МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ГРАБІК КОСТЯНТИН ІГОРОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____ О. В. Зелінська
«____»_____ 2025 р.

РОЗРОБКА ПІДСИСТЕМИ ДЛЯ ТЕМАТИЧНОГО ПІДБОРУ КНИГ

Спеціальність 122 Комп'ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
Р. М. Бабаков, професор
кафедри інформаційних технологій,
к. т. н., доцент
_____

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____

Вінниця – 2025

АНОТАЦІЯ

Грабік Костянтин Ігорович. Розробка підсистеми для тематичного підбору книг. Спеціальність 122 «Комп'ютерні науки», освітня програма «Комп'ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця, 2025.

У бакалаврській роботі розглядається проблема ефективного тематичного підбору книг у сучасному інформаційному просторі, який характеризується значним зростанням цифрових бібліотек та онлайн-сервісів.

Обґрунтовано актуальність розробки спеціалізованої підсистеми, що автоматизує процес підбору літератури за тематичними критеріями та відповідає індивідуальним інформаційним потребам користувачів.

У роботі проведено аналіз сучасних методів і технологій, досліджено існуючі системи та сервіси, розроблено архітектуру та алгоритми функціонування нової підсистеми, а також здійснено її програмну реалізацію та тестування.

Розроблена підсистема має практичну цінність для цифрових бібліотек, освітніх платформ і сервісів пошуку літератури.

Ключові слова: Алгоритмізація, підбір книг, автоматизація, рекомендаційні алгоритми, системи рекомендацій, Python, gRPC, Docker, FastAPI.

ABSTRACT

Hrabik Kostiantyn. Development of a subsystem for thematic selection of books. Specialty 122 «Computer Science,» educational program «Computer Science.» Vasyl Stus Donetsk National University, Vinnytsia, 2025.

The bachelor's thesis considers the problem of effective thematic selection of books in the modern information space, which is characterized by a significant growth of digital libraries and online services.

The relevance of developing a specialized subsystem that automates the process of selecting literature by thematic criteria and meets the individual information needs of users is substantiated.

The work analyzes modern methods and technologies, investigates existing systems and services, develops the architecture and algorithms for the functioning of the new subsystem, and also carries out its software implementation and testing.

The developed subsystem has practical value for digital libraries, educational platforms and literature search services.

Keywords: Algorithmization, book selection, automation, recommendation algorithms, recommendation systems, Python, gRPC, Docker, FastAPI.

ЗМІСТ

ВСТУП

У сучасному інформаційному суспільстві обсяг доступних даних про книги зростає в геометричній прогресії. За статистикою, щодня генерується понад 2,5 квінтильйона байтів інформації, значна частина яких складається з текстового контенту. Це створює нові виклики для читачів: надлишок книжкових фондів та електронних ресурсів ускладнює вибір літератури, що найповніше відповідає індивідуальним інтересам користувача. Традиційні пошукові системи, які переважно опираються на введені ключові слова, не враховують індивідуальне вподобання чи контекст користувача. У зв'язку з цим надається перевага рекомендаційним системам – автоматизованим програмам, що на основі даних про інтереси, поведінку та вподобання користувачів формують персоналізовані пропозиції відповідної літератури.

Рекомендаційні системи стали новітнім інструментом фільтрації інформації в онлайн-середовищах і вже довели свою ефективність у багатьох галузях. Застосування таких систем знижує час пошуку і вибору книжок, підвищує задоволеність користувачів і забезпечує більшу залученість аудиторії. Особливо це актуально в умовах масового розповсюдження електронних бібліотек та онлайн-магазинів, де кожен читач очікує отримати рекомендацію книги, що точно відповідає його уподобанням. Бурхливий розвиток технологій машинного навчання, обробки природної мови та великих даних відкриває нові можливості для вирішення задач персоналізації контенту, зокрема тематичного підбору літератури. Таким чином, розробка підсистеми тематичного підбору книг стає надзвичайно актуальною задачею, оскільки дозволяє вирішувати проблему інформаційного перевантаження і забезпечувати користувачеві ефективний доступ до релевантних книжкових матеріалів.

Метою даного дослідження є розробка алгоритмічної підсистеми тематичного підбору книг, яка забезпечить персоналізовані рекомендації літератури на основі аналізу інтересів та вподобань користувачів. У межах поставленої мети передбачається дослідити математичні та обчислювальні підходи до побудови рекомендаційних моделей, а також створити ефективний

прототип системи, здатний інтегруватися у бібліотечні чи інформаційно-пошукові платформи для автоматизованого підбору книг відповідно до тематичних запитів користувачів.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1. Аналіз предметної області. Вивчити існуючі рекомендаційні системи та підходи до персоналізованого підбору контенту, зокрема приклади книгосховищ і сервісів рекомендацій (Amazon, Goodreads та ін.), що демонструють застосування рекомендаційних алгоритмів у літературній сфері.
2. Огляд методів аналізу інтересів користувачів. Дослідити методології побудови профілю користувача та моделі його уподобань (колаборативна фільтрація, контентна фільтрація, гібридні моделі, кластеризація на основі тематичних ознак тощо).
3. Визначення вихідних даних. Охарактеризувати необхідний набір даних для реалізації системи: метадані книжок (заголовки, автори, тематичні маркери, анотації, тематичні категорії та класифікаційні коди) і дані про користувачів (історія читання, оцінки, явно зазначені інтереси).
4. Розробка алгоритму тематичного підбору. Сформулювати математичні моделі та алгоритми, які б забезпечили тематичне порівняння книг і профілю користувача; наприклад, застосування векторних моделей тексту, схожості тем, методів факторизації матриць або нейронних мереж.
5. Проектування підсистеми. Спроєктувати архітектуру програмного модуля або сервісу, який реалізує вибрані алгоритми тематичного підбору книг. Визначити інтерфейс взаємодії з користувачем та з іншими компонентами інформаційної системи.
6. Реалізація та експериментальна перевірка. Розробити прототип підсистеми (наприклад, веб-додаток або модуль в існуючій бібліотечній системі), провести тестування на репрезентативних

даних і оцінити якість рекомендацій за критеріями точності та задоволення користувачів.

Об’єктом дослідження є процес автоматизованого підбору книжкових ресурсів за темами та інтересами користувачів у цифровому середовищі. Іншими словами, це інженерно-інформаційна система або підсистема інформаційно-пошукового ресурсу (бібліотеки, електронної книжкової платформи тощо), яка здійснює відбір літературних творів на основі заданих критеріїв тематики та уподобань читача.

Предметом дослідження є методи, моделі та алгоритми тематичного підбору літератури у складі рекомендаційних систем. Зокрема, це включає: моделі профілювання користувача (аналіз його читацької поведінки і уподобань), методи аналізу та зіставлення тематичних характеристик книг (класифікація, семантичний аналіз), математичні підходи до формування рекомендацій (колаборативні та контентні методи, гібридні рішення, машинне навчання і т.д.), а також критерії оцінки ефективності таких рекомендацій.

Теоретичне значення дослідження полягає у систематизації та адаптації сучасних підходів до рекомендованих систем в контексті книжкових тематик. Робота сприятиме кращому розумінню того, як поєднати інформаційно-пошукові технології з методами штучного інтелекту для точного тематичного добору літератури. Теоретичні результати можуть бути використані для подальших наукових досліджень у галузях інформаційно-пошукових систем, машинного навчання та обробки природної мови.

Практичне значення полягає в створенні функціонального програмного продукту, який може бути інтегрований в існуючі бібліотечні системи або онлайн-платформи для читачів. Така підсистема рекомендацій дозволить автоматично генерувати найбільш релевантні книжкові пропозиції, зменшивши зусилля користувача при пошуку потрібної літератури і підвищивши його задоволеність сервісом. Завдяки використанню персоналізованих рекомендацій з урахуванням інтересів читача підвищується ефективність інформаційно-бібліотечного обслуговування, що важливо для сучасних електронних бібліотек і

освітніх проектів. Упровадження отриманих результатів сприятиме збереженню часу користувачів та підвищенню якості доступу до знань, що відповідають їхнім тематичним запитам.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИМОГ ДО ПІДСИСТЕМ

1.1 Методи представлення інтересів користувача

Рекомендаційні системи (RS) призначені для прогнозування того, які об’єкти (книги, фільми тощо) можуть зацікавити певного користувача на основі інформації про попередні вподобання користувачів[1, с. 10]. У сучасній літературі виділяють декілька основних підходів до представлення інтересів користувача і формування рекомендацій: фільтрація на основі контенту, колаборативна фільтрація, підходи на основі знань, гібридні методи, а також новітні техніки на кшталт embedding-представлень користувачів/об’єктів[1, с. 11]. Кожен з цих підходів має свої переваги і недоліки та використовується для моделювання профілю користувача з метою максимально точного передбачення його уподобань.

1.1.1 Контентна фільтрація

Методи, засновані на контенті, формують профіль користувача на основі властивостей і характеристик тих об’єктів, які він вподобав у минулому[1, с. 12]. Ідея контентної фільтрації полягає у рекомендації нових елементів, подібних до тих, що користувач вже оцінював позитивно. У випадку з книгами це можуть бути спільні атрибути творів: жанри, тематики, автори, ключові слова, сюжетні елементи тощо. Наприклад, якщо користувачеві сподобалася книга певного жанру або автора, система може рекомендувати інші книги з подібними жанровими ознаками чи того ж автора. Таким чином будується індивідуальний профіль інтересів, який відображає контентні вподобання користувача.

Застосування контентної фільтрації є популярним у багатьох сервісах інтернет-магазинів, онлайн-бібліотек і стримінгових платформ. Наприклад, у відеосервісі Netflix або книжкових каталогах рекомендації часто ґрунтуються на категоріях контенту (жанр, тематика), щоб новий користувач одразу міг отримати деякі релевантні поради навіть без історії взаємодій[1, с. 13]. Основна перевага підходу, заснованого на контенті, – це можливість працювати з інформацією про самі об’єкти і видавати рекомендації на основі схожості контенту. Завдяки цьому

метод не залежить від наявності великої кількості інших користувачів чи оцінок і менш вразливий до проблеми холодного старту для нових об’єктів (якщо відома їх контентна інформація). Недоліками є те, що система не враховує колективного досвіду інших користувачів – рекомендації обмежуються лише тим, що вже схоже на вподобання користувача. Через це можливе замикання на вузькому колі тем (низька новизна), а на початкових етапах, поки даних про вподобання замало, точність рекомендацій може бути невисокою[1, с. 14]. Також контентні методи потребують якісних метаданих про об’єкти (описів книг, жанрових рубрик тощо) та засобів їх обробки (наприклад, NLP для аналізу анотацій книг).

1.1.2 Колаборативна фільтрація

Колаборативна (спільна) фільтрація – найстаріший і один із найпоширеніших підходів до побудови рекомендацій. Він грунтується на аналізі поведінки та вподобань групи користувачів для виявлення шаблонів, спільних для багатьох людей[1, с. 15]. Простими словами, метод колаборативної фільтрації рекомендує користувачу ті об’єкти, які сподобалися іншим користувачам зі схожими смаками. На відміну від контентного підходу, тут профіль користувача формується не явними атрибутами книг, а списком оцінених книг (позитивно чи негативно), і порівнюється з профілями інших користувачів. Якщо знайдено групу “сусідів” – читачів, які мали багато спільних оцінок з цим користувачем – то книжки, високо оцінені тими сусідами, але ще не читані цільовим користувачем, рекомендовано йому. Аналогічно можна співставляти схожі об’єкти: якщо дві книги отримують високі оцінки від одних і тих самих людей, вони вважаються подібними, і користувачеві, що вподобав одну з них, варто запропонувати іншу.

Колаборативна фільтрація була успішно застосована в різних доменах – від інтернет-магазинів до стримінгових сервісів[1, с. 15]. Класичним прикладом є алгоритм рекомендацій на Amazon.com, де було впроваджено масштабовану схему item-to-item співфільтрації: для кожного товару знаходяться подібні товари на основі історії спільних покупок, і при перегляді товару користувач бачить

розділ «Customers who bought this item also bought…»[2, с. 10]. Цей підхід виявився дуже ефективним у реальному часі і добре масштабується на величезні масиви даних Amazon[2, с. 15]. Перевагою колаборативних методів є те, що вони автоматично враховують складні вподобання, не потребуючи явного опису властивостей книг: система може знайти неочевидні зв’язки між творами на основі поведінки великої кількості читачів. Також такі рекомендації здатні пропонувати щось несподіване (вихід за межі прочитаних жанрів), оскільки спираються на колективний досвід. Недоліки колаборативної фільтрації: по-перше, проблема холодного старту – неможливо дати хорошу рекомендацію новому користувачу, який ще не залишив оцінок, або для нової книги без читачів[2, с. 21]. По-друге, для побудови якісних рекомендацій потрібна досить велика база даних взаємодій (оцінок) від користувачів; при малій кількості даних рекомендації будуть ненадійними. По-третє, чисто колаборативні алгоритми менш прозорі та інтерпретовані – важко пояснити користувачу, чому рекомендовано ту чи іншу книгу, окрім як “такі як ви її вподобали”[2, с. 27].

Колаборативні алгоритми поділяють на підходи пам’яті (memory-based) та модельні (model-based) [2, с. 30]. У перших рекомендація обчислюється безпосередньо на основі матриці оцінок (методи сусідів – user-based та item-based), тоді як у других спочатку будують математичну модель (наприклад, методом факторизації матриці або з застосуванням машинного навчання), і вже на основі параметрів цієї моделі роблять прогноз рейтингу чи вибір топ-N рекомендацій. Зокрема, в нетфлікс-подібних задачах великого розміру широко застосовують латентно-факторні моделі – методи декомпозиції матриці “користувач-об’єкт” (наприклад, SVD, ALS) для знаходження прихованих факторів вподобань. Кожен користувач і кожна книга при цьому проектуються у спільний багатовимірний простір – вектори-«фактори», які чисельно відображають уподобання користувача та характеристики книги. Добуток цих векторів дає прогноз оцінки. Така факторизація фактично створює embedding-представлення для користувачів і об’єктів, що описує їх у вигляді набору числових характеристик (далі розглянемо embeddings окремо). В цілому,

колаборативна фільтрація залишається однією з базових технологій побудови RS, і більшість великих платформ використовують принаймні її елементи в поєднанні з іншими методами[3, с. 40].

1.1.3 Підходи на основі знань та інші

Окремою категорією є методи рекомендацій, засновані на знаннях (knowledge-based). Вони не покладаються ні на схожість контенту, ні на статистику спільних оцінок, а використовують доменні знання про предметну область та про самого користувача для генерації рекомендацій[3, с. 44]. Наприклад, у випадку книжкових рекомендацій знаннями можуть бути експертні правила або онтології: система може враховувати літературні уподобання, цілі читання (навчальна література, художня тощо), рівень складності тексту, мову, наявність в бібліотеці і т.д., і на основі цього виводити, які книги найбільше відповідають потребам користувача. Knowledge-based системи часто реалізують принцип case-based reasoning: підбір об’єкта, схожого на зазначені користувачем бажані характеристики (наприклад, “книга, схожа на X, але про Y”). Також до цієї категорії іноді відносять простіші демографічні підходи, де рекомендації ґрунтуються на належності користувача до певної категорії (наприклад, окремі добірки книг для підлітків, для фахівців галузі, для певного вікового сегмента тощо) [3, с. 51].

Методи на основі знань мають перевагу інтерпретованості та контрольованості – можна явно пояснити, чому рекомендується той чи інший твір (правило або характеристика). Вони корисні, коли немає достатніх даних про оцінки (новий користувач або книга) або коли рекомендації мають відповідати строгим критеріям (наприклад, рекомендація навчальної літератури за навчальним планом). Недолік – висока вартість розробки і підтримки: потрібно акумулювати значний обсяг знань, залучати експертів, підтримувати актуальність правил. Тому чисті knowledge-based системи зустрічаються рідко; частіше елементи знань додають до інших методів для покращення їх роботи.

