує “golden set”, розмічає релевантність, запускає оцінювання якості роботи системи;
- Зовнішні джерела - сайти або ж API, звідки береться контент для використання користувачами.

Обговоривши більш-менш акторів та їх ключову роль при взаємодії з системою, обговорімо тепер прецеденти і зв’язки між ними.

Головними прецедентами Користувача є “Виконати пошук”, “Переглянути результати” і “Перехід за посиланням”. Серед цих прецедентів, у “Виконати пошук” і “Переглянути результати” також наявні зв’язки.

“Виконати пошук” включає в себе:

- Визначення мови запиту;
- Генерація підсвічування;
- Формування списку кандидатів.

“Переглянути результати” розширюється з:

- Сортування;
- Застосування фільтрів.

Таким чином, ми можемо зрозуміти головну ціль Користувача при взаємодії з системою. Користувач шукає інформацію, необхідну йому, через програмне забезпечення. При введенні запиту, система автоматично визначає мову запиту, формує список “кандидатів” на цей запит і генерує підсвічування ключових слів або ж речень, пов’язаних зі запитом. Після цього користувач переглядає надані йому результати, з можливістю сортування результатів за релевантністю і датою, має можливість застосовувати фільтри за сайтом, мовою або ж за конкретним часовим періодом. В кінцевому результаті, при знаходження необхідної йому інформації, користувач переходить за посиланням

на необхідний йому сайт для подальшої роботи з необхідною йому інформацією.

Як зазначено раніше, Адміністратор є керуючою частиною системи, підтримує її роботоспроможність і якість видачі результатів. Прецедентами Адміністратора є:

- Керування джерелами даних;
- Запуск інгесту та індексації;
- Керування конфігурацією;
- Керування доступами ролей.

Адміністратор має можливість додавати або ж оновлювати джерела даних програмного забезпечення, підтримує роботу дедуплікації й індексації джерел до BM25 і, за потреби, до векторного індексу. Орієнтуючись на статистику успішних запитів або ж збоїв в системі, в подальшому керує станом індексів через конфігурацію і керує наявними в системі ролями для забезпечення надійної та безпечної роботи з програмою.

Обговоримо тепер Аналітика. Загалом ключова задача аналітика - збір необхідної інформації для подальшої оцінки системи. Прецедентами Аналітика являються:

- Створення “golden set” запитів;
- Розмітка релевантності запитів;
- Офлайн оцінка якості;
- Генерація звіту про якість.

Прецедент “Генерація звіту про якість” включає в себе:

- Статистичні тести.

Аналітик дозволяє оновлювати і покращувати якість роботи інтелектуальної системи, створюючи та оновлюючи “golden set” запити, дозволяє аналізувати якість ефективності релевантності результатів через

розмітки, робить офлайн-оцінку якості роботоспроможності системи через навантажувальні тести і порівняння конфігурацій і генерує звітність результатів якості, спираючись на статистичні тести.

Таким чином, було розглянуто ключові ролі в системи та їх взаємодію з нею. Завдяки цьому, є загальне розуміння, як програмне забезпечення інтелектуальної системи пошуку інформації повинно працювати, його межі та вимоги, яких повинен дотримуватись кінцевий програмний продукт.

2.2. Діаграма послідовності

Діаграма послідовності - це ще один різновид діаграми в UML. Діаграма послідовності відображає взаємодії об’єктів, впорядкованих за часом. Зокрема, такі діаграми відображаються задіяні об’єкти та послідовність надісланих повідомлень.

На діаграмі послідовності вертикальними лініями, або ж “лініями життя”, показуються різні процеси або об’єкти, які існують одночасно, а горизонтальними стрілками - повідомлення, якими вони обмінюються між собою у порядку їх виникнення. Це дозволяє специфікувати прості сценарії виконання процесів у графічній формі.

Діаграма послідовності повинна складатися з наступних компонентів:

- Актори;
- Лінії життя;
- Повідомлення, що викликаються цими акторами;
- Значення, що повертаються;
- Вказівки на будь-які цикли або області ітерацій.

Діаграми послідовності використовуються для уточнення логіки виконання сценарію, виявлення точок ітерацій, залежностей та потенційних

причин затримок в процесі, синхронізації розуміння роботи процесу між усіма учасниками команди (аналітики, розробники, тестувальники) і служить як основа для специфікації інтерфейсів. Також тестувальники використовують діаграми послідовності для написання інтеграційних тестів.

В рамках розробки програмного продукту було доречно розробити діаграму послідовності для такого сценарію як виконання пошуку. В цьому підрозділі будуть обговорені всі компоненти та їх участь в процесі. На рисунку 2.2 зображена створена діаграма послідовності.

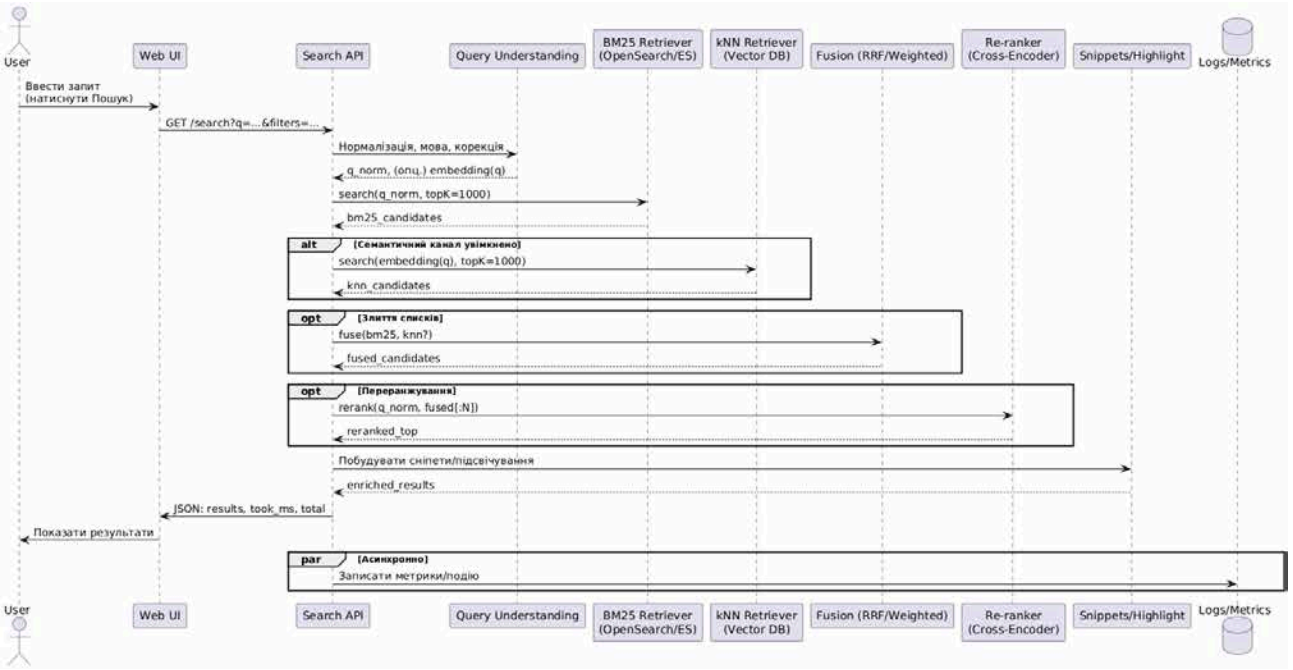

Рис. 2.2 Діаграма послідовності для сценарію “Виконати пошук”

Перш за все, розглянемо “лінії життя”, описані в діаграмі послідовності та їх роль в процесі:

- Користувач (User) - ініціює запит та взаємодіє з інтерфейсом програми;
- Веб-клієнт (Web UI) - відправляє HTTP-запити до API, відображає користувачеві відповіді на інтерфейсі;
- Сервіс пошуку (Search API) - оркеструє кроки пошуку, з’єднує підсистеми, формує результат пошуку на запит;

- Модуль “розуміння запиту” (Query Understanding) - нормалізація запиту, визначення мови, базова орфографічна корекція, побудова структурованого запиту для BM25, обчислення ембеддингу запиту;
- Ретрівер BM25 (BM25 Retriever) - повертає список кандидатів на основі інвертованого індексу;
- Ретрівер kNN (kNN Retriever) - повертає семантичних кандидатів за ембеддингом запиту;
- Модуль злиття (Fusion) - об’єднує списки у єдиний, ранжований список;
- Переранжер (Re-ranker) - уточнює порядок у топ-N знайдених кандидатів;
- Генератор сніпетів (Snippets/Highlight) - формує уривки з підсвіченими збігами;
- Логи/Метрики (Logs/Metrics) - приймає асинхронні події спостережуваності, такі як час, коди, лічильники.

