

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Рекомендаційна система для платформи з продажу
цифрових товарів»**

Виконав студент групи ПЗм-23-1
ЄВДОКИМОВ Владислав Вікторович

Керівник роботи:
МОРОЗОВ Андрій Васильович

Рецензент:
ЄФРЕМОВ Юрій Миколайович

Житомир – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

зав. кафедри інженерії
програмного забезпечення
Тетяна ВАКАЛЮК
«01» жовтня 2024 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ЄВДОКИМОВ Владислав Вікторович**

Керівник роботи: **МОРОЗОВ Андрій Васильович**

Тема роботи: **«Рекомендаційна система для платформи з продажу цифрових товарів»**,

затверджена Наказом закладу вищої освіти від **«01» жовтня 2024 р.**
№ 520/с

Вихідні дані для роботи: **Предмет дослідження – методи, алгоритми та технології розробки та впровадження рекомендаційних систем для платформ електронної комерції, зокрема для продажу цифрових товарів.**

Об'єкт дослідження – рекомендаційна система для автоматизації персоналізованих пропозицій на основі аналізу даних, її структура, компоненти, механізми функціонування.

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Морозов А.В.	01.10.2024р.	01.10.2024р.
2	Морозов А.В.	02.10.2024р.	02.10.2024р.
3	Морозов А.В.	03.10.2024р.	03.10.2024р.

РЕФЕРАТ

Кваліфікаційна робота освітнього ступеня «магістр» присвячена розробці та впровадженню рекомендаційної системи для автоматизації персоналізованих пропозицій на основі аналізу даних.

Пояснювальна записка складається зі вступу, трьох розділів та списку використаної літератури. Обсяг роботи становить 78 сторінок, включаючи 59 ілюстрацій, 8 таблиць, 1 додаток та 40 посилань на зовнішні джерела.

Метою роботи є створення ефективної рекомендаційної системи на основі алгоритмів машинного навчання, яка забезпечує високу точність і різноманітність рекомендацій. У роботі досліджено актуальні алгоритми рекомендацій, такі як моделі на основі контенту та гібридні підходи, із подальшою їх адаптацією під специфічні задачі.

Проаналізовано сучасні підходи до нормалізації даних, які дозволяють уніфікувати показники взаємодії користувачів із системою, включаючи кількість унікальних користувачів, продажі та кліки. Реалізовані алгоритми протестовано, проведено порівняння їхньої точності, покриття та здатності забезпечувати новизну у рекомендаціях.

Робота включає тестування створеної системи на практичних даних, аналіз її ефективності та формулювання висновків щодо можливих шляхів удосконалення.

КЛЮЧОВІ СЛОВА: РЕКОМЕНДАЦІЙНА СИСТЕМА, МАШИННЕ НАВЧАННЯ, АЛГОРИТМ НА ОСНОВІ КОНТЕНТУ, ГІБРИДНА МОДЕЛЬ, АНАЛІЗ ДАНИХ.

ABSTRACT

The qualification work of the educational degree "master" is devoted to the development and implementation of a recommender system for the automation of personalized offers based on data analysis.

The explanatory note consists of an introduction, three chapters and a list of used literature. The volume of work is 78 pages, including 59 illustrations, 8 tables, 1 appendices and 40 references to external sources.

The goal of the work is to create an effective recommender system based on machine learning algorithms, which provides high accuracy and variety of recommendations. The work examines current recommendation algorithms, such as content-based models and hybrid approaches, with their subsequent adaptation to specific tasks.

Modern approaches to data normalization, which allow unifying indicators of user interaction with the system, including the number of unique users, sales and clicks, are analyzed. The implemented algorithms were tested, and their accuracy, coverage, and ability to provide novelty in recommendations were compared.

The work includes testing the created system on practical data, analyzing its effectiveness and formulating conclusions regarding possible ways of improvement.

KEYWORDS: RECOMMENDER SYSTEM, MACHINE LEARNING, CONTENT-BASED ALGORITHM, HYBRID MODEL, DATA ANALYSIS.

