Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мережних технологій

Магістерська робота

освітній ступінь – магістр

на тему: «**Порівняльне середовище рекомендаційних систем**»

Виконала: студентка 2-го року навчання,

освітньо-наукової програми «Інженерія програмного забезпечення», 121

Щербатюк Аліна Олексіївна

Керівник Глибовець А.М.,_____

доктор технічних наук, професор

Рецензент_____

(прізвище та ініціали)

Магістерська робота захищена з оцінкою _____

Секретар ЕК _____

«____» _____ 2025 р.

Київ – 2025

ЗМІСТ

Анотація

У роботі досліджено підходи до створення універсальної рекомендаційної системи, здатної підтримувати підключення різнорідних рушіїв та опрацьовувати довільні датасети зі структурованими атрибутами. Розроблено серверний застосунок, що реалізує єдине REST-API, модуль імпорту даних із конфігураційним описом, адаптерну архітектуру для інтеграції алгоритмів, а також механізм уніфікованого зберігання інформації про користувачів, об’єкти та події у взаємодії. Обґрунтовано вибір технологій FastAPI, PostgreSQL, Gorse та бібліотеки Implicit, реалізовано взаємодію з рушіями на різних мовах програмування. На основі експериментального тестування із використанням датасетів MovieLens та Goodbooks продемонстровано можливості запропонованого рішення щодо гнучкого управління джерелами даних і порівняння результатів декількох рекомендаційних алгоритмів.

Вступ

У сучасному інформаційному середовищі рекомендаційні системи відіграють ключову роль у персоналізації цифрових сервісів, покращенні користувацького досвіду та підвищенні ефективності взаємодії між користувачем і контентом. Розвиток цієї галузі супроводжується стрімким зростанням кількості алгоритмів, реалізацій і фреймворків, що ускладнює їх безпосереднє порівняння та інтеграцію в єдине середовище. Потреба порівнювати та комбінувати різні рушії рекомендацій у будь-яких предметних областях без необхідності побудови окремої інфраструктури для кожного з них зумовлює актуальність створення уніфікованого програмного рішення.

Метою даної роботи є створити середовище для дослідження методів порівняння та комбінування рекомендаційних алгоритмів, а також підходів до уніфікації даних у рекомендаційних системах. На основі проведеного

аналізу реалізовано гнучке програмне середовище з єдиним API-інтерфейсом, яке дозволяє підключати різні рушії рекомендацій, обробляти довільні набори даних та формувати рекомендації незалежно від предметної області.

Об'єктом дослідження є архітектурні рішення, алгоритми та програмні технології, що застосовуються у побудові сучасних систем рекомендацій. Предметом дослідження є методи уніфікації вхідних даних, порівняння та комбінування результатів роботи різних рекомендаційних алгоритмів, а також способи організації універсального прикладного інтерфейсу для керування взаємодією з рекомендаційними рушіями.

У межах дослідження розв’язуються наступні завдання:

• Аналіз існуючих підходів до побудови та інтеграції рушіїв рекомендацій

• Проєктування універсального сховища подій і атрибутів із можливістю обробки довільних датасетів

• Розробка адаптерної архітектури для підключення різних рушіїв

• Реалізація єдиного API для керування навчанням моделей і формування рекомендацій

• Проведення експериментальної оцінки точності та швидкодії системи на прикладових датасетах

Запропоноване рішення дозволяє ефективно поєднувати декілька алгоритмів рекомендацій, швидко інтегрувати нові рушії та адаптуватися до різних форматів вхідних даних без зміни основної інфраструктури.

Розділ 1. Теоретична складова

1.1. Класифікація рекомендаційних систем

Рекомендаційні системи класифікують за типами алгоритмів, що застосовуються для передбачення вподобань користувачів. Основними класами є колаборативна фільтрація, фільтрація на основі вмісту, гібридні

методи, методи на основі знань та алгоритми, що використовують глибоке навчання. У цьому підрозділі розглянуто особливості кожного з цих класів алгоритмів, з акцентом на їх застосування в API-орієнтованих системах персоналізованих рекомендацій.

1.1.1 Колаборативна фільтрація

Колаборативна фільтрація (CF) є одним з найпоширеніших класів , що використовуються у побудові рекомендаційних систем. Її принцип базується на припущенні, що користувачі з подібними вподобаннями в минулому, ймовірно, проявлятимуть подібну поведінку в майбутньому. Цей клас алгоритмів не вимагає знання характеристик самих об’єктів (наприклад, категорій товарів або описів фільмів), а ґрунтується виключно на історії взаємодій користувачів з об’єктами.

Існує два основні способи реалізації колаборативної фільтрації: memory-based та model-based.

Memory-based передбачає обчислення схожості між користувачами або об'єктами на основі їхніх історичних взаємодій. Існують два основні різновиди цього підходу:

- User-based CF — система ідентифікує користувачів зі схожими вподобаннями та рекомендує об'єкти, які ці користувачі позитивно оцінили.
- Item-based CF — система шукає об'єкти, схожі до тих, що користувач уже оцінив, і рекомендує їх.

Для обчислення схожості зазвичай використовуються метрики на кшталт косинусної схожості або коефіцієнта Пірсона. Типовими алгоритмами, що реалізують цей підхід, є k-Nearest Neighbors (k-NN), реалізований як у user-

based, так і в item-based варіантах. Попри простоту та інтуїтивність, ефективність memory-based методів погіршується зі збільшенням обсягів даних, особливо в умовах високої розрідженості [1].

Model-based підхід ґрунтується на побудові узагальнюючої моделі на основі даних про попередні взаємодії користувачів. Найпоширенішим прикладом є матрицева факторизація, де матриця оцінок користувачів розкладається на добуток двох матриць меншої розмірності, що відображають латентні (приховані) характеристики користувачів та об'єктів. Особливого поширення набули алгоритми SVD (сингулярне розкладання), SVD++ (з урахуванням неявної взаємодії), а також ALS (alternating least squares), який ефективний для обробки великих обсягів імпліцитних даних [2].

Серед інших моделей варто виділити факторизаційні машини (FM) — алгоритми, що дозволяють враховувати не лише ідентифікатори користувачів та об’єктів, а й додаткові ознаки (наприклад, категорії товарів або вік користувача). Такі моделі добре масштабуються та забезпечують більшу гнучкість у використанні додаткової інформації [3].

У контексті API-орієнтованих систем рекомендацій колаборативна фільтрація часто реалізується за допомогою бібліотек на кшталт Surprise або LightFM, які дозволяють тренувати модель на батчах статичних даних та використовувати її для прогнозування оцінок або формування списків рекомендованих об'єктів. Наприклад, у бібліотеці Surprise реалізовано кілька варіантів алгоритмів колаборативної фільтрації — від простих KNN до розширених моделей типу SVD++.

1.1.2 Фільтрація на основі вмісту

Фільтрація на основі вмісту (Content-Based Filtering, CBF) ґрунтується на припущенні, що користувачеві сподобаються об'єкти, подібні до тих, які він

оцінив позитивно у минулому. Цей клас алгоритмів ґрунтується на аналізі атрибутів об'єктів і побудові індивідуального профілю користувача на основі історії взаємодій.

У простішому випадку об'єкти описуються векторами характеристик — наприклад, жанрами, категоріями, ціною тощо. Профіль користувача формується на основі сукупності ознак об'єктів, яким він надавав перевагу. Далі система обчислює ступінь схожості між профілем користувача та потенційними об'єктами, найчастіше — за допомогою косинусної схожості або евклідової відстані [4].

Класичні алгоритми CBF часто використовують метод TF-IDF для обробки текстових описів. Також застосовуються прості моделі машинного навчання, такі як: наївний байєсівський класифікатор, дерева рішень та логістична регресія для передбачення вподобань на основі атрибутів [5].

Сучасні алгоритми розширюють можливості CBF завдяки використанню глибокого навчання. Сучасні алгоритми розширюють можливості CBF завдяки використанню глибокого навчання. Наприклад, у сферах обробки мультимедійного контенту (музика, відео, подкасти) застосовуються моделі, що формують векторні представлення об'єктів за допомогою згорткових нейромереж (CNN), рекурентних мереж (RNN) або трансформерів. Це дає змогу автоматично витягувати ключові ознаки з тексту, зображень або аудіо [6].

Перевагою контентної фільтрації є здатність рекомендувати нові об'єкти, які ще не мають відгуків, що особливо актуально у cold-start сценаріях. Крім того, цей підхід дозволяє створювати персоналізовані пояснення: наприклад, «рекомендуємо тому, що вам сподобалась стаття про штучний інтелект».

Однак CBF має й обмеження. Найбільш критичне з них — ефект вузької спеціалізації (over-specialization), коли користувачу пропонуються лише дуже схожі об’єкти, без достатнього рівня новизни. Ще однією проблемою є необхідність у якісному описі об'єктів, адже без достатньої кількості атрибутів ефективність системи різко знижується.