Обговоривши лінії життя та їх участь у процесі виконання сценарію “Виконати пошук”, варто тепер обговорити взаємодію між ними. Основним потоком подій є:

- Користувач - UI - введення запиту, запуск пошуку;
- UI - API - надсилання HTTP-запиту з параметрами, такими як:
  - q - рядок запиту;
  - size - розмір результатів на запит;
  - Опціональні фільтри за сайтом, мовою, проміжком часу.
- API - Query Understanding - синхронний виклик з рядком запиту для його нормалізації, визначення мови, орфографічну корекцію;
- API - Ретрівер BM25 - виклик пошуку з тілом, включаючи фільтри, з поверненням масиву кандидатів та їх метаданими;

- API - Ретрівер kNN - опціональний потік подій, за наявності увімкненого семантичного каналу, виконується пошук з поверненням масиву кандидатів;
- API - Fusion - опціональний потік подій, оформляє переранжирування списку кандидатів у єдиний список;
- API - Snippets/Highlight - витягує фрагменти тексту для сформованого фінального списку, підсвічує збіги з запитом, обрізає до 1-2 фрагментів;
- API - UI - відповідь від JSON з результатами;
- UI - Користувач - рендеринг списку на користувацькому інтерфейсі, з яким користувач може взаємодіяти за допомогою фільтрів або сортування;
    - У випадку фільтрації/сортування, повторюється подія “UI - API” з новими параметрами.

Паралельно до основної серії подій, ми маємо також паралельну подію “API - Logs/Metrics”. Це є не блокуюча подія, пов’язана з відправками записів і не повинна впливати на час відповіді користувачу на його запит.

Передумовами процесу виконання пошуку, описаного в діаграмі послідовності є такі пункти:

- Індекс BM25 створений і наповнений;
- API доступне;
- Векторний індекс синхронізований. Опціонально для загальної роботи виконання процесу;
- Конфігурація таймаутів встановлена.

Післяумовами процесу виконання пошуку повинні бути пункти:

- Користувач отримує результат;
- В логах з’являється запис процесу;

- В разі збоїв семантики або переранжування - зафіксовані прапорці деградації.

Таким чином, завдяки діаграмі послідовності було описано процес виконання сценарію “Виконати пошук”. Завдяки цьому, ми можемо детальніше розглянути процеси, які відбуваються під час надсилання користувачем запиту в систему, а саме які компоненти обробляють цей запит і за яких умов. Не менш важливим пунктом є логування результатів, що в подальшому забезпечить аналітичну оцінку якості роботи програмного забезпечення інтелектуальної системи пошуку інформації.

2.3. Діаграма активності

Діаграма активності - це ще одна діаграма UML, що візуалізує потік дій у системі або бізнес-процесі, подібного до розширеної блок-схеми. Діаграма активності показує, як одна дія може перейти до іншої і може зображувати послідовні, паралельні, розгалужені або ж одночасні потоки. Дії можуть виконуватись користувачами або ж програмними компонентами.

Основними елементами діаграми активності є:

- Дії;
- Потоки керування;
- Вузли розгалуження або злиття;
- Паралелізація;
- Пули;
- Початковий і кінцевий вузли.

Діаграми активності використовують для зрозумілого описування бізнес-процесів або алгоритмів: їх послідовність дій, умови, цикли,

паралельність та обробку винятків; використовуються для узгодження вимог між аналітиками, розробниками і тестувальниками.

На рисунку 2.3 зображена створена діаграма активності для процесу пошуку:

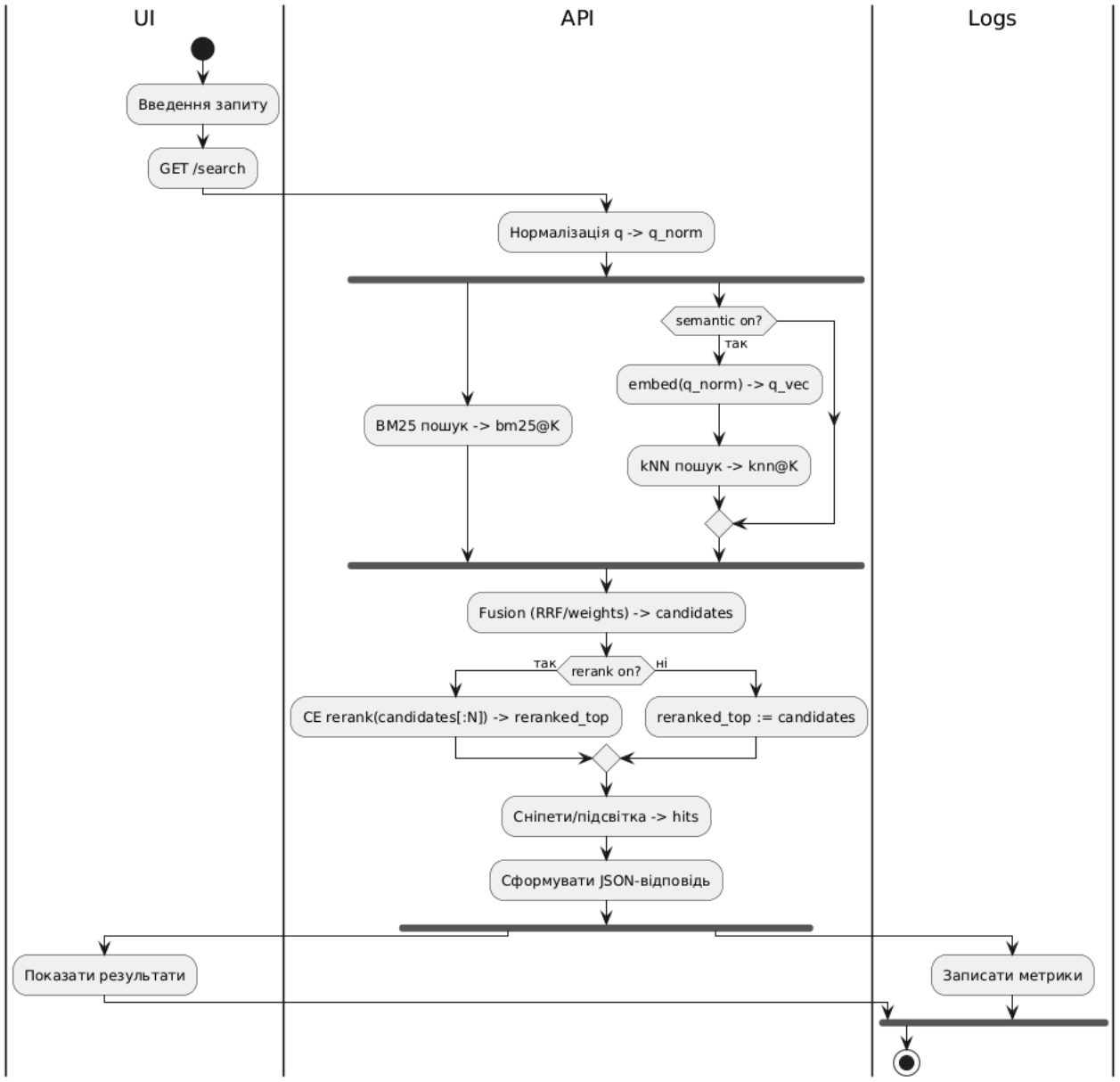

Рис. 2.3 Діаграма активності

Розглянемо більш детальніше компоненти цієї діаграми. Почнемо з пулів, наявних в діаграмі:

- Користувач (UI) - відповідає за введення пошукових запитів, і тим самим відправку HTTP-запитів. В кінці отримує відображення результатів на введений запит;
- API - оркеструє кроки, виклики ретріверів або модулів, формує відповідь на запит в якості JSON-резпонсу;
- BM25 індекс - лексичний пошук з інтегрованим індексом;
- Векторний індекс kNN - опціональний семантичний пошук за ембеддингами;
- Переранжер (Rerank) - опціональне ранжування порядку списку для топ-N;
- Генератор сніпетів - формує уривки й підсвічує збіги з пошуковим запитом;
- Логи/метрики - асинхронне журналювання.

Опишемо тепер потік керування й даних покроково:

- Користувач вводить запит q, задає за власним бажанням фільтри;
- Надсилається запит до API для подальшої обробки даних;
- Система нормалізує запит, визначає мови і робить орфографічну корекцію. Вихідними даними є q_norm;
- В подальшому, в залежності від увімкнення семантичного каналу, система працюватиме або з BM25 та kNN, або виключно з BM25;
    - В гілці BM25 йде обов’язковий лексичний пошук, вхідними даними є q_norm, вихідні дані вказані як bm25@k, тобто сформований список кандидатів;
    - В гілці kNN відбувається семантичний пошук. Будуються ембеддинги і пошук за метрикою cosine/dot. Вихідні дані вказані як knn@k.

- У разі отримання даних виключно від лексичного пошуку, переранжування (Fusion) не є потрібним. У випадку отримання даних від лексичного і семантичного пошуків, відбувається переранжування списків. Як вихідні дані отримуємо reranked_top в обох випадках;
- Після того, як відбулось переранжування або ж система отримала список від BM25, генеруються сніпети, використовуючи значення з reranked_top. Вирізаються 1-2 релевантних фрагментів, які формуються у сніпет. Вихідні дані отримуємо як масив об’єктів hits;
- Після генерації сніпетів, система формує JSON-відповідь;
- На гілку Користувача відправляються результати пошуку на подальшу з ними взаємодію;
- На гілку Логування, не блокуючи користувача, відправляються асинхронно дані для логування подій.

