КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра системного аналізу та теорії прийняття рішень

Кваліфікаційна робота
на здобуття ступеня бакалавра

за спеціальністю 124 Системний аналіз

на тему:

"РЕКОМЕНДАЦІЙНІ СИСТЕМИ В ОНЛАЙН-ТОРГІВЛІ"

Виконав студент 4-го курсу
Арзамасцев Владислав Олександрович

(підпис)

Науковий керівник:
асистент, кандидат технічних наук
Махно Михайло Федорович

(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри системного аналізу та теорії прийняття рішень
«01» червня 2023 р.,
протокол № 13
Завідувач кафедри
О. Г. Наконечний

(підпис)

Київ – 2023

ЗМІСТ

ВСТУП

В епоху тотальної цифровізації неможливо уявити бізнес, який би не надавав послуги онлайн. Кожен із нас щодня використовує онлайн-сервіси для пошуку інформації, стрімінгу контенту чи шоппінгу. Дуже часто кількість товарів і послуг, які надають компанії, є просто величезною. Аби не втратити клієнтів, компанії мають динамічно підлаштовуватись під кожного споживача, пропонуючи йому лише найбільш релевантні товари – саме цю проблему розв'язують рекомендаційні системи.

Актуальність обраної теми полягає в тому, що кількість користувачів онлайн-сервісів кожного дня зростає, і необхідно постійно оптимізовувати обслуговування кожного клієнта. Крім того, із послабленням карантинних обмежень у розвинених країнах, з'явилась необхідність утримати клієнтів на стрімінгових та розважальних сервісах.

Аналіз останніх досліджень і публікацій засвідчує, що на сьогоднішній день рекомендаційні системи залишаються найпотужнішим інструментом для збільшення зацікавленості користувачів інтернет-ресурсів, і є рушійною силою сучасної онлайн-торгівлі. Моделі, що використовують корпорації-гіганти, відмінно справляються з задачею утримання клієнтів, проте є надто складними для малих і середніх бізнесів – як з точки зору обслуговування, так і за об'ємом розрахунків. Задача побудови простої моделі, яка давала б точні і в той же час різноманітні рекомендації, залишається актуальною.

Метою дипломної роботи є дослідження різних моделей, які використовують у рекомендаційних системах, висвітлення проблем, які постають перед інженерами при їх проектуванні та імплементації, а також моделювання і побудова рекомендаційної системи для онлайн-магазину книжок на основі даних про інтеракцію покупців із товарами.

Для досягнення цієї мети поставлені такі ***завдання***:

а) дослідити і порівняти найпопулярніші моделі рекомендаційних систем:
   1) Collaborative Filtering;
   2) Content-Based Filtering;
   3) Knowledge-Based рекомендаційні системи;

б) визначити, чи можливо зробити передбачення рейтингів більш точним за допомогою комбінування тих чи інших моделей;

в) описати проблеми, які виникають при тренуванні та використанні рекомендаційних систем, навести способи розв'язання цих проблем;

г) навести алгоритми, які використовуються у різних моделях рекомендаційних систем;

д) побудувати модель рекомендаційної системи для книжкового інтернет-магазину;

е) навести програмну реалізацію для побудованої моделі;

є) оцінити точність результату та викласти міркування щодо можливого збільшення точності передбачень.

Методи дослідження: теорії, наукове моделювання, наукове дослідження, спостереження, вимірювання, дедукції та індукції.

Результати дослідження можуть бути використані при підборі, моделюванні та реалізації рекомендаційної системи для малого бізнесу, а також для аналізу та покращення існуючих рішень.

Особистим внеском є проведення аналізу алгоритмів, що розв'язують дану проблему, а також побудова моделі та імплементація рекомендаційної системи, яка була б найбільш підходящою для даної задачі.

Структура дипломної роботи відповідає меті та завданням дослідження і складається зі вступу, 3 розділів, 10 підрозділів, висновків, списку використаних джерел, додатків.

1 ОСНОВНІ МОДЕЛІ РЕКОМЕНДАЦІЙНИХ СИСТЕМ

1.1 Історія розвитку рекомендаційних систем

Рекомендаційні Системи (Recommender Systems, RSs) – підклас систем для фільтрування інформації, що полегшують прийняття рішень для користувача, звужуючи множину усіх опцій і пріоритезуючи ці опції в поточному контексті. Пріоритезація будується на основі даних, зібраних з користувача як індивідуума, з групи користувачів, з навколишнього середовища та даних про самі опції. Рекомендаційні системи – найпотужніший інструмент, який використовують великі бізнеси для зацікавлення та утримання користувачів, збільшення конверсії, суми середнього чека та покращення користувацього досвіду.

Ідея використовувати комп'ютер для моделювання поведінки людей – не нова. Прототип рекомендаціної системи з'явився у 1979 році, створений Елейн Річ у США. У рамках дослідження побудови моделі індивідуума на основі якоїсь малої кількості інформації про нього, було створено систему Grundy. Система повинна була задавати користувачеві серію питань та будувати стереотипи, на основі яких юзеру пропонувалась книга, що мала б йому сподобатись [1]. На той час, це була революційна ідея, оскільки це був перший крок до персоналізації автоматизованих сервісів.

Перша ефективна рекомендаційна система, що реалізовувала ідею розбиття користувачів на групи, називалась GroupLens, 1994 рік. GroupLens збирала рейтинги новин у мережі Usenet і використовувала їх для передбачення того, наскільки читачі були б зацікавлені у статті до того, як вони її відкриють. Ця система була однією з перших, у якій алгоритми передбачення було повністю автоматизовано. Прогноз будувався на основі паттернів у історії рейтингів.

На початку 90-х з'явилась нова ідея – фільтрація на основі контенту (Item-Based підхід). Дослідження Music Genome Project (1999 рік) показало, що у музичного треку є 450 генів, які описують якісь конкретні його властивості і є пов'язаними між собою. Було створено алгоритм, який визначав, наскільки дві пісні є схожими одна на одну, порівнюючи їхні "геноми". Нові пісні, подібні до тих, що користувач уже прослухав, з'являлись у його рекомендаціях. Такий підхід зарекомендував себе як більш ефективний за умови наявності малої кількості даних про споживачів, ніж підхід GroupLens. На сьогоднішній день, цей алгоритм є основою популярного (250 тис. користувачів) інтернет-сервісу Pandora Radio.

Уперше ці два підходи було скомбіновано у 1994 році студентами Стенфорду у рекомендаційній системі Fab. Метою системи було полегшити пошук статей і новин у Інтернеті. Творці Fab вирішили спочатку об'єднати статті за загальними темами (такими як "фінанси" чи "розваги"), після чого для кожної категорії проводилось ранжування – "наскільки цій конкретній людині цікава конкретна стаття". Такий підхід дозволив зменшити кількість фідбеків юзерів, які необхідні системі для функціонування та збагатив результуючу вибірку новинами, які відносяться до релевантних тем та які сподобались найбільше іншим юзерам.

Сучасні великі корпорації інвестують багато людських і фінансових ресурсів у створення алгоритмів, які підбирали б найбільш підходящий контент для користувачів. Їхні пропрієтарні алгоритми вирізняються як ефективністю, так і складністю. Наприклад, у 2006 році стрімінгова компанія Netflix запустила конкурс із призовим фондом в 1 мільйон доларів на створення алгоритму, який видавав би кращі рекомендації, ніж їх власний – CineMatch. Приз дістався у 2009 році команді, яка презентувала комплексне рішення зі 107 алгоритмів і покращила значення середньоквадратичного відхилення із 0.9525 до 0.8722. Система-призер лягла в основу поточної рекомендаційної системи Netflix, яка для рекомендацій враховує історію переглядів одного користувача і групи користувачів, поточний сезон, свята,

навігацію по веб-сайту, вигляд постерів, рейтинги та багато іншого. Згідно з Аіш Фентоном, 80% контенту, який споживають клієнти, було запропоновано їхнім пропрієтарним алгоритмом [2]. За даними Macrotrends, у 2022 році компанія заробила 31.616 мільярдв доларів США, і переважну частину прибутку було отримано якраз завдяки персоналізації контенту.

Іншою компанією-гігантом, яка залежить від рекомендаційних систем, є Amazon. Згідно зі статтею McKinsey & Company "How retailers can keep up with consumers", Amazon отримує 35% відсотків покупок за допомогою свого пошукового алгоритму A10. Навідміну від алгоритмів у пошукових системах типу Google чи DuckDuckGo, A10 під час пошуку враховує не лише ключові слова, а й історію продажів, коефіцієнт конверсії та фідбек покупців.

Отже, рекомендаціні системи – важлива частина великого бізнесу та онлайн-торгівлі. В залежності від домену корпорації, рекомендаційні системи можуть як складатись із одного простого алгоритму, так і бути комплексними, багатошаровими системами. Персоналізація у сфері послуг дозволяє корпораціям більш ефективно продавати свій товар, тому компанії зацікавлені у дослідженнях алгоритмів, що моделюють поведінку клієнтів, полегшують прийняття рішень та відбраковують контент чи послуги, які не приносять прибутку.

1.2 Collaborative Filtering

Collaborative Filtering – метод передбачення рейтингу товару, що базується на даних про взаємодію інших користувачів із товарами. Основна ідея цього методу полягає в тому, щоб розбити множину користувачів на групи по їхнім уподобанням, і використати уже існуючі рейтинги групи юзерів, щоб обчислити рейтинг іншого юзера з цієї групи. Головне припущення, яке ми робимо при використанні Collaborative Filtering моделей – "Якщо користувачі дали якусь оцінку в минулому, то

вони дадуть таку саму оцінку і в майбутньому" (history repeats itself). Алгоритми, що реалізують Collaborative Filtering, потребують великої кількості даних по доменній області. Це можуть бути як **явні** дані – наприклад, рейтинг товару від 1 до 5 чи кількість позитивних відгуків, так і **неявні** дані – наприклад, скільки секунд користувач провів за переглядом товару, чи додав товар у корзину, чи залишив коментар і тд.

Ідею Collaborative Filtering можна реалізувати різними способами. Існує 2 концептуально різних сімейства алгоритмів, які належать до Collaborative Filtering: Memory-Based і Model-Based.

Memory-Based, також відомі як Neighbourhood-Based алгоритми, належать до найбільш ранніх відкриттів у Collaborative Filtering. Алгоритми, що належать до цього сімейства зазвичай дуже прості в реалізації та ґрунтуються на простих ідеях:

а) User-Based CF. Нехай у нас є **цільовий користувач** – тобто користувач, для якого підбираються рекомендації. Нехай також у нас є n користувачів, m товарів і матриця R_{n*m} рейтингів – елемент $r_{i,j}$ представляє оцінку юзера і товару j. Задача – визначити значення елемента $r_{i,j}$, якщо цільовий користувач і ще не проглянув (чи не оцінив) товар j. Для цього ми вибираємо N юзерів, подібних до цільового і рахуємо зважене середнє їхніх рейтингів, використовуючи коефіцієнт подібності цільового юзера і $s_{i,k}$ до юзера k як вагу:

$$r_{i,j} = \frac{\sum_k s_{i,k} * r_{k,j}}{|\sum_k s_{i,k}|}. \quad (1)$$

Недоліком такого підходу є те, що користувачі можуть бути **упередженими** – тобто хтось частіше високо оцінює контент, а хтось є більш строгим, і проставляє оцінки менші за середні. Для того, щоб більш точно передбачити

$r_{i,j}$, ми маємо позбутись цього упередження. Наприклад, порахуємо середній бал усіх рейтингів для кожного з юзерів і позначимо його $\bar{r}$. $\bar{r_i}, \bar{r_k}$ - середні рейтинги цільового та k-го з подібних юзерів відповідно. Тоді шукаємо неспотворене упередженнями $r_{i,j}$:

$$r_{i,j} = \bar{r_i} + \frac{\sum_k s_{i,k} * (r_{k,j} - \bar{r_k})}{|\sum_k s_{i,k}|}. \quad (2)$$

Для пошуку коефіцієнта подібності $s_{i,k}$ найчастіше використовують 3 підходи:

1) рахують евклілову відстань між u_i, u_k (найпростіший і найгірший спосіб):

$$s_{i,k} = \sqrt{\sum_{f=1}^{m} (r_{i,f} - r_{k,f})^2}; \quad (3)$$

2) рахують косинус кута між u_i, u_k - чим косинус ближчий до 1, тим подібніші користувачі; якщо косинус прямує до нуля, то кореляція відсутня; якщо косинус прямує до -1, маємо анти-кореляцію:

$$s_{i,k} = \frac{r_i * r_k}{|r_i| * |r_k|} = \frac{\sum_{f=1}^{m} r_{i,f} * r_{k,f}}{\sqrt{\sum_{f=1}^{m} r_{i,f}^2 * \sum_{f=1}^{m} r_{k,f}^2}}; \quad (4)$$

Але так порахований коефіцієнт буде спотворений упередженістю. Для розв'язання цієї проблеми, косинус "центрують" (цю величину також називають коефіцієнтом Пірсона);

3) рахують коефіцієнт Пірсона (центрований косинус), та сама інтерпретація, що і для косинуса:

$$s_{i,k} = \frac{\sum_{f=1}^{m}(r_{i,f} - \bar{r_i}) * (r_{k,f} - \bar{r_k})}{\sqrt{\sum_{f=1}^{m}(r_{i,f} - \bar{r_i})^2 * \sum_{f=1}^{m}(r_{k,f} - \bar{r_k})^2}}; \quad (5)$$

б) Item-Based Collaborative Filtering. Математично дуже схожа на User-Based CF, але відрізняється концептуально. При реалізації Item-Based підходу, ми теж намагаємось передбачити рейтинг неоціненого товару j юзером i. Для цього ми вибираємо підмножину товарів S, подібних до товару j, $|S| = N$. Вважатимемо 2 товари подібними, якщо вони отримали схожу оцінку від одного і того ж користувача. Тоді для передбачення рейтингу товару рахуватимемо зважене середнє рейтингів подібних товарів, які оцінив цільовий користувач. [3]

Item-Based методи часто пропонують більш релевантні рекомендації у порівнянні із User-Based методами, тому що у процесі передбачення рейтингу ми більше використовуємо вже проставлені рейтинги цільового юзера.