1.1.4 Гібридні методи

Жоден з наведених вище підходів не є універсальним: кожен має обмеження (як описано, контентний і колаборативний методи страждають від протилежних проблем, а знання-вмісні системи важко масштабувати). Тому в сучасних рекомендаційних системах поширені гібридні підходи, що поєднують кілька методів для досягнення кращих результатів[3, с. 5-10]. Мета гібридизації – компенсувати недоліки одного методу сильними сторонами іншого[3, с. 23]. Наприклад, класичний підхід – комбінувати контентну і колаборативну фільтрацію: спочатку використовувати колаборативний підхід, але для нових користувачів чи об’єктів (коли нема достатньо оцінок) перемикатися на контентний режим. Інший варіант – рекомендувати результат обох методів і об’єднувати списки. Можливе також глибше поєднання: включення контентних ознак в модель колаборативної фільтрації, або навпаки – використання колаборативно визначених переваг при контентному порівнянні[3, с. 35-47]. Є приклади побудови єдиної моделі, що інтегрує одночасно правила обох типів, наприклад факторизаційні машини (Factorization Machines) або графові підходи з урахуванням різнотипних зв’язків (коли користувачі, книги, автори, жанри утворюють знаннєву графову структуру).

Практично всі великі онлайн-сервіси сьогодні використовують гібридні рекомендатели, що дозволяє їм забезпечити і високу точність, і прийнятну новизну/диверсифікованість результатів[3, с. 51-63]. Зокрема, соціальні книжкові мережі (розглянуті далі) поєднують колаборативні алгоритми з елементами на основі контенту і залучають явні дії користувачів (рейтинги, «віртуальні полиці» тощо) як сигнали для покращення рекомендацій[3, с. 65-78].

1.1.5 Embedding-представлення і глибинне навчання

Останнім часом бурхливого розвитку набули методи побудови рекомендаційних моделей на основі embeddings – вбудованих представлень, що отримуються з даних за допомогою нейронних мереж або інших методів машинного навчання[3, с. 81-95]. Embedding у контексті RS – це відображення

дискретних об’єктів (таких як ID користувача, ID книги, або інших ознак) у багатовимірні безперервні вектори таким чином, що схожі за вподобаннями об’єкти отримують близькі вектори[4]. Такі векторні представлення здатні захоплювати приховані зв’язки і патерни, покращуючи якість рекомендацій. Наприклад, якщо два романи часто читають одні й ті самі люди, модель embedding розташує їхні вектори близько один до одного, навіть якщо за жанром вони різні – тобто виявиться неочевидна, але важлива спільність у аудиторії. Embeddings-вектори можуть навчатися різними способами: шляхом факторизації матриці оцінок (що еквівалентно тренуванню простого нейронного автоенкодера), через згорткові або рекурентні нейронні мережі (особливо для послідовних рекомендацій, аналізу тексту рецензій тощо), або за допомогою методів типу Word2Vec/Item2Vec (розглядаючи послідовності книг, які читає користувач, як «речення», і слова в них – як книги)[4]. Результатом є щільні вектори розмірністю порядку 50–200, які можна використовувати для оцінки близькості користувача і книги (через косинусну подібність чи скалярний добуток) або як ознаки у складнішій моделі.

Переваги embedding-підходів – висока гнучкість і точність: вони можуть врахувати дуже багато факторів одночасно (взаємодії, тексти, метадані, соціальні зв’язки тощо) і виявляти складні нелінійні залежності. В сучасних масштабних RS такі методи є основою: наприклад, алгоритми YouTube, Goodreads та ін. використовують багаторівневі нейронні архітектури для навчання вбудованих представлень на основі історії поведінки користувача[5]. Недоліки – велика обчислювальна складність тренування моделей, потреба у великих наборах даних для уникнення перенавчання, а також складність інтерпретації: embedding-вектори є «чорними скриньками», тому пояснити, чому система рекомендує ту чи іншу книгу, нелегко[5]. Для подолання останнього виклику розробляється напрям інтерпретованих нейронних рекомендацій, де моделі намагаються будувати пояснення до векторних ознак (наприклад, прив’язувати виміри embedding до конкретних жанрів чи тегів)[5]. Загалом, embedding-представлення

суттєво підвищили якість рекомендацій у багатьох сферах, тому їх обґрунтовано обрано для побудови математичного ядра даної підсистеми.

1.2 Вимоги до якості математичного представлення інтересів

Оцінюючи вибір підходу для представлення інтересів користувача, слід визначити критерії якості цього представлення. Від нього залежить успіх усієї підсистеми, адже невдалий профіль користувача призведе до нерелевантних рекомендацій. Основні вимоги до моделі представлення такі:

- Точність та релевантність. Модель уподобань повинна забезпечувати високоточні рекомендації – ті, що справді відповідають смакам користувача. Точність вимірюють метриками на зразок RMSE (для прогнозування рейтингів) чи Precision/Recall, nDCG, MAP (для списків рекомендацій)[7, с. 76]. Важливо, щоб представлення захоплювало основні вподобання користувача і дозволяло відрізняти цікаві для нього книги від нецікавих. Релевантність результатів тісно пов’язана з точністю: система має рекомендувати саме той контент, який користувач очікує побачити або який його приємно здивує. При цьому, окрім простого збігу з минулими інтересами, часто оцінюють і новизну та диверсифікованість рекомендацій – тобто здатність моделі пропонувати не лише передбачувані, але й різнопланові та несподівані книги, що можуть зацікавити користувача[7, с. 77]. Баланс між точністю і новизною є важливим критерієм якості RS.
- Адаптивність і динамічність. Уподобання користувачів змінюються з часом, тому представлення повинно бути адаптивним – здатним оновлюватися під впливом нової інформації. Це означає підтримку динамічного профілю: якщо користувач почав читати книги нового жанру, модель повинна достатньо швидко відобразити цю зміну і почати рекомендувати нові об’єкти. Адаптивність стосується і стійкості до проблеми холодного старту – якісне представлення повинно хоча б частково працювати, навіть коли даних мало

(наприклад, використовуючи апріорні знання про схожі книги або демографічну інформацію). У наукових працях запропоновано підходи, що враховують контекст використання як фактор для адаптації: модель може змінювати свої рекомендації залежно від часу доби, сезону, місцезнаходження або поточного настрою користувача[7, с. 77]. Загальна вимога – гнучкість моделі в умовах динамічного середовища (нові книги, нові рецензії, тренди) та можливість масштабування на більшу кількість користувачів і даних без втрати якості.

- Інтерпретованість та прозорість. Важливо, щоб представлення інтересів (а отже і робота рекомендаційного алгоритму) була по можливості зрозумілою для людини. Інтерпретованість означає, що можна пояснити, які фактори впливають на рекомендації: напряму (через зрозумілі ознаки, як-то «користувач полюбляє жанр фантастики, тому рекомендовано цей роман») або опосередковано (через пояснення від моделі – наприклад, показати цитати з рецензій, що пов’язують дві книги). Дослідження показують, що наявність пояснень до рекомендацій підвищує довіру користувачів та їх задоволеність системою[7, с. 78]. Прозорий алгоритм також легше коригувати й підтримувати – розробники можуть краще розуміти, чому модель дає певні результати[7, с. 79]. Тому, хоча висока точність часто досягається «чорними скриньками» (нейронними мережами), у підсистемах, що працюють з культурним контентом (книгами), бажано забезпечити баланс між точністю і пояснюваністю. Мій підхід приділяє увагу тому, щоб моделі уподобань користувачів були хоча б частково трактованими (наприклад, через прив’язку latent-факторів до жанрових тегів) і дозволяли генерувати прості пояснення типу «Рекомендується, тому що ви високо оцінили Такого-то автора» або «...жанр фентезі».

- Інші якості. Серед інших вимог можна зазначити стійкість до шумів і аномалій (представлення не повинно різко спотворюватися через поодинокі випадкові оцінки, помилки даних чи зумисні маніпуляції), обчислювальну ефективність (модель має дозволяти швидке оновлення і швидку генерацію рекомендацій, навіть при великій кількості користувачів та книг) та узагальнюваність (можливість використання моделі на суміжних задачах або перенавчання на нових даних без втрати попередньо здобутих знань). Усі ці характеристики мають бути враховані при розробці математичного ядра підсистеми рекомендацій, щоб забезпечити високу якість її роботи.

1.3 Огляд наукових досліджень з рекомендаційних систем та факторів вибору контенту

Системи рекомендацій інтенсивно досліджуються з кінця 90-х років і до сьогодні, набувши широкого застосування у різних доменах: електронна комерція, медіа-контент, соціальні мережі, електронне навчання тощо[8]. Класична робота Resnick & Varian (1997) ввела сам термін recommender system як інструмент боротьби з інформаційним перевантаженням користувачів у цифрову епоху[8]. Відтоді з’явилося багато методів (див. розділ 1.1), і вчені зосередили увагу як на алгоритмічних удосконаленнях, так і на розумінні факторів, що впливають на сприйняття рекомендацій користувачами. Останнє включає вивчення психологічних, соціальних і контекстних аспектів: чому користувач обирає той чи інший контент із запропонованого. Розглянемо кілька ключових факторів, висвітлених у наукових публікаціях.

Жанрові вподобання і тематичні інтереси. Безумовно, базовим фактором вибору книг є жанрово-тематичні смаки читача. Більшість рекомендаційних алгоритмів мають на меті саме виявити ці стійкі інтереси – наприклад, любов до історичних романів, наукової фантастики чи певного автора – і пропонувати новинки, що відповідають цьому профілю. У роботі Aggarwal (2016) відзначено, що персоналізація контенту навколо вподобаних категорій є ядром цінності RS[9]. В академічних дослідженнях широко використовуються датасети з

жанровою розміткою книг і оцінками, щоб вивчати, як моделі вчаться розпізнавати прихильність користувача до жанру та експлуатувати це для рекомендацій[9]. З іншого боку, надмірне фокусування тільки на жанрі може знижувати якість рекомендацій – користувачі нерідко хочуть розширювати горизонти. Тому останні алгоритми включають метрики диверсифікації, щоб урізноманітнити список рекомендацій і показати щось за межами основних тем користувача, підтримуючи інтерес і залученість[9].

Емоційний стан і настрій. Все більше уваги приділяється афективним факторам – тому, як емоції користувача впливають на його вибір контенту. На інтуїтивному рівні зрозуміло, що людина обирає книгу відповідно до настрою: комусь у певний момент хочеться легкої розважальної літератури, а іншим разом – серйозного, глибокого чтива. Емоційна забарвленість твору (наприклад, веселий роман, меланхолійна драма, мотиваційна література) може або притягувати, або відштовхувати читача залежно від його внутрішнього стану. У роботі Polignano et al. (2021) підкреслено, що емоції відіграють важливу роль у рішеннях користувача щодо контенту, і рекомендательні моделі мають навчитися враховувати “емоційний контекст” для надання доречніших порад[10]. З’явився напрям emotion-aware recommender systems: дослідники пропонують аналізувати тон і емоції у відгуках або соціальних мережах, щоб зрозуміти емоційний вплив книг[10]. Наприклад, у статті Lutan & Badica (2023) розроблено методи рекомендації книг, що використовують емоції з рецензій: спочатку визначають, які емоційні реакції викликають книги у читачів (через аналіз тексту відгуків – радість, сум, здивування тощо), а тоді рекомендують книги, емоційний профіль яких відповідає вподобанням користувача[10]. Результати показали, що врахування емоційної складової покращує довіру до рекомендацій та різноманітність списку, оскільки користувач бачить більш «людяні» поради, а не просто автоматичний збіг за жанром. Отже, науковці дійшли згоди, що впровадження афективних даних (навіть таких непрямих, як смайли у відгуках чи тон коментарів) збагачує модель користувача і покращує якість рекомендацій у сферах, де емоції важливі (художня література, музика, кіно).

Контекст використання і ситуаційні фактори. Контекст – це сукупність обставин, за яких користувач взаємодіє з системою: час, місце, платформа, соціальне оточення, ціль використання тощо. Контекстуальна інформація може суттєво впливати на вибір контенту. Наприклад, книжкові вподобання можуть змінюватись між буднями і вихідними, або залежати від сезону (попит на літературу певної тематики зростає під час свят, подій). Context-aware рекомендаційні системи (CARS) досліджуються з метою використання таких даних для тоншої персоналізації[11]. Adomavicius et al. (2011) запропонували тристоронню модель “користувач – об’єкт – контекст”, де уподобання визначаються не статично, а як функція контексту (наприклад, U любить X жанр при читанні вранці vs ввечері може любити інший жанр). Інші роботи вводять поняття сесійних рекомендацій: коли система аналізує короткостроковий контекст поточної сесії (останні перегляди книг) і надає рекомендації, релевантні саме до цієї сесії, навіть якщо вони відхиляються від глобального профілю користувача. У новинарних і відео-рекомендаційних сервісах контекст (наприклад, геолокація, поточні тренди) вже став невід’ємним компонентом моделей[11]. Для книжкових рекомендацій контекст теж має значення: дослідження відзначають, що читачі можуть мати різні читацькі ролі – наприклад, студент шукає книгу для навчання (контекст «навчання») або ту ж особу цікавить художній роман для відпочинку (контекст «дозвілля»). У системі Goodreads користувачі навіть можуть вказувати “currently reading”, “want to read” – своєрідні контекстні статуси, що також дозволяє зрозуміти, для чого користувач збирається читати книгу. У науковій статті Світлани Водолазької (2013) про книжкові соцмережі зазначено, що такі статуси і теги («віртуальна полиця прочитаного, бажаного») використовуються як сигнали очікувань користувача і допомагають системі краще вичленувати його інтереси та цілі читання[11]. Отже, врахування контексту – часового, цільового та соціального – розглядається як важливий напрям покращення рекомендацій, і сучасні підсистеми повинні бути здатні інтегрувати ці фактори.

Вплив соціального середовища і популярності. Ще один аспект – соціальні фактори: на вибір книги може впливати те, що читають друзі, загальний рейтинг твору, популярність у масовій культурі. Рекомендаційні моделі в соціальних мережах, таких як Goodreads чи LiveLib, враховують мережу контактів: часто пропонуються книги, які високо оцінили друзі користувача або люди зі схожими смаками[11]. Також рейтингові списки (“топ-100 книг року”, “найпопулярніші фентезі місяця”) самі по собі виконують рекомендаційну функцію і впливають на вибір – це своєрідна базова рекомендаційна система, заснована на глобальній популярності. Дослідники попереджають про ефект “багато хто обирає – і я оберу”: алгоритми, що занадто підсилюють популярні об’єкти, можуть призводити до монокультури, коли всі читають одне і те ж[11]. Тому сучасні RS балансують між персоналізацією під користувача і загальними трендами. Наукові роботи з fairness і diversity у рекомендаціях пропонують метрики, що зменшують вплив тільки популярності на рекомендації, щоб кожен користувач отримував щось індивідуальне, а не тільки бестселери. Водночас повне ігнорування соціального доказу теж небажане – адже рекомендація дуже нішевого контенту може не сподобатись, якщо він занадто далекий від звичного кола читання. Оптимальним є урахування соціального фактору як ще однієї ознаки моделі: багато алгоритмів додають фактор популярності або соціальні зв’язки до профілю користувача (наприклад, граф довіри між користувачами, модель впливу друзів), що дозволяє трохи змістити рекомендації у бік того, що схвалено спільнотою.