ЗМІСТ

РЕФЕРАТ	5
ABSTRACT	6
ЗМІСТ	7
ВСТУП	6
РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ.....	8
1.1 Наукова концепція та формалізовані моделі, методи та алгоритми	8
1.2 Завдання, що вирішуються із використанням даних наукових розробок	16
1.3 Результати впровадження відомих алгоритмів у програмні продукти	19
1.4 Сучасні проблеми та перспективи розвитку інформаційних технологій	20
1.5 Постановка задачі для проведення дослідження	22
1.6 Обґрунтування технологій та засобів розробки програмного забезпечення.....	23
Висновки до розділу 1	24
РОЗДІЛ 2. ПРАКТИЧНА РЕАЛІЗАЦІЯ АЛГОРИТМІВ ІНТЕЛЕКТУАЛЬНОГО АНАЛІЗУ ДАНИХ	26
2.1 Характеристика джерела даних для проведення аналізу.....	26
2.2 Побудова моделі аналізу і практична реалізація	30
Висновки до розділу 2	55
РОЗДІЛ 3. ОРГАНІЗАЦІЯ ЗАСОБІВ ВІЗУАЛІЗАЦІЇ ТА АНАЛІЗУ ДАНИХ.....	56
3.1 Результати проведеного аналізу в таблицях діаграмах і графіках	56
3.2 Аналіз результатів	69
Висновки до розділу 3	72
ВИСНОВКИ	74
ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ	75

ВСТУП

Актуальність теми. Рекомендаційні системи з використанням штучного інтелекту (ШІ) набувають все більшого значення в сучасному цифровому світі, відіграючи ключову роль у покращенні користувацького досвіду та підвищенні ефективності бізнес-процесів. Ці системи дозволяють надавати персоналізовані рекомендації, що сприяє збільшенню продажів, залученості користувачів та їх задоволеності.

Використання ШІ для створення рекомендаційних систем забезпечує обробку великих обсягів даних у режимі реального часу, дозволяючи враховувати індивідуальні переваги користувачів та їх поведінкові особливості. Це дає можливість пропонувати найрелевантніші продукти чи послуги, що сприяє підвищенню конверсії та лояльності клієнтів.

Такі системи можуть застосовуватись у різних сферах, включаючи електронну комерцію, медіа, соціальні мережі, онлайн-навчання та багато інших. Вони використовують передові методи машинного навчання та аналізу даних, що дозволяє постійно вдосконалювати алгоритми та підвищувати точність рекомендацій.

Рекомендаційні системи з використанням ШІ також сприяють оптимізації бізнес-процесів, надаючи аналітичні інструменти для розуміння поведінки користувачів та прогнозування трендів. Це дозволяє бізнесам приймати обґрунтовані рішення та розробляти ефективні стратегії розвитку.

Загалом, рекомендаційні системи з використанням штучного інтелекту є важливою складовою цифрового світу, сприяючи підвищенню якості обслуговування користувачів та оптимізації бізнес-процесів.

Метою кваліфікаційної роботи є розробка моделей рекомендаційних систем з використанням штучного інтелекту та реалізація засобів обробки даних для покращення якості рекомендацій.

Завданням кваліфікаційної роботи є:

- моделювання рекомендаційних алгоритмів;
- розробка методів збору та обробки даних;
- впровадження алгоритмів машинного навчання для створення рекомендацій;

- аналіз ефективності рекомендаційних систем.

Предметом дослідження є методи та алгоритми розробки рекомендаційних систем на основі штучного інтелекту.

Об'єктом дослідження є інформаційні процеси та засоби обробки даних для створення рекомендаційних систем.

Практичним завданням є розробка рекомендаційної системи для платформи storez.me, яка займається реалізацією цифрових товарів. Система розробляється з урахуванням специфіки роботи платформи, її асортименту та метрик, що використовуються для аналізу, з метою підвищення конверсій серед клієнтів та користувачів. У процесі роботи проводяться експерименти щодо пошуку слабких місць системи, що дозволить підвищити її рентабельність, адаптувати до реальних умов і видавати точні рекомендації. Розробка виконана з урахуванням сучасних підходів у сфері штучного інтелекту та обробки даних і спрямована на покращення взаємодії користувачів з платформою та підвищення обсягу продажу.

Науковою новизною є моделювання рекомендаційних алгоритмів та методів збору та обробки даних, які можна використовувати для створення рекомендаційних систем.

В процесі виконання кваліфікаційної роботи були використані наступні методи дослідження: аналітичні, математичні, алгоритмічні, статистичні та методи машинного навчання.