У API-системах CBF реалізується через побудову векторів ознак об'єктів і збереження їх у базі даних або індексі. Профіль користувача формується динамічно при запиті та використовується для обчислення схожості. Наприклад, у LightFM можна поєднувати ознаки об'єктів із латентними факторами, що забезпечує гібридний ефект без складного додаткового моделювання.

1.1.3 Гібридні методи

Гібридні рекомендаційні системи поєднують кілька класів алгоритмів (зазвичай колаборативну та контентну фільтрацію) з метою покращення точності, стійкості та універсальності. Такий підхід дозволяє компенсувати недоліки кожного з методів — наприклад, проблема cold-start у колаборативній фільтрації вирішується за допомогою контентних ознак, а обмеження вузької спеціалізації CBF згладжується за рахунок аналізу поведінки інших користувачів.

У науковій літературі запропоновано кілька стратегій гібридизації [7]:

- Зважене об’єднання (weighted hybrid) — результати декількох моделей комбінуються з використанням наперед визначених або адаптивних ваг. Наприклад, фінальний рейтинг може розраховуватися як 70% колаборативного прогнозу + 30% контентного.

- Перемикання (switching) — система обирає, який алгоритм застосувати залежно від контексту або наявності даних. Наприклад, новим користувачам система може пропонувати рекомендації на основі атрибутів, а досвідченим — колаборативні.
- Змішування (mixed hybrid) — об'єкти, рекомендовані різними методами, об’єднуються в єдиний список. Це дозволяє збільшити різноманітність і покрити ширший спектр вподобань.
- Каскадне комбінування (cascade) — одна модель генерує попередній список кандидатів, який потім уточнюється або фільтрується іншою моделлю. Наприклад, CF формує топ-50 об'єктів, а потім CBF ранжує їх за релевантністю.
- Мета-рівень (meta-level) — одна модель використовується для створення ознак або профілів, які потім подаються на вхід іншій моделі.

Гібридні системи мають переваги у гнучкості, якісному охопленні та вищій точності. Наприклад, у конкурсі Netflix Prize фінальне рішення включало понад сотню моделей, об’єднаних у складну ансамблеву структуру, що забезпечило рекордну точність прогнозів [8].

У сучасних реалізаціях гібридні підходи набули нових форм завдяки використанню нейронних мереж. Так, моделі типу Neural Collaborative Filtering (NeuMF) поєднують матричну факторизацію та багатошарові нейронні мережі для моделювання взаємодій між користувачами та об’єктами [9].

У контексті API-рекомендацій, гібридизація може бути реалізована на рівні алгоритмів або на рівні відповідей сервера. Наприклад, система може

одночасно зберігати результати колаборативної і контентної фільтрації, а при запиті повертати зважене або позиційно перемішане об'єднання. У SaaS-платформах, таких як Recombee, гібридний підхід реалізовано за замовчуванням: результати комбінуються у фоні, забезпечуючи баланс точності, новизни та охоплення.

1.1.4 Методи на основі знань

Рекомендаційні системи, що використовують знання, будуються на попередньо визначених правилах або логіці, які дозволяють підбирати об'єкти, що відповідають конкретним запитам користувача. На відміну від методів, що ґрунтуються на даних взаємодій, ці системи орієнтовані на фіксовані критерії відповідності, що є особливо актуальним у сферах з високими вимогами до точності, наприклад у медицині, страхуванні чи банківській справі [10].

До основних типів належать:

- Rule-based системи, які застосовують набір заздалегідь сформульованих умов (наприклад, "обрати товар, якщо ціна менша за N та категорія — електроніка"). Такі рішення часто реалізуються через логічні конструкції або спеціальні механізми фільтрації.
- Case-based системи, що використовують приклади раніше успішно задоволених запитів і шукають аналогії до нових випадків. Це дозволяє пропонувати об’єкти, які схожі на ті, що вже відповідали потребам інших користувачів у подібних ситуаціях [10].

Головна перевага цього підходу — можливість працювати навіть у повній відсутності даних про попередні взаємодії, тобто вирішувати проблему cold-start. Крім того, система може генерувати чіткі пояснення, чому було

рекомендовано саме цей варіант — наприклад, «відповідає вказаному бюджету і підтримує потрібну функцію» [11].

У межах API-сервісів такий підхід зазвичай реалізується у вигляді обробки вхідних параметрів — наприклад, через передачу структурованих JSON-запитів, де зазначені обмеження й бажані характеристики. Рекомендації в цьому випадку можуть ґрунтуватися не на складних алгоритмах, а на прямому порівнянні об’єктів з умовами, заданими клієнтом. У деяких випадках такі системи працюють як частина фільтраційного модуля, що поєднується з іншими алгоритмами у складі гібридної архітектури.

Хоча методи на основі знань не мають такої гнучкості, як моделі машинного навчання, вони залишаються ефективними у випадках, коли об'єктивні вимоги користувача можна формалізувати, а ризик помилкової рекомендації є критичним [11].

1.1.5 Підходи з використанням глибокого навчання

В останнє десятиліття нейромережеві моделі стали активно впроваджуватись у рекомендаційні системи, відкриваючи можливості для роботи з великими обсягами неструктурованих даних та моделювання складних взаємозв’язків між користувачами й об’єктами. Підходи, що використовують глибоке навчання, дозволяють інтегрувати контент, поведінкову інформацію, часові фактори й навіть знання з графів у єдину систему [12].

Одним із найвідоміших напрямів є нейронна колаборативна фільтрація (Neural Collaborative Filtering, NCF), яка замінює традиційну операцію скалярного добутку у матричній факторизації на багатошарову нейронну мережу. Це дає змогу моделювати нелінійні залежності між користувачами та об’єктами, враховуючи складні патерни вподобань. Прикладом є модель

NeuMF, яка поєднує переваги MF і багатошарових персептронів для досягнення вищої точності [13].

Інший напрям — автокодери, які навчаються реконструювати вхідні дані (наприклад, вектор оцінок користувача) і водночас вивчають компактне латентне представлення. У рекомендаціях це дозволяє ефективно відновлювати відсутні значення. Автокодери є особливо ефективними для top-N задач у системах із розрідженими даними [14].

Рекурентні нейронні мережі (RNN), зокрема GRU або LSTM, активно використовуються у послідовних рекомендаціях, де важливо враховувати порядок дій користувача. Наприклад, після перегляду смартфона користувач із великою ймовірністю зацікавиться аксесуарами. Модель GRU4Rec показала високу ефективність у таких сценаріях, особливо у випадках коротких сесій або рекомендацій "у моменті" [15].

Ще потужнішим інструментом стали трансформери, які дозволяють моделювати довготривалі залежності у взаємодіях. У моделі SASRec історія користувача розглядається як послідовність, аналогічна до речення в обробці природної мови. Більш сучасна BERT4Rec моделює взаємозв’язки у двох напрямках, що дозволяє досягти ще кращої точності при прогнозуванні наступного об’єкта [16].

Крім того, графові нейронні мережі (GNN) стали ефективним способом обробки даних у вигляді взаємозв’язаних сутностей — користувачів, товарів, категорій тощо. Такі моделі, як NGCF чи PinSage, дозволяють поширювати інформацію між вузлами графа, забезпечуючи глибше розуміння контексту [17].

Переваги глибинного навчання полягають у гнучкості, можливості використання різних типів даних (текст, зображення, час, графи) і високій

точності у багатьох сценаріях. Водночас ці підходи мають суттєві недоліки: потреба у великих обсягах даних для навчання, складність налаштування, висока обчислювальна вартість і часто супроводжуються низькою інтерпретованістю [11].

У контексті систем рекомендацій із API-архітектурою такі моделі частіше використовуються у вигляді попередньо навчених компонентів, результати яких зберігаються у базі або кеші. Наприклад, рекомендації, згенеровані трансформером або автокодером, можуть періодично оновлюватись офлайн, а API просто повертає відповідні списки на запит клієнта. Такий підхід дозволяє зберегти ефективність глибинних моделей, водночас не ускладнюючи онлайн-сервіс.

1.2 Методи гібридизації та комбінування результатів

У багатьох випадках поєднання декількох рекомендаційних алгоритмів забезпечує кращу якість системи, ніж використання одного методу. Гібридизація дозволяє компенсувати слабкі сторони окремих моделей, забезпечити більшу адаптивність до різних сценаріїв і підвищити точність прогнозування. На практиці це особливо корисно у системах, що мають API-архітектуру, де важливо надавати стабільні результати для користувачів з різним рівнем даних, не втрачаючи швидкодії. Залежно від мети, яку переслідує система, можна комбінувати результати у різний спосіб: через злиття оцінок, зміну порядку елементів у списках або навіть повну заміну алгоритму залежно від контексту.

1.2.1 Алгоритмічне об’єднання

Одним із найпоширеніших технічних підходів до гібридизації є зважене об’єднання результатів декількох моделей. Такий механізм може бути реалізований у вигляді ансамблю, де кожен алгоритм повертає оцінку або

список, а система формує фінальний результат, поєднуючи їх за заданими або динамічними вагами.