Ще однією перевагою Item-Based методів є їхня стабільність. Як правило, у компанії набагато більше клієнтів, ніж товарів. Тому 2 юзера швидше за все матимуть невелику кількість спільних товарів, які вони оцінили, зате 2 товари швидше за все матимуть багато користувачів, які оцінили їх обидва. Це робить Item-Based фільтрацію більш стійкою до шилінгових атак (shilling attacks), коли група атакуючих "отруює" рекомендаційну систему великою кількістю упереджених рейтингів, щоб змусити рекомендаційну систему пропонувати вигідний їм товар.

Проте у Item-Based методів є і недоліки, і найбільший із них – одноманітність рекомендацій. User-Based алгоритми пропонують набагато менш очевидні товари, тому що у процесі підбору рекомендації ми використовуємо дані інших людей з різними смаками. Тому User-Based алгоритми довше утримують користувача на інтернет-ресурсі.

Model-Based методи, у яких узагальнена модель даних створюється наперед з використанням методів машинного навчання і видобутку даних (data-mining). Таким чином, ми можемо розділити процес тренування (або побудови моделі) та власне передбачення. Існує багато Model-Based Collaborative Filtering алгоритмів, наприклад, Bayes Networks, Decision trees, Singular Value Decomposition, Latent Dirichlet Allocation.

Model-Based системи найчастіше перетворюють вихідну матрицю фідбеків для зменшення розмірності задачі. Latent Factor моделі, такі як Singular Value Decomposition, Principal Component Analysis чи Аутоенкодери, стискають матрицю фідбеків до матриць меншої розмірності, що містять латентні фактори. **Латентними факторами** називатимемо фактори, які можна тільки вивести із математичної моделі, побудованої на інших, явних факторах, які можна побачити чи виміряти. Наприклад, нехай юзер високо оцінив фільми "Матриця", "Інтерстеллар" і "Ефект Метелика". З такого набору фільмів ми можемо зробити висновок, що користувачу цікаво переглядати фантастику, і йому було б доцільно порекомендувати інші фільми такого ж жанру. У нашому прикладі, висока оцінка даних фільмів – це явні фактори, а уподобання жанру "фантастика" – латентний фактор, який ми вивели штучно. По суті, матрична факторизація дозволяє нам вивести множину латентних факторів, а також отримати інформацію, яким чином конкретний користувач і конкретний товар пов'язані з цими факторами. Це дає велику перевагу над Neighbourhood-Based алгоритмами – якщо 2 користувачі не оцінили жодного спільного товару, ми все одно можемо знайти зв'язок між ними завдяки латентним факторам (їхнім смакам).

Взагалі кажучи, Model-Based рекомендаційні системи часто мають ряд переваг над Memory-Based системами:

– User-Based та Item-Based алгоритми мають O(n^2) і O(m^2) просторову складність. На відміну від них, розмір навченої моделі зазвичай набагато

менший, ніж розмір початкової матриці рейтингів, отже, потрібно менше ресурсів на комп'ютері;

– вони є набагато більш стійкими до overfitting. Overfitting – проблема, що виникає у машинному навчанні, коли на результат дуже сильно впливають шуми в даних. Це стається, коли алгоритм дуже складний чи гнучкий, і вловлює шуми, замість того, щоб вловлювати паттерн в даних. Overfitted модель покаже відмінний результат на тренувальному датасеті і не спрацює на нових, небачених даних. Завдяки узагальненому підходу Model-Based алгоритмів, ми часто можемо уникнути overfitting. Більше того, такі моделі можна зробити цілком стійкими за допомогою регуляризації;
– Model-Based методи є швидшими як у передбаченні. Завдяки попередній обробці даних, результуюча модель є більш компактною і узагальненою, тому потрібно проводити менше обчислень;
– Memory Based алгоритми не можуть видобувати латентні фактори, а тому для товари із малою кількостю рейтингів ризикують не потрапити у рекомендації. З цієї ж причини, рекомендації для новоприбулих користувачів будуть менш точними.

Отже, незважаючи на те, що Memory-Based алгоритми є більш простими, вони за своєю природою евристичні, і не завжди дають хороший результат, особливо для великих матриць фідбеків. Для задач великої розмірності було б доцільно комбінувати їх із Model-Based методами. Наприклад, для передбачення рейтингу користувача можна використати зважене середнє рейтингів, отриманих із Memory-Based і Model-Based моделей. Значення ваг можна отримати за допомогою методів крос-валідації.

Варто зазначити, що частина Neighbourhood-Based методів може бути формально представленою у вигляді регресійних моделей, тож не можна сказати, що

Model-Based підхід є чимось окремим і принципово новим. Моделі, що враховують латентні фактори, набули своєї популярності після завершення конкурсу Netflix Prize, хоч вони і були запропоновані набагато раніше в контексті неповних датасетів.

1.3 Content-Based Filtering

Навідміну від Collaborative Filtering методів, які базуються на кореляції між рейтингами груп користувачів, Content-Based Filtering методи базуються на історії рейтингів самого цільового користувача, а також на властивостях товарів, які сподобались йому найбільше. Таким чином, рейтинги інших користувачів мають незначний вплив на рекомендації, якщо взагалі хоч якось на них впливають.

Крім концептуально іншого підходу до надання рекомендацій, Content-Based системи використовують інші джерела даних. Замість матриці фідбеків, Content-Based рекомендаційні системи зазвичай використовують аналіз опису контенту та аналіз профіля користувача для передбачення рейтингів:

– ***аналіз опису контенту.*** Із опису товару, який надає виробник, вибираються атрибути, які пояснюють призначення та властивості товару. Наприклад, часто використовується текстовий опис товарів. Для порівняння товарів можна використати Term Frequency – Inverse Document Frequency статистику. Для цього визначаємо величини

$$TF = \frac{(Frequencyoccurrenceof T in D)}{(Numberoftermsin D)}. \quad (6)$$

$$IDF = lg\left(\frac{(Amountofdocuments)}{(Numberofdocumentsthatcontaintermt)}\right). \quad (7)$$

$$TF\text{-}IDF = Tf * IDF. \quad (8)$$

І для кожного товару складаємо вектори, компонентами яких є TF-IDF величина для кожного ключового слова в описі (або тегу). Далі визначаємо подібні товари, використовуючи одну із формул (3), (4) або (5). [4]

– ***аналіз профіля користувача***, який згенеровано на основі фідбеку про переглянуті товари. Ідея полягає в тому, щоб асоціювати цільового користувача і ключові атрибути товарів, які йому сподобались, і на основі цих асоціацій рекомендувати товари, у яких ключові атрибути є схожими на вже уподобані. Подібність товарів за атрибутами також можна визначати за формулами (3), (4) або (5). Базовим прикладом побудови профілю є використання набору розмічених тренувальних документів із описом товарів, рейтинги юзерів як мітки, і класифікаційна модель, яка пов'язує атрибути товарів із рейтингами. Побудова профіля користувача насправді може бути різною для різних задач: в одному випадку краще використовувати явний фідбек (оцінки переглянутим товарам), в іншому – неявний (дії на сторінці, кількість секунд перегляду товару). Також можна запросити у користувача створити самим свій профіль на основі ключових слів, свого віку, статі, інтересів, сфери діяльності – цей підхід має дещо спільне із Knowledge-Based рекомендаційними системами. [5]

Таким чином, Content-Based рекомендаційні системи мають декілька переваг над Collaborative Filtering системами, оскільки вони не потребують рейтингів інших користувачів. Завдяки орієнтації на атрибути товарів, Content-Based алгоритми менш вразливі до Cold Start Problem – за умови наявності гарного опису товару. Це дозволяє легко вводити в оборот нові товари і не перейматись тим, що вони не з'являться в рекомендаціях. Отже, цей вид рекомендаційних систем краще підходять для бізнесів, які часто вводять та виводять з обігу нові товари, а також для ситуацій,

коли в наявності є велика кількість інформації про товар. Крім того, Content-Based фільтрація дозволяє створити більш прозору систему рекомендацій: користувач отримує товар, подібний до тих, що йому сподобались у минулому. Цей вид систем також легше масштабується, тому він більше підходить для систем із дуже великою кількістю користувачів. Також Content-Based алгоритми автоматично підлаштовуються під повсякденні потреби конкретного покупця, що робить можливим рекомендацію нішевих товарів, у яких мало хто зацікавлений.

На жаль, у цього виду рекомендаційних систем є і недоліки. Так само, як і з Collaborative Filtering, передбачити рейтинги товарів для нового користувача може бути складно, оскільки його профіль іще не побудовано. Масштаб проблеми можна зменшити, якщо попросити користувачів надати деталі про свої вподобання при реєстрації, і потім для кожної категорії повертати найпопулярніші товари. Після цього можна будувати профіль за неявним фідбеком. Ще одним недоліком Content-Based систем є одноманітність рекомендацій. Оскільки ми не враховуємо історію інтеракцій інших покупців із товарами, процес урізноманітнення рекомендацій не є тривіальним. Тому цей вид систем погано впорається із завданням утримувати користувача на ресурсі якомога довше. Останнім недоліком
Content-Based алгоритмів є те, що вони потребують більшої кількості даних. Для генерації якісних рекомендацій треба мати якомога більш детальні описи товарів і дані про поведінку кожного користувача платформи. Натомість, Collaborative Filtering алгоритми потребують лише даних про взаємодію між юзерами і товарами і потребує лише поверхневих описів.

1.4 Knowledge-Based рекомендаційні системи

Ми розглянули 2 види рекомендаційних систем: Collaborative Filtering та Content-Based системи. Функціонування обох видів та адекватність рекомендацій

залежить від уже наявного фідбеку – від матриці рейтингів та рейтингів цільового юзера відповідно. Неважко зрозуміти, що за умов нестачі інформації про фідбек Collaborative Filtering алгоритми не зможуть адекватно виділяти групи користувачів, а Content-Based методи не зможуть побудувати профіль цільового юзера. Ця проблема називається "проблемою холодного старту" або Cold Start Problem. Collaborative Filtering системи найбільш вразливі до цієї проблеми, вони дають неточні рекомендації для нових юзерів і нових товарів. Content-Based алгоритми більш стійкі до холодного старту у сенсі додавання нових товарів, але з новими користувачами вони справляються не дуже добре.

Однак проблема холодного старту може виникнути не тільки "на старті", вона може бути постійною пробемою в залежності від домену бізнесу. Так, є товари, які купують рідко або які дуже специфічні для конкретного покупця – наприклад, предмети розкоші, автомобілі, житло, дорога електроніка, нішеві товари. Побудувати рекомендації описаними вище підходами неможливо, оскільки:

- товари купують рідко, тому ні скласти профіль юзера, ні побудувати матрицю рейтингів не вдастся – товари вже "застаріють" і не будуть цікаві користувачеві;
- товари складно описати у feature термінах. Наприклад, житло можна обирати за кількістю поверхів, кімнат, ванних кімнат, наявності балкону, метражу, локації, наявності інфраструктури поруч, просторості, ремонту, поверху, ціни, освітлення і так далі. Неможливо групувати такі товари за характеристиками, і також складно відслідковувати консистентну історію рейтингів. [6]

Для розв'язання цих проблем застосовують Knowledge-Based рекомендаційні системи. Ці системи напряму взаємодіють із юзером, запитуючи у нього критерії пошуку товарів та запам'ятовуючи фідбек, і постійно підлаштовуються під користувача на основі цієї комунікації. Рекомендації, які дає Knowledge-Based

система у процесі інтерактивного фідбеку, базуються на доменних знаннях про товари. По суті, "доменні знання" – це таблиця типу "Високорівнева характеристика => Конкрентні атрибути товару". Наприклад, у доменній області нерухомості, можна зустріти подібну таблицю (рисунок 1):

Високорівневі знання	**Властивості атрибутів товару**
Просторий світлий будинок	Кількість кімнат >= 5 Кількість вікон у кожній кімнаті >= 2
Квартира для студента	Кількість кімнат <= 2 Ціна <= 200$/міс

Рисунок 1 – Приклад доменних знань для агентства з продажу і аренди нерухомості

На високому рівні, алгоритм інтерактивного фідбеку можна описати так:

- клієнт робить запит, запит потрапляє у рекомендаційну систему;
- після цього, алгоритм системи виконує пошук по доменним знанням, отримуючи множину конкретних атрибутів товару, в яких зацікавлений клієнт;
- знайдені атрибути використовуються для запиту товарів із бази даних;
- якщо користувач задоволений вибіркою, він обирає найбільш підходящий параметр, якщо не задоволений – юзер коригує запит і повертається на перший крок.

Видно, що алгоритм чітко наводить на концептуальну різницю між Knowledge-Based системами та Collaborative Filtering / Content-Based системами: навідміну від останніх, Knowledge-Based системи базовані на чітких специфікаціях бажаного товару, а не на історії інтеракцій з товарами, надаючи при цьому вищий рівень персоналізації.

Традиційно, Knowledge-Based системи поділяються на класи за способом обробки запитів:

- ***Constraint-Based*** системи. При інтеракції, юзер вводить обмеження на атрибути товару, і рекомендаційна система використовує доменні знання для пошуку найкращого кандидата (Як у прикладі з будинком вище). Крім того, рекомендаційна система може також використати атрибути самого цільового юзера – наприклад, порекомендувати більш дешеву і меншу квартиру бездітному молодому користувачеві;
- ***Case-Based*** системи. Цей вид систем заснований на Case-Based reasoning алгоритмах, що потребують бази даних про попередній досвід розв'язання проблем (case base), і на її основі пропонують розв'язання нової проблеми. Дані в case base складаються зі специфікації проблеми та способів розв'язання. Нові проблеми розв'язуються таким чином: вибирається існуюча проблема, зі специфікацією, найближчою до даної, і розв'язання знайденої проблеми адаптується під нові умови. [7]

Легко бачити, що ні Constraint-Based, ні Case-Based підходи не можуть бути реалізовані в рамках Collaborative Filtering / Content-Based рекомендаційних систем.