Апробація роботи. За результатами дослідження подано до друку наукову статтю у фахове видання:

1. Євдокимов В. В., Морозов А.В., Вакалюк Т. А. Формалізовані методи та алгоритми для створення рекомендаційних систем. Актуальні проблеми автоматизації та інформаційних технологій. (подано до друку).

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ

1.1 Наукова концепція та формалізовані моделі, методи та алгоритми

Рекомендаційна система, яка працює на основі штучного інтелекту, – це алгоритм машинного навчання, який навчається ранжувати або оцінювати продукти чи користувачів. Вона розроблена для передбачення оцінок, які користувач може виставити конкретному елементу, і повертає ці прогнози користувачеві у вигляді впорядкованого списку. Ця технологія використовується багатьма популярними компаніями, такими як Google, Amazon і Netflix, для підвищення залучення користувачів до своїх платформ. Наприклад, Spotify може рекомендувати пісні, схожі на ті, які ви раніше слухали або вподобали, щоб зберегти вас використовувати їхню платформу для прослуховування музики. Amazon може рекомендувати продукти користувачам на основі даних, які вони зібрали про конкретного користувача [1].

Хоча системи рекомендацій дуже корисні, їх часто сприймають як "чорні скрині", оскільки моделі, створені цими компаніями, не завжди легко інтерпретуються. Користувачі можуть не розуміти, чому робиться певна рекомендація, але часто вони вважають ці рекомендації корисними та відповідними їхнім потребам та інтересам. Результати часто стосуються того, що користувач потребує чи хоче, але може бути несвідомий цього до тих пір, поки це йому не рекомендують.

Є багато способів створення рекомендаційної системи, і підходи можуть варіюватися від алгоритмічних і формульних до зосереджених на моделюванні. Ці підходи включають PageRank, колаборативне фільтрування, заснований на контенті та передбаченні зв'язків. Однак важливо відзначити, що складність не завжди перекладається в хорошу ефективність, і часто прості рішення та впровадження дають найкращі результати. Наприклад, великі компанії, такі як Reddit, Hacker News та Google, використовують прості формульні втілення рекомендаційних систем для просування контенту на своїх платформах [1].

Визначення того, що визначає хорошу рекомендацію, є завданням, з яким багато компаній все ще борються. Визначення "добрих" рекомендацій допомагає оцінити ефективність побудованої рекомендаційної системи. Якість рекомендації може бути оцінена за допомогою різних тактик, які вимірюють охоплення та точність. Точність - це частка правильних рекомендацій відносно загальної кількості можливих рекомендацій, тоді як охоплення вимірює частку об'єктів у просторі пошуку, яку система може рекомендувати. Метод оцінювання рекомендації повністю залежить від набору даних і підходу, використаного для генерації рекомендації.

Системи рекомендацій мають кілька концептуальних схожостей з проблемою класифікації та моделювання регресії. У ідеальній ситуації компанії хотіли б дізнатися, як реальні користувачі реагують на рекомендації та відстежувати метрики навколо користувача для вдосконалення своїх рекомендацій. Однак це часто важко досягти.

Розглянемо основні типи рекомендаційних систем на основі штучного інтелекту.

Колаборативне фільтрування – це техніка, що використовується в рекомендаційних системах для прогнозування інтересів та вподобань користувача на основі даних і закономірностей від багатьох користувачів. Основний принцип колаборативного фільтрування полягає в тому, що якщо два користувачі мають схожі смаки щодо одного продукту, то вони також імовірно мають схожі смаки до інших продуктів. Існують два основних підходи до колаборативного фільтрування: основані на пам'яті та модельні [1].

Підхід колаборативного фільтрування походить від людської поведінки – люди, які шукають цікаву річ, про яку вони мало знають, наприклад, фільм для оренди у відеопрокаті, зазвичай покладаються на друзів, які можуть порекомендувати їм товари, які вони пробували і які їм сподобалися. Людина, що звертається за порадою, використовує (невелику) спільноту друзів, які знають її смаки і можуть зробити хороші прогнози щодо того, чи сподобається їй певний товар. Однак в Інтернеті доступна більша спільнота, яка може порекомендувати

товари нашому користувачу, але ці люди мало або зовсім не знають один одного. Концептуально, мета системи колективного фільтрування полягає в тому, щоб визначити тих користувачів, чиї смаки в товарах є передбачуваними для смаків певної особи (зазвичай її називають "сусідством"), і використовувати їхні рекомендації для складання списку товарів, які можуть бути їй цікаві [2].