У контексті API-систем ці операції зазвичай виконуються у сервісному шарі або окремому модулі агрегування. Наприклад, SaaS-платформи типу Recombee інтегрують декілька моделей фоном, автоматично комбінуючи їхні результати. Подібний принцип може бути застосований і в локальних рішеннях, де мета-моделі або агрегатори відповідають за вибір найбільш релевантних об’єктів на основі множини прогнозів.

1.2.2 Позиційне ранжування

Позиційне ранжування передбачає формування списку рекомендацій за заздалегідь визначеною логікою, яка базується не на об’єднанні оцінок, а на плануванні позицій, які повинні займати об’єкти у фінальному списку. Такий підхід дозволяє системі демонструвати результати кількох алгоритмів одночасно, чергуючи або перемішуючи їх у певному порядку. Це корисно у випадках, коли потрібно зберігати різноманітність або виконувати бізнес-вимоги щодо розміщення об’єктів.

Одним із прикладів реалізації є метод чергування, коли, наприклад, кожен другий елемент у списку отримується з контентної моделі, а інші — з колаборативної. Інший варіант полягає у використанні фіксованих слотів: перші три позиції займають об’єкти, отримані з персоналізованої моделі, далі додаються популярні товари або новинки. Таке ранжування дає змогу збалансувати точність і новизну, а також враховувати цілі бізнесу, наприклад, просування певних товарів або брендів [9].

Позиційне ранжування часто застосовується у великих e-commerce платформах або новинних агрегаторах. Наприклад, рекомендований блок може містити поєднання персональних статей, найпопулярніших за день та

матеріалів з обраної тематики. У цьому випадку кожен підсписок формує свій алгоритм, але остаточне розміщення контролюється логікою шаблону, який визначає позиції. Це дозволяє гарантувати, що важливі або стратегічні елементи завжди будуть помітні користувачу.

У рамках API-сервісів подібний механізм може бути реалізований на рівні мікросервісної архітектури, де окремі модулі формують рекомендації за своїм алгоритмом, а спеціальний блок формує підсумковий список, враховуючи позиції для кожної категорії результатів. Таким чином, навіть якщо алгоритми не узгоджені між собою, система все одно повертає цілісний результат, структурований згідно з заданими правилами.

Перевагою цього методу є передбачуваність і контрольованість поведінки системи. Однак він вимагає ретельного дизайну шаблонів і логіки формування позицій, оскільки занадто жорсткі правила можуть негативно впливати на персоналізацію або сприйняття результатів користувачем.

1.2.3 Вибіркова маршрутизація

Цей підхід передбачає, що система вибирає один із доступних алгоритмів для генерації рекомендацій залежно від поточного контексту, типу користувача або специфіки даних. На відміну від алгоритмічного об’єднання чи позиційного ранжування, де залучаються результати кількох моделей одночасно, у випадку вибіркової маршрутизації виконується лише один рекомендатор, який вважається найбільш релевантним для даного сценарію.

Найчастіше вибір алгоритму ґрунтується на простих правилах. Наприклад, якщо користувач новий і ще не має історії взаємодій, система застосовує Content-Based Filtering або популярнісний алгоритм. Якщо ж користувач активний і має багато оцінок або кліків, обирається колаборативна

фільтрація. Такі сценарії особливо актуальні для систем, що мають обмежені ресурси або потребують швидкої відповіді [7].

Інший варіант маршрутизації — це використання класифікатора або meta-моделі, яка прогнозує, який з алгоритмів найкраще впорається із завданням у поточному випадку. Такий підхід частіше застосовується у великих системах, де на основі історичних даних було виявлено закономірності в ефективності різних моделей для окремих груп користувачів або запитів.

У прикладних реалізаціях, зокрема у SaaS-рішеннях або API-системах, маршрутизація може виконуватись на рівні бекенду. Наприклад, деякі сервіси (наприклад, Amazon Personalize або Recombee) дозволяють конфігурувати логіку вибору рекомендаторів для різних подій: окремий алгоритм для головної сторінки, інший — для схожих товарів або результатів пошуку.

Реалізація маршрутизації в рамках власного API може здійснюватись шляхом перевірки вхідних параметрів. Якщо, наприклад, у запиті передається історія з менш ніж трьома оцінками, застосовується fallback-модель. Під fallback-моделлю зазвичай мається на увазі резервна або запасна модель, яка використовується у випадках, коли основна модель не може бути застосована або не має достатньо даних. І навпаки, за наявності достатньо інформації — викликається основна модель рекомендацій. Це дозволяє створювати адаптивні системи, які гнучко реагують на змінні умови і забезпечують стабільну якість сервісу без потреби у надмірній обчислювальній складності.

Перевагою вибіркової маршрутизації є оптимальне використання ресурсів та покращення користувацького досвіду в умовах неоднорідних даних. Водночас, для ефективної роботи системи потрібно добре продумати

правила маршрутизації або навчити класифікатор, що здатен робити точний вибір алгоритму в реальному часі.

1.2.4 Зважене об'єднання результатів (метод Борда)

Зважене об’єднання, також відоме як метод Борда, передбачає комбінування рекомендаційних списків або оцінок, сформованих різними алгоритмами, шляхом розрахунку агрегованого показника на основі заздалегідь визначених ваг. Цей метод дозволяє ефективно поєднувати декілька джерел інформації, зберігаючи при цьому контроль над внеском кожної моделі у фінальний результат.

У практиці систем, що ґрунтуються на API-архітектурі, зважене об’єднання часто реалізується шляхом поелементного обчислення сумарної оцінки для кожного об’єкта за формулою виду рекомендаційний_бал = w1 * score1 + w2 * score2 + ... + wn * scoren, де score — це оцінка, надана конкретним алгоритмом, а w — відповідна вага. Такий підхід є гнучким, оскільки дозволяє регулювати вплив кожної моделі відповідно до контексту або результатів попереднього тестування [8].

Крім роботи зі значеннями оцінок, зважене об'єднання може застосовуватись також до позицій у списках рекомендацій. Наприклад, якщо об'єкт займає перше місце у списку одного алгоритму та п’яте — в іншому, то підсумковий рейтинг може враховувати обидві позиції із відповідними вагами. Цей підхід дозволяє враховувати рангову інформацію навіть за відсутності порівнюваних числових оцінок.

У деяких системах використовується стохастичний варіант методу, коли вибір елементів з різних списків виконується із заданою ймовірністю, пропорційною вагам. Такий підхід сприяє підвищенню різноманітності

рекомендацій та дає змогу уникати надмірної концентрації на одному алгоритмі.

Вагові коефіцієнти можуть задаватися вручну або визначатися емпіричним шляхом за результатами крос-валідації. У складніших конфігураціях система може автоматично оновлювати ці значення на основі продуктивності моделей у конкретному сегменті користувачів або сценарії використання.

У SaaS-рішеннях зважене об'єднання часто застосовується як частина базової логіки формування результатів. Наприклад, у Recombee внутрішній ансамбль комбінує результати колаборативної фільтрації, контентного аналізу та популярнісної моделі, забезпечуючи збалансований список рекомендацій без потреби у ручному налаштуванні.

Перевагою цього підходу є його простота, передбачуваність і можливість гнучкого налаштування. Водночас, ефективність зваженого об’єднання значною мірою залежить від правильного вибору коефіцієнтів, особливо у випадках, коли якість моделей суттєво відрізняється або змінюється з часом.

1.3 Метрики оцінювання якості рекомендацій

Оцінювання якості рекомендаційної системи є важливою складовою її розробки та вдосконалення. Метрики дозволяють кількісно виміряти ефективність моделі, враховуючи різні аспекти: точність передбачення оцінок, релевантність ранжування, охоплення, новизну та різноманітність результатів. Вибір конкретних метрик залежить від завдань, які вирішує система, та типу даних, з якими вона працює.

1.3.1 Метрики точності

Ці метрики застосовуються у задачах, де модель прогнозує оцінки користувачів для об'єктів. Найчастіше використовують:

- MAE (Mean Absolute Error) — середня абсолютна похибка;
- RMSE (Root Mean Square Error) — корінь квадратний із середньої квадратичної похибки.

RMSE надає більшу вагу більшим відхиленням, тому є чутливою до значних помилок, тоді як MAE рівномірно враховує всі похибки, демонструючи стійкість до викидів [10].

1.3.2 Метрики ранжування

У більшості практичних застосувань важливо не лише передбачити оцінки, а й сформувати список найбільш релевантних об’єктів. Для цього використовують:

- Precision@K — частка релевантних об’єктів серед перших K у списку;
- Recall@K — частка знайдених релевантних об’єктів від усіх можливих;
- NDCG (Normalized Discounted Cumulative Gain) — враховує позиції релевантних об’єктів у списку, надаючи вищу вагу першим позиціям [11].

1.3.3 Метрики покриття та новизни

Покриття (coverage) відображає частку об’єктів у базі, які система здатна рекомендувати. Чим вище значення — тим більший потенціал для різноманітності.