Важливо звернути увагу на вузли рішень, що присутні на діаграмі активності:

- Semantic on/off - за недоступності векторного індексу або якщо він вимкнений, то гілка kNN пропускається, в діагностиці це помічається як semantic=false;
- Rerank on/off - окрім лексичного, за наявності результатів із семантичного пошуку також, ранжування відбувається. В іншому випадку ранжування пропускається, дані йдуть на подальшу обробку.

Можливо помітити паралельність і синхронізацію подій на діаграмі. Конкретно маються на увазі процеси лексичного і семантичного пошуків (BM25 і kNN) та відправка відповіді на UI та логування подій. Пошуки є незалежними від один одного, як було підмічено вище, і виконуються одночасно. У випадку ж з логуванням і відправки відповіді користувачеві, то телеметрія не повинна

впливати на латентність відповіді - події являються також паралельними один до одної.

Завдяки діаграмі активності, можливо було більш детальніше розглянути процес пошуку в системі, його обробку запиту і відправку кінцевого результату користувачеві та для логування.

РОЗДІЛ 3. РОЗРОБКА СИСТЕМИ

3.1. Вибір інструментарію для створення програмного забезпечення

Для розробки будь-якого програмного забезпечення розробник повинен проаналізувати, які він інструменти буде використовувати при розробці проекту. Розуміння інструментарію є дуже важливою частиною під час розробки програмного забезпечення, оскільки розробник без знання над чим і з чим він працює банально не є ефективним розробником.

Почнімо з двох найголовніших аспектів: мови програмування і середовища розробки (IDE). Для розробки програмного забезпечення інтелектуальної системи пошуку інформації було обрано мову програмування Python версії 3.11 або більше. Зумовлено це тим, що новіші версії підтримують більше функціоналу для розробки програмного забезпечення. В якості ж IDE використовуватимуться такі середовища розробки як PyCharm або ж Notepad++. Окрім цього, було обрано обширний список додаткових інструментів та бібліотек для спрощення розробки або ж покращення якості функціоналу програмного забезпечення. Але для початку, опишемо ж вибір мови програмування та обрані середовища розробки.

Python є високорівневою об’єктно орієнтованою мовою програмування, яка вважається однією з найпопулярніших мов програмування в світі. Мова програмування запозичила свою назву з британського телевізійного шоу “Монті Пайтон” і характеризується своєю суворою динамічною типізацією. Сам же Python був розроблених ще на початку 1990-х років нідерландським програмістом Гвідо Россумом. Основними характеристиками цієї мови програмування є:

- Логічний синтаксис, завдяки якому код легко читається й засвоюється програмістом;
- Гнучкість і масштабованість, що дозволяє адаптувати високорівневу логіку та розширяти складні застосунки, якщо виникне така необхідність;
- Швидкість в плані написання, на відміну від інших мов програмування;

- Python є інтерпретованою мовою програмування, що робить можливим написання коду в будь-якому текстовому файлів на будь-якій платформі, при цьому з можливістю успішного запуску програми;
- Популярність мови, що дозволяє знайти значну частину рішень для проблем під час розробки в мережі Інтернет.

Однак, незважаючи на швидкість написання коду, використовуючи Python, універсальність мови призводить до повільності в плані компіляції й запуску програм, велику ресурсозатратність для роботи і прив'язаність до системних бібліотек.

Python використовується у багатьох сферах, знову ж таки, через універсальність цієї мови програмування. Серед цих сфер найпопулярнішими є:

- Розробка програмних застосунків в будь-якому напрямку;
- Розробка серверної частини мобільних застосунків, що є найпопулярнішим напрямом;
- Розробка комп'ютерних ігор або ж їх компонентів;
- Вбудовані системи для різних пристроїв, таких як внутрішні платформи управління банкоматами;
- Скрипти і плагіни до вже реалізованих програм для автоматизації процесів або ж створення інших рішень;
- Автоматизація тестування;
- Машинне навчання - є основною мовою для написання алгоритмів і аналітичних застосунків у сфері Machine Learning.

Останній пункт є найбільш вагомою причиною вибору мови програмування Python для подальшої розробки програмного забезпечення інтелектуальної системи пошуку інформації. Лексична простота і універсальність дозволять спростити час на розробку програмного продукту без використання більш комплексних інструментів або бібліотек, якби розробка проводилась з використанням іншої мови програмування.

Повернімося ж до середовища розробки. Серед них було вирішено опиратися максимум на два середовища - PyCharm та Notepad++, з використанням командного вікна Windows для дозавантаження додаткових бібліотек.

PyCharm - це інтегроване середовище розробки від компанії JetBrains, компанії, яка є популярним розробником таких IDE як WebStorm (JavaScript), Rider (.NET), CLion (C та C++), IntelliJ IDEA (Java і Kotlin) і PyCharm (Python). Середовище розробки надає широкий спектр інструментів для полегшення написання коду, таких як розумне автодоповнення, підсвічування синтаксису, а також підтримує веб-розробку та роботу з базами даних.

PyCharm дозволить спростити налагодження коду, що дозволить значно скоротити час на виправлення синтаксичних помилок, дозволяє встановлювати брейк-пойнти для подальшого дебагінгу коду; надає можливість рефакторингу - зміни структури коду без втрати функціональності. Підтримуючи ініціативу інших IDE від JetBrains, PyCharm підтримує систему контроль версій, інтеграцію з Git, Mercurial та іншими системами, що дозволяє працювати з історією змін. Окрім вище перерахованих переваг, PyCharm підтримує такі фреймворки як HTML, CSS і JavaScript, що надасть нам потенціал у розробці веб-адаптації програмного додатку з використанням сучасних технологій.

Окрім PyCharm, додатковим середовищем розробки було обрано Notepad++. Обговоримо ж причину вибору цього текстового редактору.

Notepad++ - це безкоштовний текстовий редактор для Windows з відкритим кодом, призначений для користувачів, яким необхідна розширена функціональність у порівнянні зі стандартним, системним “Блокнотом”. Як і PyCharm, Notepad++ надає перевагу у підсвічуванні синтаксичних помилок для багатьох мов програмування, підтримку одночасного редагування декількох файлів у інтуїтивному інтерфейсі та можливістю розширення його функцій за допомогою багатьох плагінів.

Notepad++ був обраний як швидка та не менш ефективна альтернатива для PyCharm, у випадку коли потрібне середовище розробки з мінімальним споживанням ресурсів комп’ютера. Завдяки потенціальній гнучкості через встановлення плагінів, Notepad++ є більшим ніж розширений текстовий редактор: функціонал програми надає можливість швидкого редагування коду на декількох вкладках за рахунок простоти. У випадку з програмним забезпеченням інтелектуальної системи пошуку інформації, Notepad++ буде аварійним варіантом у використанні середовища розробки: коли потребуватиметься термінова зміна коду без значних затрат ресурсів комп’ютера.

Розглянувши основну частину інструментарію при розробці програмного забезпечення інтелектуальної системи пошуку інформації, перейдемо до ключових бібліотек, плагінів та інструментів, які гратимуть не менш важливу роль у функціонуванні програмного додатку.

Почнемо ж з “ядра” пошуку, а саме інструментів, які гратимуть ключову роль у його розробці. До них входять OpenSearch, Python, FastAPI, opensearch-py, React, JSONB та декілька бібліотек, пов’язаних з обробкою пошукового запиту і дедуплікацією результатів.

Пошуковим рушієм було обрано OpenSearch, що працює як розподілений пошуковий рушій і аналітична платформа (є розгалуженою версією ElasticSearch 7), що надає інвертовані індекси, ранжування BM25, підсвічування

фрагментів, агрегацію, реплікацію та API. Також OpenSearch має опціональну підтримку векторних полів kNN. Функціонал в проєкті, який підтримуватиметься: зберігання та індексація документів, лексичний і семантичний пошуки, фільтри і сортування, формування і підсвічування збігів.

Python в цьому випадку буде працювати додатково як частина бекенд-сервісу і буде включати в себе обробку пошукових запитів, бекенд API, експерименти з моделями і офлайн оцінювання метрик.

FastAPI є високопродуктивним фреймворком для REST API на Python з валідацією схем, автогенерацією документації для API і асинхронністю. Функціонал, що покриватиметься: ендпоїнти, технічні ендпоїнти, CORS для фронтенд-частини. Використовуватиметься для мінімізації коду для якісного API і зручної документації.

Opensearch-py є офіційним python-клієнтом до OpenSearch. функціонально підтримуватиме створення індексів і мапінгу, роботу з пошуковими запитами, керування шардингом/репліками (за потреби). Є стабільним клієнтом з повним покриттям API, спрощує інтеграцію без “ручних” HTTP-запитів.