Підсумовуючи, наукові публікації підтверджують багатовимірність проблеми рекомендацій: щоб успішно підібрати книгу читачеві, недостатньо просто знати його жанрові вподобання. Треба врахувати комплекс факторів – від стабільних (улюблені теми, автори) до динамічних (настрій, контекст, тренди) – і збалансувати їх у моделі. Це зумовило появу численних гібридних дослідницьких напрямів: контекстно-орієнтовані RS, емоційно-орієнтовані RS, соціально-орієнтовані RS тощо. У моїй підсистемі, зважаючи на її тематичну спрямованість (книги), доцільно включити принаймні частину таких факторів:

базове ядро моделі має відображати стійкі літературні інтереси користувача, але архітектура повинна дозволяти адаптацію до контексту і врахування соціальних сигналів у майбутньому розвитку.

1.4 Порівняння існуючих систем рекомендацій книг

На ринку існує декілька відомих систем і сервісів, що надають книжкові рекомендації. Розглянемо чотири з них – Goodreads та Google Books– з точки зору використаних підходів і їх сильних/слабких сторін.

1.4.1 Goodreads

Goodreads – одна з найбільших глобальних соціальних мереж для любителів книг, нині належить Amazon. Вона поєднує елементи каталогізації прочитаного, оглядів, соціальної взаємодії і рекомендацій. Рекомендаційна підсистема Goodreads побудована головно на колаборативній фільтрації: сервіс збирає рейтинги (оцінки від 1 до 5 зірок) мільйонів користувачів і на їх основі формує персональні списки рекомендацій. Коли користувач оцінив достатню кількість книжок, алгоритм знаходить інших користувачів зі схожими оцінками і пропонує книги, які ті оцінили високо, але цільовий користувач ще не читав. Також Goodreads використовує елементи контентного підходу – наприклад, враховує жанри книжок на полицях користувача. Сильна сторона Goodreads – величезна база даних контенту, згенерованого користувачами: мільйони рейтингових пар «користувач–книга», рецензій, цитат, тегів. Це дає алгоритмам багатий матеріал для навчання і тонкого налаштування рекомендацій під різні смаки. Крім того, є соціальний фактор: користувач може бачити, що читають його друзі, переглядати списки популярних книг, вступати в книжкові клуби – усе це побічно виконує рекомендаційну функцію (через непрямий колаборативний фільтр за інтересами спільнот).

Втім, слабкі сторони також відомі. Алгоритми Goodreads нерідко критикують за дещо застарілий підхід і брак інновацій: вони можуть видавати доволі передбачувані рекомендації (наприклад, надто орієнтуватися на середні рейтинги і популярність). Користувачі відзначають, що система часто рекомендує

бестселери або книги тих самих авторів, яких уже читали, замість того щоб відкрити щось нове. Такий ефект пов’язаний з тим, що чиста колаборативна фільтрація схильна посилювати популярні об’єкти і пропонувати «найочевидніший» вибір. Крім того, для новачків на Goodreads рекомендації з’являються лише після оцінювання певної кількості книг, що створює бар’єр холодного старту. Соціальні дослідження також показали, що присутня маніпулятивна компонента: користувачів можуть приваблювати високорейтингові твори просто через їх статус, що зменшує різноманітність читання[12]. Незважаючи на ці недоліки, Goodreads залишається потужною платформою: її сильна сторона – спільнота. Рекомендація підкріплюється реальними відгуками і обговореннями, що додає довіри. Отже, Goodreads демонструє ефективність колаборативного підходу на великій спільноті, але потребує удосконалень в напрямі диверсифікації та сучасних методів (які, ймовірно, Amazon поступово інтегрує).

1.4.2 Google Books

Google Books – сервіс від Google, що надає пошук і попередній перегляд оцифрованих книг. Хоча Google Books не є суто рекомендаційною платформою, він містить елементи рекомендацій у вигляді секцій «Related books» та персональних порад у додатку Google Play Books. Сильна сторона Google – це контентно-орієнтований підхід у поєднанні з потужністю пошукових технологій. Компанія проіндексувала тексти мільйонів книг, тому може визначати семантичну схожість між ними. Коли користувач переглядає книгу, Google Books пропонує схожі за змістом видання (наприклад, ті, що часто згадуються разом, мають схожі ключові фрази). Фактично, використовується контентна рекомендація на основі повнотекстового аналізу та метаданих. В екосистемі Google Play Books є також AI-підсилені рекомендації: у 2023 році додано кнопку «Get recommendations», що за допомогою ШІ шукає книги, які можуть сподобатись користувачу на основі його бібліотеки та історії читання. Це, ймовірно, реалізовано як комбінація колаборативних сигналів (агрегованих через

обліковий запис Google) і контентного ранжування за тематичною близькістю. Перевагою Google є і контекстний пошук: користувач може шукати конкретні теми, цитати – і отримувати у видачі рекомендації книг, де це є. Таким чином, Google Books добре справляється з задачами відкриття літератури з конкретної тематики.

Слабкі сторони Google Books як рекомендаційного сервісу: відсутність розвиненої соціальної складової і явного рейтингового механізму. Рекомендації від Google доволі неперсоналізовані у порівнянні з Goodreads чи Amazon – вони більше схожі на результати пошуку схожих книг. Нема можливості оцінити книгу зірками чи бачити, що прочитали друзі (Google+ було інтегровано в минулому, але зараз соцмережа закрита). Тому Google не знає “смаків” користувача настільки глибоко, як сервіси кшталт Goodreads (якщо лише користувач не зберігає детальну історію читання на Google Play). Ще одна слабкість – рекомендації змішані з пошуком: користувач може не розрізняти, де алгоритм радить новинку, а де просто видає популярний результат. Для моєї підсистеми досвід Google Books показує цінність аналізу контенту: використання індексації текстів, анотацій, тегів для тематичного підбору книг. Я планую застосувати подібний принцип (тематичне групування через embeddings), але уникнемо недоліку недостатньої персоналізації, поєднавши це з профілем користувача.

Аналіз Goodreads і Google Books показав, що успішна система рекомендацій книг повинна поєднувати колективний досвід (рейтинг, оцінки, поведінка мас користувачів) з особистими вподобаннями і контентним розумінням книг. Goodreads робить наголос на соціальній і колаборативній складовій, Google – на алгоритмічному аналізі контенту. Моя підсистема прагне об’єднати ці сильні сторони: використати колаборативні embedding-моделі для виявлення прихованих інтересів, врахувати тематичну схожість книг для тематичного підбору, і забезпечити масштабованість та інтеграцію в продуктову інфраструктуру через механізми мікросервісів і API.

1.5 Аналіз інструментів розробки

1.5.1 Мова програмування

Для розробки підсистем рекомендацій існує кілька популярних мов програмування – зокрема Python, Java та Scala. Кожна з них має свої переваги у контексті побудови рекомендаційних сервісів. Python сьогодні особливо поширений у сфері машинного навчання та рекомендативних алгоритмів завдяки багатій екосистемі бібліотек і простоті прототипування. Як зазначається у роботі Майкла Екстранда, інструментарій LensKit спочатку був реалізований на Java (у 2010 р.), але його нова версія повністю перейшла на Python, щоб скористатися «широкою та зростаючою екосистемою Python для наукових обчислень», що включає бібліотеки scikit-learn, TensorFlow, PyTorch тощо. Це підкреслює, що Python забезпечує дослідникам і розробникам більшу гнучкість і доступ до сучасних інструментів машинного навчання, спрощуючи реалізацію складних моделей. Окрім того, синтаксис Python є відносно простим, а спільнота – великою, що пришвидшує розробку та налагодження системи рекомендацій.

Java традиційно використовувалася в масштабованих серверних застосунках і також має напрацювання для рекомендаційних систем. Існують бібліотеки на Java, наприклад Apache Mahout або LibRec, що реалізують алгоритми колаборативної фільтрації. Java відома продуктивністю та строгістю типізації, що може бути корисним у великих промислових рішеннях. Деякі масштабні рекомендаційні сервіси будувалися з використанням Java/Scala. Зокрема, компанія Netflix використовує алгоритм ALS (Alternating Least Squares) на базі фреймворку Apache Spark (Scala) для побудови власної системи рекомендацій. Використання Scala/Java та Spark виправдане для обробки дуже великих обсягів даних у розподіленому середовищі, де потрібна висока продуктивність і паралелізм. Scala, як JVM-мова, тісно інтегрована з екосистемою великих даних (той же Apache Spark написаний на Scala). Scala поєднує переваги ООП і функціонального підходу, що сподобалося розробникам систем обробки потоків подій та рекомендацій у реальному часі. Наприклад, відомо, що Spotify свого часу активно застосовував Scala/Spark для

рекомендаційного рушія в умовах big data. Однак поріг входження для Scala вищий, ніж для Python, а екосистема спеціалізованих бібліотек для рекомендацій менш розвинена.

При порівнянні мов слід врахувати баланс між продуктивністю та швидкістю розробки. Мови на кшталт C++ забезпечують найвищу швидкодію алгоритмів, але розробка вимагає значно більше часу і зусиль. Java/Scala пропонують високу продуктивність і масштабованість, що важливо для enterprise-рішень з мільйонами користувачів. Натомість Python вирізняється простотою реалізації моделей і наявністю величезної кількості готових інструментів для аналізу даних та побудови моделей. Багато сучасних досліджень і прототипів рекомендаційних алгоритмів виконуються саме на Python. Це обумовлено тим, що на етапі експериментів і розробки точність і гнучкість важливіші за максимально можливу продуктивність коду. До того ж, повільні місця Python-коду можуть бути оптимізовані за рахунок бібліотек на C/C++ (що використовується у тих же бібліотеках TensorFlow, NumPy тощо).

З огляду на вищезазначене, для розробки підсистеми тематичних рекомендацій книг обрано Python. По-перше, Python має широкий вибір готових рішень для реалізації алгоритмів рекомендацій (розглянутих нижче), що значно скоротить час розробки. По-друге, поріг входження для нових розробників нижчий – код на Python є компактнішим і легшим для розуміння, що важливо в умовах курсового/дипломного проєкту. По-третє, екосистема Python активно підтримується спільнотою дослідників рекомендаційних систем: є приклади як успішних наукових експериментів, так і промислових впроваджень на Python. Зважаючи на тематику проєкту – рекомендації книг – обсяги даних і навантаження не настільки великі, щоб вимагати використання Scala/Java. Тому Python є оптимальним вибором, оскільки забезпечує достатню продуктивність (за рахунок високорівневих бібліотек) і швидкість розробки.

1.5.2 Бібліотеки Python для реалізації системи

Ефективність і швидкість створення рекомендаційної системи значною мірою залежать від вибору бібліотек та фреймворків. У екосистемі Python наявні як універсальні бібліотеки машинного навчання, так і спеціалізовані інструменти, призначені саме для побудови рекомендативних моделей. Розглянемо найбільш відомі з них та проаналізуємо, які підходять для підсистеми тематичного підбору книг.

Scikit-learn – популярна загальна бібліотека машинного навчання на Python, що містить реалізації класичних алгоритмів класифікації, кластеризації, регресії і тощо. Хоча scikit-learn не надає модулів, спеціально заточених під рекомендації, її засоби можуть бути використані для базових підходів. Наприклад, для content-based рекомендацій можна використати перетворення текстових описів книг у вектори (TF-IDF) та обчислення схожості (косинусна відстань або KNN з scikit-learn). Також scikit-learn містить алгоритми для розв’язання завдання зважування рейтингів (наприклад, методи матричної факторизації можна реалізувати вручну або через SGD-регресію). Втім, відсутність готових інструментів для роботи з розрідженими матрицями «користувач-елемент» і розрахунку метрик якості рекомендацій робить scikit-learn менш зручним для побудови повноцінної рекомендаторної системи, ніж спеціалізовані бібліотеки.

Surprise – спеціалізована бібліотека Python, призначена для розробки та аналізу моделей колаборативної фільтрації з явними рейтингамиsurpriselib.com. Вона надає зручні інструменти для завантаження даних (наприклад, вбудовані датасети MovieLens), розбиття на тренувальні/тестові множини, розрахунку метрик (RMSE, MAE) та готові реалізації популярних алгоритмів прогнозування рейтингів: від простих підходів (середнє, k-ближніх сусідів) до матричної факторизації (SVD, SVD++, NMF тощо)surpriselib.com. Surprise дозволяє швидко експериментувати з алгоритмами колаборативної фільтрації, порівнювати їх якість та підбирати гіперпараметри, що корисно для академічних досліджень. Однак, сфера застосування Surprise обмежена задачами прогнозування рейтингів. Бібліотека не підтримує роботи з implicit feedback (неявними оцінками, такими

як перегляди чи покупки) і не враховує контент-ознаки (атрибути користувачів або елементів)surpriselib.com. Отже, для моєї підсистеми, де важливо врахувати тематичну близькість книг (що можна розглядати як контентні ознаки), Surprise не дасть можливості побудувати гібридну або контент-орієнтовану модель – вона придатна лише для чистої колаборативної фільтрації на основі рейтингів.

LightFM – сучасна бібліотека рекомендерних алгоритмів, що поєднує підхід колаборативної фільтрації та врахування контентних ознак (гібридні рекомендації). LightFM було розроблено інженерами компанії Lyst і відкрито у вигляді Python-бібліотеки з акцентом на простоту використання та масштабованість. В основі LightFM – модель факторизації матриці, яка представляє користувачів і об'єкти (в моєму випадку книги) як суми латентних векторів їхніх ознак. Це означає, що бібліотека дозволяє додавати до моделі будь-які фічі – наприклад, жанри чи тематику книги, або демографічні дані користувачів – і таким чином вирішує проблему cold start для нових книг чи нових користувачів. LightFM підтримує як явний фідбек (рейтинги), так і неявний фідбек (інформацію про перегляди, покупки тощо). Для оптимізації моделі реалізовано ефективні алгоритми навчання з розрідженими даними – зокрема, Bayesian Personalised Ranking (BPR) і Weighted Approximate-Rank Pairwise (WARP) для роботи з implicit feedback. Важливо, що внутрішньо LightFM використовує Cython (C-розширення для Python), завдяки чому масштабування на багатоядерних системах відбувається доволі легко і продуктивно. У документації зазначено, що бібліотека успішно застосовується в промислових системах – зокрема, на сервісах Lyst та Catalant – і демонструє можливість обробляти дуже великі датасети. Таким чином, LightFM є одним з найкращих кандидатів для моєї задачі: модель гібридної рекомендації книг за тематикою можна побудувати, використовуючи ознаки книг (жанри, ключові слова, автори) в поєднанні з матрицею взаємодій користувачів (оцінки, перегляди тощо). Це дозволить рекомендувати книги схожої тематики навіть новим користувачам або нові книги існуючим користувачам, що неможливо при чистій колаборативній фільтрації.

Implicit – ще одна спеціалізована Python-бібліотека, орієнтована на алгоритми колаборативної фільтрації для неявного фідбеку. Вона надає високопродуктивні реалізації таких алгоритмів, як Alternating Least Squares (ALS) та Bayesian Personalized Ranking, оптимізовані для розріджених матриць великого розміру. Implicit написана з акцентом на швидкодію (використовує обчислення на рівні матриць у SciPy та власні оптимізації на C++), тому підходить для сценаріїв, де система має обробляти великі об'єми даних про взаємодії користувачів. Для мого проєкту implicit може бути корисною, якщо зосередитись лише на колаборативній фільтрації (наприклад, рекомендації книг на основі поведінки користувачів без врахування контенту). Але якщо необхідно врахувати тематичний зміст книг, implicit доведеться поєднувати з іншими методами (наприклад, відфільтровувати чи ранжувати результати за схожістю контенту), оскільки ця бібліотека не працює з ознаками контенту прямо.