Новизна (novelty) характеризує здатність системи рекомендувати менш популярні або невідомі користувачу об’єкти. Метрика серендипності (serendipity) додатково оцінює несподіваність рекомендацій при збереженні їх корисності [12].

1.3.4 Вибір метрик для оцінювання результатів експерименту

У системах, що працюють в офлайн-режимі або не мають постійного зворотного зв’язку від користувачів, зазвичай використовують точкові метрики (MAE, RMSE) та метрики ранжування (Precision@K, NDCG). У таких випадках важливо дотримуватись принципів розділення даних на тренувальні та тестові набори, аби уникнути витоку інформації.

Комбінування кількох метрик дозволяє сформувати багатогранне уявлення про якість системи, виявити її сильні та слабкі сторони, а також приймати обґрунтовані рішення щодо подальшої оптимізації [13].

1.4 Уніфікація даних

Однією з ключових проблем у побудові ефективних рекомендаційних систем є уніфікація даних. У процесі інтеграції джерел інформації розробникам доводиться працювати з різнотипними форматами даних: оцінками користувачів, метаданими об’єктів, логами активності, а також контекстними ознаками (місцезнаходження, час, пристрій тощо). Складність полягає не лише в синтаксичній несумісності форматів, а й у глибшому семантичному рівні, де значення атрибутів може кардинально відрізнятися залежно від джерела.

Найбільш поширеною проблемою є відмінність у схемах атрибутів. Наприклад, у одному джерелі фільми можуть мати жанри як окремі категорії, а в іншому — у вигляді списку ключових слів. Це ускладнює

побудову уніфікованого профілю об’єкта. Аналогічно, деякі джерела можуть мати відсутні або неповні дані, що потребує застосування технік заповнення пропусків або згладжування [14]. У таких випадках важливу роль відіграють методи попередньої обробки даних, які дозволяють зберегти структурну цілісність датасету та зменшити ризик спотворення рекомендацій.

Ще одним викликом є семантична узгодженість атрибутів. Ідентичні на перший погляд ознаки можуть мати різне значення залежно від контексту. Наприклад, атрибут "тип контенту" може означати жанр, формат або джерело. Для вирішення цієї проблеми використовують методи нормалізації, мапінгу значень, а також побудову онтологій або графів знань, які дозволяють формалізувати відношення між сутностями [15]. Графи знань, зокрема, відкривають можливість побудови рекомендацій на основі семантичної близькості між об'єктами, що особливо корисно у системах, орієнтованих на пояснюваність результатів.

Уніфікація користувацьких і контекстних атрибутів є критичною для гібридних систем, які поєднують колаборативні та контентні підходи. Наприклад, факторизаційні машини (Factorization Machines, FM) вимагають однакового представлення як ID, так і ознак користувачів, об’єктів і контексту. Це дозволяє будувати єдину узагальнену матрицю ознак, яка забезпечує ширші можливості для моделювання взаємозв’язків. У системах, що працюють із декількома мовами, важливо також враховувати локалізацію назв, описів та категорій, що вимагає застосування механізмів мовної стандартизації або перекладу.

У системах, що реалізують рекомендації через API, необхідно ще на етапі розробки стандартизувати структуру вхідних даних. Це стосується як формату оцінок (наприклад, rating: float, timestamp: int), так і опису об’єктів

(title: str, category: list[str]). Сучасні API-рішення нерідко підтримують передачу атрибутів у вигляді JSON-об'єктів із гнучкою схемою, проте внутрішня система має конвертувати ці дані до узгодженого формату для тренування моделей. Іншими словами, має бути передбачено шар трансформації, який виконує роль посередника між сирими даними та модельними представленнями.

Зростаючу роль у процесі уніфікації відіграють стандартизовані схеми обміну, наприклад, Open Recommendation Specification (ORS), які визначають формати подання даних про користувачів, об’єкти та взаємодії. Такі підходи спрощують інтеграцію систем у масштабних мультиплатформених середовищах. Крім того, застосування формальних специфікацій спрощує процес тестування, оновлення моделі та впровадження нових джерел даних без необхідності суттєвих змін у логіці системи [16].

1.5 Огляд існуючих рішень

Рекомендаційні системи дедалі частіше реалізуються у вигляді гнучких бібліотек або сервісів із відкритим API, які можна легко інтегрувати в різні програмні продукти без складного налаштування. У цьому підрозділі наведено огляд популярних рішень, що застосовуються у наукових дослідженнях, MVP-проєктах та у виробничих середовищах.

Surprise — легка у використанні бібліотека для побудови моделей на основі явних рейтингів. Підтримує низку класичних алгоритмів, зокрема KNN, SVD, SVD++, BaselineOnly. Основна перевага — простота та зручна інтеграція у невеликі прототипи. Обмеженням є відсутність підтримки імпліцитних даних і контекстних ознак [17].

LightFM — поєднує контентну фільтрацію з латентними моделями, підтримує як явні, так і імпліцитні дані, працює з атрибутами користувачів та об’єктів. Може тренуватися за метриками ранжування. Завдяки сумісності з API бібліотеки scikit-learn активно використовується у стартапах і продуктивних середовищах [18].

LensKit — орієнтований на освітні та дослідницькі цілі. Підтримує багато класичних алгоритмів і експериментальні конфігурації, проте проєкт не оновлюється і не рекомендується для production-використання.

RecBole — сучасний фреймворк на базі PyTorch із підтримкою понад 70 моделей, включаючи deep learning архітектури (наприклад, трансформери та графові нейромережі). Забезпечує гнучкість, але потребує налаштування, тому більше підходить для наукових задач або великих систем [19].

Gorse — автономна система з REST API, яка реалізує гібридний підхід. Підтримує кластеризацію, cold-start, автоматичне оновлення моделей, персоналізацію. Простий запуск, конфігурація через YAML, наявна панель адміністратора. Добре підходить для швидкої інтеграції в мікросервісну архітектуру [20].

Сьогодні спостерігається чітка тенденція до впровадження готових, модульних та масштабованих фреймворків є тенденція до використання готових, модульних та масштабованих фреймворків, які скорочують час розробки та підвищують гнучкість рекомендаційних систем.

Таблиця. Порівняльна характеристика рекомендаційних фреймворків

Назва	Тип алгоритмів	Підтримка імпліцитних даних	Архітектура / API	Основні переваги	Недоліки
Surprise	KNN, SVD, SVD++	Ні	Локальна (без API)	Простота інтеграції, швидкий старт	Тільки явні рейтинги
LightFM	Content-based + факторизація	Так	Через Python API	Підтримка додаткових атрибутів, метрики ранжування	Обмежена документація
LensKit	Різні (дослідницькі)	Обмежена	Академічна платформа	Гнучкість для експериментів	Застарілий, не підтримується активно
RecBole	Deep learning, GNN, трансформери	Так	Через PyTorch / скрипти	Великий набір моделей, гнучкість	Високий поріг входу, складність
Gorse	Гібридні, популярніні	Так	REST API, кластеризація	Автоматичне оновлення, cold-start, SaaS-ready	Обмежена кастомізація внутрішніх моделей

Розділ 2. Практична реалізація

2.1 Архітектура системи рекомендацій

У межах реалізації системи рекомендацій застосовано модульну архітектуру, відповідно до якої кожен компонент виконує строго визначену функцію. Такий підхід забезпечує гнучкість проєктування, спрощує налагодження й тестування, а також створює передумови для подальшого розширення функціональності без потреби у суттєвих змінах до загальної структури.

Система складається з трьох ключових компонентів: сервера прикладного програмного інтерфейсу (API), рушія рекомендацій та системи управління базами даних (СУБД). Усі компоненти розгорнуто у відокремлених контейнерах за допомогою Docker, що дозволяє уникнути проблем сумісності бібліотек, конфліктів середовищ виконання та складнощів із мережею. Завдяки такому рішенню забезпечується відтворюваність результатів на різних інфраструктурних платформах.

Сервер API реалізовано засобами мови Python із використанням фреймворку FastAPI, що забезпечує асинхронну обробку запитів, декларативне визначення маршрутів і автоматичну генерацію документації. Він приймає запити клієнтів, передає дані до рушія рекомендацій, зберігає інформацію про користувачів та об'єкти у базі даних і повертає відповіді у форматі JSON. Основні маршрути реалізовано за адресами /feedback, /recommend, /train.

З метою підтримки кількох алгоритмів було реалізовано адаптерну архітектуру, яка дозволяє підключати різні рушії через єдиний інтерфейс взаємодії. У поточній реалізації інтегровано два рішення: Gorse — автономний рушій із вбудованим REST API, який реалізує методи

колаборативної та контентної фільтрації; та бібліотеку Implicit — Python-реалізацію алгоритму латентної факторизації ALS, орієнтовану на імпліцитні відгуки. Така структура дозволяє легко додавати нові рушії без необхідності змінювати інші компоненти системи.