Окрім ядра, будуть використовуватись також низка бібліотек для покращення якості та продуктивності програмного забезпечення для виконання функціоналу пошуку даних. Цими бібліотеками є:

- feedparser - парсер для RSS;
- requests - HTTP-клієнт;
- trafilatura - точний екстрактор основного тексту зі сторінок;
- beautifulsoup4 - розбір і очищення HTML, є резервом до бібліотеки trafilatura;
- python-dateutil - бібліотека для парсингу дат або ж часових поясів;
- langdetect - бібліотека для визначення мови тексту.

Основними їх функціями будуть зчитування фідів, перехід на повні статті, завантаження HTML, виділення основного контенту, уніфікація дат публікацій, анотація документів мовою для фільтрації та індексації.

В якості збереження метаданих у сховищах баз даних використовуватиметься JSONB, даючи можливість швидко фільтрувати метадані.

Для хешування і дедуплікації використовуватимуться такі бібліотеки як hashlib і simhash. Hashlib хешуватиме дані для знаходження точних дублікатів, simhash виконуватиме пошук схожих текстів або фрагментів тексту. Основним функціоналом в проєкті є уникнення повторів даних в джерелах і зменшення шуму й індексного розміру.

Для побудови користувацького інтерфейсу, або ж UI, використовуватимуться React та TailwindCSS. React є доволі популярною бібліотекою для побудови UI з компонентів, TailwindCSS надає змогу використовувати класи для швидкої й адаптивної стилізації інтерфейсу. Основний функціонал очевидний - розробка інтерфейсу для пошуку з використанням мінімальної збірки, що дозволить створити швидкий, охайний та інтуїтивний UI, простий для використання й подальшого розширення.

FAISS використовуватиметься як частина для семантичного пошуку, використовуючи метод kNN у векторних просторах. Бібліотека надаватиме нам можливість генерації kNN-кандидатів для семантичного пошуку, що дозволить розширити гібридний пошук для програмного забезпечення.

Для підтримки роботи з різними середовищами використовуватиметься python-dotenv. Це дозволить параметризувати OpenSearch для простої керованості конфігурацією як для локального так і для навчального середовищ.

Для юніт та інтеграційного тестування коду на мові Python використовуватиметься pytest. Це дозволить оформити перевірку обробки пошукових запитів, підтримку тестування пошукового API і загалом забезпечує регресійну стабільність і загальну якість коду для розроблюваного програмного забезпечення.

Перейдемо до опціональних бібліотек, які можуть бути чудовими доповненнями програмного забезпечення інтелектуальної системи пошуку інформації у майбутньому.

sentence-transformers - фреймворк і колекція моделей для ембеддингів речень або абзаців. Функціонально допоможе оформити векторизацію часток документів й запитів, допоможе з побудовою семантичних індексів і з підтримкою тестування різних моделей під українську й англійську мови.

cross-encoder - клас моделей для переранжування: на вхід буде отримувати пару “запит-документ” і повертатиме релевантність. Функціонал включатиме уточнення порядку для top-N кандидатів після складання списку і загалом покращить якість релевантності результатів пошуку.

Redis - сховище даних типу “ключ-значення” з дуже низькою затримкою. Дана бібліотека допоможе кешувати часті запити/відповіді та їх сніпети і допоможе зі зниженням затримки без основної архітектури.

Даний інструментарій надасть надійний BM25-базис, мінімальний код при розробці бекенду, прозору обробку запитів пошуку і керовані метадані, контрольовану семантику з приростом якості у топ-N для результатів, простий але інтуїтивний UI для користувача і, перш за все, якісний програмний продукт. Саме за цієї причини був обраний саме такий інструментарій в подальшій

розробці програмного забезпечення інтелектуальної системи пошуку інформації.

3.2. Алгоритмізація та програмування програмних модулів

Алгоритмізація та програмування програмних модулів для програмного забезпечення інтелектуальної системи пошуку інформації є доволі комплексним підходом, який потребує детального планування того, як повинен бути реалізованим функціонал програмного забезпечення. Це є дуже важливим процесом, в кінцевому результаті якого користувач повинен бачити простий, але в той же час доволі якісний за стандартами пошуковий інструмент.

Попередньо обравши інструментарій, необхідно налаштувати фреймворк, а саме завантажити всі необхідні залежності для початку розробки програмного забезпечення інтелектуальної системи пошуку інформації. Завдяки тому, що було використано мову програмування Python, всі необхідні бібліотеки можливо завантажити через командну строку Windows.

Найпростішим методом буде зібрати всі бібліотеки в txt-файл і оформити їх завантаження через pip install. Приклад такої збірки зображено на рисунку 3.1.

```
fastapi==0.115.*
uvicorn[standard]==0.30.*
opensearch-py==2.5.*
feedparser==6.0.*
requests==2.32.*
trafilatura==1.7.*
beautifulsoup4==4.12.*
python-dateutil==2.9.*
langdetect==1.0.*
python-dotenv==1.0.*
faiss-cpu==1.8.*
sentence-transformers==3.*
redis==5.*
```

Рис. 3.1 Бібліотеки в requirements.txt

Провівши просту маніпуляцію через команду “pip install -r requirements.txt”, ми можемо одразу завантажити всі необхідні бібліотеки для Python без зайвої потреби вводити кожну команду вручну через pip install. Це вже скорочує час на налаштування фреймворку.

В подальшому, складаємо конфігурацію в .env. На рисунку 3.2 зображена така конфігурація, на рисунку 3.3 зображено підключення цієї конфігурації через Python:

```
OS_HOST=http://localhost:9200
OS_USERNAME=
OS_PASSWORD=
INDEX_NAME=techblogs_bm25

REDIS_URL=redis://localhost:6379/0
ENABLE_SEMANTIC=false
CE_TIMEOUT_MS=250
```

Рис. 3.2 Конфігурація в.env

```python
from dotenv import load_dotenv
import os

load_dotenv()
OS_HOST = os.getenv("OS_HOST", "http://localhost:9200")
```

Рис. 3.3 Підключення конфігурації

В подальшому прості бібліотеки імпортуємо на верхньому рівні модулів, складні ж або опціональні можемо імпортувати всередині функцій. Це зроблено для того, щоб базовий функціонал міг працювати без цих бібліотек. На рисунку 3.4 зображений приклад імпортування бібліотеки через функцію.

```python
def embed_texts(texts):
    try:
        from sentence_transformers import SentenceTransformer
    except ImportError:
        raise RuntimeError("Semantic module not installed")
    model = SentenceTransformer("intfloat/multilingual-e5-base")
    return model.encode(texts, normalize_embeddings=True)
```

Рис. 3.4 Імпортування бібліотеки sentence_transformers через функцію

При подальшій ініціалізації функції, бібліотека без проблем буде підтягуватись і працювати коректно. Глобальні ж клієнти будуть ініціалізовані один раз при старті сервісу, або ж через події запуску, як це буде відбуватися з FastAPI.

Поговоримо про кешування та прискорення завантажень. Wheels кешуються автоматично в локальному каталозі, моделі, такі як sentence_transformers будуть кешуватися в домашній теці, однак рекомендується підготувати офлайн-кеш і вказати змінні середовища. Для Redis будуть кешуватися результати популярних запитів, щоб знизити затримку.

Завдяки PyCharm, можливо уникнути ризиків “зламаних оновлень” і оновлювати бібліотеки, за потреби, в окремій гілці. Також тепер є необхідність моніторити сумісність бібліотек. Наприклад, після оновлення бібліотеки opensearch-py необхідно перевіряти функціонал підсвічування і роботу фільтрів.

Таким чином, ми організували фреймворк і завантажили в нього всі необхідні для початку роботи бібліотеки. Реальна робота над розробкою програмного забезпечення починається конкретно після цього етапу.

Після встановлення всіх необхідних бібліотек, відбувається ініціалізація самого проєкту та його налаштувань. Для початку представимо файлову структуру фреймворку, яка зображена на рисунку 3.5.