TensorFlow та PyTorch – потужні фреймворки глибинного навчання, які також широко застосовуються для побудови рекомендаційних систем, особливо в задачах, де потрібні складні моделі (нейронні мережі, секвенційні моделі, обробка текстів тощо). Обидва фреймворки підтримують створення та навчання нейронних мереж будь-якої архітектури, що дає максимальну гнучкість у розробці алгоритму рекомендацій. Зокрема, на базі TensorFlow розроблено спеціалізовану бібліотеку TensorFlow Recommenders (TFRS), яка спрощує побудову end-to-end пайплайнів рекомендательних систем – від підготовки даних до навчання і оцінки моделей. TFRS містить готові блоки для факторизації, дворівневих моделей (candidate retrieval + ranking), реалізації метрик оцінки рекомендацій тощо. Аналогічно, в екосистемі PyTorch з'являються власні рішення для рекомендацій (наприклад, TorchRec від Facebook або компоненти NVIDIA Merlin для рекомендацій на GPU). Перевага використання загальних фреймворків полягає у високій гнучкості: можна будувати глибокі гібридні моделі, враховувати послідовність дій користача (рекомендації на основі послідовностей, як у задачі next-item prediction), використовувати техніки NLP для аналізу описів книг тощо. Водночас, недоліком є складність та великий обсяг

роботи, необхідний для реалізації повного циклу – на відміну від спеціалізованих бібліотек (Surprise, LightFM тощо), доведеться самостійно реалізувати багато «обгорток» – підготовку розріджених даних, обчислення метрик на валідації, відбір рекомендацій для кожного користувача тощо. Для невеликого академічного проєкту використання TensorFlow/PyTorch має сенс лише якщо необхідно дослідити сучасні нейронні підходи (скажімо, модель Transformer для рекомендацій). В іншому разі, доцільніше взяти готовий інструмент з реалізацією базових алгоритмів.

RecBole – це уніфікована бібліотека рекомендаційних систем, розроблена академічною спільнотою (групою RUCAIBox, КНР) як платформу для досліджень і розробки нових алгоритмів. Вона написана на Python і надає єдиний інтерфейс до більш ніж 100 реалізованих алгоритмів, охоплюючи чотири основні категорії: загальні рекомендації, послідовні рекомендації, контекстно-орієнтовані та знання-орієнтовані рекомендації. RecBole можна розглядати як свого роду «лабораторію» алгоритмів: на її основі легко виконувати експерименти, порівнювати різні методи на стандартизованих наборах даних, проводити тонке налаштування моделей. Для практичного впровадження конкретної системи (наприклад, рекомендацій книг) RecBole, можливо, буде надлишковою – адже вона містить багато зайвого функціоналу. Проте, з огляду на поставлену задачу, варто відзначити, що до складу RecBole входять як класичні моделі (той же LightFM, SVD, ALS), так і сучасні – наприклад, графові нейронні мережі, секвенційні моделі на основі Transformer, тощо. Це означає, що при бажанні експериментувати з найновішими підходами (наприклад, враховувати граф зв’язків між книгами або модель уваги для визначення подібності за описами) – RecBole може надати готові реалізації. В рамках бакалаврського проєкту використання RecBole виглядає виправданим лише на етапі дослідження – для вибору найкращого алгоритму. Проте для безпосереднього впровадження підсистеми на практиці, більш легковагі спеціалізовані бібліотеки (такі як LightFM або Surprise) можуть бути простішими у інтеграції.

Висновок щодо вибору бібліотек. Для побудови підсистеми тематичного підбору книг найбільш придатними є бібліотеки, що підтримують гібридні рекомендації або легко розширюються контентними ознаками. Виходячи з аналізу, оптимальним вибором є LightFM – як інструмент, що вже містить усе необхідне для врахування описових ознак книг і історії взаємодій користувачів. LightFM поєднує відносну простоту використання з продуктивністю, перевіреною на практиці (виробничі впровадження в компаніях). В якості альтернативи або доповнення можна використати Surprise – для базового прототипування моделей колаборативної фільтрації і оцінки якості, а також implicit – для швидкого виконання алгоритмів факторизації на implicit-даних. Якщо ж постане задача істотно ускладнити модель (наприклад, додати семантичний аналіз текстів книг для визначення тематики), можна інтегрувати рішення на базі TensorFlow/PyTorch. Втім, на початковому етапі LightFM має забезпечити необхідну якість рекомендацій за рахунок комбінування даних про рейтинги/вподобання і контентних характеристик книг.

1.5.3 Веб-фреймворки для реалізації API

Для надання результатів роботи рекомендаційної підсистеми іншим компонентам (наприклад, фронтенду чи іншим мікросервісам) необхідно створити веб-сервіс із певним API. Розглянемо три популярні Python-фреймворки для розробки веб-сервісів та REST API: Flask, Django (REST Framework) та FastAPI – і порівняємо їх у контексті задачі побудови мікросервісу рекомендацій. За результатами аналізу обґрунтуємо вибір FastAPI як оптимального рішення.

Flask – мікрофреймворк для Python, відомий своєю мінімалістичністю і гнучкістю. Flask надає базові можливості маршрутизації URL, обробки запитів і формування відповідей, але практично не нав’язує структуру проекту. Завдяки цьому Flask підходить для невеликих сервісів або прототипів: розробник сам обирає необхідні компоненти (базу даних, ORM, валідацію даних, авторизацію тощо) і підключає їх у міру потреби. Для задач машинного навчання Flask довгий

час був де-факто стандартом, оскільки дозволяв просто обгорнути модель у веб-API з мінімальними накладними витратами. Однак, Flask працює у контексті WSGI (синхронна обробка запитів) і не підтримує асинхронність з коробки. Це означає, що для високонавантажених сценаріїв (де потрібна обробка багатьох одночасних запитів з мінімальною затримкою) Flask може поступатися продуктивністю більш сучасним рішенням. До того ж, розробнику Flask-додатку доведеться самостійно реалізувати або інтегрувати багато стандартних для API речей – документацію ендпойнтів (напряму Flask цього не генерує), схеми валідації вхідних даних, обробку CORS і т.д. Існують розширення (flask-restful, flask-swagger тощо), але їх потрібно підключати та налаштовувати окремо. Отже, Flask – це легка вага з точки зору початкового коду, але в довгостроковій перспективі при розростанні API може знадобитися суттєвий власний або сторонній код для підтримки зручностей, які в інших фреймворках є «із коробки».

Django – повноцінний високорівневий фреймворк для розробки веб-застосунків на Python, що дотримується концепції «Battery included» (максимум необхідного у комплекті). Для створення RESTful API на Django зазвичай використовується Django REST Framework (DRF) – потужний додатковий модуль, який забезпечує серіалізацію даних, визначення ViewSet-ів, автоматичну генерацію документації та інтерактивного браузу API, а також безліч вбудованих функцій (аутентифікація, пагінація, контроль доступу тощо). DRF є дуже популярним рішенням для складних веб-застосунків, де API тісно інтегрований з базою даних через ORM Django. Основна перевага – висока швидкість розробки типових CRUD-застосунків: багато коду генерується або налаштовується декларативно, і розробник може сконцентруватися на логіці. Для мого завдання (підсистема рекомендацій) використання Django/DRF могло б бути виправданим, якби система була частиною більшого веб-сайту або вимагала розгорнутої адмін-панелі, аутентифікації користувачів тощо. Проте в рамках мікросервісної архітектури, де рекомендаційна підсистема існує окремо і спілкується з іншими через API, підключення повного Django може виглядати надмірним. Django –

досить важкий фреймворк: запуск навіть простого сервісу на Django потребує більше ресурсів, а його синхронна природа (до версій 3.x) означає, що для обробки багатьох паралельних запитів потрібні додаткові воркери або сервіси (наприклад, через gunicorn). Хоча Django 4 підтримує Async views, екосистема DRF та більшості плагінів досі значною мірою орієнтована на синхронний підхід. За даними неформальних тестів і порівнянь, продуктивність FastAPI (асинхронного) на порядки вища, ніж у зв'язки Django+DRF, в сценаріях з високою конкуренцією запитів. Таким чином, якщо головна мета – швидкий API-сервіс для рекомендацій, Django REST Framework може виявитися занадто громіздким і повільним рішенням, яке до того ж ускладнить деплой через більший імідж Docker та більше залежностей.

FastAPI – сучасний фреймворк (випуск з 2018 р.), спеціально створений для швидкої реалізації API на базі стандартів OpenAPI та JSON Schema. Основна відмінність FastAPI – використання асинхронного сервера (ASGI, на базі Uvicorn/Starlette) і типізація Python (type hints) для автоматичного генерування схем валідації і документації. По суті, FastAPI поєднує переваги Flask (легкість, мінімалістичність) з перевагами Django/DRF (автоматичні багато чого) без значних накладних витрат. Розглянемо ключові властивості FastAPI і як вони виглядають у контексті моєї задачі:

- Висока продуктивність і асинхронність. За рахунок використання асинхронного сервера Uvicorn і внутрішнього ядра Starlette, FastAPI показує один з найкращих результатів серед Python-фреймворків у незалежних бенчмарках (TechEmpower тощо). За повідомленнями розробників, продуктивність FastAPI порівнювана з Node.js та Go для реальних сценаріїв API. Для нас це означає, що сервіс зможе обслуговувати більше запитів з мінімальними затримками, що важливо, якщо рекомендації будуть викликатися інтерактивно (наприклад, при відкритті сторінки користувача з персональними рекомендаціями – запит має відпрацювати за десятки мілісекунд). Асинхронна модель дозволяє ефективно використовувати ресурси

при зверненні до бази даних чи інших сервісів: поки чекаємо на відповідь, потік не блокується і може обробляти інші запити. У випадку рекомендаційного сервісу, де можливо знадобиться робити зовнішні запити (наприклад, до бази даних книг), асинхронність – великий плюс.

- Автоматична генерація документації API. FastAPI автоматично генерує інтерактивну документацію (Swagger UI та Redoc) для всіх ендпойнтів на основі описів і типів, зазначених у коді. Достатньо запустити сервіс, і за адресою /docs можна побачити гарний інтерфейс з переліком усіх методів, їхніми параметрами та прикладами відповідей. Для мого мікросервісу це дуже зручно – інші розробники (наприклад, ті, хто пишуть фронтенд) легко зрозуміють, як викликати API рекомендацій, які дані очікуються і що повертається. У Flask довелося б писати цю документацію вручну або використовувати сторонні бібліотеки; в Django/DRF документація є, але налаштування займає більше часу. Таким чином, FastAPI підвищує зручність інтеграції: API, що самодокументується, зменшує можливі помилки використання.
- Вбудована валідація та серіалізація даних. За допомогою Pydantic (бібліотека для декларативного оголошення моделей даних) FastAPI автоматично перевіряє вхідні параметри (query, body) на відповідність очікуваним типам і форматам. Наприклад, якщо ендпойнт очікує об’єкт із полем user_id: int та списком favorite_genres: list[str], то при передачі невірних типів (нечислового user_id або списку жанрів іншого формату) клієнт отримає зрозумілу помилку 422 із поясненням, які поля некоректні. Це значно спрощує розробку надійного API. У контексті моєї системи: можна визначити Pydantic-модель для запиту рекомендацій (наприклад, містить ідентифікатор користувача, необов’язково – жанр/тему для фільтрації) та для відповіді (список рекомендованих книжок з

полями назва, автор тощо). Далі FastAPI подбає про те, щоб ці моделі правильно заповнювались і віддавались у вигляді JSON. Flask таких можливостей не пропонує (все вручну), DRF має Serializers (аналог Pydantic), але їх використання трохи складніше синтаксично.

- Простота коду та структура. Завдяки використанню аннотацій типів, код обробників FastAPI дуже подібний до звичайних Python-функцій. Це робить його читабельним і зрозумілим навіть для менш досвідчених розробників або data science-інженерів, які можуть не мати глибокого бекграунду у веб-розробці. JetBrains відзначає, що FastAPI стрімко набирає популярність серед спільноти: за 2021–2023 роки його використання зросло з 21% до 29% розробників Python, і серед спеціалістів з даних він посів друге місце за популярністю (31% опитаних використовують FastAPI). Популярність означає наявність активної спільноти і безлічі прикладів, що теж плюс для нас. Структура проекту на FastAPI може бути зведена до кількох файлів (main.py для запуску, кілька модулів з роутами/логікою, схема даних), що вписується в парадигму мікросервісу – нічого зайвого.

Зважаючи на ці фактори, було вирішено використовувати FastAPI для реалізації веб-сервісу рекомендацій. У порівнянні з Flask, FastAPI надає значно більше можливостей «з коробки», що зменшить обсяг додаткового коду (особливо щодо документації і валідації). А у порівнянні з Django REST Framework, FastAPI виграє в продуктивності та простоті – я отримую легкий і швидкий сервіс, який легко розгорнути в контейнері та масштабувати при потребі. Варто зазначити, що FastAPI вже прийнятий багатьма великими компаніями (серед користувачів згадуються Uber, Netflix, Microsoft та ін.) для створення високонавантажених сервісів. Отже, обраний стек (Python + FastAPI) цілком відповідає сучасним промисловим трендам у розробці систем рекомендацій.

1.5.4 Системи ізоляції та деплою

При розробці і впровадженні програмної системи важливо забезпечити її ізоляцію від оточення та портативність – тобто можливість згортати додаток у стандартний пакет для розгортання на різних серверах. Історично для цих цілей використовувалися віртуальні машини (Virtual Machines, VM), але останнє десятиліття домінує підхід контейнеризації, флагманом якого є технологія Docker. Розглянемо відмінності між традиційною віртуалізацією (VM) та контейнерами, а також між двома популярними контейнерними платформами – Docker і його бездемонним аналогом Podman. Обґрунтуємо вибір Docker як засобу ізоляції та деплою моєї підсистеми.

Віртуальні машини забезпечують ізоляцію на рівні апаратного забезпечення: гіпервізор емулює окремий сервер, на якому запускається повноцінна гостьова операційна система, всередині якої вже працює потрібний додаток. Перевага підходу – високий ступінь ізоляції (кожна VM – це фактично окремий сервер з власним ядром ОС, драйверами тощо) і можливість запускати різні ОС на одному фізичному вузлі. Однак, за це платиться ціною значних накладних витрат: кожна VM споживає ресурси на підтримку власної ОС, багато пам’яті та CPU йде на дублювання базових функцій ОС для кожної VM. Завантаження і розгортання VM – порівняно повільний процес (рахунок на хвилини), а образи VM займають гігабайти. Для масштабування сервісів це не оптимально: якщо потрібно запустити 10 копій веб-сервісу, доведеться запустити 10 повних ОС усередині VM, що дуже неефективно.

Контейнери ізолюють середовище на рівні процесів та операційної системи, використовуючи можливості ядра (Linux namespaces, cgroups). Контейнер – це по суті набір процесів, які бачать ізольоване файлове середовище і системні ресурси, але при цьому використовують ядро хостової ОС спільно з іншими контейнерами. Таким чином, контейнери значно «легші» за VM: зазвичай образ контейнера включає тільки необхідні бібліотеки та залежності додатку, без зайвих компонентів ОС (ядра тощо). Розмір контейнерів вимірюється мегабайтами, старт відбувається за секунди. Це дозволяє ефективно

масштабувати системи: запуск додаткових екземплярів сервісу в контейнері відбувається швидко і майже лінійно масштабується по ресурсах. Завдяки відсутності гіпервізора, накладні витрати мінімальні – наприклад, 10 контейнерів з веб-сервісом можуть спільно використовувати одне ядро Linux і не дублювати його, як у випадку 10 VM. Контейнери також портативні: один і той самий образ можна запускати де завгодно – локально, на сервері, у хмарі – і він буде працювати ідентично, оскільки містить увесь необхідний рантайм і залежності. Ця портативність і легкість зробили контейнеризацію стандартом де-факто для деплойменту мікросервісів.