Для зберігання структурованих даних про користувачів та об'єкти застосовано PostgreSQL з розширенням pgvector, що забезпечує підтримку векторного подання ознак. Це створює підґрунтя для реалізації векторного пошуку та гібридних стратегій у подальшому розвитку системи.

Усі сервіси об'єднано в єдину віртуальну мережу Docker, що дозволяє здійснювати взаємодію між контейнерами за іменами, визначеними у конфігураційному файлі docker-compose.yml. Зокрема, сервер API звертається до СУБД під іменем db, а до Gorse — під іменем gorse. Ініціалізація всієї інфраструктури здійснюється командою docker compose up -d, після чого сервіси автоматично запускаються та починають обробку запитів відповідно до своїх функціональних ролей.

Передбачено зручний інтерфейс для перевірки функціональності: Swagger-документацію за адресою http://localhost:8000/docs та адміністративну панель Gorse — http://localhost:8088.

Таким чином, запропонована архітектура забезпечує гнучкість, масштабованість і розширюваність системи, а також спрощує подальший супровід і модернізацію її компонентів.

Для зберігання метаданих про користувачів та об’єкти, а також можливого подальшого розширення системи на основі контентної фільтрації, використано PostgreSQL з розширенням pgvector. Обраний підхід дає змогу

уніфіковано зберігати числові ознаки у векторному форматі, що є важливою передумовою для реалізації векторного пошуку або гібридних стратегій у майбутньому.

Усі контейнери системи об’єднані в єдину віртуальну мережу Docker. Це дає можливість кожному сервісу звертатися до іншого за фіксованим іменем, що відповідає назві контейнера у конфігураційному файлі docker-compose.yml. Наприклад, сервер API звертається до бази даних як до db, а до рушія Gorse — як до gorse. Встановлення з’єднання відбувається автоматично під час запуску системи.

Запуск усієї інфраструктури здійснюється одною командою — docker compose up -d. Після цього всі сервіси ініціалізуються та починають обробку запитів відповідно до призначених ролей. Користувач має доступ до інтерфейсу Swagger за адресою http://localhost:8000/docs для перевірки API та до вебінтерфейсу Gorse на http://localhost:8088.

У процесі створення архітектури було важливо забезпечити не лише працездатність, а й зручність підтримки, можливість швидкого масштабування й адаптації під нові потреби. Саме тому було реалізовано чітке розмежування обов’язків між модулями, забезпечено мінімальну залежність між компонентами та спрощено налаштування середовища. Такий підхід дозволяє зосередитися на вдосконаленні окремих частин системи без необхідності повного її перепроєктування.

2.2 Формат імпорту даних на основі YAML

Однією з ключових функцій розробленої системи є можливість попереднього імпорту користувацьких даних для подальшої генерації рекомендацій. Для цього було реалізовано підтримку структури у форматі YAML (YAML Ain’t Markup Language), яка дає змогу зручно, у

декларативному вигляді, задавати інформацію про користувачів, об'єкти та їх взаємодії. Формат було обрано через його читабельність, широке поширення у DevOps-практиках та зручну інтеграцію з Python.

Файл імпорту містить три логічні блоки: users, items та feedback. У блоці users перераховуються ідентифікатори користувачів та їх атрибути (наприклад, стать, вік або географічна категорія), які за потреби можна використовувати для побудови гібридних моделей або фільтрації результатів. У блоці items зберігаються об'єкти рекомендацій: фільми, товари або інші сутності, що можуть бути рекомендовані. До кожного об'єкта можна додати описові характеристики (жанр, тег, ціна тощо), які в перспективі можуть бути векторизовані та використані для побудови контентної фільтрації. Блок feedback містить записи про взаємодії між користувачами та об'єктами. Ці взаємодії описуються як триплети у форматі: користувач — тип взаємодії — об'єкт, наприклад: user1 like item4.

Таке представлення забезпечує гнучкість: кожен блок можна оновлювати незалежно, а типи взаємодій задавати у довільному вигляді (наприклад, like, view, click, rate, тощо). Це дозволяє використовувати однаковий формат для налаштування різних рушіїв рекомендацій, у тому числі Gorse та Implicit, кожен із яких по-своєму інтерпретує вхідні дані. У випадку Gorse ці дані можуть бути передані через REST API або підвантажені через фоновий процес обробки даних. У разі Implicit, YAML-файл перетворюється у відповідну розріджену матрицю взаємодій, що використовується для обчислення латентних факторів.

Структура YAML-файлу дозволяє легко адаптувати систему до нових задач без необхідності зміни коду. Наприклад, додавання нового типу взаємодії або нової характеристики об’єкта не потребує модифікації API-сервера чи бази даних, достатньо оновити файл імпорту. Це спрощує тестування

моделей, перенавчання на нових вибірках, а також прискорює перевірку гіпотез при зміні структури даних.

На практиці імпорт YAML-файлу відбувається через спеціальний маршрут у системі або ж використовується як початковий спосіб завантаження даних до рушія. В обох випадках підтримується валідація вмісту на рівні схеми, що дозволяє виявляти помилки форматування або неочікувані типи даних ще до початку обробки.

Обраний формат дозволяє уніфікувати початкове завантаження даних, забезпечити гнучке описання об’єктів та користувачів, а також підтримати роботу з різними рушіями без дублювання коду або створення окремих структур для кожного випадку. Його використання значно пришвидшує процес інтеграції нових даних у систему, що є особливо цінним у дослідницькому та прототипному сценарії.

2.3 Реалізація адаптерної архітектури

У рамках реалізації системи рекомендацій було прийнято рішення застосувати адаптерну архітектуру. Такий підхід дозволяє абстрагувати логіку взаємодії з конкретними рушіями рекомендацій та забезпечити єдиний інтерфейс, через який основний застосунок взаємодіє з обчислювальними модулями. Це створює передумови для розширення системи, спрощує інтеграцію нових алгоритмів і дозволяє варіювати конфігурацію залежно від потреб користувача або особливостей даних.

У системі було реалізовано абстрактний базовий клас, що визначає основні методи, необхідні для інтеграції будь-якого рушія. Серед них методи для обробки запитів на отримання рекомендацій, навчання моделей, запису зворотного зв’язку від користувачів та ініціалізації рушія. Цей клас є

частиною внутрішньої структури системи та задає контракт, якого мають дотримуватися всі конкретні реалізації.

На основі цього базового інтерфейсу було створено окремі адаптери для кожного з двох використовуваних рушіїв. Адаптер GorseAdapter взаємодіє з системою Gorse через HTTP-запити до REST API. Він включає функції надсилання зворотного зв'язку, ініціалізації рекомендаційного процесу, запуску тренування моделі та отримання результатів у структурованому вигляді. У свою чергу, адаптер ImplicitAdapter працює із бібліотекою Implicit локально, без потреби у зовнішньому API, що дозволяє використовувати її у задачах з обмеженим доступом до мережі або в режимі офлайн.

Механізм реєстрації рушіїв базується на використанні асоціативного контейнера, в якому кожному ідентифікатору відповідає конкретний адаптер. Під час старту системи або при отриманні запиту від клієнта сервер визначає, який рушій має бути використаний, та динамічно створює відповідний об'єкт адаптера. Це дозволяє запускати одну й ту саму систему з різними конфігураціями без зміни коду.

Структура адаптерів підтримує розширення: при необхідності до системи можна додати новий рушій, реалізувавши окремий адаптер, який наслідує базовий клас та перевизначає ключові методи. Це забезпечує високу гнучкість і масштабованість архітектури, що є особливо важливим у задачах дослідження ефективності різних алгоритмів або при адаптації системи під нові типи даних.

Обраний підхід також дозволяє забезпечити тестованість кожного адаптера окремо, що спрощує виявлення помилок та підвищує надійність усієї системи. Адаптери реалізовані таким чином, щоб взаємодія з ними була

незалежною від внутрішньої логіки рушія, завдяки чому система залишається узгодженою та керованою навіть при значних змінах в окремих її модулях.

2.4 REST API для інтеграції рушіїв та доступу до рекомендацій

Інтеграція системи рекомендацій з іншими компонентами реалізована за допомогою REST API, що забезпечує гнучкий та стандартизований механізм обміну даними. Такий підхід дозволяє спростити доступ до функціональності системи з боку зовнішніх застосунків, а також реалізувати віддалену взаємодію між сервісами в межах розподіленої інфраструктури. Інтерфейс побудовано за принципами REST (Representational State Transfer), що забезпечує простоту, масштабованість та зручність використання. Його реалізовано з використанням фреймворку FastAPI**,** який підтримує асинхронну обробку запитів і автоматичну генерацію документації, що значно спрощує інтеграцію та тестування системи.

2.4.1 Основні маршрути API

API реалізує кілька ключових маршрутів, що відповідають за обробку вхідних даних, запуск алгоритмів навчання та генерацію рекомендацій. До основних належать:

- /feedback — приймає історію взаємодій користувача з об’єктами (наприклад, рейтинги або перегляди);
- /recommend — повертає список рекомендованих об’єктів на основі наявних даних;
- /train — ініціює процес тренування моделей (локально або в зовнішньому рушії);

Кожен маршрут підтримує HTTP-методи відповідно до своєї функції, наприклад, POST для надсилання даних або GET для отримання рекомендацій.