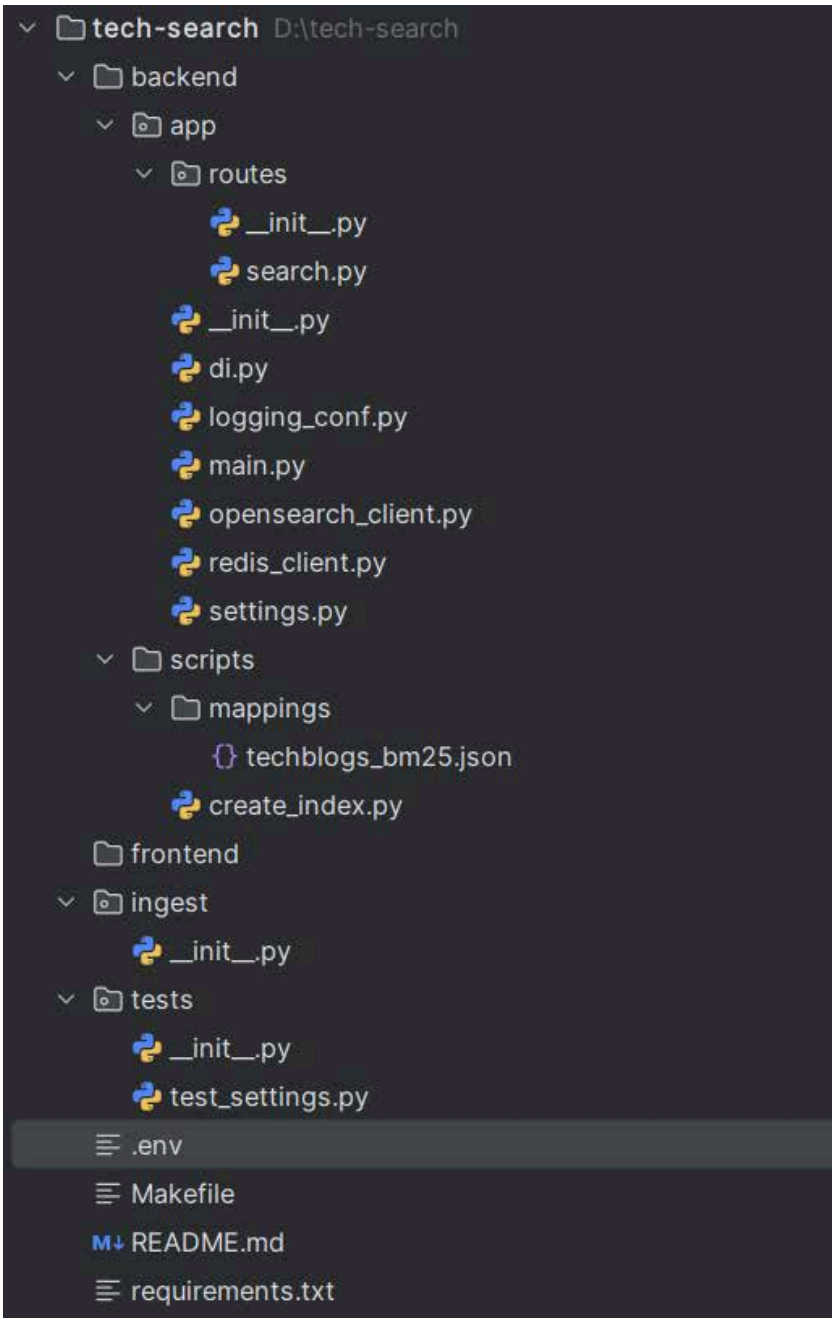

Рис. 3.5 Файлова структура проєкту

В requirements попередньо збережений список бібліотек, який можна використовувати для їх відповідного оновлення через pip -install. В .env зберігається шаблон конфігурації, представлений вище, однак з новими, доданими конфігураціями:

.env

```
# OpenSearch
OS_HOST=http://localhost:9200I
OS_USERNAME=
OS_PASSWORD=
OS_INDEX=techblogs_bm25
OS_TLS_VERIFY=false

# API
API_HOST=0.0.0.0
API_PORT=8000
CORS_ALLOW_ORIGINS=*

# Redis (опц.)
REDIS_URL=redis://localhost:6379/0
ENABLE_CACHE=false
CACHE_TTL_SECONDS=900

# Пошукові параметри
DEFAULT_PAGE_SIZE=10
MAX_PAGE_SIZE=50
BM25_TOPK=1000
FUSION_K0=60
ENABLE_SEMANTIC=false
ENABLE_RERANK=false
RERANK_TOPN=100

# Ліміти/таймаути
OS_TIMEOUT_SECONDS=20
RERANK_TIMEOUT_MS=250
```

Рис. 3.6 Оновлені шаблони конфігурації в .env

Подальші налаштування конфігурацій зберігаються в settings.py, код якого зображено на рис. 3.7 і 3.8:

```python
load_dotenv()

class AppSettings(BaseModel):  1 usage
    # OpenSearch
    os_host: str = Field(default=os.getenv("OS_HOST", "http://localhost:9200"))
    os_username: str | None = os.getenv("OS_USERNAME")
    os_password: str | None = os.getenv("OS_PASSWORD")
    os_index: str = os.getenv("OS_INDEX", "techblogs_bm25")
    os_tls_verify: bool = (os.getenv("OS_TLS_VERIFY", "false").lower() == "true")
    os_timeout_seconds: int = int(os.getenv("OS_TIMEOUT_SECONDS", "20"))

    # API
    api_host: str = os.getenv("API_HOST", "0.0.0.0")
    api_port: int = int(os.getenv("API_PORT", "8000"))

    cors_allow_origins: List[str] = Field(
        default_factory=lambda: [o.strip() for o in os.getenv("CORS_ALLOW_ORIGINS", "*").split(",")]
    )

    # Redis (optional)
    redis_url: str | None = os.getenv("REDIS_URL")
    enable_cache: bool = (os.getenv("ENABLE_CACHE", "false").lower() == "true")
    cache_ttl_seconds: int = int(os.getenv("CACHE_TTL_SECONDS", "900"))
```

Рис. 3.7 Код налаштувань з settings.py

```python
    # Search pipeline
    default_page_size: int = int(os.getenv("DEFAULT_PAGE_SIZE", "10"))
    max_page_size: int = int(os.getenv("MAX_PAGE_SIZE", "50"))
    bm25_topk: int = int(os.getenv("BM25_TOPK", "1000"))
    fusion_k0: int = int(os.getenv("FUSION_K0", "60"))
    enable_semantic: bool = (os.getenv("ENABLE_SEMANTIC", "false").lower() == "true")
    enable_rerank: bool = (os.getenv("ENABLE_RERANK", "false").lower() == "true")
    rerank_topn: int = int(os.getenv("RERANK_TOPN", "100"))
    rerank_timeout_ms: int = int(os.getenv("RERANK_TIMEOUT_MS", "250"))

    @field_validator("default_page_size", "max_page_size")
    @classmethod
    def _validate_page_sizes(cls, v: int) -> int:
        if v <= 0:
            raise ValueError("page size must be positive")
        return v

settings = AppSettings()
```

Рис. 3.8 Код налаштувань з settings.py для search pipeline

В logging_conf.py описуємо логування і додаємо функціонал очищення хендлерів, для уникнення дублювання при швидкому перезавантаженні.

```python
import logging
import sys

def configure_logging(level: str = "INFO") -> None:
    fmt = "%(asctime)s %(levelname)s [%(name)s] %(message)s"
    handler = logging.StreamHandler(sys.stdout)
    handler.setFormatter(logging.Formatter(fmt))
    root = logging.getLogger()
    root.setLevel(level)
    root.handlers.clear()
    root.addHandler(handler)
```

Рис. 3.9 Код з logging_conf.py

Фабрика клієнта OpenSearch описана у файлі opensearch_client.py, зображеному на рисунку 3.10:

```python
from __future__ import annotations
from opensearchpy import OpenSearch
from .settings import settings

def make_opensearch() -> OpenSearch:
    http_auth = None
    if settings.os_username and settings.os_password:
        http_auth = (settings.os_username, settings.os_password)

    client = OpenSearch(
        hosts=[settings.os_host],
        http_auth=http_auth,
        use_ssl=settings.os_host.startswith("https"),
        verify_certs=settings.os_tls_verify,
        timeout=settings.os_timeout_seconds,
    )
    return client
```

Рис 3.10 Код з opensearch_client.py

На рисунку 3.11 зображена реалізація функціоналу Redis:

```python
from __future__ import annotations
from typing import Optional
import redis
from .settings import settings

_redis: Optional[redis.Redis] = None

def get_redis() -> Optional[redis.Redis]:
    global _redis
    if not settings.enable_cache or not settings.redis_url:
        return None
    if _redis is None:
        _redis = redis.from_url(settings.redis_url, decode_responses=True)
    return _redis
```

Рис. 3.11 Код з redis_client.py

В di.py (рис. 3.12) реалізовані залежності та ресурси для FastAPI. Search.py описує базові маршрути. В main.py складуємо FastAPI (рис. 3.13), а скрипт створення індексу OpenSearch розподіляється на мапінг через JSON (рис. 3.14) та реалізацію create_index.py (рис. 3.15). Запуск відбувається через команду “python -m backend.scripts.create_index”. Мінімальний тест конфігів для pytest реалізований у test_settings.py (рис. 3.16).

```python
from __future__ import annotations
from contextlib import asynccontextmanager
from fastapi import FastAPI
from .logging_conf import configure_logging
from .opensearch_client import make_opensearch
from .redis_client import get_redis

@asynccontextmanager
async def lifespan(app: FastAPI):
    # init logging
    configure_logging("INFO")
    # прогріваємо клієнтів, щоб ловити помилки на старті
    try:
        os_client = make_opensearch()
        os_client.info()  # ping
    except Exception as e:
        import logging
        logging.getLogger(__name__).warning("OpenSearch not reachable: %s", e)

    try:
        _ = get_redis()
    except Exception as e:
        import logging
        logging.getLogger(__name__).warning("Redis not reachable: %s", e)

    yield
```

Рис. 3.12 Код з di.py

```python
from __future__ import annotations
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from .di import lifespan
from .settings import settings
from .routes.search import router as search_router

app = FastAPI(lifespan=lifespan, title="Tech Search API")

app.add_middleware(
    CORSMiddleware,
    allow_origins=(["*"] if "*" in settings.cors_allow_origins else settings.cors_allow_origins),
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

app.include_router(search_router)
```