На сьогодні найпоширенішим інструментом контейнеризації є Docker. Docker – це платформа з відкритим кодом, яка дозволяє будувати, розповсюджувати і запускати контейнери. Docker складається з докер-демона (фоновий сервіс, що керує контейнерами), формату образів (Docker Image) та реєстрів образів (DockerHub тощо). Docker значно спростив життя розробників: щоб упакувати мою підсистему рекомендацій у контейнер, достатньо написати Dockerfile (де зазначити базовий образ, наприклад, Python 3.10, скопіювати код програми, встановити залежності і запустити сервіс). На виході отримуємо Docker-образ – бінарний пакунок, що містить мою програму. Цей образ можна запустити на будь-якому сервері з Docker (Linux, Windows, macOS) і він гарантовано працюватиме однаково. Docker забезпечує ізоляцію файлової системи, мережі, змінних середовища для кожного контейнера, що дозволяє запускати навіть конфліктні між собою додатки на одному хості.

Переваги Docker, які важливі для мене:

- Популярність та екосистема. Docker став синонімом контейнерів і має найбільшу спільноту та екосистему інструментів. Існують сотні тисяч готових образів на Docker Hub для різних технологій. Зокрема, я можу використати офіційний образ Python як базу. У разі потреби легко знайти образи баз даних, кешувальників і т.д. Docker має зрілий інструментарій для композиції сервісів (docker-compose), оркестрації (Docker Swarm, а головне – Kubernetes повсюдно підтримує Docker-образи). Тобто, обравши

Docker, я не ризикую сумісністю чи підтримкою – це промисловий стандарт. Як відзначають в оглядах, Docker пропонує широку екосистему попередньо зібраних образів та інструментів, що мінімізує трудовитрати на налаштування середовища. Практично всі сучасні open-source проекти мають Dockerfile для швидкого запуску, що підтверджує доцільність цього вибору і для мого проекту.

- Простота використання. Docker дозволяє запускати мій сервіс локально в точнісінько такому ж середовищі, як буде на продакшені. Це виключає проблему "працює на моєму комп’ютері – не працює на сервері". Крім того, контейнер можна ізолювати ресурсно (виділити певну кількість CPU, RAM), щоб забезпечити стабільну роботу без взаємного впливу з іншими сервісами.
- Швидкість деплою і оновлень. Оскільки образ контейнера включає всі залежності, розгортання нової версії зводиться до завантаження нового образу і перезапуску контейнера. Це займає кілька секунд або хвилин, тоді як оновлення VM або налаштування серверів вручну – значно довше. Для частих оновлень (наприклад, тонке налаштування моделі рекомендацій і частий деплой нових параметрів) Docker підходить ідеально.

Окрім Docker, набувають популярності альтернативні інструменти контейнеризації, зокрема Podman. Podman було розроблено з акцентом на безпеку: на відміну від Docker, який працює через демона від імені привілейованого користувача, Podman не має центрального демона і може запускати контейнерні процеси без привілеїв (rootless mode). Крім того, Podman сумісний з Docker-образами (використовує той самий формат OCI). Фактично, Podman забезпечує такий самий користувацький інтерфейс (команди podman run/build аналогічні docker run/build), тож перейти на нього технічно нескладно. Перевага Podman – вищий рівень безпеки за замовчуванням (немає постійного root-вразливого демона) та краща інтеграція з системами на кшталт Kubernetes (через CRI-O). Однак, Docker наразі суттєво випереджає Podman за популярністю і екосистемою. Багато інструментів розраховано саме на Docker, більшість

довідкових матеріалів – про Docker. Тому у виборі між Docker і Podman зазвичай враховують контекст: якщо вимоги безпеки критичні (наприклад, середовище з багатьма неконтрольованими користувачами на хості) – можливо, Podman кращий. У моєму випадку, коли мікросервіс буде розгорнуто в ізольованому середовищі (наприклад, власний VPS або хмарний інстанс під моїм контролем), переваги Docker переважують. Docker простіше налаштувати та він має звичніший робочий процес. До того ж, багато хостинг-платформ (AWS, GCP, Heroku і ін.) мають пряму підтримку деплою Docker-образів.

Отже, для ізоляції середовища виконання та розгортання підсистеми рекомендацій обрано Docker. Я створюю Docker-образ, що містить все необхідне: код сервісу (FastAPI застосунок), модель/бібліотеки рекомендацій (LightFM чи інші) та залежності (наприклад, Python-бібліотеки). Це дозволить гарантовано отримати однаковий результат на машині розробника, на тестовому сервері і на продакшені. Контейнеризація сприятиме і простоті масштабування – за необхідності можна запустити кілька контейнерів з рекомендаційним сервісом і балансувати між ними навантаження. Вибір Docker також відповідає галузевим практикам: практично всі сучасні мікросервісні архітектури будуються на Docker-контейнерах, що забезпечують переносимість і легке керування життєвим циклом сервісів.

1.5.5 Протоколи комунікації між сервісами

У розподіленій системі, де різні компоненти (сервіси) спілкуються між собою, дуже важливим є вибір протоколу і формату взаємодії. Мій рекомендаційний модуль, ймовірно, працюватиме як окремий сервіс, до якого звертається, скажімо, основний веб-додаток або мобільний бекенд для отримання рекомендацій. Розглянемо кілька підходів до організації API між сервісами: REST, GraphQL, gRPC та Thrift. Кожен з них має свої переваги і недоліки, але для мого випадку (внутрішня взаємодія сервісів в межах однієї системи) оптимальним, як буде показано, є gRPC.

REST (Representational State Transfer) – найпоширеніший стиль веб-API, заснований на протоколі HTTP і форматі передачі повідомлень типово JSON або XML. REST підкреслює використання стандартних методів HTTP (GET, POST, PUT, DELETE тощо) для маніпуляції ресурсами, і є дуже гнучким та простим для споживачів. Завдяки своїй простоті та прив'язці до веб-технологій REST став фактично універсальною мовою спілкування між різними системами через інтернет. Переваги REST: його легко тестувати (можна зробити запит через браузер або curl), існує безліч клієнтських реалізацій для будь-якої мови, а також він людино-читаємий (JSON можна прочитати і зрозуміти). Однак, REST має й недоліки. По-перше, надлишковість даних: клієнт часто отримує більше інформації, ніж потребує, або робить кілька запитів для отримання пов'язаних даних з різних ресурсів. Це було однією з причин появи GraphQL. По-друге, REST не визначає формального контракту взаємодії – лише угоди. Хоча існують OpenAPI/Swagger для документації, немає автоматичного контрактного зв'язування між клієнтом і сервером: зміна API може спричинити помилки, якщо клієнт не оновлено, і це виявиться лише під час виконання. Нарешті, текстовий формат JSON додає накладні витрати на серіалізацію/десеріалізацію і більше навантажує мережу (у порівнянні з бінарними протоколами). Тим не менш, для взаємодії «клієнт-сервер» (наприклад, між фронтендом і бекендом) REST залишається одним з найкращих виборів завдяки простоті.

GraphQL – це мова запитів для API, відкрита Facebook у 2015 році. GraphQL надає клієнту гнучкість у виборі саме тих даних, які йому потрібні, через єдиний ендпойнт. Замість кількох REST-запитів до різних ресурсів, клієнт GraphQL може одним запитом отримати складну, пов'язану структуру даних, визначивши, які поля потрібні. Це вирішує проблему надлишковості і множинних запитів: наприклад, мобільний додаток може одним викликом завантажити і інформацію про користувача, і список рекомендованих книг з потрібними полями, і статус підписки – замість трьох різних REST-викликів. GraphQL-сервер має строгий схемний контракт: визначаються типи даних і можливі запити/мутації, клієнт може інспектувати цю схему. Переваги GraphQL проявляються саме в клієнт-

серверному сценарії, особливо для складних інтерфейсів (як новинна стрічка Facebook, де його і придумали). У випадку моєї системи, GraphQL міг би бути корисним, якщо б фронтенд хотів гнучко комбінувати дані – скажімо, запитувати рекомендації книг з різними фільтрами в одному запиті або об'єднувати з іншими даними. Але у внутрішній комунікації мікросервісів GraphQL використовується рідко. Це зумовлено тим, що GraphQL додає складності на серверній стороні (потрібно реалізувати резолвери для кожного поля, оптимізувати запити, враховувати версію схеми). До того ж, GraphQL працює поверх HTTP (зазвичай), використовуючи текстовий JSON, тому за продуктивністю він не перевершує REST, а інколи може бути і повільнішим через необхідність сформувати та розібрати великий JSON з вкладеними об'єктами. Ще один нюанс – кешування: з REST-ресурсами можна використовувати HTTP-кеш, проксі тощо; для GraphQL це складніше через гнучкість запитів (хоча існують клієнтські бібліотеки, що кешують результати запитів). Таким чином, GraphQL – чудовий інструмент для публічного API (коли багато різних клієнтів з різними потребами даних), але для простого сценарію "бекенд сервіс запитує рекоммендер-сервіс" – він ускладнить систему без суттєвої вигоди.

RPC (віддалений виклик процедур)-стилі, представлені тут gRPC та Thrift, підходять для комунікації сервіс-сервіс і ставлять на перше місце ефективність та суворий контракт між сторонами. І gRPC, і Thrift використовують схему (IDL – Interface Definition Language), на основі якої генерується код клієнта і сервера для різних мов. Це означає, що після визначення інтерфейсу (методів, повідомлень) і сервер, і клієнт "знають" структуру даних на етапі компіляції, що виключає багато помилок (у REST/GraphQL такого немає – помилки можуть виявитися тільки під час виконання). Розглянемо особливості кожного з цих двох протоколів.

Apache Thrift – це фреймворк RPC, спочатку розроблений Facebook (2000-ні) і переданий в Apache. Thrift надає власну мову опису інтерфейсів, яка підтримує багато мов програмування (C++, Java, Python, PHP, Ruby і ще десяток). Сервіс на Thrift визначається через файли .thrift, де оголошуються структури

даних і сервіси (методи з параметрами). На основі цього файл Thrift-компілятор генерує код клієнтських бібліотек і серверних обгорток для потрібних мов. Перевага Thrift – крос-мовна і крос-платформна сумісність: можна написати сервер на Java, клієнт на Python, і вони “говоритимуть” через Thrift-протокол без проблем. Thrift підтримує різні транспортні протоколи (TCP, HTTP) і формати серіалізації (у тому числі бінарний, компактний тощо). Відомо, що Thrift дуже швидкий і маловитратний: у бенчмарках він часто випереджає REST+JSON за швидкістю і latency. Наприклад, ще 10 років тому Thrift був базою для мікросервісів Twitter, Uber та інших, хто будував власні внутрішні платформи. Недоліки Thrift – певна складність у налаштуванні (потрібно керувати згенерованими кодами, сумісністю версій) та менша популярність у нових проектах (нині його частково витіснив gRPC). Проте Thrift досі успішно застосовується в проектах, де потрібна ефективна взаємодія між різноплатформенними компонентами – наприклад, у big data системах (Hadoop екосистема) або у фінансових застосунках. Отже, Thrift – перевірений часом інструмент, який акцентується на продуктивності та ефективному крос-мовному RPC.

gRPC – сучасний RPC-фреймворк, розроблений Google і представлений у 2015 році. По суті, gRPC є наступником/аналогом Thrift, що використовує найкращі напрацювання Google (Protocol Buffers, HTTP/2) для досягнення високої продуктивності. В gRPC інтерфейс сервісу описується файлом .proto (IDL – Protocol Buffers). На основі нього генерується код для клієнтів і серверів (підтримується багато мов – C++, Java, Go, Python тощо). На відміну від Thrift, який має власний транспорт, gRPC працює поверх HTTP/2 – це дає ряд переваг: мультиплексування запитів через одне з’єднання, потокова передача (streaming) даних від сервера до клієнта і навпаки, ефективне стиснення заголовків. Повідомлення серіалізуються у бінарному форматі Protocol Buffers, який дуже компактний і швидко парситься. У підсумку, gRPC забезпечує мінімальні накладні витрати і високу швидкість обміну повідомленнями – цей протокол створювався саме для високонавантажених мікросервісів. Більше того, gRPC має

вбудовану підтримку стрімів: сервіс може відправляти клієнту потік результатів (що зручно, наприклад, для нотифікацій чи поступового надходження даних), а також двобічний стрім (bi-directional streaming) для інтерактивних сценаріїв. У контексті рекомендацій це могло б бути використано для реального часу оновлення рекомендацій, хоча моя система, ймовірно, не потребує такого (пакетна видача списку книг в одному запиті). Однак, ця можливість свідчить про гнучкість gRPC.

Google та інші компанії широко застосовують gRPC у своїх платформах – наприклад, всі внутрішні API Google перейшли на нього, Uber також перейшов з HTTP на gRPC для частини сервісів, де потрібна була ефективність. За рекомендаціями експертів, gRPC доцільно використовувати для “внутрішнього” зв’язку між серверами, тоді як GraphQL – для зовнішніх клієнтських API. Моя ситуація якраз така: рекомендаційна підсистема – це бекенд-сервіс, який викликатиметься іншими бекенд-сервісами (наприклад, основним API додатку). Тут пріоритет – швидкість і надійність виклику, а не гнучкість запиту. gRPC перевершує REST за швидкодією в середовищах з високим QPS (запитів в секунду) і низькою затримкою, оскільки зменшує мережевий трафік і використовує постійні з’єднання. Крім того, описавши контракт у .proto, Я можу згенерувати клієнтський код і на стороні основного сервісу викликати метод типу GetBookRecommendations(user_id) майже як локальну функцію – з автогенерованими класами запиту/відповіді. Це спрощує розробку і усуває цілий клас потенційних помилок (неправильні URL, помилки у форматі JSON, невідповідність версій). Варто також згадати, що gRPC підтримує такі корисні функції, як вбудоване балансування навантаження, перевірка стану (health check), трасування викликів – все це важливо у мікросервісній архітектурі.

Щодо порівняння gRPC vs Thrift: обидва близькі за ідеологією. Thrift дещо старіший і має більше свобод у налаштуванні (можна вибрати транспорт/протокол), але gRPC пропонує більш сучасний стек: HTTP/2, що полегшує проходження через проксі, і Protocol Buffers третьої версії (open-source, широко підтримуються). В результаті, gRPC зараз більш популярний у нових

проектах. В моєму випадку, зважаючи що всі компоненти будуть на Python, використання gRPC цілком виправдане: є готова підтримка в Python (grpcio), а також у разі потреби легко підключити клієнтів на інших мовах. Thrift теж можна було б використати, але це додало б складності (окрема генерація, можливо більші зусилля на конфігурацію сервера). Отже, обираємо gRPC як протокол міжсервісної комунікації.

Підсумовуючи, gRPC має наступні переваги для мого проекту:

- Висока ефективність і низькі затримки. Бінарна передача даних і HTTP/2 дають швидший обмін, що важливо для швидкої видачі рекомендацій у реальному часі.
- Суворий контракт і автогенерація коду. Зменшується кількість помилок інтеграції; розробка спрощується.
- Можливість стрімінгу. Це може знадобитися, наприклад, якщо вирішимо реалізувати потокове відправлення рекомендацій по мірі обрахунку або оновлення (хоча поки не планується, але технологія дозволяє масштабуватися в цьому напрямку).
- Широка підтримка і активне використання в галузі. Є багато матеріалів, прикладів. Зокрема, великі компанії підтвердили переваги gRPC для мікросервісів (Google, Netflix, Dropbox тощо). Це мінімізує ризики – технологія зріла і перевірена на виробництві.