2.4.2 Обробка запитів на тренування та рекомендації

Процес обробки запитів реалізовано у кілька етапів. Дані, що надходять до API, передаються безпосередньо до відповідного рушія рекомендацій через адаптер.

У разі використання Gorse, API надсилає запит до зовнішнього REST-сервісу через HTTP. У разі використання Implicit, викликається локальна Python-функція, яка реалізує алгоритм ALS (Alternating Least Squares). Ця логіка інкапсульована в адаптерному шарі, який забезпечує уніфіковану взаємодію незалежно від типу рушія. Така архітектура спрощує масштабування системи, полегшує додавання нових алгоритмів і дозволяє налаштовувати поведінку залежно від типу запиту чи сценарію використання.

2.4.3 Формат запитів та відповідей

Для забезпечення сумісності між модулями системи та зовнішніми клієнтами, усі дані передаються у форматі JSON. Приклад запиту до маршруту /recommend може мати вигляд:

```
{
  "user_id": "user123",
  "top_k": 10
}
```

У відповідь система повертає масив об'єктів з відповідними ідентифікаторами та (за потреби) супутніми метаданими:

```
{
```

```
  "recommendations": [
    {"item_id": "item789", "score": 0.95},
    {"item_id": "item456", "score": 0.91}
  ]
}
```

Аналогічно, маршрут /feedback приймає масив об'єктів взаємодії, де кожен запис включає ідентифікатор користувача, об'єкта, тип дії та часову мітку.

```
[
  {"user_id": "user123", "item_id": "item456", "event": "view", "timestamp": "2025-06-05T08:15:00Z"},
  ...
]
```

Це забезпечує універсальність і дозволяє адаптувати систему до будь-якого типу подій, що можуть бути використані для навчання моделей у подальшому.

2.5 Реалізація збереження проміжних даних у фреймворці

У рамках реалізації системи було передбачено механізм збереження проміжних даних, необхідних для роботи алгоритмів рекомендацій, а також для підтримки відстеження історії взаємодій користувачів. Це дозволяє забезпечити повторюваність експериментів, проводити аналіз ефективності моделей та реалізовувати функції персоналізації.

Для зберігання даних використовується система керування базами даних PostgreSQL, розгорнута в окремому контейнері Docker. Як рушій збереження обрано SQLAlchemy, який дозволяє описувати структуру таблиць за допомогою ORM-моделей та виконувати операції читання/запису уніфіковано для всіх компонентів системи.

У фреймворку реалізовано збереження таких типів інформації:

- події взаємодії користувача з об’єктами (рейтинги, перегляди, кліки);
- профілі користувачів та метаінформація про об'єкти;
- статуси виконання тренування та внутрішні службові позначки (наприклад, оброблені сесії).

Дані подій зберігаються у таблиці feedback, де кожен запис містить ідентифікатор користувача, об'єкта, тип події та часову мітку. Це дає змогу накопичувати повну історію взаємодій для подальшого навчання моделей, аналізу чи перевалідації результатів.

Для забезпечення підтримки векторного пошуку, у базі активовано розширення pgvector, що дозволяє зберігати ознаки у форматі багатовимірних векторів. Це є основою для потенційної реалізації більш складних гібридних стратегій, зокрема семантичного зіставлення профілів та об'єктів.

Всі операції доступу до бази інкапсульовано в окремий модуль store.py, що реалізує низькорівневу логіку читання та запису. Завдяки цьому доступ до даних можна централізовано контролювати, не порушуючи цілісність архітектури.

Такий підхід до обробки проміжних даних забезпечує не лише ефективність зберігання, але й розширюваність системи, даючи змогу підключати додаткові модулі обробки або аналізу без необхідності зміни основного застосунку.

2.6 Алгоритм позиційного ранжування

У процесі об’єднання результатів від кількох незалежних рушіїв виникла необхідність у формуванні єдиного списку об’єктів, що враховує

рекомендації з різних джерел. Для цього реалізовано алгоритм позиційного ранжування, який забезпечує агрегування списків на основі порядку об’єктів у кожному з них без урахування абсолютних значень оцінок.

2.6.1 Принцип позиційного голосування

Позиційне голосування ґрунтується на тому, що кожен об’єкт у списку отримує кількість балів, обернено пропорційну до своєї позиції. Наприклад, об’єкт на першому місці отримує N балів, на другому — N–1 тощо. Якщо об’єкт відсутній у списку конкретного рушія, він не отримує жодного балу. Це дозволяє комбінувати результати від різних моделей, не покладаючись на їхні внутрішні метрики.

2.6.2 Застосування методу Борда

Для обчислення фінального рейтингу застосовано адаптований метод Борда. Кожен рушій формує власний топ-N список, на основі якого об’єкти отримують бали відповідно до своєї позиції. Сума балів з усіх джерел визначає фінальне положення об’єкта у зведеному списку. Такий підхід забезпечує узгоджене ранжування з урахуванням усіх джерел рекомендацій, що особливо важливо при використанні гібридної архітектури системи.

2.6.3 Приклад обчислення результатів

Припустимо, що два рушії сформували рекомендаційні списки з п’яти об’єктів:

- рушій A: `[item1, item2, item3, item4, item5]`
- рушій B: `[item3, item2, item6, item5, item1]`

Об'єктам призначаються бали від 5 до 1 залежно від позиції у кожному списку. У разі відсутності — 0 балів. Підсумковий рейтинг розраховується як сума балів з обох списків:

Об’єкт	**Бали від A**	**Бали від B**	**Підсумок**
item1	5	1	6
item2	4	4	8
item3	3	5	8
item4	2	0	2
item5	1	2	3
item6	0	3	3

Фінальний список формується за спаданням сумарного балу: `[item2, item3, item1, item5, item6, item4]`

Такий підхід дозволяє ефективно поєднувати результати від кількох моделей без потреби уніфікації їхніх внутрішніх форматів або переобчислення метрик, що суттєво спрощує інтеграцію нових рушіїв у систему.

2.7 Уніфікація даних для підтримки мультиалгоритмічного середовища

У мультиалгоритмічному середовищі, яке включає кілька рушіїв рекомендацій з різними вимогами до формату вхідних даних, ключову роль відіграє процес уніфікації. У межах цієї системи було реалізовано підхід, який дозволяє подати дані у стандартизованому вигляді, придатному як для

традиційних моделей колаборативної фільтрації, так і для алгоритмів на основі імпліцитних відгуків або контентних ознак.

На вхід системи надходять дані у вигляді трійок користувач–об’єкт–оцінка. Для забезпечення узгодженості інформації всі ідентифікатори користувачів та об’єктів нормалізуються до строкового типу, що виключає конфлікти між рушіями, які розрізняють числові та символьні ключі. Крім того, оцінки, що надходять у вигляді різних типів (цілі числа, дійсні значення, булеві мітки), приводяться до єдиного числового формату, узгодженого з вимогами обраного алгоритму.

Усі події взаємодії зберігаються у внутрішньому репозиторії уніфікованих записів, який реалізовано на рівні бази даних PostgreSQL. Зокрема, використовується спеціальна таблиця interactions, де для кожної події фіксується user_id, item_id, score, timestamp та тип дії (наприклад, перегляд, лайк, оцінка тощо). Така структура дозволяє не лише тренувати моделі, а й здійснювати фільтрацію даних для різних рушіїв на етапі формування запиту.

У випадку використання рушія Gorse дані із бази автоматично імпортуються через API або передаються у вигляді CSV-файлів, що генеруються скриптом manage.py. Для Implicit адаптер здійснює прямий вибір даних із таблиці, перетворюючи їх у матрицю взаємодій на основі scipy або pandas. Це дозволяє зберігати гнучкість і одночасно гарантувати однакове джерело даних для всіх рушіїв.

Уніфікація також охоплює підтримку семантичних атрибутів — таких як категорії товарів або демографічні характеристики користувачів — які можуть використовуватися в контентних або гібридних моделях. Для цього в системі передбачено додаткові таблиці items та users, що містять

структуровану інформацію про об’єкти та профілі, яка може бути використана рушіями, що підтримують фільтрацію за ознаками.

Таким чином, уніфікований підхід до представлення та зберігання даних дозволяє забезпечити ефективну роботу мультиалгоритмічного середовища, знижуючи ризики несумісності між компонентами та полегшуючи розширення системи новими типами моделей у майбутньому.

2.8 Використання відкритих датасетів

2.8.1 Вибір датасетів

Для оцінювання роботи реалізованої системи було обрано публічні датасети, які широко застосовуються у наукових публікаціях та конкурсах з рекомендаційних алгоритмів. Вибір таких джерел дозволяє як реплікувати результати інших дослідників, так і забезпечити об’єктивність під час порівняння якості модулів.