Рис. 3.13 Код з main.py. Локальний запуск реалізований через команду “uvicorn backend.app.main:app --reload”

```json
{
  "settings": {
    "index": {
      "number_of_shards": 1,
      "number_of_replicas": 0
    },
    "analysis": {
      "analyzer": {
        "default": { "type": "standard" }
      }
    }
  },
  "mappings": {
    "properties": {
      "url":        { "type": "keyword" },
      "site":       { "type": "keyword" },
      "title":      { "type": "text" },
      "body":       { "type": "text" },
      "language":   { "type": "keyword" },
      "published_at": { "type": "date", "format": "strict_date_optional_time||epoch_millis" }
    }
  }
}
```

Рис. 3.14 Мапінг OpenSearch через json (techblogs_bm25.json)

```python
from __future__ import annotations
import json
from pathlib import Path
from opensearchpy import OpenSearch
from backend.app.settings import settings
from backend.app.opensearch_client import make_opensearch

def main():  1 usage
    client: OpenSearch = make_opensearch()
    mapping_path = Path(__file__).parent / "mappings" / "techblogs_bm25.json"
    body = json.loads(mapping_path.read_text(encoding="utf-8"))

    if client.indices.exists(settings.os_index):
        client.indices.delete(index=settings.os_index)

    client.indices.create(index=settings.os_index, body=body)
    print(f"Created index: {settings.os_index}")

if __name__ == "__main__":
    main()
```

Рис. 3.15 Реалізація скрипту в create_index.py

```python
from backend.app.settings import settings

def test_settings_defaults():
    assert settings.os_host
    assert settings.os_index
    assert settings.default_page_size > 0
    assert settings.max_page_size >= settings.default_page_size
```

Рис. 3.16 Реалізація тестових конфігурацій в test_settings.py

Ініціалізувавши проєкт, наступним етапом являється розробка пайплайну інгесту джерел для інтелектуальної системи. На рисунку 3.17 зображена файлова структура інгесту файлів:

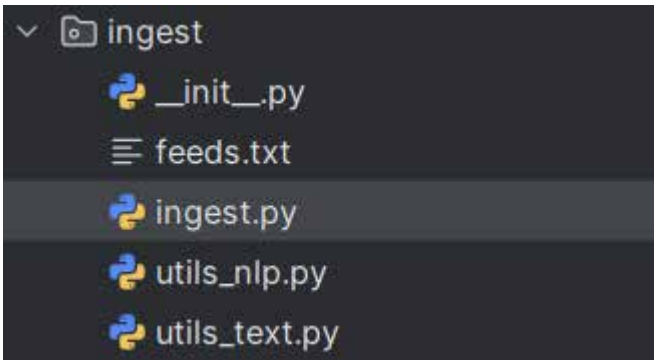

Рис. 3.17 Файлова структура інгесту

Конфігурація джерел вказана в txt файлі під назвою feeds. Утиліти для тексту, дат і дедуплікації даних описані в utils_text.py. Утиліти для виявлення мови через langdetect і хешування через simhash реалізовані в utils_nlp.py (рис. 3.18). Сам же функціонал інгесту реалізований в ingest.py.

```python
from __future__ import annotations
from langdetect import detect
from datasketch import MinHash
import re

def detect_lang(text: str) -> str:  2 usages
    t = re.sub(r"[^A-Za-zА-Яа-яІіЇїЄєҐґ0-9 ]+", " ", text)[:1000]
    try:
        return detect(t)
    except Exception:
        return "und"

def minhash_sig(text: str, num_perm: int = 64) -> MinHash:  1 usage
    m = MinHash(num_perm=num_perm)
    for token in set(text.lower().split()):
        m.update(token.encode("utf-8"))
    return m
```

Рис 3.18 Утиліти розпізнавання мови і хешування в utils_nlp.py

Дизайн та створення індексу в OpenSearch разом зі скриптом було створено і вказано вище. Аналогічно Вдалось створити базовий пошук через BM25, наведений на рисунках 3.18, 3.19 та 3.20

```python
@router.get("/search", response_model=SearchResponse)  1 usage (1 dynamic)
def search(
    q: str = Query(..., description="Пошуковий запит"),
    size: int = Query(settings.default_page_size, ge=1, le=settings.max_page_size),
    site: Optional[str] = Query(None),
    language: Optional[str] = Query(None),
    date_from: Optional[str] = Query(None),
    date_to: Optional[str] = Query(None),
):
    client = make_opensearch()

    must = [{"multi_match": {"query": q, "fields": ["title^3", "body"]}}]
    filters = []
    if site:
        filters.append({"term": {"site": site}})
    if language:
        filters.append({"term": {"language": language}})
    if date_from or date_to:
        rng = {}
        if date_from:
            rng["gte"] = date_from
        if date_to:
            rng["lte"] = date_to
        filters.append({"range": {"published_at": rng}})
```

Рис. 3.18 Алгоритм базового пошуку BM25 (частина 1)

```python
body = {
    "query": {"bool": {"must": must, "filter": filters}},
    "highlight": {
        "fields": {
            "body": {"fragment_size": 160, "number_of_fragments": 2},
            "title": {"fragment_size": 80, "number_of_fragments": 1},
        },
        "pre_tags": ["<mark>"],
        "post_tags": ["</mark>"],
    },
    "_source": ["url", "site", "title", "published_at", "language"],
    "size": size,
}
```

Рис. 3.19 Алгоритм базового пошуку BM25 (частина 2)

```
res = client.search(index=settings.os_index, body=body)
hits: List[Hit] = []
for h in res["hits"]["hits"]:
    src = h.get("_source", {})
    hi = h.get("highlight", {})
    snippet = None
    if "body" in hi:
        snippet = " … ".join(hi["body"])[:600]
    elif "title" in hi:
        snippet = hi["title"][0]
    hits.append(
        Hit(
            id=h.get("_id"),
            url=src.get("url"),
            site=src.get("site"),
            title=src.get("title"),
            published_at=src.get("published_at"),
            language=src.get("language"),
            snippet=snippet,
        )
    )

return SearchResponse(
    total=res["hits"]["total"]["value"],
    took_ms=res.get("took", 0),
    hits=hits,
)
```

Рис. 3.20 Алгоритм базового пошуку BM25 (частина 3)

Переходимо до фронтенд частини програмного продукту. Використовуючи React + TailwindCSS, маємо змогу створити простий інтерфейс без складного білду. На рисунку 3.21 зображена проста файлова структура фронтед-частини програмного додатку:

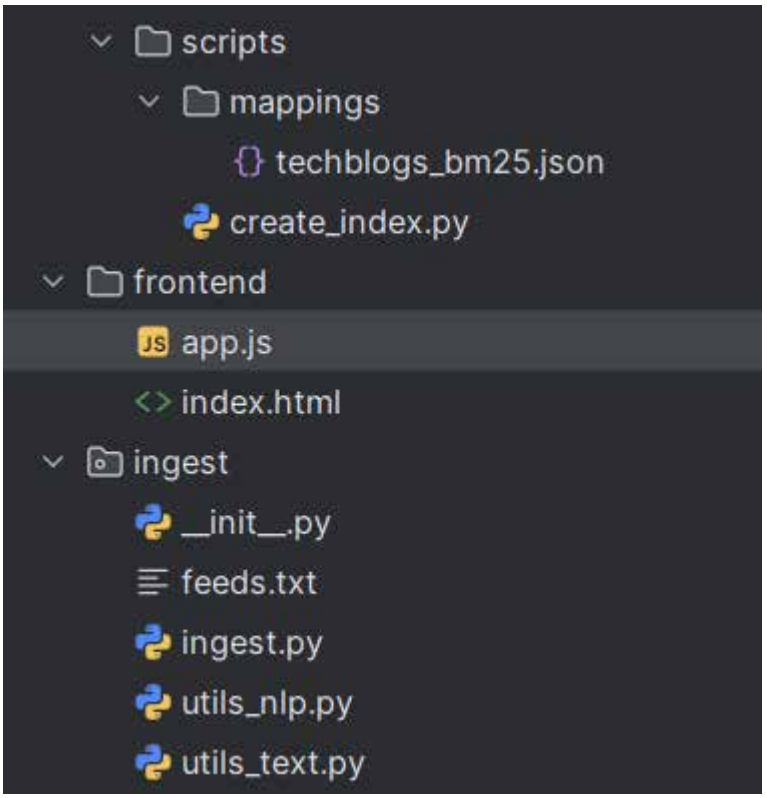

Рис. 3.21 Файлова структура фронтенд-частини програмного забезпечення. Збережена у папці frontend

Таким чином, в даному розділі було покрито інструментарій програмного забезпечення інтелектуальної системи пошуку інформації, продемонстровано частину алгоритмів і структуру самої програми і тим самим описано, який функціонал матиме кінцевий продукт. Однак варто зазначити, що структура програми дозволяє розширити її потенціал в якості реалізації додаткового функціоналу, що покращить якість роботи та зручність використання для користувача.