Як було зазначено вище, і Constraint-Based, і Case-Based системи використовують постійну комунікацію з користувачем для покращення результатів. Незалежно від доменної області, Knowledge-Based рекомендаційні системи використовують 3 форми взаємодії з користувачем. Їх можна використовувати окремо або поєднувати, в залежності від потреб конкретного бізнесу. Ці форми інтеракції можна визначити наступним чином:

- ***Conversational recommendation***. Побажання юзера отримуються в рамках "циклу фідбеку", тобто інтерактивної розмови. Форму розмови із користувачем варто використовувати у ситуації, коли доменна область є складною, і швидко визначити необхідні атрибути товарів важко. Крім того,

користувач може сам погано розбиратись у предметній області, і в рамках розмови можна деталізувати його побажання;

- ***Search-Based recommendation***. Висновок про потреби юзера робиться із відповідей на серію запитань, які звужують множину релевантних товарів. Наприклад, "Вам потрібен приватний будинок чи пентхаус?". Такі алгоритми добре інтегруються із Constraint-Based рекомендаційними системами або з пошуковими системами, що підтримують кон'юнктивні запити;
- ***Navigation-Based recommendation***. У такій формі, рекомендаційна система дає юзеру рекомендацію, а користувач відповідає на неї "критикою" – тобто інструкцією, як покращити рекомендацію. Наприклад, "Я б хотів квартиру на поверхах з четвертого по дев'ятий". Це називається "Unit Critique", тобто це одна зміна поточного запиту. Також використовують "Compound Critique" – декілька одночасних змін запиту і "Dynamic Critique" – зміна запиту, що враховує частину (або всю) історії змін запиту. Цей вид комунікації добре інтегрується із Case-Based рекомендаційними системами.

Варто зазначити, що більшість Knowledge-Based систем потребують представлення описів товарів у реляційній парадигмі (У вигляді таблиць), а не у вигляді тексту, як це було у Content-Based системах. Це є наслідком складності парадигми доменних знань, які легше подаються саме у вигляді відношень.

Отже, Knowledge-Based рекомендаційні системи краще підходять для опису товарів, що потребують високого ступеня персоналізації, а також у ситуаціях, коли неможливо отримати достатньо рейтингів для побудови простішої рекомендаційної системи. Knowledge-Based алгоритми стійкі до проблеми холодного старту, і для функціонування вони потребують невеликої кількості даних про цільового користувача. Їхніми недоліками є складність в імплементації, більш строгі вимоги до подання даних про товари, необхідність бази доменних знань і більша інтрузивність у діяльність користувача.

2 КОМПОЗИТНІ РЕКОМЕНДАЦІЙНІ СИСТЕМИ. ЕЛЕМЕНТИ МАШИННОГО НАВЧАННЯ У РЕКОМЕНДАЦІЙНИХ СИСТЕМАХ

2.1 Гібридні рекомендаційні системи

У попередньому розділі ми розглянули 3 основні види рекомендаційних систем: Collaborative Filtering, Content-Based та Knowledge-Based системи. В залежності від конкретної задачі, домену бізнесу та наявних ресурсів доцільно застосовувати той чи інший підхід. Та як було сказано раніше, кожен вид рекомендаційної системи має свої обмеження: так, Collaborative Filtering особливо вразливий до проблеми холодного старту, а Content-Based системи продукують одноманітні результати. З метою розв'язання цих проблем, на практиці різні рекомендаційні системи об'єднують у єдину систему. Такі композитні системи називаються ***Гібридними*** рекомендаційними системами.

Можна виділити 3 основних дизайни Гібридних рекомендаційних систем:

– ***Ensemble*** дизайн. У цьому дизайні базові рекомендаційні системи існують окремо, і результат їхньої незалежної роботи комбінується якимось чином у єдиний список рекомендацій. У наших позначеннях, формально це можна описати так: R^k_{m*n} - матриця рейтингів, порахована k-м алгоритмом (або базовою рекомендаційною системою) для m користувачів і n товарів, $k \in \{1, \dots, q\}$. $r^k_{i,j}$ - рекомендація юзеру i на товар j, згенерована алгоритмом k. Варто зазначити, що $r^k_{i,j}$ відрізнятимуться лише для тих фідбеків, які ми ще не спостерігали. Дуже часто результуюча рекомендація рахується як зважене середнє результатів незалежних компонентів:

$$R^* = \frac{\sum_{i=1}^{q} (s_i * R^i_{m*n})}{\sum_{i=1}^{q} s_i}. \quad (9)$$

Можливо також подавати результат роботи i-го алгоритму на вхід i+1 -го алгоритму, об'єднуючи їх у ланцюжок. Останній елемент і буде шуканим розв'язком:

$$R^* = F_q\left(F_{q-1}(\dots F_1(R))\right). \quad (10)$$

тут F_k - алгоритм k-ї рекомендаційної системи, R – початкова матриця фідбеків;

- ***Monolithic*** дизайн. Для реалізації цього дизайну створюється алгоритм, який одночасно реалізує декілька парадигм, наприклад, Knowledge-Based і Content-Based підходи. Навідміну від Ensible дизайну, у Monilithic алгоритмі не можна чітко визначити, які компоненти відносяться до яких парадигм. Цей дизайн дозволяє більш сильно зв'язати між собою різні джерела даних;
- ***Mixed*** системи. Аналогічно з Ensible дизайном, тут об'єднуються різні рекомендаційні системи, але їхні результати не об'єднуються в один, а презентуються всі разом.

Згідно з Робіном Бурком, є 7 підходів до побудови Гібридних рекомендаційних систем, що забезпечують найкращий результат. Ці підходи також дозволяють позбутись недоліків рекомендаційних систем, які з'являються при їх застосуванні окремо одна від одної. Найчастіше при реалізації цих підходів, за основу беруть рекомендаційну систему на основі Collaborative Filtering та намагаються збільшити її точність та / або позбутись проблеми холодного старту. Нижче наведено класифікації Гібридних рекомендаційних систем:

- ***Weighted***. У цьому виді Гібридних систем ми обчислюємо кінцеву рекомендацію як агрегат рекомендацій, наданих кожною наявною базовою

рекомендаційною системою. Часто використовують зважене середнє (формула (9)) або лінійну комбінація результатів. Перевагою даного виду є простота реалізації та можливість легко адаптувати гібрид під нові умови (треба просто змінити ваги). Їхній недолік – потреба в тому, щоб компоненти давали більш-менш консистентні рекомендації, а цього може бути важко досягти на практиці (Так, Collaborative Filtering методи дають поганий результат на рідковживаних товарах) (рисунок 2).

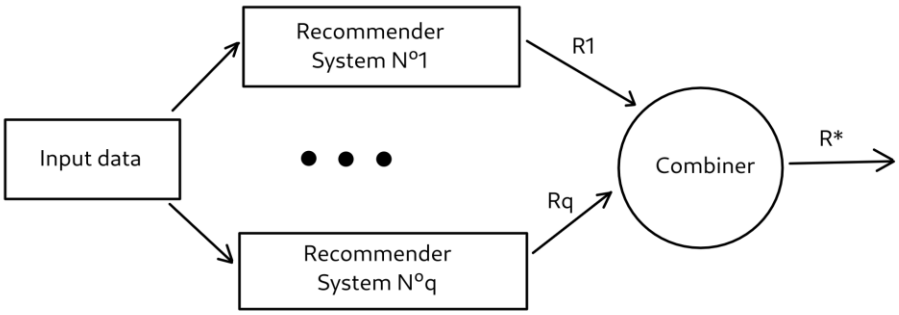

Рисунок 2 – Weighted Hybrid Recommender System

– ***Switching***. В залежності від контексту, гібрид використовує якийсь конкретний компонент для надання рекомендації. Наприклад, для юзерів, які щойно зареєструвались, доцільно використовувати Knowledge-Based систему, а коли у нас буде достатньо даних для побудови користувацького профіля – використовувати Content-Based систему (рисунок 3).

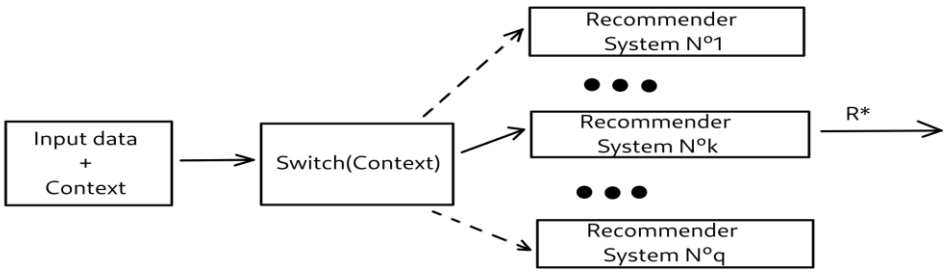

Рисунок 3 – Switching Hybrid Recommender System

– ***Mixed***. Даний вид Гібридних рекомендаційних систем найбільше підходить для ситуацій, коли необхідно надавати дуже багато рекомендацій одночасно. В залежності від контексту, ми обираємо, які частини гібриду будуть генерувати рекомендації на запит користувача, після чого чого паралельно їх генеруємо та повертаємо агрегований результат. Mixed Гібридні системи застосовують для побудови рекомендацій у складних доменах і їх можна часто побачити в парі з Knowledge-Based системами (рисунок 4).

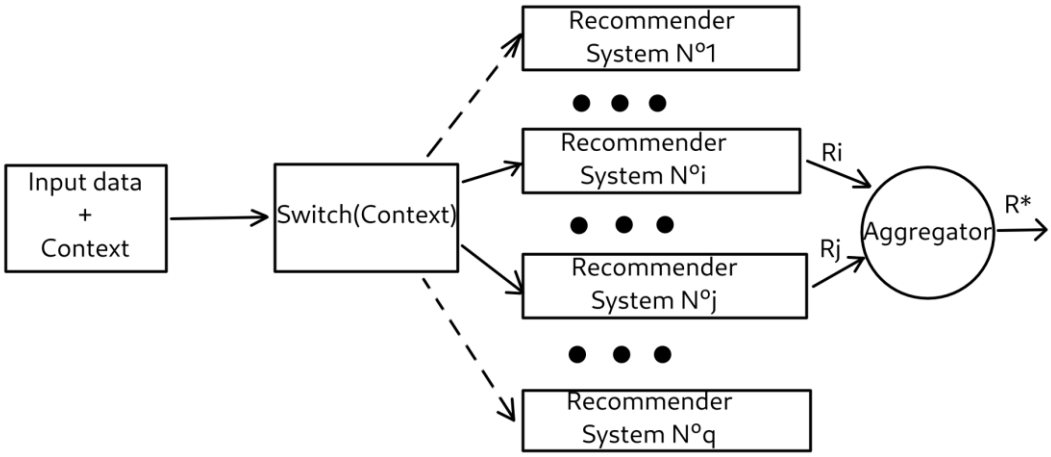

Рисунок 4 – Mixed Hybrid Recommender System

– ***Feature Combination***. У гібридних рекомендаційних системах, що реалізують Feature Combination архітектуру, ми маємо головну рекомендаційну систему і допоміжну. Головна система використовує рекомендації та / або додаткові дані (Additional Features), запропоновані допоміжною системою під час генерації рекомендацій. Перевага даного підходу заключається в тому, що головна рекомендаційна система бере до уваги результати допоміжної та коригує власний результат, і при цьому головна система не залежить напряму від даних, необхідних для

допоміжної системи. Наприклад, якщо головна система реалізує Content-Based підхід, а допоміжна – Collaborative Filtering підхід, то на старті (зразу після реєстрації) головна рекомендаційна система може ігнорувати результати допоміжної, оскільки та не здатна надати адекватні рекомендації на даному етапі. По мірі того, як користувач переглядає/оцінює все більше і більше товарів, можна поступово брати до уваги рекомендації допоміжної системи, урізноманітнюючи вибірку. Варто зазначити, що Гібридні системи, які реалізують Feature Combination підхід, є монолітними системами
(рисунок 5).

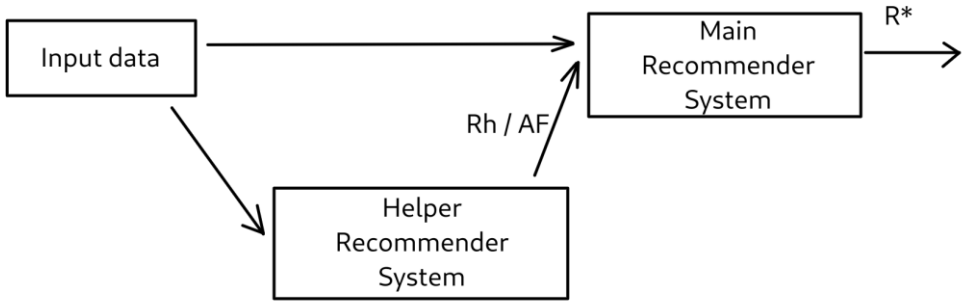

Рисунок 5 – Feature Combination Hybrid Recommender System

– ***Cascade***. Цей вид систем характерний тим, що ми вибудовуємо ієрархічну структуру із базових рекомендаційних систем. Головна рекомендаційна система створює рекомендацію, а підлегла – очищує результат, наприклад, обирає між двома рівноцінними рекомендаціями кращу. Cascade гібридні системи ефективніші в плані обчислень ніж Weighted, оскільки останні застосовують всі наявні в гібриді рекомендаційні системи до всіх товарів і всіх користувачів, без якоїсь додаткової фільтрації. Крім того, цей вид систем дає кращий результат на розріджених датасетах, бо підлегла система

очищує вже згенеровані рекомендації, а не створює повністю нові, тож на другому етапі шуми в даних не спотворюють результат (рисунок 6).

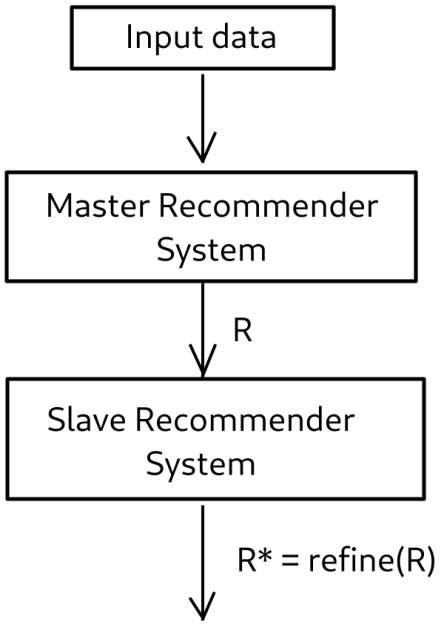

Рисунок 6 – Cascade Hybrid Recommender System

– ***Feature Augmentation***. Так само, як і в Cascade системах, результат роботи однієї рекомендаційної системи подається на вхід іншій. Але є суттєва відмінність: підлегла система в Cascade гібриді не брала до уваги результат роботи батьківської системи. Вона незалежно генерувала свої рекомендації і використовувала їх для фільтрування та репріоретизації рекомендацій, наданих головною системою. У випадку Feature Augmentation гібридів, результат роботи попередньої системи розцінюється як додаткові властивості (Additional Features) і цей результат буде використовуватись у процесі генерації рекомендацій наступною системою. Підхід з використанням додаткових властивостей для наступної рекомендаційної системи в ланцюжку дозволяє нам покращити якість рекомендацій, не змінюючи при цьому основну модель (рисунок 7).