Першим з них є MovieLens 100K — класичний набір даних, який містить 100 000 рейтингових оцінок від 943 користувачів 1682 фільмам. Цей датасет має чітку структуру, відсутність пропусків, а також супровідні метадані, що дозволяє оцінити як алгоритми з явним навчанням, так і контентно-орієнтовані моделі. Крім того, його розмір оптимальний для тестування швидкості та стабільності системи під навантаженням у локальному середовищі.

Другим протестованим датасетом став Goodbooks-10K, що містить понад 6 мільйонів оцінок 10 тисяч книжок. Цей набір охоплює більш складну структуру переваг користувачів і дозволяє перевірити, як система справляється з більшими обсягами даних, нерівномірним розподілом активності та зміною домену з фільмів на книги. Також він є релевантним

для перевірки cold-start сценаріїв та поведінки різних рушіїв при високій кількості рідкісних об'єктів.

2.8.2 Методика проведення порівняння

Перед використанням кожен датасет було адаптовано до формату, сумісного з рушіями системи: кожен запис представлено у вигляді трійки користувач–об’єкт–оцінка. При цьому проводилася попередня нормалізація і фільтрація записів, зокрема видалення одиничних взаємодій, що не дозволяють ефективно навчатись.

У тестах було використано два основні способи взаємодії:

CLI-імпорт даних у рушії через manage.py, що дозволяє забезпечити попередню обробку та збереження у базу;

API-імпорт через маршрут /feedback, що моделює реальні сценарії постачання даних із клієнтської частини.

Після імпорту виконувалося тренування моделей із використанням Gorse або Surprise. Вихідні рекомендації формувалися через /recommend для підмножини тестових користувачів. Метрики якості включали:

Precision@K — частка релевантних об'єктів серед рекомендованих;

Recall@K — охоплення релевантних об’єктів;

Coverage — частка об’єктів, які з’явилися у хоча б одній рекомендації.

Для підвищення надійності оцінки було використано крос-валідацію за користувачами з декількома повторними вибірками, що дозволило мінімізувати вплив випадкових флуктуацій у поведінці алгоритмів.

2.8.3 Аналіз отриманих результатів

Результати експериментів свідчать про здатність реалізованої системи ефективно працювати з датасетами різного обсягу та домену. Зокрема, для MovieLens було отримано високі значення точності при використанні алгоритмів матричної факторизації з Surprise (Precision@5 ≈ 0.60, Recall@5 ≈ 0.42). Натомість Gorse забезпечив ширше охоплення рекомендацій (Coverage ≈ 0.91), завдяки вбудованим механізмам кластеризації та підтримці неявної взаємодії.

При роботі з Goodbooks-10K обидва рушії продемонстрували стабільну продуктивність. У випадку Surprise середнє значення RMSE склало близько 0.85, а MAE — 0.70. Точність рекомендацій була дещо нижчою, що пов’язано з високою розрідженістю та складністю структури вподобань користувачів у цьому наборі.

Таким чином, реалізована платформа довела свою здатність масштабуватись, працювати з різнорідними джерелами даних, а також комбінувати вихідні результати для покращення загального рівня рекомендацій. Це відкриває перспективи для подальшого впровадження методів гібридизації та персоналізованої агрегації рекомендацій.

Висновки

1. Результати та особливості дослідження

У межах реалізації системи було успішно побудовано багатомодульну архітектуру для рекомендаційних сервісів, що дозволяє порівнювати результати різних рушіїв за єдиною схемою інтеграції. Основною перевагою реалізованого підходу є адаптерна структура, яка забезпечує гнучкість у підключенні нових алгоритмів та спрощує тестування. У якості

рушіїв було використано Gorse та Surprise, кожен з яких демонструє власні переваги у різних сценаріях.

Система підтримує як REST API-взаємодію, так і CLI-інструменти для завантаження, тренування та отримання рекомендацій. Це створює основу для інтеграції з іншими модулями або клієнтськими застосунками. Використання PostgreSQL з підтримкою pgvector відкриває можливості для розширення системи у напрямку гібридних рекомендацій з векторним пошуком.

2. Можливості розвитку та удосконалення проєкту

У перспективі система може бути доповнена підтримкою нових рушіїв, зокрема на базі нейронних мереж або knowledge-graph. Також можливим є впровадження автоматичного вибору алгоритму залежно від характеристик вхідних даних. Важливим напрямом розвитку є розширення функціональності API, зокрема реалізація історичної статистики, кастомізації логіки агрегування рекомендацій та інструментів для A/B-тестування.

Система також дозволяє адаптуватися до нових доменів — від електронної комерції до освітніх платформ — шляхом заміни датасетів і специфікацій моделей. Подальше вдосконалення інтерфейсної частини дозволить зробити її зручнішою для аналітиків та дослідників.

3. Практичне застосування

Запропоноване рішення може бути використано для побудови внутрішніх систем персоналізованих рекомендацій у малих та середніх проєктах, де необхідна швидка інтеграція з мінімальними витратами. Платформа особливо корисна для дослідницьких цілей, оскільки дозволяє легко змінювати конфігурації та тестувати різні стратегії ранжування.

Також система може бути базою для проведення академічних експериментів з різними типами даних, адаптаціями алгоритмів, а також вивченням впливу гібридних стратегій. Відкритість архітектури сприяє подальшій кастомізації, розширенню сценаріїв використання та підвищенню якості рекомендацій у практичних умовах.

Список використаних джерел

1. Sarwar B., Karypis G., Konstan J., Riedl J.. Item-based collaborative filtering recommendation algorithms // *Proceedings of the 10th international conference on World Wide Web*. 2001. P. 285–295. URL: https://dl.acm.org/doi/10.1145/371920.372071.
2. Koren Y., Bell R., Volinsky C.. Matrix factorization techniques for recommender systems // *Computer*. Vol. 42, No. 8. 2009. P. 30–37. URL: https://doi.org/10.1109/MC.2009.263.
3. Rendle S.. Factorization machines // *IEEE ICDM*. 2010. P. 995–1000. URL: https://doi.org/10.1109/ICDM.2010.127.
4. Pazzani M., Billsus D.. Content-based recommendation systems // *The Adaptive Web*. 2007. P. 325–341. URL: https://doi.org/10.1007/978-3-540-72079-9_10.
5. Lops P., De Gemmis M., Semeraro G.. Content-based recommender systems: State of the art and trends // *Recommender Systems Handbook*. 2011. P. 73–105. URL: https://doi.org/10.1007/978-1-4899-7637-6_3.
6. Van den Oord A., Dieleman S., Schrauwen B.. Deep content-based music recommendation // *Advances in Neural Information Processing Systems*. 2013. P. 2643–2651. URL: https://proceedings.neurips.cc/paper_files/paper/2013/file/4d5bcbdf456d5f2b2ca2b56c5c5d7cfa-Paper.pdf.
7. Burke R.. Hybrid web recommender systems // *The Adaptive Web*. 2007. P. 377–408. URL: https://doi.org/10.1007/978-3-540-72079-9_12.
8. Bell R., Koren Y., Volinsky C.. Modeling relationships at multiple scales to improve accuracy of large recommender systems // *Proceedings of the 13th ACM SIGKDD*. 2007. P. 95–104. URL: https://doi.org/10.1145/1281192.1281210.
9. He X., Liao L., Zhang H., Nie L., Hu X., Chua T.S.. Neural collaborative filtering // *Proceedings of the 26th International Conference on World Wide Web*. 2017. P. 173–182. URL: https://doi.org/10.1145/3038912.3052569.
10. Jackson P.. Introduction to expert systems. Addison Wesley. 1998. URL: https://archive.org/details/introductiontoex0000jack.
11. Zhang Y., Chen X.. Explainable recommendation: A survey and new perspectives // *Foundations and Trends® in Information Retrieval*. Vol. 14, No. 1. 2020. P. 1–101. URL: https://doi.org/10.1561/1500000066.

12. Chen M., Beutel A., Covington P., Jain S., Belletti F., Chi E.H.. Top-K off-policy correction for a REINFORCE recommender system // *RecSys '19*. 2019. P. 1–9. URL: https://dl.acm.org/doi/10.1145/3298689.3346996.

13. Sun F., Liu J., Wu J., Pei C., Lin X., Ou W., Jiang P.. BERT4Rec: Sequential recommendation with bidirectional encoder representations from transformer // *CIKM*. 2019. P. 1441–1450. URL: https://doi.org/10.1145/3357384.3357895.

14. Perny P., Zucker J.-D.. Preference-based search and multi-criteria decision making // *Encyclopedia of Artificial Intelligence*. 2009. URL: https://www.igi-global.com/gateway/chapter/10379.

15. Di Noia T., Ostuni V.C., Rosati J., Tomeo P., Sciascio E.D.. An analysis of the semantic gap in content-based recommender systems // *User Modeling and User-Adapted Interaction*. Vol. 26, No. 3. 2016. P. 219–257. URL: https://doi.org/10.1007/s11257-016-9170-9.