РОЗДІЛ 4. РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

4.1. Апаратні та програмні вимоги до реалізації побудованої системи

Після перевірки роботи програмного забезпечення, продукт є готовим до свого релізу, але перед цим варто зазначити з якими системами даний продукт буде працювати: які апаратні і програмні забезпечення варто завантажити, щоб користувач не мав жодних проблем з запуском і роботою програми.

Поговоримо про операційні системи, які підтримуються програмним забезпеченням. Програма сама по собі є доволі гнучкою і не потребує системних компонентів від операційних систем, робота програми і підтримка нею ОС не є ключовими факторами, тому програма спокійно може працювати на таких операційних системах:

- Microsoft Windows;
- Linux;
- Mac OS.

Ці операційні системи є найпопулярнішими і функціонал програми був успішно протестований на ОС від Microsoft Windows та Linux. Гіпотетично, програма повинна без проблем також підтримувати систему Mac OS.

Підсумуємо мінімальні технічні вимоги для використання програмного забезпечення:

- Рекомендується остання 64-бітна версія ОС;
- Процесор Intel Core i3 або AMD;
- Підключення до Інтернету для доступу до джерел. Рекомендується якісне підключення до 100 Мбіт/с для якісної роботи пошукового рушія;
- До 300 Мб доступного місця на жорсткому диску

Це є загальні вимоги до програмного забезпечення інтелектуальної системи пошуку інформації. Через власну простоту, програма не є занадто потребуючою в рамках системного забезпечення комп’ютером. Програма має простий та інтуїтивний інтерфейс і надає змогу користувачеві знаходити і взаємодіяти з необхідною йому інформацією в рамках технічних блогів.

4.2. Хід виконання дослідження

Дійшовши до кінцевого результату в обсязі розробки програмного забезпечення інтелектуальної системи пошуку інформації, тепер додаток можна застосовувати як базис для виконання дослідження. Дослідження полягає у підтвердженні гіпотези ефективності гібридного пошуку, використовуючи BM25 (лексичний пошук) і kNN (семантичний пошук), провести аналіз результатів пошуків і порівняти отримані результати один з одним. Такий підхід надасть простий, але об’єктивний доказ успіху проведення дослідження. Для початку було поставлено задачу зібрати всі необхідні метрики для дослідження.

Почнімо підготовку даних для аналізу. В якості порівняльних результатів були обрані такі атрибути:

- Тип пошуку;
- Конфігурація;
- Відсоток покриття nDCG@10;
- Відсоток покриття MRR@10;
- Відсоток покриття Recall@100;
- Затримка p50;
- Затримка p95.

Для початку опишемо, що означають такі метрики, як nDCG@10, MRR@10, Recall@100, p50 і p95.

nDCG (normalized Discounted Cumulative Gain) визначає якість ранжування результатів з урахуванням позицій і ступенів релевантності до пошукового запиту. nDCG@10 означає розранунок в топ-10. Визначається за формулою:

$$\mathrm{DCG@10} = \sum_{i=1}^{10} \frac{gain_i}{\log_2(i+1)}$$

Рис. 4.1 Формула розрахунку nDCG, де gain - актуальність документа, і - позиція

MRR (Mean Reciprocal Rank) використовується для оцінки якості роботи пошукового рушія, враховуючи перші 10 результатів (топ-10) і обчислюючи виявлення там першого релевантного результату. Високий відсоток означатиме те, що система часто надає правильну відповідь ще на перших 10 результатах. MRR@10 означає розрахунок в межах 10 запитів. Визначається за формулою:

$$\mathrm{MRR} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{\mathrm{rank}_i}$$

Рис. 4.2 Формула розрахунку MRR, де n - кількість запитів, rank - позиція запиту, i - запит

Важливо враховувати, що якщо перший релевантний результат виходить за рамки n, то MRR еквівалентне до нуля.

Recall - метод обчислення частки всіх релевантних результатів до запиту, які потрапили в список. Recall@100 означає розрахунок релевантних результатів з топ-100. Формула обчислення доволі проста і зображена на рисунку 4.3:

$$\mathrm{Recall@100} = \frac{\#\{\text{релевантні в топ-100}\}}{\#\{\text{усі релевантні в колекції для запиту}\}}$$

Рис. 4.3 Формула розрахунку Recall

Метрики p50 і p95 вимірюють затримку в сервісі при обробці запиту, де число означає відсоток від загальної кількості запитів. Таким чином, p50 обчислює половину вимірів, p95 обчислює 95% від запитів. Зумовлено це тим, щоб показати з p50 типовий досвід користувача з програмним забезпеченням і якістю роботи додатку при піковій роботі, використовуючи p95.

Обговоривши виміри, приступимо ж до їх загального вимірювання. Для спрощення роботи і запобігання розрахування всіх даних вручну, було створено декілька скриптів, які зроблять всю складну роботу й одразу продемонструють результати. Для nDCG@10, MRR@10 і Recall@100 було розроблено алгоритм, зображений на рисунку 4.4 та конфігурації, з якими буде проводитись дослідження. Конфігурації та кінцевий результат обчислень буде зазначено нижче в таблиці.

```python
import ir_measures, pandas as pd
from ir_measures import nDCG, MRR, Recall

def load_per_query_metrics(qrels_path, run_path):  2 usages
    qrels = ir_measures.read_trec_qrels(qrels_path)
    run = ir_measures.read_trec_runs(run_path)
    metrics = [nDCG@10, MRR@10, Recall@100]
    rows = []
    for qid, q_measures in ir_measures.iter_calc(metrics, qrels, run):
        rec = {"qid": qid}
        for m, v in q_measures.items():
            rec[str(m)] = float(v)
        rows.append(rec)
    return pd.DataFrame(rows)

bm25_df  = load_per_query_metrics( qrels_path: "eval/qrels.csv",  run_path: "eval/run_BM25.txt")
hybrid_df = load_per_query_metrics( qrels_path: "eval/qrels.csv",  run_path: "eval/run_HYBRID.txt")
```

Рис. 4.4 Алгоритм обчислення метрик nDCG@10, MRR@10 і Recall@100

Перейдемо до обчислення затримок (латентності) використовуючи метрики p50 і p95. Для цього також було створено алгоритм розрахунку цих метрик (рис. 4.5) і зібрані логи для конфігурацій.

```python
import json, numpy as np

def load_pcts(path, path_filter="/search"):  2 usages
    vals = []
    with open(path, encoding="utf-8") as f:
        for line in f:
            rec = json.loads(line)
            if rec.get("path")==path_filter:
                vals.append(rec["ms"])
    return np.percentile(vals, 50), np.percentile(vals, 95)

p50_bm25, p95_bm25   = load_pcts("logs/latency_BM25.jsonl")
p50_hyb,  p95_hyb    = load_pcts("logs/latency_HYBRID.jsonl")
```

Рис. 4.5 Алгоритм обчислення метрик p50 і p95

Створивши алгоритми та отримавши необхідні нам значення, можемо їх продемонструвати в таблиці 4.1.

Тип	Конфіг	nDCG@10 (%)	MRR@10 (%)	Recall@100 (%)	p50 (мс)	p95 (мс)
BM25	k1=1.6, b=0.75, title^3	33.2	24.8	71.2	180	520
Гібрид (BM25+kNN)	FAISS @1000, k0=60	36	26.5	75.2	230	690
Гібрид (BM25+kNN) і ранжування	ранжування топ-100	38.5	28.6	76.3	290	790

Таблиця 4.1 Результати якості роботи програмного забезпечення інтелектуальної системи пошуку інформації з різними типами пошуку, де k1 - насичення (розрахунок знайдення ключових слів), b - нормалізація довжини документа, title^3 - збільшення ваги збігів втричі, k0 - ступінь злиття списків кандидатів

З того, що ми бачимо, можемо зробити такі висновки:

- При порівнянні BM25 та гібриду, гібрид підвищив nDCG@10 на 0.028 (2.8%), MRR@10 на 0.017 (1.7%), Recall@100 на 0.04 (4%), однак p95 зросло на 170 мс;
- Додавання ранжування до гібриду дало 0.025 (2.5%) до nDCG@10, MRR@10 зросло на 0.021 (2.1%), що є значним зростом. Recall@100 зріс на 0.011 (1.1%) разом з затримкою p95 до 100 мс.

Для подальшого дослідження необхідно зрозуміти, які саме конфігурації зробили такий приріст. Для цього було зроблено порівняльний аналіз таких конфігурацій:

- Гібрид проти BM25;
- Гібрид з k0=30 проти k0=60;
- Гібрид з обчисленням в 500 кандидатів проти 1000;
- Гібрид і ранжування проти гібриду.

Подальші результати цього дослідження наведені в таблиці 4.2, з описаним приростом або спаданням якості пошуку.