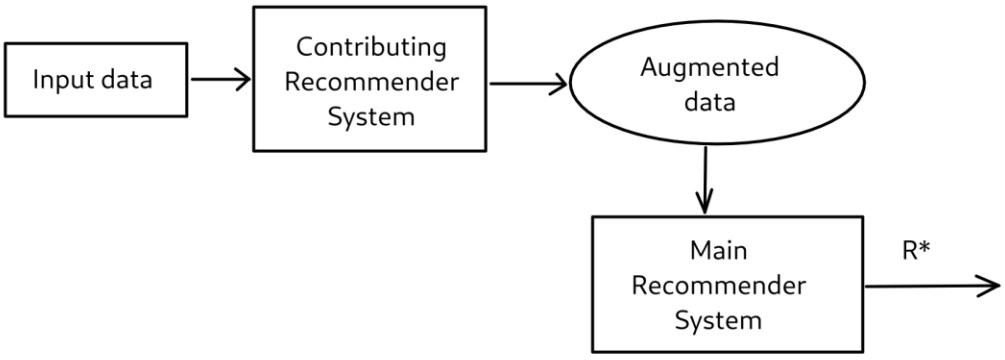

Рисунок 7 – Feature Augmentation Recommender System

– ***Meta-Level***. Ще один спосіб об'єднати рекомендаційні системи в ланцюжок. Проте при реалізації Meta-Level Гібридних систем, на вхід одній рекомендаційній системі подається модель, згенерована іншою рекомендаційною системою. Цим вони відрізнюються від Feature Augmentation методів: у останніх навчена модель генерувала додаткові дані, які передавались на вхід в іншу рекомендаційну систему. У Meta-Level системах, ціла модель стає вхідними даними. Типова комбінація – Content-Based і Collaborative Filtering системи. Тут навчена модель, яка подається на вхід Collaborative Filtering системі, представляє собою компрессоване подання інтересів цільового користувача, і колаборативному механізму буде легше знайти користувачів, чиї інтереси співпадали б з інтересами цільового користувача, ніж якби ми оперували сирою матрицею рейтингів. Ці системи також належать до систем із монолітним дизайном. [8, 9, 10]

Отже, оскільки різні рекомендаційні системи дають кращі результати в залежності від умов використання, природньо постає питання їхньої комбінації для підвищення ефективності та надійності. Розглянуті в цьому розділі методи побудови Гібридних рекомендаційних систем дозволяють не тільки збільшити точність та

швидкість генерації рекомендацій, а й позбутись недоліків базових рекомендаційних систем. Саме тому Гібридні рекомендаційні системи доцільніше застосовувати на практиці, аніж окремі базові системи. Варто зазначити, що усі лідери Netflix Prize – Гібридні рекомендаційні системи із Ensemble дизайном.

2.2 Моделі з латентними факторами

Як було згадано раніше, на практиці матриця фідбеків є дуже розрідженою. Існують різні способи зменшити її розмірність та покращити представлення даних задля більш ефективного використання у Collaborative Filtering та Content-Based Filtering рекомендаційних системах. Зазвичай у машинному навчанні для цього використовують Singular Value Decomposition (SVD). SVD – спосіб факторизації матриць, він використовує структуру матриці, де рядки позначають юзерів, а стовпці– товари. Факторизація полягає в тому, щоб представити матрицю фідбеків R як

$$R = UV^T, \tag{11}$$

де U і V – ортогональні матриці.

Розглянемо випадок, коли матриця R повністю вказана. В такому випадку можна провести факторизацію алгоритмом Truncated SVD. В цьому алгоритмі, ми розкладемо матрицю R на 3 матриці:

$$R \approx Q_k * \Sigma_k * P_k^T. \tag{12}$$

Тут Q_k, Σ_k, P_k - матриці розміру m*k, k*k, k*n відповідно. У стовпчиках матриці Q_k знаходяться k найбільших власних векторів матриці $R * R^T$, а у стовпчиках матриці

P_k- найбільші власні вектори матриці $R^T * R$. Матриця Σ_k складається з квадратних коренів k найбільших власних значень матриці Q_k або P_k.

Цей розклад дозволяє нам багато чого дізнатись: матриця Q_k представляє собою відношення юзерів до латентних факторів, матриця Σ_k представляє собою силу (важливість) латентних факторів, а матриця P_k - подібність товарів до латентних факторів [11, 12]. Щоб мати розклад у вигляді (11), матрицю Σ_k зазвичай поглинає Q_k:

$$\begin{aligned} U &= Q_k * \Sigma_k \\ V &= P_k \\ R &\approx U * V^T . \end{aligned} \tag{13}$$

Задачу пошуку пошуку ортогональних матриць U і V можна поставити наступним чином:

$$J = \frac{1}{2} * \|R - U * V^T\|^2 \to min. \tag{14}$$

Розглянемо ***простий ітеративний підхід до SVD***, що розв'язує оптимізаційну проблему, коли матриця R не повністю визначена. Спершу ми повинні центрувати матрицю R, віднявши від кожного рядка його середнє значення – μ_i. Отриману матрицю позначимо як R_c. У центрованій матриці замінимо невказані значення нулями. Після чого знайдемо розклад матриці R_c у вигляді (13), застосувавши SVD. Позначимо $\overline{u_i}, \overline{v_j}$ - і-й рядок матриці U та j-й рядок матриці V. Тоді передбачений рейтинг юзера і для товару j обчислюється наступним чином:

$$\hat{r_{i,j}} = u_i * v_j + \mu_i \tag{15}$$

Проте отриманий рейтинг насправді не є надійним, бо значення є "упередженим" – через початкову ініціалізацію неспостережених значень середнім арифметичним всіх рейтингів у рядку і. Для того, щоб позбутись упередження, наведену вище процедуру повторюють ітеративно за наступним алгоритмом:

- ***ініціалізація:*** Ініціалізувати неспостережені значення в рядку і середнім арифметичним цього рядка - μ_i. Отримали матрицю R_c;
- ***крок 1:*** Привести R_c до вигляду $Q_k * \Sigma_k * P_k^T$ використовуючи SVD;
- ***крок 2:*** Замінити елементи R_c, які стояли на місці неспостережених значень, відповідними значенями матриці $Q_k * \Sigma_k * P_k^T$. Повернутись на крок 1.

Зазвичай проводять 5-10 ітерацій для досягнення правдоподібних передбачень.

У випадку коли матриця R містить дуже багато невказаних значень, обчислення наведеним алгоритмом можуть застрягти у локальному оптимумі. Щоб цього не сталось, у процедуру вводять предиктори, що передбачають упередження юзера і стосовно товару j. Нехай до початку процедури ми отримали від предиктора $B_{i,j}$ - упередження юзера і до товару j. Після чого від усіх спостережених елементів $r_{i,j}$ матриці R віднімають $B_{i,j}$ і запускають наведений вище алгоритм. Після закінчення, $B_{i,j}$ додається назад до спостережених елементів. Даний підхід є більш надійним через кращу ініціалізацію.

Простий ітеративний підхід на практиці дорого реалізовувати, тому що він вимагає повністю визначеної матриці. Більш ефективний підхід – додати обмеження на ортогональність до моделі, що оптимізується. Оптимізацію можна провести із використанням методів градієнтного спуску.

Позначимо множину спостережених значень матриці R через S. Матимемо наступну оптимізаційну задачу (із регуляризацією):

$$J = \frac{1}{2}\sum_{(i,j)\in S}\left(r_{i,j} - \sum_{s=1}^{k} u_{i,s} * v_{j,s}\right)^2 + \frac{\lambda_1}{2}\sum_{i=1}^{m}\sum_{s=1}^{k} u_{i,s}^2 + \frac{\lambda_2}{2}\sum_{j=1}^{n}\sum_{s=1}^{k} v_{j,s}^2, \qquad (16)$$

де U і V – ортогональні матриці

Нагадаємо, що у машинному навчанні, регуляризація - це техніка, що використовується для зменшення перенавчання (overfitting) моделі на тренувальному датасеті та поліпшення її здатності узагальнюватись на нові, невідомі дані. Регуляризація допомагає зменшити цей ефект, додаванням додаткового члену до функції втрат в процесі навчання моделі. Цей додатковий член вимагає від моделі не тільки зведення функції втрат на тренувальному датасеті, але і зменшення значень ваг, що використовуються в моделі. Це дозволяє моделі бути менш чутливою до окремих даних в тренувальному датасеті, та здатною до кращого узагальнення на нові дані. Задача (16) є складнішою через введення обмеження на ортогональність. Для ефективного розв'язання краще використовувати напрямлений градієнтний метод [13].

2.3 Використання Data Mining технік у рекомендаційних системах

Data Mining – також відомий як Knowledge Discovery in Data (KDD) – це процес знаходження закономірностей у великих датасетах. Data Mining використовується організаціями для оптимізації процесів прийняття рішень за допомогою аналізу даних. Зазвичай KDD використовують для досягнення однієї з двох цілей: або для більш детального і високорівневого опису датасетів, або для передбачень за допомогою машинного навчання. У процесі майнінгу даних, Data Scientist-и описують дані за допомогою спостережень, кореляцій та асоціацій, після чого класифікують та кластеризують дані за допомогою класифікаційних та регресійних методів.

Однією з технік майнінгу даних, що дозволяють видобувати із даних знання, є Асоціативні правила. На високому рівні, асоціативні правила є твердженнями типу "якщо-то" і вони допомагають знайти вірогідність того, що між якимись об'єктами існує відношення. Майнінг даних за допомогою асоціативних правил широко застосовується у торгівлі для збільшення обсягів продажу, а також для знаходження кореляцій у датасетах, пов'язаних із медициною. [14]

Асоціативні правила виникли при розв'язанні задачі знаходження відношень між товарами у супермаркетах, тому цілком природньо, між ними та Collaborative Filtering існує певна подібність. Тому важливо підкреслити різницю між ними: Асоціативні правила не дають інформації про особисті вподобання якогось клієнта, зате вони дозволяють встановити відношення між елементами кожної транзакції. Уведемо деякі основні поняття: Нехай у нас є дискретна множина наявних товарів I потужності n. Транзакцією T ми назвемо підмножину I. Базою даних транзакцій $\Psi = \{T_1, T_2, \ldots, T_m\}$ ми назвемо множину із m транзакцій. Асоціативне правило – це імплікація, яка складається із антецедента A і консеквента C, $A \subset I$, $C \subset I$ і позначає кореляцію (а не наслідок) (рисунок 8).

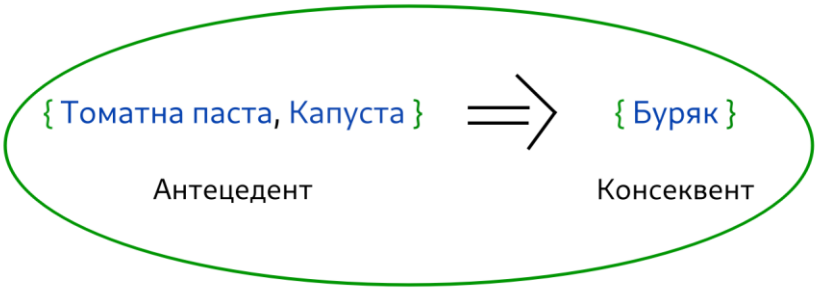

Рисунок 8 – Приклад асоціативного правила

Також введемо поняття набору товарів: $IS = A \cup C$.

Для того, щоб оцінити силу відношення між групами товарів, є 3 важливі метрики:

– Support або значення підтримки – це відношення, яке показує, наскільки часто набір товарів зустрічається в усіх транзакціях:

$$Support(\{X\}=>\{Y\}) = \frac{|\{T:X\subset T\wedge Y\subset T\}|}{|\Psi|}. \quad (17)$$

Якщо 2 товари ріддко зустрічаються у транзакціях, то значення підтримки близьке до нуля. Тоді немає сенсу утворювати з цих товарів правила, оскільки у нас недостатньо інформації, щоб оцінити відношення між цими товарами.

– Confidence або упевненість – імовірність появи консеквента у корзині користувача, якщо там є антецеденти:

$$Confidence(\{X\}=>\{Y\}) = \frac{|\{T:X\subset T\wedge Y\subset T\}|}{|\{T:X\subset T\}|}. \quad (18)$$

Чим вище значення упевненості, тим сильніша асоціація між двома елементами, однак є проблема: Якщо антецедентом виступає набір товарів, які купують дуже часто, то значення упевненості може бути спотворене і не відображатиме реальної кореляції.