Звичайно, gRPC – не панацея: якщо б мій сервіс був відкритим для сторонніх клієнтів, можливо, варто було б зробити REST або GraphQL API для зручності зовнішніх розробників. Але оскільки система внутрішня, контрольована мною, і я можу генерувати клієнти під мої ж сервіси – gRPC виглядає найкращим вибором для швидкодії і строгого визначення інтерфейсів.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ПІДСИСТЕМИ ПІДБОРУ

2.1 Архітектура підсистеми рекомендацій

Підсистема рекомендацій книг спроєктована як окремий модуль, відповідальний за генерування персоналізованих книжкових рекомендацій. Архітектурно це автономний сервіс (мікросервіс), що взаємодіє з іншими компонентами системи через чітко визначені інтерфейси API. Основними компонентами підсистеми є двигун рекомендацій (алгоритмічне ядро, яке обчислює рейтинги чи релевантність книг для користувачів), сховище даних (бази даних, що містять інформацію про книги, користувачів та їхні взаємодії), та інтерфейс служби (веб-API для отримання запитів і видачі результатів). Для реалізації інтерфейсу використано фреймворк FastAPI на Python, що забезпечує швидку обробку HTTP-запитів та зручну інтеграцію з ML-моделлю. Внутрішня комунікація сервісу з іншими мікросервісами може здійснюватися через gRPC – це дозволяє отримувати дані профілю користувача або деталі про книги з інших підсистем із мінімальною затримкою. Сам сервіс розгорнуто в Docker-контейнері, що забезпечує портативність та ізоляцію середовища. Таким чином, архітектура підсистеми є багатошаровою: на рівні даних – бази даних книг та користувацьких оцінок; на рівні логіки – модель рекомендацій (алгоритм LightFM); на рівні представлення – API сервіс для запитів рекомендацій.

Користувач (через клієнтський застосунок або веб-інтерфейс) надсилає запит на отримання книжкових рекомендацій до API підсистеми. Підсистема отримує запит (позначено як крок 1 на рисунку) і далі звертається до бази даних для отримання необхідних даних – інформації про профіль користувача, його історію вподобань, а також метадані кандидатних книг (крок 2). Після отримання даних підсистема використовує ядро рекомендацій – попередньо навчену модель LightFM – для обчислення рейтингових оцінок або ймовірності зацікавлення користувача кожною з потенційних книг (крок 3). Модель на основі цих даних генерує список найбільш релевантних книжок для даного користувача. Нарешті, API формує відповідь, що містить топ-N рекомендованих книжок, і надсилає її користувачеві (крок 4). Така модульна архітектура спрощує підтримку та

масштабування: модель можна оновлювати чи перенавчати незалежно, а підключення через FastAPI/gRPC забезпечує сумісність із різними клієнтами та іншими сервісами системи.

2.2 Логіка та методи генерації рекомендацій

Підсистема підбору реалізує логіку рекомендацій на основі сучасних підходів машинного навчання. Загалом існує декілька підходів до рекомендацій: контент-орієнтований, колаборативний та гібридний. У контент-орієнтованих системах рекомендацій для визначення подібних елементів використовуються характеристики самих об'єктів (атрибути контенту). Кожна книга описується набором ознак (жанр, автор, ключові слова тощо), і будується профіль користувача на основі вподобаного ним контенту. Система на основі вмісту порівнює ці описи та рекомендує користувачеві нові книги, схожі на ті, що йому вже сподобалися в минулому. Такий підхід добре працює для нових користувачів або для рекомендацій у межах чітко визначених інтересів, адже враховує тільки явні характеристики книг та вподобань.

Натомість, колаборативна фільтрація базується на аналізі поведінкових даних багатьох користувачів. У колаборативних системах будується матриця взаємодій R розмірності $|U|\times|I|$, де рядки відповідають користувачам $u \in U$, стовпці — об'єктам (книгам) $i \in I$, а значення r_{ui} відображають факт взаємодії (наприклад, оцінка або перегляд) між користувачем u і книжкою a. На основі цієї розрідженої матриці система шукає закономірності: наприклад, *user-based* колаборативний підхід визначає схожих між собою користувачів та рекомендує книги, які вподобали схожі користувачі. *Item-based* підхід, навпаки, визначає схожість між самими книгами за історією взаємодій усіх користувачів і радить користачеві ті книги, які схожі на вже переглянуті ним. Колаборативна фільтрація дозволяє виявляти приховані вподобання на основі колективного досвіду: вона може рекомендувати книги, які користувач сам не шукав би, але які сподобалися іншим з подібними смаками. Водночас є й проблеми, притаманні колаборативним методам, зокрема проблема холодного старту (коли для нових

користувачів або нових книг недостатньо даних взаємодій) та проблема шпаруватості даних.

З огляду на переваги та недоліки зазначених підходів, у даному проєкті обрано гібридний підхід для побудови рекомендаційної підсистеми. Гібридна система поєднує колаборативні та контентні методи, щоб враховувати як колективні патерни вподобань, так і атрибутивні характеристики книг. Зокрема, використано модель LightFM, яка реалізує латентно-факторну (factorization) модель рекомендацій з можливістю врахування додаткових ознак контенту та профілю користувача. Модель LightFM належить до класу моделей з прихованими факторами: вона асоціює з кожним користувачем і кожною книгою вектор у просторі прихованих ознак, навчаючись таким чином, щоб скалярний добуток цих векторів був вищим для пар "користувач-книга", які справді взаємодіють. Важливо, що LightFM є гібридною моделлю: латентні вектори формується як сума векторів всіх ознак користувача або книги. Наприклад, книга може мати ознаки "жанр: фентезі" і "автор: Толкін", користувач – ознаки "улюблений жанр: фентезі" і т.д.; модель навчить для кожної ознаки свій embedding-вектор, а сумуючи вектори ознак отримуємо вектор книги q_i та вектор користувача p_u. Таким чином, якщо користувачеві подобаються фентезійні романи певного автора, скалярний добуток $p_u \cdot q_i$ для книги того ж жанру/автора буде більшим. Завдяки цьому LightFM може одночасно врахувати як минулі взаємодії (колаборативна складова), так і атрибути об'єктів (контентна складова) для видачі рекомендацій.

Алгоритм генерації рекомендацій у підсистемі працює наступним чином. Коли надходить запит від користувача, система формує множину кандидатів – книг, які потенційно можуть бути рекомендовані. До множини кандидатів можуть включатися: книги, що мають схожий жанр чи тематику з уже переглянутими (контентний аспект), а також книги, які популярні серед схожих за вподобаннями користувачів (колаборативний аспект). Для кожного кандидата модель LightFM

обчислює оцінку переваги $\hat{r}_{ui}$ – числовий показник того, наскільки ймовірно книга is сподобається користувачу u. Далі здійснюється сортування кандидатів за спаданням цієї оцінки, і відбираються топ-N книг з найвищими значеннями $\hat{r}_{ui}$. Перед поверненням результату можуть застосовуватися додаткові бізнес-правила, наприклад фільтрація вже придбаних або прочитаних книг, забезпечення різноманітності рекомендацій тощо. У результаті користувач отримує кінцевий список рекомендацій, сформований на основі збалансованого урахування його смаків і глобальних трендів системи. Така логіка забезпечує персоналізований підбір контенту і може адаптуватися до змін поведінки користувачів завдяки регулярному оновленню моделі на нових даних.

2.3 Математична формалізація задачі рекомендацій

Задачу рекомендації книг можна формально представити як задачу передбачення вподобань користувачів на просторі пар користувачів UUU та об’єктів (книг) III. Маємо множину користувачів:

$$U = \left\{ u_1, u_2, ..., u_{|U|} \right\}$$

та множину книг:

$$I = \left\{ i_1, i_2, ..., i_{|I|} \right\}$$

Нехай матриця взаємодій

$$R = \left\{ r_{ui} \right\}$$

розмірності $|U| \times |I|$ описує відомі оцінки або інтерес користувача *u* до книги *i*. Значення r_{ui} можуть бути явними (наприклад, рейтинг від 1 до 5) або бінарними неявними (наприклад, r_{ui} = 1 якщо користувач переглянув чи придбав книгу, і r_{ui} = 0 якщо немає взаємодії). Матриця R є розрідженою, оскільки кожен

окремий користувач зазвичай взаємодіє лише з малою часткою доступних книг. Метою системи є побудувати функцію рекомендацій

$$f : U \times I \to \square \ ,$$

яка ставить у відповідність кожній парі (u,i) певну прогнозовану оцінку

$$\hat{r}_{ui} = f(u,i),$$

що відображає корисність або цікавість книги і для користувача *u*. На основі цієї функції можна ранжувати множину книг для кожного користувача та рекомендувати топові елементи з найвищими $\hat{r}_{ui}$. Формально, для кожного u будується впорядкований список:

$$\mathrm{Re}\, c(u) = Top - N\left\{\hat{r}_{ui} : i \in I\right\},$$

де ***N*** – потрібна кількість рекомендацій.

Одним із найпопулярніших підходів до побудови функції *(u,i)* є латентно-факторна модель, що використовує матричну факторизацію рейтингової матриці. Ідея факторизації полягає у представленні великих розріджених матриць через добуток менших матриць (матриць прихованих факторів) з метою виявлення прихованих закономірностей у даних. В контексті рекомендацій це означає, що матриця R розкладається наближено на дві матриці: матрицю прихованих факторів користувачів F^U розмірності $|U| \times k$ та матрицю прихованих факторів предметів F^I розмірності $k \times |I|$ (де k – кількість факторів). Рядок F^U відповідає вектору $p_u \in \square^k$ для користувача u, а стовпчик F^I – вектору $q_i \in \square^k$ для книги i. При цьому добуток матриць $F^U \square F^I$ відтворює відому частину матриці *R* і дає наближення для невідомих значень. Інакше кажучи, модель припускає, що прогнозована оцінка може бути обчислена як скалярний добуток латентних векторів:

$$\hat{r}_{ui} = p_u \bullet q_i,$$

де $p_u = (p_{u1}, ... p_{uk})$ – вектор прихованих факторів користувача *u*, а $q_i = (q_{i1}, ... q_{ik})$ – вектор факторів книги i. Ці вектори не відомі наперед – їхні значення отримуються в результаті навчання моделі на основі наявних даних матриці R. Модель факторизації підбирає значення p_u і q_i так, щоб для всіх відомих пар $(u,i) \in R$ прогноз $\hat{r}_{ui}$ якомога точніше наближав реальне значення r_{ui} . Для цього зазвичай мінімізують функцію помилки (застосовується метод градієнтного спуску або інші алгоритми оптимізації). Наприклад, у випадку явних рейтингових даних популярним є квадратичний критерій помилки:

$$E = \sum_{(u,i) \in \Re_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda(\|p_u\|^2 + \|q_i\|^2),$$

де сума береться по всіх відомих взаємодіях у тренувальному наборі даних, а $\lambda(\|p_u\|^2 + \|q_i\|^2)$ – член регулярізації, що запобігає перенавчанню моделі (покарання за занадто великі значення компонент векторів). В результаті мінімізації цього функціоналу отримуються такі p_u та q_i, які найкраще пояснюють наявні вподобання користувачів.

Для підвищення точності прогнозування модель може враховувати також систематичні зсуви у даних – наприклад, той факт, що деякі користувачі в середньому ставлять вищі оцінки, а деякі книги отримують кращі оцінки за інших. Тому розширена формула прогнозу додає відповідні біаси (зміщення) користувача і предмета, а також глобальний середній рейтинг μ:

$$\hat{r}_{ui} = \mu b_u + b_i p_u \bullet q_i,$$

де b_u – індивідуальне зміщення (схильність до вищих чи нижчих оцінок) для користувача u, b_i – зміщення для об’єкта i, а $p_u \bullet q_i$ – взаємодія латентних

векторів, як і в базовій моделі. Врахування b_u та b_i дозволяє моделі краще персоналізувати прогноз та компенсувати загальні тенденції в даних.

У випадку неявних даних (коли r_{ui} сигналізує лише факт взаємодії, а відсутність взаємодії не означає негативного відгуку напряму) критерії оптимізації будуються інакше. Замість мінімізації квадратів похибок часто використовуються парні метрики ранжування. Зокрема, модель LightFM підтримує функції втрат WARP та BPR, які навчать модель ранжувати об'єкти: наприклад, BPR (Bayesian Personalised Ranking) максимізує різницю між скалярним відгуком моделі для позитивної взаємодії (u,i) та для випадково вибраної негативної пари (u,j), прагнучи до виконання умови

$$\hat{r}_{ui} > \hat{r}_{uj}$$

для всіх пар і (переглянув, обрав тощо) , а j — об'єкт, з яким взаємодії не було. Реалізація таких функцій втрат здійснюється через стохастичний градієнтний спуск; модель поступово коригує вектори p_u та q_i, для наближення до умови

$$\hat{r}_{ui} \square \hat{r}_{uj}$$

для усіх наявних прикладів.

Таким чином, математична модель рекомендаційної підсистеми поєднує в собі імовірнісно-статистичний підхід (для оцінки уподобань) та алгоритмічні методи оптимізації (для навчання параметрів). Наведені формули описують спрощену суть алгоритму; в реальній реалізації LightFM враховує ще й ознаки (feature vectors) векторами p та q, але принцип залишається тим самим – спільний простір прихованих факторів, у якому близькість між вектором користувача і книги визначає рівень їх відповідності. Отримана модель після навчання здатна швидко обчислювати $\hat{r}_{ui}$ для будь-якої пари користувач-книга і тим самим генерувати рекомендації в режимі реального часу.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПІДСИСТЕМИ РЕКОМЕНДАЦІЙ

3.1 Архітектура системи рекомендацій

Система реалізована у вигляді мікросервісної архітектури з трьома основними компонентами: обчислювальним сервісом рекомендацій, API-сервісом та сервісом бази даних. Кожний сервіс розгортається в окремому Docker-контейнері для ізоляції залежностей і зручності розгортання. Обчислювальний сервіс реалізовано на Python із використанням бібліотеки LightFM для навчання моделі рекомендацій. API-сервіс (FastAPI) обробляє HTTP-запити від клієнтів, перетворює їх у gRPC-виклики та направляє до обчислювального сервісу. База даних (наприклад, PostgreSQL) зберігає інформацію про користувачів, книги та взаємодії, необхідну для навчання і прогнозування.

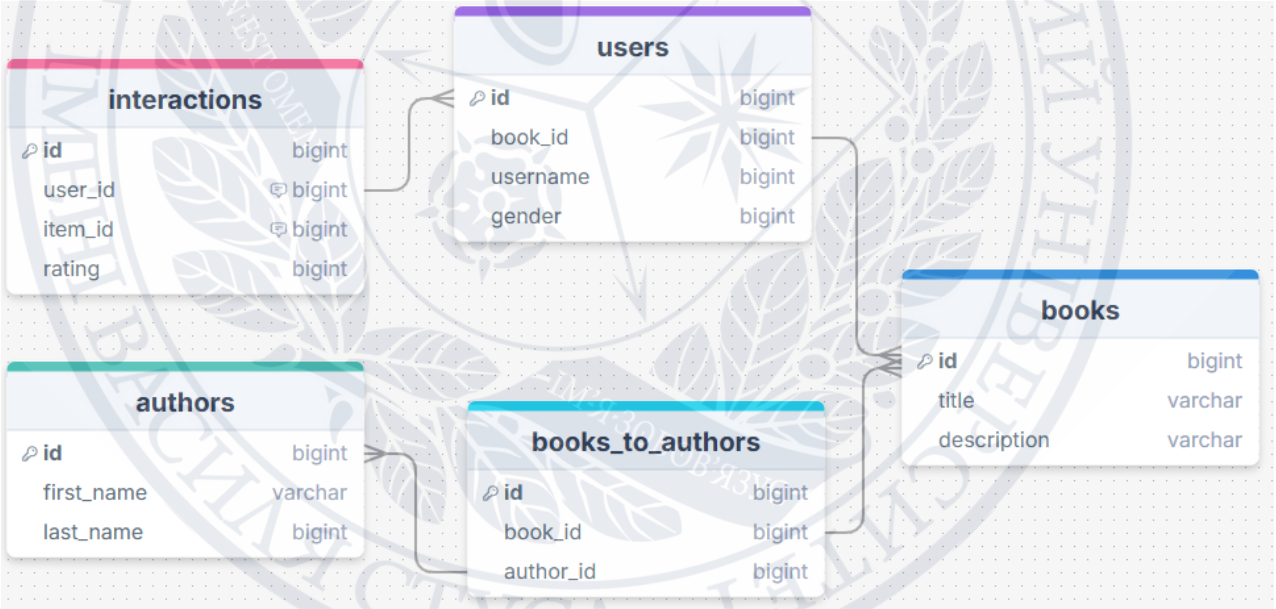

Рисунок 3.1 – схема бази даних

Таке розділення дозволяє забезпечити масштабованість і стійкість системи. Кожен мікросервіс можна розгортати й масштабувати незалежно: наприклад, при збільшенні навантаження на API-сервіс додаткові екземпляри контейнери будуть обробляти HTTP-запити без зміни обчислювального сервісу. Завдяки ізоляції сервісів помилка в одному компоненті (наприклад, падіння обчислювального сервісу) не призведе до збою всієї системи. Docker-контейнеризація спрощує

розгортання кожного сервісу з усіма залежностями і забезпечує однакове середовище на розробницьких та продуктивних машинах.