Конфіг	Δ nDCG@10 (%)	Δ MRR@10 (%)	Δ Recall@100 (%)	Δ p95 (мс)	Δ p50 (мс)
Гібрид/ BM25	+2.8	+1.7	+4	+170	+40
Гібрид (k0=30/ k0=60)	-0.65	-0.31	-0.42	-20	-5
Гібрид (topK=500/ topK=1000)	-0.41	-0.22	-0.6	-35	-10
Гібрид+ ранжування/ Гібрид	+2.52	+2.14	+1.1	+100	+25

Таблиця 4.2 Порівняльний аналіз роботи програмного забезпечення при різних конфігураціях

Спираючись на результати розрахунків, наведених з таблиці 4.2, можна дійти до таких висновків:

- Як описано вище, гібридна версія пошуку дає кращі результати за виключно лексичний, однак наявне збільшення в затримці обробки запитів;
- Зменшення k0 призвело до втрати якості на 0.0065 для nDCG@10, 0.0031 для MRR@10 та 0.0042 для Recall@100, затримка для p50 та p95 мізерно зменшилась до -5мс та -20мс. Зміна призвела до втрати якості та ледве відчутне зниження в затримці;
- Зменшення topK аналогічно знизило якість до 0.0041 для nDCG@10, 0.0022 для MRR@10, 0.006 для Recall@100 за рахунок незначного зменшення затримки для p50 (-10мс) та p95 (-35мс). Зміна призвела до втрати в якості за рахунок невеликого зменшення латентності;
- Гібридний пошук з ранжуванням надав значний приріст в якості для всіх метрик в порівнянні з чистим гібридом, однак помірно збільшує затримку.

Проаналізувавши отримані дані, можемо тепер детально обговорити їх і дійти до заключних висновків.

4.3. Обговорення отриманих результатів

Отримавши та проаналізувавши дані, обговоримо, що призвело до таких результатів у першу чергу.

Як і очікувалось, перехід від BM25 до гібриду (BM25+kNN) дав стабільний приріст у якості. Це означає, що семантичний канал додає документи, які лексичний не знаходить. Ціна підвищення якості - збільшення затримки, однак на прийнятний рівень за стандартами пошукових сервісів.

Додавання ранжування до гібридного пошуку піднімає якість виданих результатів, що було очікуваним результатом, оскільки ранжування списків дає більш точну оцінку документів. Однак з якістю зростає також і затримка (p95 = 790мс), що компромізує баланс між швидкістю та якістю. В якості простого рішення можливо реалізувати ранжування запитів як опцію для користувача, що надасть вибір між якістю та швидкістю за потреб.

Збільшення параметру k0 продемонструвало покращення якості пошуку за ледве відчутного збільшення в часі, що говорить про те, що при слабшому впливі на вищі позиції списків покращує видачу точних результатів. Аналогічне можливо сказати й про збільшення обробки списку кандидатів з 500 до 1000, що покращує якість результатів пошуку за помірної латентності при обробці пошукового запиту.

Додатково, то введення запитів для пошуку як коротких запитів опрацьовується гібридним пошуком з ранжуванням краще, ніж з BM25. Семантичний пошук також допомагає з видачею результатів, релевантних до пошуку, навіть якщо точний збіг відсутній - це стосується більше випадків синонімії та перефразування.

ВИСНОВКИ

Було розроблено та оцінено програмне забезпечення інтелектуальної системи пошуку з використанням гібридного типу пошуку. В кінцевому результаті, програма забезпечує стабільну релевантність відповідей за прийнятної затримки з підтримкою багатомовності та прозорої подачі актуальних результатів.

Підсумувавши все дослідження, можна дійти до висновку, що гібридизація пошуку з використанням BM25 та kNN покращує і видачею результатів, і їхньою якістю за незначної затримки. Ранжування дає додатковий і найбільший приріст для десятки перших результатів, за рахунок додавання затримки при обробці запиту. Це можливо компенсувати простим функціоналом увімкнення або вимкнення ранжування при пошуку, наданого користувачеві: за рахунок цього, користувач самостійно і свідомо матиме можливість обирати між швидкістю видачі результатів або ж їх потенційною якістю.

Стандартні параметри для злиття списків кандидатів та їх максимального списку обробки виявилися найоптимальнішими для даних: невелика ціна при затримці за стабільний приріст в якості результатів.

Таким чином, було доведено актуальність переходу з виключного лексичного або семантичного пошуків до їхнього гібриду: це покращує якість результатів з не критичним збільшенням латентності. Потенційна оптимізація самої програми дозволить зменшити цю затримку в майбутньому без збиття балансу між якістю та затримкою.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

- Довідник Python - https://docs.python.org/uk/3/tutorial/index.html
- Про Python - https://uk.wikipedia.org/wiki/Python
- FastAPI - https://fastapi.tiangolo.com/
- Документація Opensearch-py - https://github.com/opensearch-project/opensearch-py?tab=readme-ov-file#readme
- Про Redis - https://uk.wikipedia.org/wiki/Redis
- Документація k6 - https://grafana.com/docs/k6/latest/
- Документація React - https://uk.legacy.reactjs.org/docs/getting-started.html
- Документація TailwindCSS - https://dev.to/munisekharudavalapati/tailwind-css-complete-documentation-3goc
- Документація OpenSearch - https://docs.opensearch.org/1.0/about/
- Документація JSONB - https://www.postgresql.org/docs/9.5/functions-json.html
- Метрики оцювання якості - https://weaviate.io/blog/retrieval-evaluation-metrics
- Метрики оцінювання затримки - https://oneuptime.com/blog/post/2025-09-15-p50-vs-p95-vs-p99-latency-percentiles/view
- Про BM25 - https://www.geeksforgeeks.org/nlp/what-is-bm25-best-matching-25-algorithm/
- Про kNN - https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
- Manning, C. D., Raghavan, P., Schütze, H. Introduction to Information Retrieval. Cambridge University Press, 2008. 581 с. URL - https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf

ДОДАТКИ

ДОДАТОК А

Фрагмент програмного коду. Вимірювання метрики якості запиту

```
import ir_measures, pandas as pd
from ir_measures import nDCG, MRR, Recall

def load_per_query_metrics(qrels_path, run_path):
    qrels = ir_measures.read_trec_qrels(qrels_path)
    run = ir_measures.read_trec_runs(run_path)
    metrics = [nDCG@10, MRR@10, Recall@100]
    rows = []
    for qid, q_measures in ir_measures.iter_calc(metrics, qrels, run):
        rec = {"qid": qid}
        for m, v in q_measures.items():
            rec[str(m)] = float(v)
        rows.append(rec)
    return pd.DataFrame(rows)  # колонки: qid, nDCG@10, MRR@10, Recall@100

bm25_df  = load_per_query_metrics("eval/qrels.csv", "eval/run_BM25.txt")
hybrid_df = load_per_query_metrics("eval/qrels.csv", "eval/run_HYBRID.txt")
```

ДОДАТОК Б

Фрагмент програмного коду. Вимірювання метрики латентності запиту

```
def load_pcts(path, path_filter="/search"):
    vals = []
    with open(path, encoding="utf-8") as f:
        for line in f:
            rec = json.loads(line)
            if rec.get("path")==path_filter:
```

```
            vals.append(rec["ms"])
    return np.percentile(vals, 50), np.percentile(vals, 95)

p50_bm25, p95_bm25   = load_pcts("logs/latency_BM25.jsonl")
p50_hyb,  p95_hyb    = load_pcts("logs/latency_HYBRID.jsonl")

delta_p50 = p50_hyb - p50_bm25
delta_p95 = p95_hyb - p95_bm25
print({"D p50 (ms)": delta_p50, "D p95 (ms)": delta_p95})
```

ДОДАТОК В

Фрагмент програмного коду. Вимірювання статистичних результатів при зміні конфігурації

```
import ir_measures, numpy as np, pandas as pd
from ir_measures import nDCG, MRR, Recall

def per_query(qrels, run):
    qrels = ir_measures.read_trec_qrels(qrels)
    run   = ir_measures.read_trec_runs(run)
    rows=[]
    for qid, ms in ir_measures.iter_calc([nDCG@10, MRR@10, Recall@100], qrels,
run):
        rows.append({"qid": qid, "nDCG@10": float(ms[nDCG@10]),
                     "MRR@10": float(ms[MRR@10]),
                     "Recall@100": float(ms[Recall@100])})
    return pd.DataFrame(rows)

base = per_query("eval/qrels.csv", "eval/run_BM25.txt")
hybr = per_query("eval/qrels.csv", "eval/run_HYBRID.txt")
ce   = per_query("eval/qrels.csv", "eval/run_HYBRID_CE.txt")

def deltas(a, b, metric):
    m = a.merge(b, on="qid", suffixes=("_a","_b"))
    return (m[f"{metric}_b"] - m[f"{metric}_a"]).values

d_ndcg = deltas(base, hybr, "nDCG@10")
```

```
# бутстреп 95% CI для середньої дельти
rng = np.random.default_rng(42)
def mean_ci(x, B=5000, alpha=0.05):
    means=[rng.choice(x, size=len(x), replace=True).mean() for _ in range(B)]
    return np.mean(x), np.quantile(means,[alpha/2,1-alpha/2])

mean, ci = mean_ci(d_ndcg)
print("D nDCG@10 =", mean, "CI95=", ci)
print("D MRR@10 =", mean, "CI95=", ci)
print("D Recall@100 =", mean, "CI95=", ci)
```