– Lift або значення підвищення інтересу – відношення імовірності мати консеквент у корзині, знаючи, що антецедент теж у корзині, до імовірності появи консеквента у корзині в цілому:

$$Lift(\{X\}=>\{Y\}) = \frac{\frac{|\{T:X\subset T\wedge Y\subset T\}|}{|\{T:X\subset T\}|}}{\frac{|\{T:Y\subset T\}|}{|\Psi|}} = \frac{Confidence(\{X\}=>\{Y\})}{\frac{|\{T:Y\subset T\}|}{|\Psi|}}. \quad (19)$$

Значення підвищення інтересу менше за 1 показує, що поява антецедента у корзині не збільшує імовірності появи консеквента, навіть якщо значення упевненості є високим. Чим більше значення Lift за 1, тим сильніша асоціація між A і C. Таким чином, значення підвищення інтересу є більш надійною метрикою, ніж значення упевненості. [15]

Існує декілька алгоритмів, що дозволяють отримати асоціативні правила із датасету. Одним із найперших та найпростіших у реалізації алгоритмів є алгоритм Apriory. Ключовим аспектом алгоритму Apriory є антимонотонність значення підтримки та значення впевненості. В основі цього алгоритму лежить припущення, що всі підмножини наборів товарів, які часто зустрічаються, повинні також часто застрічатись (Принцип Apriory); якщо набір товарів зустрічається нечасто, то і множини, які включають в себе даний набір товарів, теж зустрічатимуться нечасто. Значення впевненості також слідує принципу антимонотоннсоті з урахуванням кількості елементів у консеквенті:

$$Confidence(\{A,B,C\}=>\{D\}) \geq Confidence(\{A,B\}=>\{C,D\}) \geq Confidence(\{A\}=>\{B,C,D\}). \quad (20)$$

Принцип антимонотонності значень підтримки та впевненості дозволить відповідно відсіяти комбінації товарів та правила, які не задовольняють обмеженню на мінімальні значення підтримки та упевненості без необхідності створювати ці комбінації та правила. Розглянемо алгоритм покроково:

- **крок 1**. k = 1. Для всіх наявних товарів порахувати значення підтримки. У список кандидатів C_k записати товари, що задовольняють обмеженню на мінімальне значення підтримки;
- **крок 2**. Збільшити поточне значення k на 1 та згенерувати список кандидатів C_k, використовуючи C_{k-1}. Таблиця C_k є результатом декартового

добутку списку C_{k-1} самого на себе за умови, що лівий рядок C_{k-1} і правий рядок C_{k-1} мають k – 2 спільних товари. Для кожного отриманого рядка потрібно порахувати значення підтримки, і якщо воно є більшим за мінімальне, то додати цей рядок в список C_k. Якщо до списку C_k було додано хоч один рядок, перейти на крок 2, інакше – перейти на крок 3;

- **крок 3**. Об'єднати списки C_k у список C, k >= 2;
- **крок 4**. Для кожного рядка таблиці C згенерувати усі можливі асоціативні правила. Для кожного правила обчислити значення упевненості. Якщо воно є більшим за мінімальне значення упевненості, ми розглядаємо значення як надійне та додаємо до результуючого списку правил Rules;
- **крок 5.** Повернути список Rules та завершити алгоритм.

Легко бачити, що алгоритм Apriory є простим для розуміння та реалізації. Однак, у нього є і недоліки. Найбільшим із них є те, що алгоритм потребує багато обчислювальних ресурсів та повільно працює на великих наборах транзакцій. Тож для додатків, які функціонують в умовах малої кількості доступних ресурсів або працюють із великими обсягами даних, краще підійдуть новіші алгоритми, такі як алгоритм Max-Miner.

Асоціативні правила можуть бути використані у рекомендаційних системах, що працюють над унарними матрицями рейтингів (матриці, що складаються з нулів та одиниць) – наприклад, це матриці, які відображають якусь разову активність користувачів: "Подивився – не подивився", "Купив – не купив" і тд. Для того, щоб застосовувати асоціативні правила у рекомендаційних системах, ми повинні побудувати їх за вищенаведеним алгоритмом з додатковою умовою: нас цікавитимуть лише правила з одним товаром у консеквенті. Після цього, для кожного конкретного користувача ми можемо будувати рекомендації наступним чином: Відбираємо асоціативні правила, де у антецеденті є якнайбільше товарів, уподобаних

цільовим користувачем, сортуємо їх за спаданням значення упевненості та відбираємо перші k значень.

Наведений вище підхід доволі простий, але рівень персоналізації доволі низький, оскільки у нас є лише інформація про те, чи купив юзер товар, чи не купив. Але наш алгоритм не враховує інформацію про те, чи насправді сподобався користувачу товар. Для розв'язання цієї проблеми ми можемо модифікувати елементи антецедента і консеквента так, щоб у них містилась інформація про те, чи сподобалися товари користувачу, який ініціював транзакцію (рисунок 9):

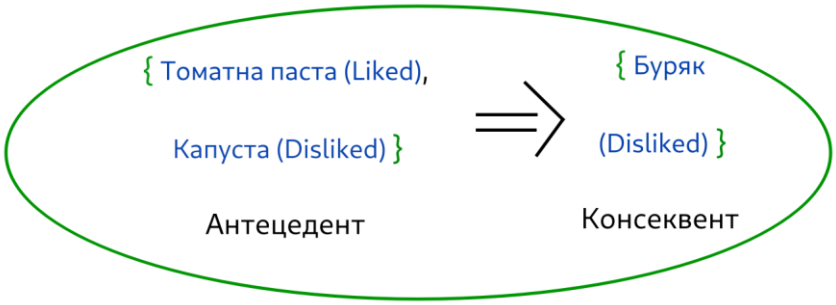

Рисунок 9 – Асоціативне правило з атрибутом уподобання

Даний підхід дозволяє нам урахувати інформацію про особисті уподобання користувача, однак алгоритм відбору асоціативних правил для рекомендацій цільовому юзеру потребує модифікації. Оскільки у кожного користувача свої уподобання, то тепер асоціативні правила можуть конфліктувати між собою. Конфліктом двох асоціативних правил ми назвемо ситуацію, коли правила мають спільні товари в консеквенті із різним значенням атрибуту уподобань. Для того, щоб розв'язати проблему конфліктів, необхідно знайти спосіб пріоритезувати правила в залежності від уподобань цільового користувача. Наприклад, можна використати значення упевненості або порівняти кількість користувачів, яким товар сподобався чи не сподобався.

Ми розглянули Item-Based модель з використанням асоціативних правил, але в принципі можливе і використання User-Based моделі. Для того, щоб отримати

правила асоціації між користувачами, достатньо повторити згадані вище алгоритми на транспонованій матриці фідбеків. Таким чином, ми отримаємо асоціації між профілями користувачів, які покажуть, з якими юзерами цільовий користувач має спільні смаки, а з якими – відмінні. Такий підхід дозволить робити більш неочікувані рекомендації, проте потребуватиме більших обчислювальних потужностей, позаяк кількість користувачів зазвичай перевищує кількість товарів.

3 РЕАЛІЗАЦІЯ РЕКОМЕНДАЦІЙНОЇ СИСТЕМИ ДЛЯ КНИЖКОВОГО ІНТЕРНЕТ-МАГАЗИНУ

3.1 Постановка задачі

Розглянемо стартап інтернет-магазин Bookstore, що продає книжки. Власники Bookstore надали нам дані про наявні книжки, користувачів та їхні покупки:

- bookAlsoEnjoy.csv (id, also_enjoy_ids) – 1354 рядки. Датасет містить інформацію про книжки, які, на думку продавця, можуть бути схожі з його товаром (1100 порожніх записів);
- bookDesc.csv (id, goodreads_desc, google_desc) – 1354 рядки. Датасет містить інформацію про опис книжок на ресурсах GoodReads та Google;
- bookList.csv (id, title, subtitle, authors, language, genre, published_date, page_count) – 1354 рядки. Датасет містить інформацію про усі наявні книги у Bookstore;
- bookReview.csv (id, text_review, rating) – 33757 рядки. Датасет містить інформацію про фідбек користувачів на книги;
- users.csv (id, email, firstName, lastName, avatarUrl) – 1000 рядків. Датасет містить інформацію про частину користувачів додатку;
- purchases.csv (id, date, receipt_id, book_id, user_id) – 15000 рядків. Датасет містить інформацію про транзакції користувачів із датасету users.csv. [16]

За тиждень необхідно розробити рекомендаційну систему та MVP проект для демонстрації персоналізації товарів за наступними умовами:

- при генерації стрічки на першому місці мають бути товари, що можуть зацікавити юзера;
- стрічка містить неочевидні товари, які можуть здивувати користувача;

- рекомендаційна система враховує, які жанри книжок подобаються більше, ніж інші;
- рекомендаційна система враховує фідбек інших користувачів.

3.2 Підбір технологій та реалізація

Підбір технологій варто почати із бази даних, оскільки вона буде ядром нашого додатку. Спершу необхідно визначитись із видом бази даних. Оскільки датасети добре розбиті по доменам, не містять багато порожніх значень та мають чітку структуру, то найкраще нам підійде якась із реляційних баз. На мою думку, найбільш підходящим варіантом буде PostgreSQL, оскільки це безкоштовна база з відкритим вихідним кодом, її легко використовувати та адмініструвати, має чудовий перфоманс навіть на великій кількості даних, добре масштабується.

Наступним кроком є проектування архітектури веб-додатка. Оскільки ми будуємо MVP, можна обійтись елементарною монолітною архітектурою (рисунок 10):

Рисунок 10 – Поточна архітектура додатку

Для фронтенду оберемо фреймворк React та бібліотеку styled-components, оскільки нам важливо швидко реалізувати MVP, а така комбінація фронтенд-технологій дозволить перевикористовувати компоненти та швидко створювати гарні елементи UI без необхідності маніпулювати HTML та CSS. Також використаємо мову

TypeScript – це допоможе уникнути помилок, пов'язаних з динамічною типзацію JavaScript.

Для бекенду також оберемо TypeScript, щоб не поскладнювати процес розробки. Використаємо платформу Node.js, яка пропонує легку розробку та масштабування server-side додатків на JS. Node.js ідеально підходить для додатків із real-life комунікацією та стрімінгом, що є важливим для нас. Крім того, платформа є безкоштовною і мультиплатформенною. Важливим кроком при створенні бекенду є вибір технології для комунікації з базою даних. Оскільки ми маємо справу із реляційною базою, ми можемо скористатись ORM – фреймворком, яким дозволяє оперувати таблицями так, наче це об'єкти. Ми можемо делегувати ORM такі процеси, як маппінг відношень до об'єктів, генерація SQL запитів, обробка транзакцій та очистка параметрів від небажаних SQL ін'єкцій. Також необіхідно обрати фреймворк для створення HTTP сервера бекенду. Оскільки нам важлива шкидкість розробки, то фреймворк повинен автоматизувати якомога більше процесів, залишаючи для нас лише роботу з бізнес-логікою. Тому найкращим вибором стане Nest.js: Модулярна архітектура дозволяє легко перевикористовувати компоненти, інтегрована підтримка TypeORM, підтримка DI, декларативна оброка HTTP запитів та відповідей.

Перейдемо до проектування рекомендаційної системи. Перед тим, як власне розробити систему, необхідно додатково структурувати дані та подати їх у вигляді, сумісному з реляційною парадигмою. На рисунку 11 наведено структуру частини бази даних, якою оперуватиме рекомендаційна система:

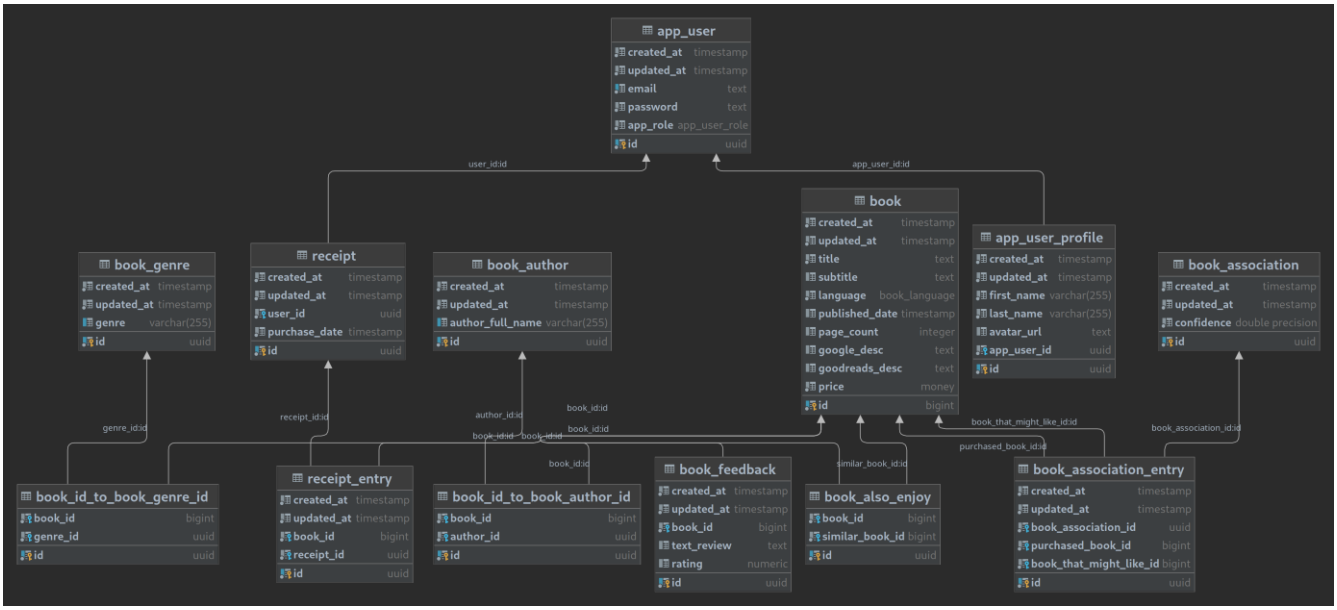

Рисунок 11 – База даних Bookstore

За умовами, рекомендаційна система повинна враховувати історію покупок самого користувача та інших користувачів. Інакше кажучи, рекомендаційна система повинна містити Collaborative Filtering та Content-Based компоненти. Крім того, нам потрібно мати товари, які корелюються зі смаками користувача, але можуть мати інший жанр або автора, тобто потрібно знайти спосіб встановити відповідності між різними авторами та жанрами, а потім відібрати топ найбільш схожих.