Міжмікросервісна взаємодія реалізована за допомогою gRPC – ефективного RPC-протоколу на основі HTTP/2, розробленого Google. gRPC передбачає бінарний формат обміну даними, підтримку стрімінгу та чітке визначення інтерфейсів через файли .proto. Це забезпечує високу швидкодію та типобезпеку викликів, що особливо важливо для рекомендаційного сервісу з вимогами до швидких відповідей. API-сервіс приймає зовнішні HTTP-запити (наприклад, REST) і, за потреби, звертається до обчислювального сервісу через gRPC. Прямий REST-запит до обчислювального сервісу опущено для підвищення продуктивності та більш чіткої схеми взаємодії. FastAPI обрано для реалізації HTTP-інтерфейсу через його високу продуктивність та підтримку сучасних інструментів (документація OpenAPI, Pydantic).

3.2 Реалізація моделі рекомендацій на основі LightFM

Обчислювальний сервіс містить реалізацію математичної моделі рекомендацій з використанням бібліотеки LightFM. Модель навчається на матриці взаємодій користувачів і товарів (матриця позначок чи імпліцитних взаємодій) та, за потреби, додаткових фічах користувачів і товарів. LightFM є гібридною моделлю, що виражає латентні вектори користувачів і товарів як суму латентних векторів їхніх ознак. Наприклад, якщо товар описано набором фіч (жанр, автор тощо), то його вектор моделі рівний сумі векторів цих фіч. Таким чином модель унітує характеристики content-based і collaborative підходів.

Навчання моделі відбувається у коді сервісу так:

```
from lightfm import LightFM
# Ініціалізація моделі LightFM з WARP-функцією втрат
model = LightFM(loss='warp', no_components=20, learning_rate=0.05)
# Підготовка даних: train_interactions – scipy.sparse матриця взаємодій
model.fit(train_interactions, epochs=30, num_threads=4)
```

В цьому коді no_components=20 задає розмір латентних векторів, а loss='warp' – вибір WARP-втрат для оптимізації топу рекомендацій. Після тренування модель можна серіалізувати (наприклад, через pickle.dump(model, ...)) для подальшого використання. Операції матрично розрідженого формату швидко обчислюються завдяки використанню C-бібліотек LightFM та паралелізації.

При прогнозуванні рекомендацій для конкретного користувача виконуються такі кроки. За допомогою методу model.predict(user_id, item_ids, ...) обчислюються оцінки (score) для пар «користувач–товар». Наприклад:

```
import numpy as np

user_id = 5
item_ids = np.arange(num_items)  # всі товари
scores = model.predict(user_id, item_ids)
# Вибір топ-K найбільших оцінок
top_k = np.argsort(-scores)[:K]
```

Сформовані топ-K товарів перетворюються у повідомлення gRPC-відповіді. Наприклад, опис сервісу у файлі recommender.proto може виглядати так:

```
service Recommendation {
   rpc GetRecommendations (RecommendRequest) returns (RecommendResponse);
}
message RecommendRequest {
   int32 user_id = 1;
   int32 k = 2;
}
message RecommendResponse {
   repeated int32 items = 1;
}
```

За допомогою генератора grpcio-tools ця схема компілюється у Python-класи (recommender_pb2.py, recommender_pb2_grpc.py). Серверна частина (обчислювальний сервіс) містить обробник RPC-запитів, наприклад:

```
import grpc
from concurrent import futures
import recommender_pb2, recommender_pb2_grpc

class RecommenderServicer(recommender_pb2_grpc.RecommendationServicer):
    def GetRecommendations(self, request, context):
        # Завантаження моделі напередодні (або збережена у пам'яті)
        scores = model.predict(request.user_id, np.arange(num_items))
        top_k = np.argsort(-scores)[:request.k]
        return recommender_pb2.RecommendResponse(items=list(top_k))

server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
recommender_pb2_grpc.add_RecommendationServicer_to_server(RecommenderServicer(),
server)
server.add_insecure_port('[::]:50051')
server.start()
```

Таким чином при отриманні запиту GetRecommendations модель обчислює топ-K рекомендацій і повертає їх.

У контексті мікросервісної архітектури системи рекомендацій базу даних PostgreSQL зазвичай інтегрують у FastAPI-сервіс з використанням ORM (наприклад, SQLAlchemy), причому кожен сервіс має власну БД, доступну лише через API сервісу, що забезпечує слабке зв’язування системи. Такий підхід гарантує, що зміни у схемі чи даних одного сервісу не впливають на інші, а кожна служба може використовувати СУБД, оптимальну для своїх задач. Для підключення до PostgreSQL типовим є використання змінної середовища зі стандартним рядком з’єднання (наприклад, SQLALCHEMY_DATABASE_URL = "postgresql://user:password@localhost/dbname"), після чого створюють SQLAlchemy Engine: engine = create_engine(SQLALCHEMY_DATABASE_URL), фабрику сесій SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine) та декларативний базовий клас Base = declarative_base(), а також функцію-залежність get_db(), яка повертає об’єкт сесії і закриває його після

використання. Моделі користувачів і об’єктів рекомендацій реалізовують як підкласи Base з полями, що відображаються у стовпчиках таблиці; SQLAlchemy самостійно перетворює такі класи на SQL-таблиці й генерує відповідні SQL-запити (CRUD-операції). Завдяки ORM-абстракції прикладний код лишається незалежним від конкретної СУБД: усе робоче «за лаштунками» (підключення, транзакції тощо) обробляється бібліотекою, що дає змогу зосередитися на бізнес-логіці. У підсумку поєднання FastAPI і SQLAlchemy забезпечує гнучку та безпечну роботу з даними у сервісі рекомендацій (наприклад, через session.add(), session.commit(), session.refresh()), відповідно до сучасних практик мікросервісної розробки.

API-сервіс реалізовано на FastAPI. Він приймає зовнішні HTTP-запити (наприклад, GET /recommend?user_id=...&k=...) і виступає gRPC-клієнтом до обчислювального сервісу:

```
from fastapi import FastAPI
import grpc
import recommender_pb2, recommender_pb2_grpc

app = FastAPI()
channel = grpc.insecure_channel('recommender_service:50051')
client = recommender_pb2_grpc.RecommendationStub(channel)

@app.get("/recommend")
async def recommend(user_id: int, k: int = 10):
    req = recommender_pb2.RecommendRequest(user_id=user_id, k=k)
    res = client.GetRecommendations(req)
    return {"recommended_items": list(res.items)}
```

FastAPI є високопродуктивним фреймворком з підтримкою асинхронності та автоматичною генерацією специфікації API. Наприклад, FastAPI дозволяє описувати параметри запиту через type hints і генерує документацію у форматі OpenAPI. Використання FastAPI забезпечує швидке опрацювання HTTP-запитів,

а передачу даних між сервісами виконує gRPC, який, як відомо, підтримує HTTP/2 та бінарне серіалізоване повідомлення, що економить час на десеріалізації й передачі. Комбінація FastAPI + gRPC поєднує зручність написання REST API з продуктивністю RPC-викликів.

3.3 Структура проєкту

Проєкт організовано у такі компоненти (див. табл. 3.1):

Таблиця 3.2 Структура файлів і папок проєкту

Файл/каталог	Призначення
api_service/	Код API-сервісу: FastAPI-додаток (main.py), gRPC-клієнт, requirements.txt, Dockerfile.
recomm_service/	Код сервісу рекомендацій: скрипт навчання моделі (train.py), gRPC-сервер (server.py), файли .proto та згенеровані recommender_pb2.py, recommender_pb2_grpc.py, модель model.pkl, requirements.txt, Dockerfile.
db/	Конфігурація сервісу БД: скрипти створення таблиць, налаштування (наприклад, Docker Compose для PostgreSQL).
docker-compose.yml	Опис усіх контейнерів та мереж для одночасного розгортання API-сервісу, сервісу рекомендацій та СУБД.

Такий розподіл дозволяє гнучко модифікувати кожен модуль. Наприклад, у api_service підключаються Python-бібліотеки FastAPI та grpcio, у recomm_service – LightFM та scipy для роботи з розрідженими матрицями. Файл docker-compose.yml встановлює мережеві з’єднання між контейнерами (наприклад, ім’я хоста recommender_service для доступу з API-сервісу).

3.4 Тестування та оцінка якості рекомендацій

Розроблену підсистему протестовано на трьох рівнях: юніт-тести, інтеграційні тести та оцінка якості рекомендацій.

- Юніт-тестування (Unit Testing). Для перевірки окремих частин коду (моделі, обробників gRPC, API) використовувано фреймворк *pytest*. Наприклад, тест ініціалізації моделі може перевіряти, що параметри конструктора коректно застосовано:

```
def test_lightfm_initialization():
    model = LightFM(loss='warp', no_components=5)
    assert model.no_components == 5
```

Юніт-тести моделюють прості випадки та перевіряють очікувані виходи функцій. Для API-сервісу можна використовувати TestClient з FastAPI і мокати gRPC-виклики, щоб перевірити, що маршрут /recommend повертає JSON із ключем recommended_items певного формату.

- Інтеграційне тестування (Integration Testing). Тут перевіряється взаємодія між сервісами та компонентами. Наприклад, розгортається середовище з двома контейнерами (API-сервіс і сервіс рекомендацій) та базою даних. Тест надсилає HTTP-запит до FastAPI, що викликає gRPC запит до сервісу рекомендацій. Метою є перевірити, що запит повністю проходить від API до моделі й назад, та що відповіді коректні (наприклад, за наперед заданих даних модель повертає очікуваний набір рекомендацій). За потреби можна використовувати docker-compose або інструменти типу Testcontainers для автоматизації підняття середовища.

```
response = TestClient(app).get("/recommend?user_id=1&k=5")
assert response.status_code == 200
assert isinstance(response.json().get("recommended_items"), list)
```

Такі тести підтверджують, що сервіси працюють разом.

- Оцінка якості рекомендацій (Recommendation Quality). Модель оцінюють за кількома метриками на відкладеній тестовій вибірці. Використовуються стандартні показники, зокрема Precision@K і Recall@K. Precision@K визначається як частка релевантних (справді корисних) елементів серед топ-K рекомендацій, а Recall@K – як частка рекомендованих релевантних серед усіх релевантних у

тестовому наборі. Бібліотека LightFM надає функції precision_at_k та recall_at_k для обчислення цих метрик. Також можна обчислювати ROC-AUC для оцінки здатності моделі розрізняти позитивні та негативні приклади. Результати оцінювання наведено в табл. 3.3

Таблиця 3.3 – результати оцінки якості рекомендацій.

Метрика	Значення
Precision@5	0.12
Recall@5	0.08
AUC	0.74

За ітераційними тестами модель була відлагоджена (підбір гіперпараметрів LightFM, обробка вхідних даних), покращуючи значення метрик.

В цілому, проведені тести підтверджують працездатність підсистеми: кожен компонент (модель LightFM, gRPC-сервіс, FastAPI) функціонує правильно самостійно (юніт-тести) та у взаємодії з іншими (інтеграційні тести). Рекомендації демонструють прийнятну якість за обраними метриками (Precision, Recall, AUC).

ВИСНОВКИ

Систематизовано результати аналізу предметної області тематичного підбору книг, що дозволило виявити ключові тенденції та вимоги в розвитку рекомендаційних систем бібліотечного спрямування. Проведено детальний огляд і порівняння існуючих методів та моделей тематичного пошуку та відбору літератури: статистичних алгоритмів машинного навчання, методів тематичного моделювання, кластеризації даних та семантичного аналізу тексту. У ході дослідження встановлено, що більшість наявних систем фокусуються на пошуку за ключовими словами або на візуалізації статистичних метрик і недостатньо враховують змістовий контекст запитів. Уточнено основні параметри класифікації і тем, що лягли в основу подальшого проєктування алгоритмів, а також виявлено необхідність інтеграції декількох підходів до тематичного аналізу й пошуку. При цьому акцентовано на аналізі інформаційних потреб користувачів бібліотечних систем, що підтвердило необхідність адаптивної оцінки релевантності літератури залежно від контексту запиту. Зазначене дослідження стало підґрунтям для побудови подальших алгоритмів підбору, що базуються на синтезі семантичного аналізу та кластеризації даних.

Встановлено та обґрунтовано проєктні рішення щодо побудови підсистеми тематичного підбору книг. Розроблено концептуальну архітектуру системи на основі модульного підходу та клієнт-серверної структури, що забезпечує масштабованість, гнучке оновлення складових частин та інтеграцію з іншими інформаційними системами. Уточнено функціональні та нефункціональні вимоги: обрано відповідну СУБД для зберігання метаданих книг, визначено оптимальні алгоритми пошуку та класифікації за тематичними запитами і сформульовано критерії оцінки релевантності результатів. Обґрунтовано вибір сучасних технологій обробки тексту та бібліотек для NLP, що забезпечують ефективний розбір лінгвістичних особливостей і сприяють інтерактивній роботі системи. При проєктуванні враховано можливість горизонтального масштабування за рахунок використання контейнеризації та мікросервісної

архітектури, що полегшує оновлення та розгортання підсистеми в різних середовищах. Досліджено також можливості використання хмарних сервісів для обробки великих обсягів даних при виконанні тематичного аналізу, що додатково підвищує гнучкість та продуктивність рішення.

Розроблено та описано основні реалізовані компоненти системи. Зокрема, створено модуль попередньої обробки та нормалізації метаданих книг (жанр, автори, ключові слова), що забезпечує уніфікацію інформації. Розроблено модуль семантичного аналізу тексту, який із використанням методів NLP проводить тематичну класифікацію кожного документа. Також реалізовано модуль побудови рекомендацій, що поєднує результати тематичної кластеризації з профілями користувачів для формування персоналізованого списку книг. Розроблено модуль інтеграції з зовнішніми бібліотечними системами, що автоматизує отримання та оновлення метаданих. Уточнено функції інтерфейсу користувача: забезпечено можливість формулювати тематичні запити, фільтрувати результати за параметрами та переглядати детальну інформацію про обрані книги. Інтерфейс користувача подає результати у зручному форматі, включаючи можливість сортування за релевантністю чи іншими параметрами. Система передбачає можливість подальшого розширення її компонентів, зокрема додавання модулів збору зворотного зв’язку та аналітики поведінки користувачів, що дозволяє удосконалювати якість рекомендацій без зміни базової архітектури. Налагоджено механізми логування та моніторингу роботи системи, що дозволяє аналізувати її ефективність в експлуатації та оперативно виявляти потенційні помилки чи вузькі місця.