16. Schroeder M., Beel J.. Open Recommendation Specification (ORS): Towards a unified recommendation standard // *Workshop on Recommendation in Multistakeholder Environments. RecSys*. 2021. URL: https://arxiv.org/abs/2110.06800.

17. Hug N.. Surprise: A Python scikit for building and analyzing recommender systems // *JMLR*. 2020. URL: https://surpriselib.com.

18. Kula M.. Metadata embeddings for user and item cold-start recommendations // *RecSys '15*. 2015. P. 14–21. URL: https://doi.org/10.1145/2792838.2809695.

19. Zhao W.X., Zhang S., He X., et al.. RecBole: Towards a unified, comprehensive and efficient framework for recommendation algorithms // *CIKM*. 2021. P. 4653–4664. URL: https://doi.org/10.1145/3459637.3482016.

20. Zheng C.. Gorse: An open-source recommender system engine based on collaborative and content-based filtering // *GitHub*. URL: https://github.com/zhenghaoz/gorse.

Додатки

Додаток А

Містить реалізацію HTTP-маршрутів /train, /feedback, /recommend та керування рушіями рекомендацій.

```
from fastapi import FastAPI, Query
import os, pandas as pd
from adapters import registry
import store, uvicorn
from typing import List

app=FastAPI()
models={}

def ensure_model(name:str):
    if name not in models:
        df=store.get_frame()
```

```
        m=registry[name]()
        m.fit(df)
        models[name]=m

@app.post("/train")
def train(engine:str=Query(...)):
    ensure_model(engine)
    return {"trained":engine}

@app.post("/feedback")
def feedback(events: List[dict]):
    df=pd.DataFrame(events)
    if 'ts' not in df: df['ts']=pd.Timestamp.utcnow()
    store.insert_interactions(df)
    return {"inserted":len(df)}

@app.patch("/item/{item_id}")
def patch_item(item_id:int, body:dict):
    store.upsert("items", item_id, body.get("attrs",{}))
    return {"status":"ok"}

@app.patch("/user/{uid}")
def patch_user(uid:int, body:dict):
    store.upsert("users", uid, body.get("attrs",{}))
    return {"status":"ok"}

@app.get("/recommend")
def recommend(user_id:int, k:int=10, engines:str="implicit"):
    names=engines.split(",") if engines!="all" else list(models)
    rank={}
    for n in names:
        ensure_model(n)
        for pos,(iid,score) in enumerate(models[n].recommend(user_id,k*2)):
            rank[iid]=rank.get(iid,0)+(k*2-pos)
    top=sorted(rank.items(), key=lambda t:t[1], reverse=True)[:k]
    return [{"item_id":iid,"score":s} for iid,s in top]

if __name__=="__main__":
    uvicorn.run("main:app",host="0.0.0.0",port=8000)
```

Додаток Б

Забезпечує створення таблиць, додавання та оновлення даних, вибірку інтеракцій для навчання моделей.

```
import os, json
from typing import Any
from sqlalchemy import create_engine, text
import pandas as pd

engine = create_engine(os.getenv("DATABASE_URL"))

DDL = '''
CREATE TABLE IF NOT EXISTS users(
  id BIGINT PRIMARY KEY,
  attrs JSONB DEFAULT '{}'::jsonb
```

```
);
CREATE TABLE IF NOT EXISTS items(
  id BIGINT PRIMARY KEY,
  attrs JSONB DEFAULT '{}'::jsonb
);
CREATE TABLE IF NOT EXISTS interactions(
  id SERIAL PRIMARY KEY,
  user_id BIGINT REFERENCES users(id),
  item_id BIGINT REFERENCES items(id),
  rating FLOAT,
  ts TIMESTAMPTZ DEFAULT now()
);
'''
with engine.begin() as con:
    con.execute(text(DDL))

def upsert(table:str, id_:int, attrs:dict[str,Any]):
    q=text(f"""
        INSERT INTO {table}(id, attrs) VALUES (:id, :attrs)
        ON CONFLICT (id) DO UPDATE SET attrs = {table}.attrs ||
EXCLUDED.attrs;
    """)
    with engine.begin() as con:
        con.execute(q, {"id":id_, "attrs":json.dumps(attrs)})

def insert_interactions(df:pd.DataFrame):
    df.to_sql("interactions", engine, if_exists="append", index=False,
method="multi")

def get_frame():
    q="""
      SELECT i.user_id, i.item_id,
             COALESCE(i.rating,1) rating,
             i.ts,
             u.attrs AS uattrs, it.attrs AS iattrs
      FROM interactions i
      JOIN users u ON u.id=i.user_id
      JOIN items it ON it.id=i.item_id;
    """
    return pd.read_sql(q, engine)
```

Додаток В

Скрипт командного рядка, що завантажує користувачів, товари й взаємодії на основі конфігурації mapping.yml.

```
import argparse, pathlib, yaml, pandas as pd, store, datetime as dt

def load_dataset(folder):
    folder=pathlib.Path(folder)
    cfg=yaml.safe_load(open(folder/'mapping.yml'))

    # users
    if 'users' in cfg:
        dfu=pd.read_csv(folder/cfg['users']['file'],
```

```
sep=cfg['users'].get('sep',','))
        key=cfg['users']['key']
        for rec in dfu.to_dict(orient='records'):
            store.upsert('users', int(rec.pop(key)), rec)

    # items
    if 'items' in cfg:
        dfi=pd.read_csv(folder/cfg['items']['file'],
sep=cfg['items'].get('sep',','))
        key=cfg['items']['key']
        for rec in dfi.to_dict(orient='records'):
            store.upsert('items', int(rec.pop(key)), rec)

    # interactions
    icfg=cfg['interactions']
    dfi=pd.read_csv(folder/icfg['file'], sep=icfg.get('sep',','))
    dfi.rename(columns={icfg['user_col']:'user_id',
icfg['item_col']:'item_id'}, inplace=True)
    if icfg.get('rating_col'):
        dfi['rating']=dfi[icfg['rating_col']]
    else:
        dfi['rating']=1
    if 'ts_col' in icfg:
        dfi['ts']=pd.to_datetime(dfi[icfg['ts_col']])
    else:
        dfi['ts']=dt.datetime.utcnow()
    store.insert_interactions(dfi[['user_id','item_id','rating','ts']])
    print("Loaded", len(dfi), "interactions")

if __name__=="__main__":
    ap=argparse.ArgumentParser()
    ap.add_argument("dataset")
    args=ap.parse_args()
    load_dataset(args.dataset)
```

Додаток Г

Здійснює імпорт користувачів, товарів та фідбеку в Gorse через HTTP-запити, запускає тренування моделі.

```
from adapters import register
from adapters.base import BaseAdapter
import os, requests, pandas as pd

@register("gorse")
class GorseAdapter(BaseAdapter):
    def __init__(self):
        self.host=os.getenv("GORSE_ADDR","http://gorse:8087").rstrip("/")
    def fit(self, df:pd.DataFrame, **_):
        users=[{"UserId":int(u)} for u in df.user_id.unique()]
        requests.post(f"{self.host}/api/data/import/users", json=users)
        items=[{"ItemId":int(i)} for i in df.item_id.unique()]
        requests.post(f"{self.host}/api/data/import/items", json=items)
```

```
fb=[{"FeedbackType":"star","UserId":int(r.user_id),"ItemId":int(r.item_id),
"Value":r.rating} for r in df.itertuples()]
        requests.post(f"{self.host}/api/data/import/feedback", json=fb)
        requests.post(f"{self.host}/api/algorithm/fit")
    def recommend(self,user_id:int,k:int=10,**_):
        r=requests.get(f"{self.host}/api/recommend/{user_id}?n={k}")
        if r.ok:
            return [(int(x['ItemId']), float(x.get('Score',1))) for x in
r.json()]
        return []
```

Додаток Д

Будує матрицю взаємодій, тренує модель ALS і генерує рекомендації на основі латентних факторів.

```
from adapters import register
from adapters.base import BaseAdapter
import pandas as pd, scipy.sparse as sps, implicit

@register("implicit")
class ImplicitALS(BaseAdapter):
    def __init__(self):
        self.model=None
        self.uid={}, {}
    def fit(self, df:pd.DataFrame, **_):
        ucat=df.user_id.astype("category")
        icat=df.item_id.astype("category")
        self.uid_map=dict(enumerate(ucat.cat.categories))
        self.iid_map=dict(enumerate(icat.cat.categories))
        mat=sps.coo_matrix((df.rating,(ucat.cat.codes,icat.cat.codes)))

self.model=implicit.als.AlternatingLeastSquares(factors=64,iterations=10)
        self.model.fit(mat)
    def recommend(self,user_id:int,k:int=10,**_):
        if self.model is None or user_id not in self.uid_map.values():
return []
        rev={v:k for k,v in self.uid_map.items()}
        rec,score=self.model.recommend(rev[user_id], self.model.user_items,
N=k)
        return [(self.iid_map[i], float(s)) for i,s in zip(rec,score)]
```