Для початку визначимо, які жанри та автори для кожного користувача є улюбленими. Оскільки ми маємо історію транзакцій, та ми точно знаємо, що кількість куплених користувачем товарів не перевищує кількох десятків, то достатньо буде порахувати частоту появи кожного жанру g чи автора a по всіх чеках цільового користувача. Позначимо ці величини LS_g і LS_a, а також визначимо множину книг, які купив цільовий користувач B_{pu}, множину авторів книги b позначимо як A_b, множину жанрів книги G_b:

$$LS_g = \frac{|\{b : g \in G_b \wedge b \in B_{pu}\}|}{max(\{1, |\{b : b \in B_{pu}\}|\})}. \quad (21)$$

$$LS_a = \frac{|\{b : a \in A_b \wedge b \in B_{pu}\}|}{max(\{1, |\{b : b \in B_{pu}\}|\})}. \quad (22)$$

У формулах (21), (22) у чисельниках – мультисети. Знаючи ці величини, ми можемо обчислити оцінки по жанрам і авторам для кожної книги. Позначимо ці оцінки як W_g і W_a, а також позначимо множину спільних жанрів книги b із жанрами, які користувач любить як

$$C_{bg} = G_b \cap \{g | \exists Book : Book \in B_{pu} \wedge g \in G_{Book}\}. \quad (23)$$

Аналогічно множина спільних авторів книги із жанрами, які користувач любить, позначимо як

$$C_{ba} = A_b \cap \{a | \exists Book : Book \in B_{pu} \wedge a \in A_{Book}\}. \quad (24)$$

Тоді ми можемо обчислити W_g і W_a наступним чином:

$$W_g = \frac{\sum_{g \in C_{bg}} LS_g}{max\{1, |C_{bg}|\}}. \quad (25)$$

$$W_a = \frac{\sum_{a \in C_{ba}} LS_a}{max\{1, |C_{ba}|\}}. \quad (26)$$

Таким чином, використовуючи оцінки W_g і W_a, ми можемо знайти найбільш релевантні книги по жанрам і авторам, виходячи з історії покупок цільового користувача. Тепер нам потрібно реалізувати колаборативну частину рекомендаційної системи. Для цього ми можемо використати асоціативні правила, щоб знайти взаємозв'язки між товарами, які різні користувачі купують разом.

Генерація асоціативних правил відбувається за алгоритмом, наведеним у розділі 2.3, з додатковою фільтрацією правил за показником Lift: нас цікавлять лише правила, у яких показник Lift більший або рівний 1 [Додаток А].

Позначимо множину асоціативних правил, де в антецеденті є куплена користувачем книга b_a:

$$AR_{b_a} = \{ar = (\{X\} => \{Y\}) : b_a \in X \wedge b_a \in B_{pu}\}. \quad (27)$$

І введемо позначення оцінки асоціативного правила W_{ar} для книги b_c, що міститься у консеквенті:

$$W_{ar} = max(\{0\} \cup \{Confidence(ar)\}), ar \in AR_{b_a}. \quad (28)$$

Асоціативні правила дозволять нам знайти закономірності у покупках користувачів, проте не для кожної пари книг знайдеться правило, яке задовольняє обмеженню на мінімальне значення упевненості. Для того, щоб зробити колаборативну частину рекомендаційної системи більш значущою, ми можемо використати рейтинги, які користувачі надали книзі. Позначимо через R_b множину числових рейтингів книги b, екземпляр рейтингу – через r_b, $r \in [1,5]$. Корисною для нас величиною буде середній рейтинг книги $\bar{r_b}$:

$$\bar{r_b} = \frac{\sum_{r \in R_b} r}{max(\{1, |R_b|\})}. \quad (29)$$

Маючи середній рейтинг книги, ми можемо зменшити упередженість оцінок W_g і W_a . Нам важливо, щоб книги, які загалом не подобаються читачам, не займали надто високі місця у стрічці цільового юзера, навіть якщо він є фанатом автора чи

жанру. В той же час, ми б хотіли, щоб популярні книжки авторів та жанрів, які юзер читає, були у нього на вищих місцях. Порахуємо неупереджені оцінки W_{gu} і W_{au} наступним чином:

$$W_{gu} = \frac{W_g + (1 - sign(\bar{r_b})) * W_g + sign(\bar{r_b}) * \bar{r_b}}{2}. \quad (30)$$

$$W_{au} = \frac{W_a + (1 - sign(\bar{r_b})) * W_a + sign(\bar{r_b}) * \bar{r_b}}{2}. \quad (31)$$

Залишилось об'єднати ці оцінки у результуючу, яку буде використано при сортуванні товарів для кожного конкретного юзера. В нашому випадку, буде цілком достатньо скористатись Weighted гібридною системою. Для середнього рейтингу книги оберемо вагу $c_1 = \frac{3}{10}$, решта вагів $c_{2-4} = 1$. Значення c_1 було обрано достатньо великим, аби високо оцінені товари переважали низько оцінені, але не настільки великим, щоб нівелювати персоналізацію, яку дають оцінки W_{gu} і W_{au} . Таким чином, фінальна оцінка W:

$$W = c_1 * \bar{r_b} + c_2 * W_{gu} + c_3 * W_{ga} + c_4 * W_{ar}. \quad (32)$$

Легко бачити, що 2-й і 3-й доданки вносять персоналізацію, 1-й та 4-й доданки – коригують оцінку відповідно до смаків інших користувачів. Крім того, за допомогою 1-го і 4-го доданку ми уникаємо проблеми холодного старту: коли новий користувач реєструється або має бідний профіль, ми презентуємо товари, які в цілому найбільше сподобались користувачам платформи та товари, які інші користувачі купили разом із тими небагатьма товарами, що є у нового користувача [Додаток Б].

Нижче наведено результати роботи даної рекомендаційної системи для двох різних користувачів (рисунки 12–16):

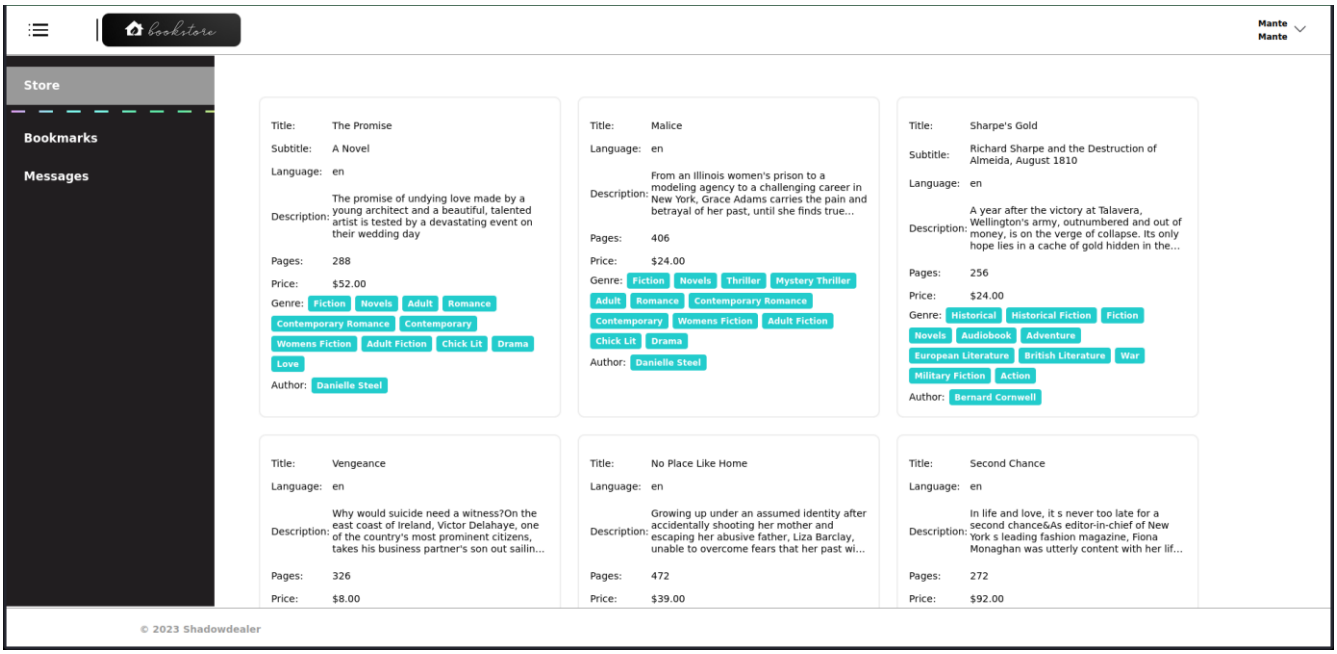

Рисунок 12 – Перші 6 книг юзера Mante

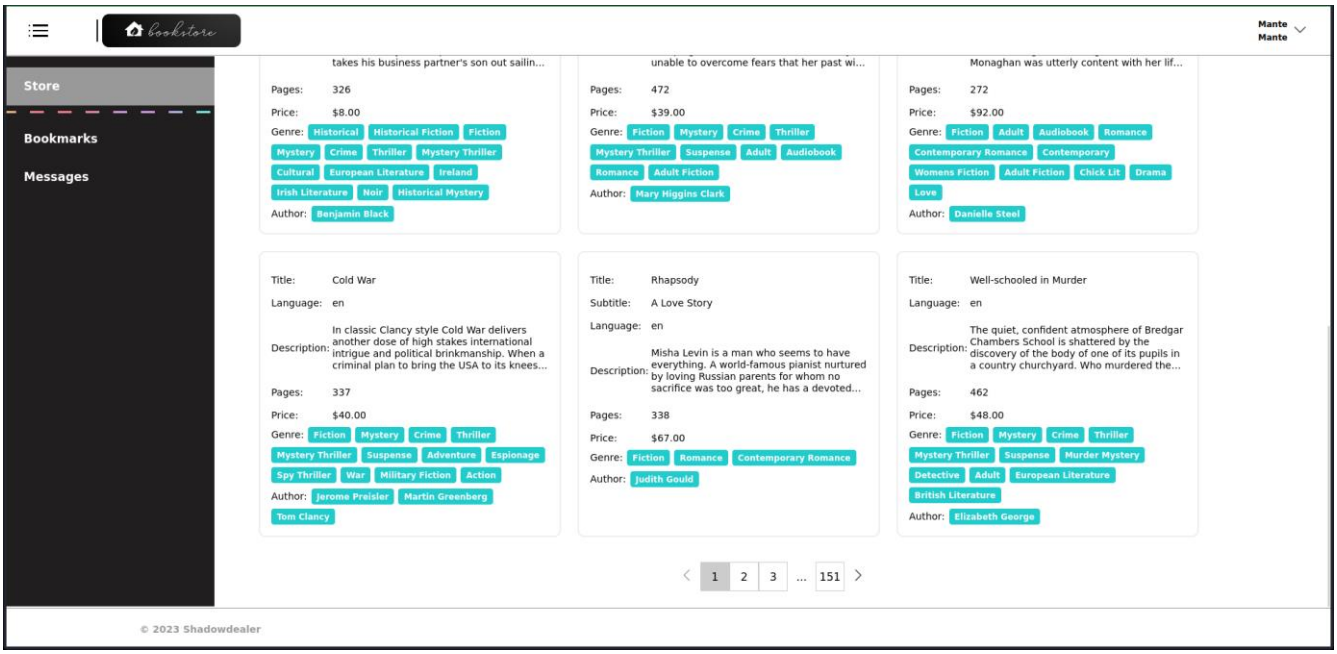

Рисунок 13 – Наступні 3 книги юзера Mante

Бачимо, що користувач Mante більше захоплюється романтикою, новелами, кримінальними драмами.