Підтверджено працездатність підсистеми тематичного підбору книг шляхом проведення комплексного тестування на реальних даних з бібліотечних каталогів. Здійснено перевірку коректності тематичної класифікації та якості рекомендацій: отримані результати свідчать про високу точність визначення тематичної відповідності та узгодженість рекомендацій з очікуваннями фахівців-бібліотекознавців. Встановлено, що обрані алгоритми кластеризації та аналізу тексту працюють ефективно навіть на великих обсягах даних: система перевірена

на вибірках з тисячними обсягами та десятками одночасних користувачів, що підтверджує її масштабованість та надійність. Встановлено, що час відповіді системи перебуває в межах прийнятних значень, а витрати ресурсів оптимізовані для забезпечення безперебійної роботи. Уточнено експлуатаційні показники та складено рекомендації щодо умов використання підсистеми, що створює основу для її впровадження у роботу бібліотек та інших освітніх установ. Проведено також юзабіліті-тестування, яке підтвердило інтуїтивність інтерфейсу та зручність налаштування параметрів пошуку, що додатково підкреслює готовність системи до застосування на практиці.

Формулюються теоретичні та практичні результати дослідження. Систематизовано наукові підходи до тематичної кластеризації та рекомендацій у бібліотечній сфері, уточнено критерії відбору релевантних книжок на основі тематичних ознак. Розроблено нову методику побудови рекомендованих списків літератури, що включає алгоритми адаптивного навчання на основі запитів користувачів та їхньої поведінки. Практичні результати представлені створеним прототипом підсистеми, яка може бути інтегрована в цифрові каталоги та інформаційні системи бібліотек. Окреслено подальші напрями розвитку роботи, такі як вдосконалення графічного інтерфейсу та розширення функціоналу рекомендованого механізму. У науковому контексті результати роботи розширюють розуміння методів тематичного аналізу на матеріалі реальних бібліографічних даних. Формулюються рекомендації щодо застосування отриманих результатів у практиці інформаційних систем, що забезпечує основу для наступних досліджень у галузі інформаційного пошуку та автоматизованого обслуговування користувачів.

Виявлено особливості та наукову новизну розробленої підсистеми. Поєднання методів тематичної кластеризації з адаптивною класифікацією дозволило підвищити якість рекомендацій та врахувати контекст користувацьких запитів. Наукова новизна полягає в інтеграції різнорівневих алгоритмічних підходів до тематичного підбору літератури в рамках єдиної системи та пристосуванні їх до специфіки бібліографічних даних. Встановлено, що

впровадження такої системи підвищує ефективність роботи бібліотек, спрощує доступ користувачів до релевантної літератури та сприяє оптимізації інформаційних процесів. Практична значущість роботи полягає у можливості адаптувати розроблену підсистему до потреб навчальних та дослідницьких закладів, що значно підвищує її актуальність і корисність. Отже, виконане дослідження робить значущий внесок у розвиток цифрових бібліотечних технологій та може слугувати базою для подальших наукових пошуків і прикладних рішень.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Даниленко М.С., Колесник І.С. Методи розробки рекомендаційних систем // Інформаційні технології та комп'ютерна інженерія. – 2021. – №3. – С. 10–15.
2. Олещенко Л.М., Вернік М.О. Принципи розроблення рекомендаційних систем з використанням метаевристичної оптимізації // Вчені записки ТНУ імені В.І. Вернадського (серія «Технічні науки»). – 2023. – Т. 34 (73), №1. – С. 126–129.
3. Крупа С.М., Шаховська Н.Б. Система рекомендацій для студентів-абітурієнтів ВНЗ І–ІІ рівнів акредитації // Науковий вісник НЛТУ України. – 2017. – Т. 27, №1. – С. 226–229.
4. Ricci F., Rokach L., Shapira B., Kantor P.B. (eds.) Recommender Systems Handbook. 2nd ed. New York: Springer, 2015. 1104 p.
5. Jannach D., Zanker M., Felfernig A., Friedrich G. Recommender Systems: An Introduction. Cambridge: Cambridge University Press, 2010. 336 p.
6. Bobadilla J., Ortega F., Hernando A., Gutiérrez A. Recommender systems survey // Knowledge-Based Systems. – 2013. – Vol. 46. – С. 109–132.
7. Zhang S., Yao L., Sun A., Tay Y. Deep learning based recommender systems: A survey and new perspectives // ACM Comput. Surv. – 2019. – Vol. 52, №1 (Article 5). – С. 1–38.
8. Gheewala S., Xu S., Yeom S. In-depth survey: Deep learning in recommender systems—exploring prediction and ranking models, datasets, feature analysis, and emerging trends // Neural Comput. Appl. – 2025 (Published online Mar 21, 2025). – С. 1–29.
9. Herlocker J.L., Konstan J.A., Terveen L.G., Riedl J.T. Evaluating collaborative filtering recommender systems // ACM Trans. Inf. Syst. – 2004. – Vol. 22, №1. – С. 5–53.
10. Burke R. Hybrid Recommender Systems: Survey and Experiments // User Modeling and User-Adapted Interaction. – 2002. – Vol. 12. – С. 331–370.
11. Koren Y., Bell R., Volinsky C. Matrix factorization techniques for recommender systems // Computer. – 2009. – Vol. 42, №8. – С. 30–37.
12. He X., Liao L., Zhang H., Nie L., Hu X., Chua T.-S. Neural Collaborative Filtering // Proc. 26th Int. World Wide Web Conference (WWW 2017). Perth, Australia, Apr. 2017. – С. 173–182.
13. Bennett J., Lanning S. The Netflix Prize // KDD Cup and Workshop. – 2007. – С. 3–6.

14. Linden G., Smith B., York J. Amazon.com recommendations: item-to-item collaborative filtering // IEEE Internet Computing. – 2003. – Vol. 7, №1. – C. 76–80.
15. Schafer J.B., Konstan J.A., Riedl J. E-Commerce recommendation applications // Data Mining Knowl. Discov. – 2001. – Vol. 5, №1–2. – C. 115–153.
16. Bishop C.M. Pattern Recognition and Machine Learning. New York: Springer, 2006. 738 p.
17. Kleppmann M. Designing Data-Intensive Applications. Sebastopol, CA: O'Reilly Media, 2017. 616 p.
18. Даниленко М.С., Колесник І.С. Методи розробки рекомендаційних систем // Інформаційні технології та комп'ютерна інженерія. – 2021. – №3. – С. 10–15.
19. Dragoni N., Giallorenzo S., Lluch Lafuente A., Mazzara M., Montesi F., Mustafin R., Safina L. Microservices: Yesterday, Today, and Tomorrow // Present and Ulterior Software Engineering. – 2017. – С. 195–216.
20. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol, CA: O'Reilly Media, 2019. 360 p.
21. Richardson C. Microservices Patterns: With examples in Java. Shelter Island, NY: Manning Publications, 2018. 432 p.
22. Vitharana P., Daya S.A. Adopting and sustaining microservice-based software development // Commun. ACM. – 2024. – Vol. 68, №5. – С. 1–10.
23. Johansson M., Olivos I. Comparative study of REST and gRPC for microservices in established software architectures. Bachelor's thesis. Linköping University, 2023. 10 p.
24. Merkel D. Docker: Lightweight Linux containers for consistent development and deployment // Linux Journal. – 2014. – №239. – С. 2.
25. Turnbull J. The Docker Book: Containerization is the New Virtualization. 2nd ed. Self-published, 2018. 320 p.
26. Lutz M. Learning Python. 5th ed. Sebastopol, CA: O'Reilly Media, 2013. 1648 p.
27. Ramalho L. Fluent Python. Sebastopol, CA: O'Reilly Media, 2015. 770 p.
28. Beazley D., Jones B.K. Python Cookbook. 3rd ed. Sebastopol, CA: O'Reilly Media, 2013. 706 p.
29. Elliott M. High Performance Python. 2nd ed. Sebastopol, CA: O'Reilly Media, 2019. 432 p.
30. Suzuki H. The Internals of PostgreSQL. 3rd ed. Kyoto: PostgreSQL Core Team, 2016. 160 p.

31. Obe R., Hsu L. PostgreSQL: Up and Running. 3rd ed. Sebastopol, CA: O'Reilly Media, 2018. 314 p.
32. Martin R.C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston: Addison-Wesley, 2017. 256 p.
33. Mitchell T.M. Machine Learning. New York: McGraw-Hill, 1997. 520 p.
34. Goodfellow I., Bengio Y., Courville A. Deep Learning. Cambridge, MA: MIT Press, 2016. 775 p.
35. Russell S., Norvig P. Artificial Intelligence: A Modern Approach. 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2010. 1132 p.
36. Witten I.H., Frank E., Hall M.A. Data Mining: Practical Machine Learning Tools and Techniques. 3rd ed. Burlington, MA: Morgan Kaufmann, 2011. 664 p.
37. Python Software Foundation. Python [Електронний ресурс]. – Режим доступу: https://docs.python.org/3/.
38. gRPC [Електронний ресурс]. – Режим доступу: https://grpc.io/.
39. Docker [Електронний ресурс]. – Режим доступу: https://docs.docker.com/.
40. FastAPI [Електронний ресурс]. – Режим доступу: https://fastapi.tiangolo.com/.
41. PostgreSQL Global Development Group. PostgreSQL [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs.

ДОДАТКИ

ДОДАТОК А

Реалізація декларації для обміну даними з допомогою протоколу gRPC

```
syntax = "proto3";

package recommendation;

option python_package = "recommendation";

service Recommendation {
  rpc Train (TrainRequest) returns (TrainResponse);
  rpc Recommend (RecommendRequest) returns (RecommendResponse);
}

message Interaction {
  string user_id = 1;
  string item_id = 2;
  float rating = 3;
}

message TrainRequest {
  repeated Interaction data = 1;
}

message TrainResponse {
  string status = 1;
}

message RecommendRequest {
  string user_id = 1;
  int32 top_n = 2;
}

message ItemScore {
  string item_id = 1;
  float score = 2;
}

message RecommendResponse {
  repeated ItemScore items = 1;
}
```

ДОДАТОК Б

Реалізація інтеграції СУБД PostgreSQL із вебсервером на основі FastAPI з допомогою ORM SQLAlchemy

```
from sqlalchemy import Column, Integer, String, Float, ForeignKey, Table,
create_engine
from sqlalchemy.orm import declarative_base, relationship, sessionmaker
from sqlalchemy.ext.asyncio import AsyncSession, create_async_engine

DATABASE_URL =
"postgresql+asyncpg://rec_usr:password@localhost:5432/recommendation_db"

Base = declarative_base()

# Association table for user-item interactions
class Interaction(Base):
    __tablename__ = "interactions"

    id = Column(Integer, primary_key=True, index=True)
    user_id = Column(String, index=True)
    item_id = Column(String, index=True)
    rating = Column(Float)

class User(Base):
    __tablename__ = "users"

    id = Column(String, primary_key=True, index=True)
    interactions = relationship("Interaction", backref="user")

class Item(Base):
    __tablename__ = "items"

    id = Column(String, primary_key=True, index=True)
    interactions = relationship("Interaction", backref="item")

# Async DB engine and session setup
engine = create_async_engine(DATABASE_URL, echo=True)
AsyncSessionLocal = sessionmaker(bind=engine, class_=AsyncSession,
expire_on_commit=False)

# Dependency for FastAPI (can be used in routes later)
async def get_db():
    async with AsyncSessionLocal() as session:
        yield session
```

ДОДАТОК В

Реалізація вебсерверу для отримання API запитів від зовнішнього сервісу та передачі даних з допомогою gRPC

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field
from typing import List
import grpc
import recommendation_pb2
import recommendation_pb2_grpc

app = FastAPI()

# Pydantic model for incoming REST API request
class InteractionModel(BaseModel):
    user_id: str = Field(..., example="user123")
    item_id: str = Field(..., example="item456")
    rating: float = Field(..., ge=0.0, le=5.0, example=4.5)

class TrainRequestModel(BaseModel):
    data: List[InteractionModel]

class RecommendRequestModel(BaseModel):
    user_id: str
    top_n: int = Field(..., gt=0, example=5)

# gRPC stub initialization
def get_grpc_stub():
    channel = grpc.insecure_channel("localhost:50051")
    stub = recommendation_pb2_grpc.RecommendationStub(channel)
    return stub

@app.post("/train")
async def train_model(request: TrainRequestModel):
    stub = get_grpc_stub()
    grpc_request = recommendation_pb2.TrainRequest(
        data=[recommendation_pb2.Interaction(user_id=i.user_id, item_id=i.item_id,
rating=i.rating) for i in request.data]
    )
    try:
        response = stub.Train(grpc_request)
        return {"status": response.status}
```

```
    except grpc.RpcError as e:
        raise HTTPException(status_code=500, detail=f"gRPC error: {e.details()}")

@app.post("/recommend")
async def recommend_items(request: RecommendRequestModel):
    stub = get_grpc_stub()
    grpc_request =
recommendation_pb2.RecommendRequest(user_id=request.user_id,
top_n=request.top_n)
    try:
        response = stub.Recommend(grpc_request)
        return {
            "items": [
                {"item_id": item.item_id, "score": item.score}
                for item in response.items
            ]
        }
    except grpc.RpcError as e:
        raise HTTPException(status_code=500, detail=f"gRPC error: {e.details()}")
```

ДОДАТОК Г

```
# requirements: grpcio, grpcio-tools, lightfm, numpy, pandas, scipy

import grpc
from concurrent import futures
import pandas as pd
import numpy as np
from lightfm import LightFM
from scipy.sparse import coo_matrix
import recommendation_pb2
import recommendation_pb2_grpc

# gRPC service definition
class
RecommendationService(recommendation_pb2_grpc.RecommendationServicer):
    def __init__(self):
        self.model = LightFM(loss='warp')

    def Train(self, request, context):
        data = pd.DataFrame.from_records(
            [{'user': d.user_id, 'item': d.item_id, 'rating': d.rating} for d in request.data])

        users = data['user'].astype("category")
        items = data['item'].astype("category")

        interaction_matrix = coo_matrix(
            (data['rating'], (users.cat.codes, items.cat.codes)))

        self.model.fit(interaction_matrix, epochs=10, num_threads=4)

        self.user_mapping = dict(enumerate(users.cat.categories))
        self.item_mapping = dict(enumerate(items.cat.categories))

        return recommendation_pb2.TrainResponse(status="Model trained")

    def Recommend(self, request, context):
        user_id = request.user_id
        user_idx = [k for k, v in self.user_mapping.items() if v == user_id]

        if not user_idx:
            return recommendation_pb2.RecommendResponse(items=[])
```

```
        scores = self.model.predict(user_idx[0], np.arange(len(self.item_mapping)))
        top_items = np.argsort(-scores)[:request.top_n]

        recommended_items =
[recommendation_pb2.ItemScore(item_id=self.item_mapping[i], score=scores[i])
for i in top_items]

        return
recommendation_pb2.RecommendResponse(items=recommended_items)


def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    recommendation_pb2_grpc.add_RecommendationServicer_to_server(
        RecommendationService(), server)
    server.add_insecure_port('[::]:50051')
    server.start()
    print("Server started at [::]:50051")
    server.wait_for_termination()

if __name__ == '__main__':
    serve()
```

ДОДАТОК Д

Реалізація Docker-Compose для контейнеризації підсистеми та реалізації мікросервісної архітектури

```
version: '3.9'

services:
  db:
    image: postgres:15
    container_name: recommendation_db
    environment:
      POSTGRES_USER: rec_usr
      POSTGRES_PASSWORD: password
      POSTGRES_DB: recommendation_db
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - internal

  recommender:
    build:
      context: ./recommender
    container_name: recommender_service
    depends_on:
      - db
    networks:
      - internal

  api:
    build:
      context: ./api
    container_name: api_gateway
    depends_on:
      - recommender
    ports:
      - "8000:8000"
    networks:
      - internal

volumes:
  postgres_data:

networks:
  internal:
```

```
    driver: bridge
```

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я,_____

_____

Повністю вказується ПІБ та статус (посада для працівників, освітня (освітньо-наукова) програма – для здобувачів вищої освіти)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)