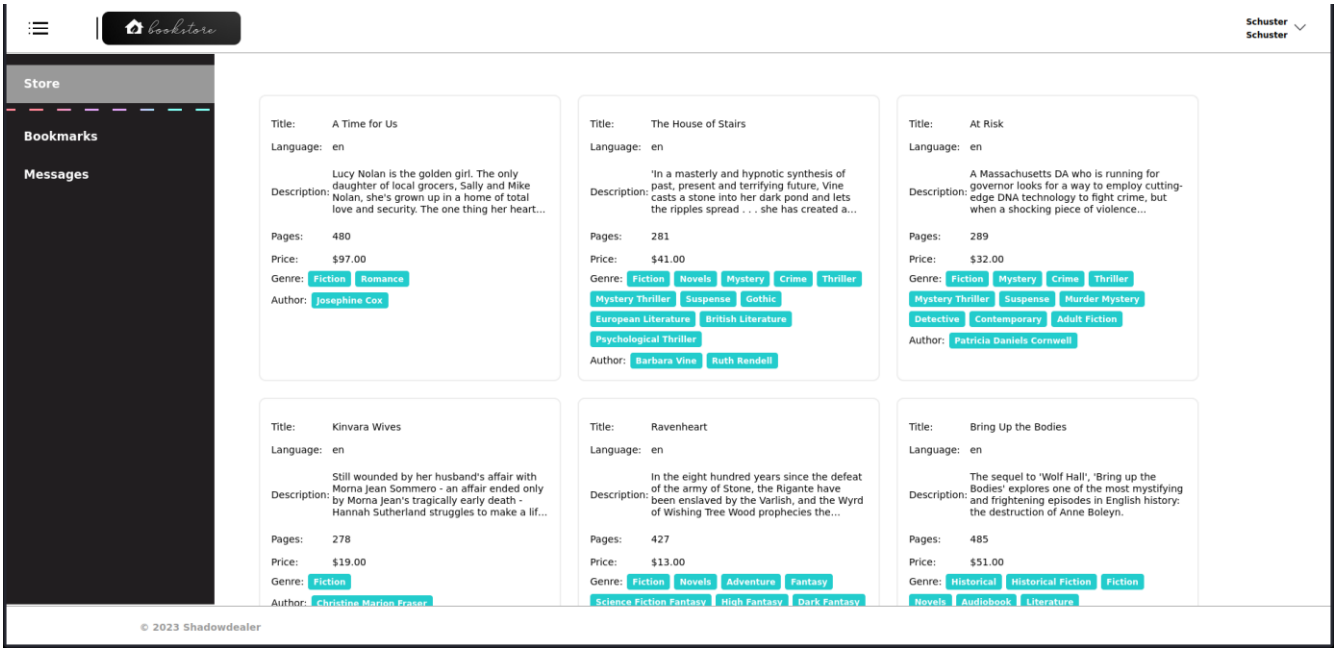

Рисунок 14 – Перші 6 книг користувача Schuster

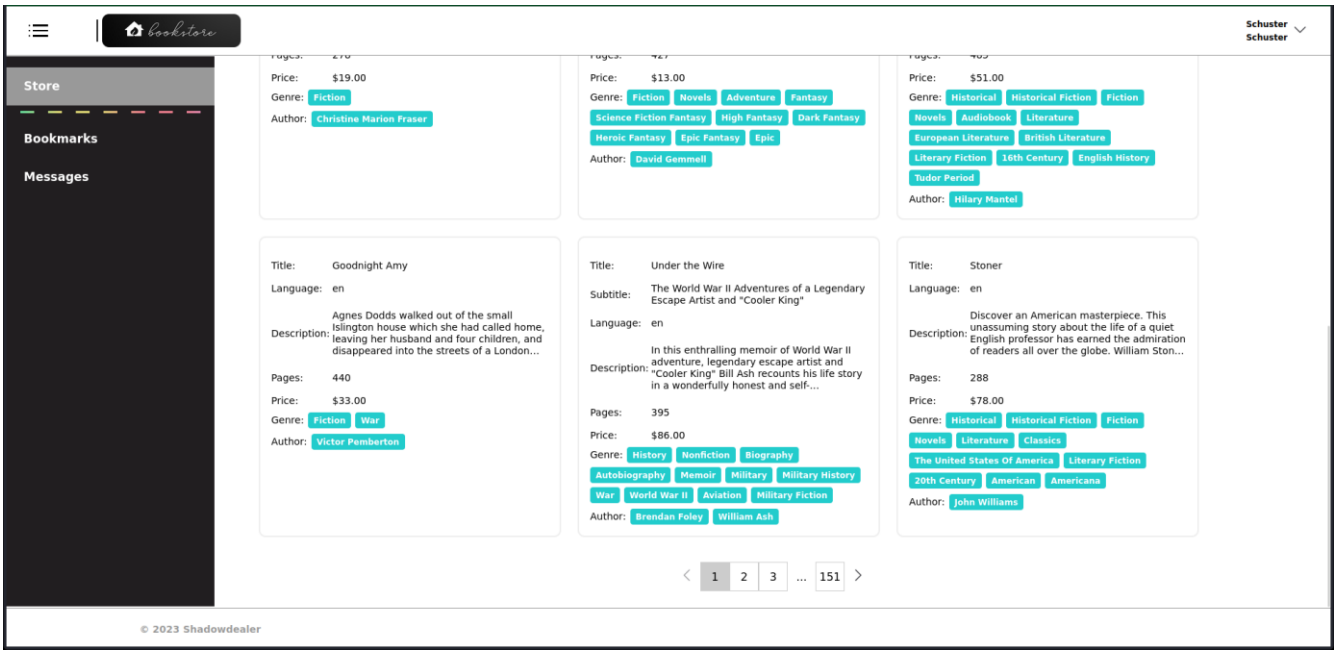

Рисунок 15 – наступні 3 книги користувача Schuster

На останніх двох рисунках ми бачимо результат роботи рекомендаційної системи для іншого користувача. Стрічка користувача Schuster відрізняється від стрічки користувача Mante, оскільки Schuster більше захоплюється фантастикою,

історією та воєнною літературою. Таким чином, кожен користувач має свою унікальну стрічку, порядок елементів у якій змінюватиметься після кожної покупки.

3.3 Оцінка проекту та рекомендаційної системи

Даний проект розв'язує поставлену задачу та цілком задовольняє поставлені умови. Проте наведена реалізація – лише MVP, і для запуску системи у продакшн необхідно внести ряд змін. Ці зміни стосуються не лише процесу генерації рекомендацій, але і веб-додатку в цілому.

Перше, на що варто звернути увагу, це архітектура додатку. Монолітні системи зазвичай важко підтримувати та тестувати. Рано чи пізно, компоненти такої системи стають сильно зв'язними, що сильно поскладнює процес удосконалення. У архітектурі на рисунку 10 є іще одна неочевидна, але серйозна вада: рекомендаційна система вбудована у монолітну систему і співіснує по сусідству із іншими частинами додатку. Це є проблемою, тому що рекомендаційні системи потребують дуже великої кількості ресурсів, і деплой рекомендаційної системи сервері, пільному з іншими частинами додатку, може призвести до відмови в обслуговуванні.

Проблему зв'язності рекомендаційної системи із іншим кодом та проблему вичерпування ресурсів допоможе розв'язати мікросервісна архітектура та використання технологій кешування. На рисунку нижче продемонстрований приклад архітектури книжкового інтернет-магазину, в якому рекомендаційну систему буде легко оновлювати та підтримувати:

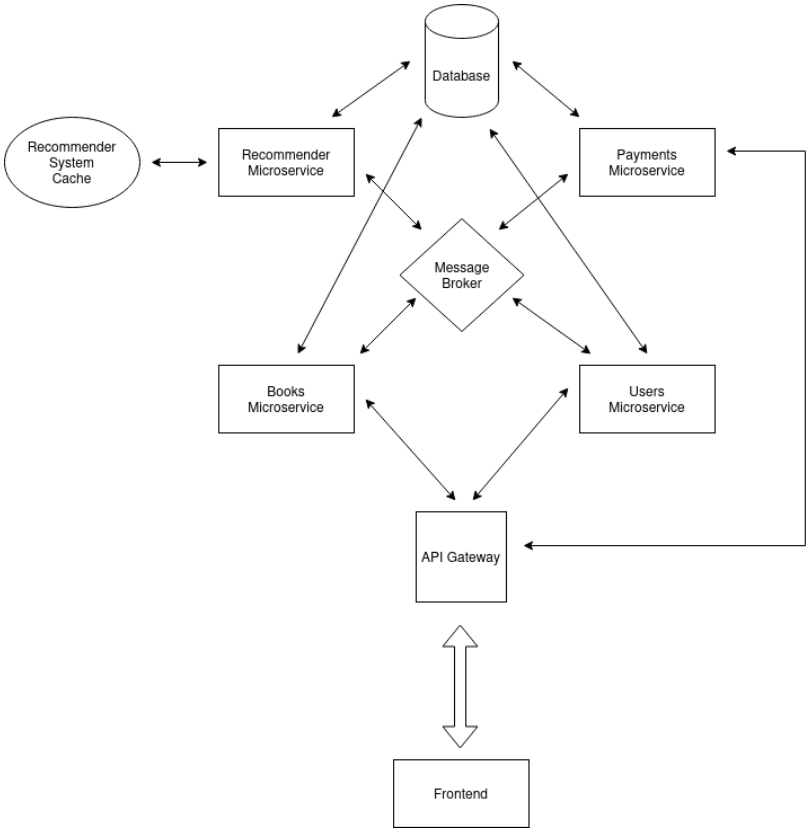

Рисунок 16 – Покращена архітектура інтернет-магазину

Другою проблемою, на яку варто звернути увагу, є недолік дизайну самої рекомендаційної системи. По-перше, наша система не враховує явний фідбек користувача: Якщо користувач купив товар і дав негативний відгук, то схожі товари йому рекомендувати вже не потрібно. По-друге, дана система не використовує неявний фідбек. Юзери не купуватимуть книги дуже часто, скоріше за все, не більше 20 штук на рік. Неявний фідбек був би дуже корисним, бо він би дозволив робити висновки про уподобання користувача, навіть якщо той нічого не купує. Так, якщо користувач купував лише триллери, але останні 10 переглянутих ним книжок є історичними, то можна зробити висновок, що у користувача змінились уподобання у жанрі, і книжкам з історичним жанром потрібно надати вищий пріорітет. По-третє, для нашого інтернет-магазину можливо додати ще одну рекомендаційну підсистему, що аналізуватиме описи та рев'ю книжок і таким чином знаходитиме подібні книжки –

розглянута нами у підрозділі 1.3 Content-Based фільтрація. Використовуючи формули (6) – (8) можна побудувати Term Frequency – Inverse Document Frequency статистику та збагатити стрічку товарами, схожими за описами.

Третьою проблемою реалізації є неефективний алгоритм присвоєння оцінок книжкам. У поточній реалізації, усі книжки завантажуються із бази даних у оперативну пам'ять та для кожної книги присвоюються оцінки на стороні сервера. Кожен користувач має чекати декілька секунд перед тим, як згенерується наступна вибірка рекомендованих товарів. Така затримка не є прийнятною для продакшну, а тому необхідно якимось чином зменшити кількість книжок, які рекомендаційна система має обробити. Для цього можна реалізувати деякі оптимізації алгоритму та кешування. Так, наприклад, якщо користувач перебуває на самому початку стрічки з товарами, то необхідно попередньо відсіяти товари, які гарантовано будуть у кінці вибірки – наприклад, книги з жанрами, які є нецікавими користувачу і які мають низький середній рейтинг. Причому фільтрація має бути реалізована на рівні бази даних. Другим кроком на шляху до оптимізації перфомансу є перенесення асоціативних правил у in-memory кеш рекомендаційної системи. Це також розв'язує структурну проблему зі зберіганням службової інформації у одній базі з даними веб-додатку. Третій необхідний крок – мемоїзація рекомендацій на якийсь період для кожного конкретного користувача. В поточній реалізації кожен перехід на нову сторінку змушує нас перераховувати оцінки по всім книжкам. Це є зайвим, оскільки ця дія ніяк не впливає на порядок рекомендованих товарів, проте потребує значної кількості обчислювальних ресурсів.

Таким чином, більшість змін направлені на архітектуру та на перфоманс усієї системи в цілому. Для того, щоб вносити суттєві зміни у процес генерації рекомендацій, необхідно у першу чергу вдосконалити збір телеметрії з користувачів, мігрувати на мікросервісну архітектуру та провести рефакторинг алгоритмів.

ВИСНОВКИ

Рекомендаційні системи з'явились незадовго після того, як Інтернет перетворився на платформу для веб-додатків. Практично одразу рекомендації стали невід'ємною частиною веб-додатків, оскільки вони дозволяють автоматично створювати контент, цікавий кожному окремому користувачу. За допомогою різних технік персоналізації, рекомендаційні системи здатні максимізувати такі важливі показники, як середня сума чеку та проведений на платформі час, а також зменшити відтік клієнтів. Реалізація рекомендаційної системи повністю залежить від доменної області бізнесу та наявних обчислювальних потужностей.

У роботі ми розглянули ключові техніки персоналізації контенту, а також способи їх комбінування. У результаті аналізу цих технік можна дійти висновку, що гібридні рекомендаційні системи є найбільш оптимальним варіантом для будь-якого бізнесу. Комбінація різних технік дає можливість розв'язати проблеми, присутні у системах, що використовують лише один із алгоритм із сімейства Collaborative Filtering, Knowledge Based та Content-Based Filtering алгоритмів. Важливим досягненням у сфері рекомендаційних систем є моделі із латентними факторами, що дозволяють знайти неочевидні зв'язки між товарами та смаками покупців, а також зменшити розмірність задачі за допомогою факторизації матриці фідбеків.

Також ми розглянули різні підходи, закладені в основу рекомендаційних алгоритмів. На практиці застосовуються як прості статичні алгоритми, так і більш складні методи машинного навчання. Рішення про застосування тих чи інших підходів приймається на основі наявних даних, наявних ресурсів та ступеня персоналізації, прийнятного для бізнесу. Важливо підкреслити, що у загальному випадку складніші методи не є кращими за статистичні, і оптимальним рішенням є їх комбінація.

Для підтвердження цієї гіпотези було створено MVP рекомендаційної системи для інтернет-магазину книжок із Ensemble дизайном, яка використовує статистичні методи для Content-Based частини та методи машинного навчання для Collaborative Filtering частини. Для досягнення оптимального балансу між рівнем персоналізації та рівнем приємних сюрпризів у стрічці, ми об'єднали Content-Based та Collaborative Filtering компоненти у Weighted гібридну рекомендаційну систему. Крім того, ми використали статистику по рейтингам для зменшення упередженості Content-Based компоненти результуючої рекомендаційної системи. Створена система розв'язала поставлену задачу і показала адекватний результат для різних користувачів. У той же час, система не є надто складною і не потребуватиме великої кількості ресурсів і вона може бути використана як генератор рекомендацій для малого бізнесу. Зазначимо, що якість рекомендацій суттєво збільшиться, якщо використовувати також неявні рейтинги та провести модифікації, запропоновані в розділі 3.3.

Таким чином, рекомендаційні системи є доволі широкою і складною темою, і багато аспектів досі є недослідженими. Так, для більш точних рекомендацій потрібно збирати і розмічувати великі обсяги даних. Крім того, проблеми холодного старту та диверсифікації неможливо розв'язати без створення гібридних рекомендаційних систем.

Наостанок хотілось би зазначити іще декілька соціальних проблем, які створюють рекомендаційні системи. Найбільш очевидною є витік або продаж особистих даних, які становлять профіль користувача. Порушення приватності є злочином, тому компанії, які використовують особисті даних, повинні більше інвестувати у інформаційну безпеку. Іншою, менш очевидною проблемою, є створення "інформаційних бульбашок" та психологічні маніпуляції. Оскільки рекомендації часто базуються на попередніх діях користувача, рекомендаційна система може обмежити доступ до різноманітної інформації, що може вплинути на розвиток людини, її знання та світогляд.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Elaine Rich, "User Modeling via Stereotypes", 1979: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog0304_3
2. Netflix Research: Recommendation, 2019: https://www.youtube.com/watch?v=f8OK1HBEgn0
3. Shuyu Luo, "Introduction to Recommender Systems", Towards Data Science, https://towardsdatascience.com/intro-to-recommender-system-collaborative-filtering-64a238194a26
4. Kevin Luk, "Introduction to TWO approaches of Content-Based Recommendation System", Towards Data Science, https://towardsdatascience.com/introduction-to-two-approaches-of-content-based-recommendation-system-fc797460c18c
5. Charu C. Aggarwal, Recommender Systems: the Textbook, Content-Based Recommender Systems, c 140.
6. "Knowledge-based recommender system", Wikipedia, https://en.wikipedia.org/wiki/Knowledge-based_recommender_system
7. Barry Smyth. "Case-based recommender systems", https://www.cs.auckland.ac.nz/~ian/CBR/recommenders.pdf
8. Burke, R. "Hybrid Recommender Systems: Survey and Experiments": https://pzs.dstu.dp.ua/DataMining/recom/bibl/Hybrid_Recommender_Systems_Survey_and_Experiments.pdf
9. Jeffrey Chiang, AnalyticsVidhya, "7 Types of Hybrid Recommendation System": https://medium.com/analytics-vidhya/7-types-of-hybrid-recommendation-system-3e4f78266ad8

10. Charu C. Aggarwal, Recommender Systems: the Textbook, Ensemble-Based and Hybrid Recommender Systems, c 201-203.
11. Denise Chen, "Recommender System – singular value decomposition (SVD) & truncated SVD", Towards Data Science https://towardsdatascience.com/recommender-system-singular-value-decomposition-svd-truncated-svd-97096338f361
12. Adrain Tam, "Using Singular Value Decomposition to Build a Recommender System", Machine Learning Mastery: https://machinelearningmastery.com/using-singular-value-decomposition-to-build-a-recommender-system/
13. Charu C. Aggarwal, Recommender Systems: the Textbook, Singular Value Decomposition, c 113-115.
14. Ben Lutkevich, "association rules", Techtarget: https://www.techtarget.com/searchbusinessanalytics/definition/association-rules-in-data-mining
15. Anisha Garg, "Complete guide to association rules", Towards Data Science, 2018: https://towardsdatascience.com/association-rules-2-aa9a77241654
16. Bookstore dataset, Github, https://github.com/avg-gh-enjoyer/bookstore-dataset

ДОДАТОК А

Алгоритм побудови асоціативних правил

1. association.service.ts

```
import { Injectable } from '@nestjs/common';
import { ReceiptService } from '../receipt/receipt.service';
import * as _ from 'lodash';
import { BookService } from '../book/book.service';
import { Book } from '../book/book.entity';
import { Rule } from "./apriory.rule.dto";
import { AssociationMapper } from "./association.mapper";
import { Repository } from "typeorm";
import { BookAssociation } from "./book.association.entity";
import { InjectRepository } from "@nestjs/typeorm";


@Injectable()
export class AssociationService {
  #stepToMinSupport = {
    1: 0.0025,
    2: 0.0003,
    3: 0.00003,
    4: 0.00003,
  };
  #minConfidence = 0.1;

  constructor(
```

```
  private readonly receiptService: ReceiptService,
  private readonly bookService: BookService,
  @InjectRepository(BookAssociation)
  private readonly associationRepository: Repository<BookAssociation>
) {}

#getMinSupport(step) {
  return this.#stepToMinSupport[step] || 0;
}

#countSupport(
  transactions: Array<Array<Book>>,
  itemSet: Array<Book>,
): number {
  const idsOfTransactionsWithElementsFromItemSet = transactions
    .map((transaction: Book[]) => transaction.map((book) => book.id))
    .filter((transaction: string[]) =>
      itemSet.every((b) => transaction.includes(b.id)),
    );

  return (
    idsOfTransactionsWithElementsFromItemSet.length / transactions.length
  );
}

#countConfidence(
  transactions: Array<Array<Book>>,
```

```
  antecedent: Array<Book>,
  consequent: Array<Book>,
): number {
  const transactionsOfBookIdsThatContainAntecedent = transactions
    .map((transaction: Book[]) => transaction.map((book) => book.id))
    .filter((transaction: string[]) =>
      antecedent.every((b) => transaction.includes(b.id)),
    );

  const transactionsOfBookIdsThatContainAntecedentAndConsequent =
    transactionsOfBookIdsThatContainAntecedent.filter(
      (transaction: string[]) =>
        consequent.every((b) => transaction.includes(b.id)),
    );

  return (
    transactionsOfBookIdsThatContainAntecedentAndConsequent.length /
    transactionsOfBookIdsThatContainAntecedent.length
  );
}

#getCommonItemsInItemSets(
  itemSet1: Array<Book>,
  itemSet2: Array<Book>,
): Array<Book> {
  const itemSet2Ids = itemSet2.map((book) => book.id);
  return itemSet1.filter((book) => itemSet2Ids.includes(book.id));
```

```
}

#joinItemSet(
  itemSets: Array<Array<Book>>,
  transactions: Array<Array<Book>>,
  step: number,
): Array<Array<Book>> {
  const out = [];
  let i = 0;
  for (const itemSet1 of itemSets) {
    i++;
    for (const itemSet2 of itemSets) {
      if (step > 2) {
        const commonItems = this.#getCommonItemsInItemSets(
          itemSet1,
          itemSet2,
        );
        if (commonItems.length !== step - 2) {
          continue;
        }
      }
      const mergedItemSet: Array<Book> = _.uniqBy(
        [...itemSet1, ...itemSet2],
        (book: Book) => book.id,
      );
      if (mergedItemSet.length < 2) {
        continue;
```

```
      }
      const support = this.#countSupport(transactions, mergedItemSet);
      if (support >= this.#getMinSupport(step)) {
        out.push(mergedItemSet);
      }
    }
    console.log(
      `Joining progress: ${
        (i / itemSets.length) * 100
      }% , joined set length = ${out.length}`,
    );
  }
  return _.uniqWith(out, (itemSet1, itemSet2) => {
    const ids1 = itemSet1.map(b => b.id).sort();
    const ids2 = itemSet2.map(b => b.id).sort();
    return _.isEqual(ids1, ids2)
  });
}

#buildRulesFromItemSet(
  itemSet: Array<Book>,
  transactions: Array<Array<Book>>,
): Array<Rule> {
  const rules: Array<Rule> = [];
  for (let i = 0; i < itemSet.length; i++) {
    const antecedent = itemSet.filter((e, index) => index !== i);
    const consequent = [itemSet[i]];
```

```
    const confidence = this.#countConfidence(
      transactions,
      antecedent,
      consequent,
    );
    const lift = confidence / this.#countSupport(transactions, consequent);
    if (confidence >= this.#minConfidence && confidence < 1 && lift >= 1) {
      const newRule: Rule = {
        antecedent,
        consequent,
        confidence,
        lift,
      };
      rules.push(newRule);
    }
  }
  return rules;
}

#saveItemSets(itemSets: Array<Array<Book>>, step: number) {
  require('fs').writeFileSync(
    `assets/association-rules/item-set-${step}.json`,
    JSON.stringify(itemSets.map(itemSet => {
      return itemSet.map(book => book.id)
    }), null, 2),
  );
}
```

```
#saveNewRules(rules: Array<Array<Rule>>, step: number) {
  require('fs').writeFileSync(
    `assets/association-rules/new-rules-${step}.json`,
    JSON.stringify(rules.map(rules => rules.map((rule) => ({
      antecedent: rule.antecedent.map(book => book.id),
      consequent: rule.consequent.map(book => book.id),
      confidence: rule.confidence,
      lift: rule.lift,
    }))), null, 2),
  );
}

#saveRules(rules: Array<Rule>, step: number | string) {
  require('fs').writeFileSync(
    `assets/association-rules/new-rules-${step}.json`,
    JSON.stringify(rules.map((rule) => ({
      antecedent: rule.antecedent.map(book => book.id),
      consequent: rule.consequent.map(book => book.id),
      confidence: rule.confidence,
      lift: rule.lift,
    })), null, 2),
  );
}

#getAssociationRules(
  books: Book[],
```

```
  transactions: Array<Array<Book>>,
): Array<Rule> {
  const res: Array<Rule> = [];
  let newRules: Array<Array<Rule>> = [];
  let lastItemSets: Array<Array<Book>> = books.map((b) => [b]);
  let step = 1;
  while (step < 4) {
    step++;
    lastItemSets = this.#joinItemSet(lastItemSets, transactions, step);
    this.#saveItemSets(lastItemSets, step);

    newRules = lastItemSets
      .map((itemSet, index) => {
        const rules = this.#buildRulesFromItemSet(itemSet, transactions);
        console.log(
          `Generating rules progress: ${
            (index / lastItemSets.length) * 100
          }%, rules length: ${rules.length}`,
        );
        return rules;
      })
      .filter((rules) => rules.length);

    if (newRules.length === 0) {
      break;
    }
    newRules.forEach((rules) => res.push(...rules));
```

```
      this.#saveNewRules(newRules, step);
    }
    return res;
  }

  async buildAssociationRules() {
    const allReceipts = await this.receiptService.getAllWithEntriesAndBooks();
    const booksAsTransactions: Array<Book[]> = allReceipts
      .map((receipt) => receipt.entries)
      .map((entries) => {
        return entries.map((e) => e.book);
      });
    const allBooks = (await this.bookService.getAllBooks());

    const allBooksWithMinSupport = allBooks
      .filter(
        (b) =>
          this.#countSupport(booksAsTransactions, [b]) > this.#getMinSupport(1),
      );
    const associationRules = this.#getAssociationRules(
      allBooksWithMinSupport,
      booksAsTransactions,
    );
    this.#saveRules(associationRules, 'final');
    const entities = associationRules.map(r =>
AssociationMapper.mapAprioryRuleToEntity(r));
    await this.associationRepository.save(entities);
```

```
  }

  async getByAntecedentIds(antecedentIds): Promise<BookAssociation[]> {
    return this.associationRepository.createQueryBuilder('association')
      .innerJoinAndSelect('association.entries', 'association_entry')
      .innerJoinAndSelect('association_entry.purchasedBook',
'association_entry_purchased_book')
      .innerJoinAndSelect('association_entry.bookThatMightLike',
'association_entry_book_that_might_like')
      .where('association_entry.purchasedBook.id IN(:...ids)', {ids: antecedentIds})
      .getMany();
  }

}
```

ДОДАТОК Б

Алгоритм рекомендаційної системи

1. recommender.system.service.ts

```
import { PaginationParams } from "../common/pagination.params";
import { Injectable } from "@nestjs/common";
import { BookService } from "../book/book.service";
import { ReceiptService } from "../receipt/receipt.service";
import { Book } from "../book/book.entity";
import { AssociationService } from "../association-service/association.service";
import * as _ from "lodash";
import { BookAssociation } from "../association-service/book.association.entity";

@Injectable()
export class RecommenderSystemService {
  #maxRating = 5;

  #ratingCoefficient = 0.3;

  constructor(
    private readonly bookService: BookService,
    private readonly receiptService: ReceiptService,
    private readonly associationService: AssociationService
  ) {
  }

  #getWeights(ids: string[]): Object {
```

```
  const sums = ids.reduce((res, curr) => {
    const existing = res[curr] || 0;
    return {
      ...res,
      [curr]: existing + 1
    };
  }, {});

  for (const id of ids) {
    sums[id] = sums[id] / ids.length;
  }
  return sums;
}

#getScoreByWeightsAndKeys(weights: Object, keys: string[]): number {
  let sum = 0;
  for (const key of keys) {
    const weight = weights[key];
    if (weight) {
      sum += Number(weight);
    }
  }
  return sum / Math.max(keys.length, 1);
}

#getBookGenreScore(
  genreWeights: Object,
```

```
  purchasedBooksGenreIds: Array<string>,
  book: Book
) {
  const booksGenreIds = book.genres.map((genre) => genre.id);
  const commonGenres =
    (_.intersection(
      purchasedBooksGenreIds,
      booksGenreIds
    ) as Array<string>) || [];
  return this.#getScoreByWeightsAndKeys(genreWeights, commonGenres);
}

#getBookAuthorScore(
  authorWeights: Object,
  purchasedBooksAuthorIds: Array<string>,
  book: Book
) {
  const booksAuthorIds = book.authors.map((author) => author.id);
  const commonAuthors =
    (_.intersection(
      purchasedBooksAuthorIds,
      booksAuthorIds
    ) as Array<string>) || [];
  return this.#getScoreByWeightsAndKeys(authorWeights, commonAuthors);
}

#getAverageRatingScore(b: { averageRating?: number }) {
```

```
  if (!b.averageRating || isNaN(Number(b.averageRating))) {
    return 0;
  }
  return (b.averageRating / this.#maxRating) * this.#ratingCoefficient;
}

#getAssociationRuleWithHighestConfidence(
  associationRules: Array<BookAssociation>,
  book: Book
): BookAssociation | undefined {
  const relevantAssociationRules = associationRules
    .filter((rule) =>
      rule.entries.some((entry) => entry.purchasedBook.id === book.id)
    )
    .sort((rule1, rule2) => rule2.confidence - rule1.confidence);
  if (relevantAssociationRules.length) {
    return relevantAssociationRules[0];
  }
  return undefined;
}

#adjustScoreToAverageRatingScore(
  score: number,
  avgRatingScore: number
): number {
  if (avgRatingScore === 0) {
    return score;
```

```
  }
  return (score + avgRatingScore) / 2;
}

async getBooksOrdered(
  userId: string,
  paginationParams: PaginationParams
): Promise<{
  books: Book[];
  totalAmount: number;
}> {
  const receipts =
    await this.receiptService.getByUserIdWithEntriesAndBooksWithAuthorsAndGenre(
      userId
    );
  const purchasedBooks = receipts
    .flatMap((receipt) => receipt.entries)
    .map((entry) => entry.book);

  const purchasedBooksIds = purchasedBooks.map((b) => b.id);
  const purchasedBooksGenreIds = purchasedBooks
    .flatMap((b) => b.genres)
    .map((genre) => genre.id);
  const genreWeights = this.#getWeights(purchasedBooksGenreIds);

  const purchasedBooksAuthorIds = purchasedBooks
    .flatMap((b) => b.authors)
```

```
    .map((author) => author.id);
  const authorWeights = this.#getWeights(purchasedBooksAuthorIds);

  const allBooks = await this.bookService.getAllBooksWithGenresAndAuthors();
  const associationRules = await this.associationService.getByAntecedentIds(
    purchasedBooksIds
  );

  const bookIdToLikeScore = {};

  for (const b of allBooks) {
    const associationRule = this.#getAssociationRuleWithHighestConfidence(
      associationRules,
      b
    );
    if (associationRule) {
      const entryWithConsequent = associationRule.entries.find(e =>
Boolean(e.bookThatMightLike));
      const consequentId = entryWithConsequent.bookThatMightLike.id;
      bookIdToLikeScore[consequentId] = associationRule.confidence;
    }
  }

  for (const b of allBooks) {
    const existingScore = bookIdToLikeScore[b.id] || 0;
    const genreScore = this.#getBookGenreScore(
      genreWeights,
```

```
    purchasedBooksGenreIds,
    b
  );
  const authorScore = this.#getBookAuthorScore(
    authorWeights,
    purchasedBooksAuthorIds,
    b
  );
  const avgRatingScore = this.#getAverageRatingScore(b);

  bookIdToLikeScore[b.id] =
    this.#adjustScoreToAverageRatingScore(genreScore, avgRatingScore) +
    this.#adjustScoreToAverageRatingScore(authorScore, avgRatingScore) +
    existingScore +
    avgRatingScore;
}

const recommenderComparator = (b1: Book, b2: Book): number => {
  return bookIdToLikeScore[b2.id] - bookIdToLikeScore[b1.id];
};

const from = paginationParams.skip;
const to = paginationParams.skip + paginationParams.limit;

return {
  books: allBooks
    .sort(recommenderComparator)
```

```
        .filter((book, index) => index >= from && index < to),
      totalAmount: allBooks.length
    };
  }
}
```