Основані на пам'яті також відомі як колаборативне фільтрування за сусідством і використовують рейтинги комбінацій користувач-продукт для прогнозування їх вподобань на основі їх оточення. Основані на користувачах колаборативні системи рекомендують продукти користувачеві на основі вподобань схожих користувачів, тоді як основані на продуктах колаборативні системи рекомендують продукти на основі схожості між елементами, розрахованою за оцінками користувачів для цих елементів [1, 3].

Модельні – використовують прогностичні моделі, що включають машинне навчання для параметризації рис, пов'язаних із набором даних, як вхідні дані моделі. Це допомагає вирішити проблему, пов'язану з оптимізацією. Модельні підходи включають дерева рішень, правила та моделі латентних факторів [1, 4].

Моделі колаборативного фільтрування є перевагами через їхню легку реалізацію, високий рівень охоплення та можливість виявлення тонких характеристик без потреби в знанні вмісту елемента. Деякі приклади алгоритмів колаборативного фільтрування включають рекомендації відео на YouTube на основі користувачів, які підписалися чи дивилися схожі відео, та рекомендації курсів на CourseEra, що ґрунтуються на інших особах, які завершили існуючі курси, які користувач вже пройшов [1].

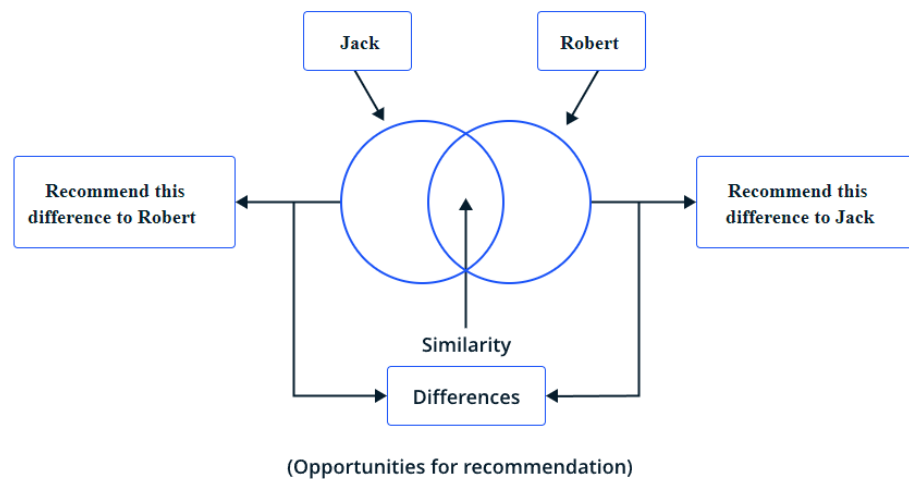


Рис. 1.1. Приклад моделі колаборативного фільтрування

На діаграмі показані два різних користувачі та їхні інтереси, разом із схожістю у їхніх смаках. Виявлено, що і Джек, і Роберт мають схожі смаки, тому інтереси Джека рекомендуються Роберту і навпаки.

Система на основі контенту – це система, яка генерує рекомендації на основі вподобань та профілю користувача, співставляючи їх з товарами, які вони раніше вподобали. Замість використання рейтингів між цільовим користувачем та іншими користувачами, моделі на основі контенту фокусуються на рейтингах, які надає цільовий користувач. Ці моделі встановлюють рівень схожості між елементами на основі атрибутів елементів, які користувач вподобав [1, 5].

Для створення системи на основі контенту потрібне потужне джерело даних на рівні елемента, пов'язане з атрибутами елемента, такими як ціна, рік публікації і т.д., разом із деяким зворотнім зв'язком користувача щодо елемента, який може бути як імпліцитним, так і експліцитним. Моделі на основі контенту особливо ефективні, коли недостатньо даних про рейтинги, оскільки вони можуть використовувати рейтинги та атрибути елементів для генерації рекомендацій.

Приклади систем на основі контенту включають продуктовий стрічку Amazon, яка рекомендує продукти, схожі на ті, які користувач раніше придбавав, та рекомендації музики Spotify. Деякі компанії, такі як Hacker Rank та Reddit, також використовують алгоритмічні підходи для рекомендацій нових публікацій користувачам на основі таких факторів, як час публікації, кількість лайків,

дизлайків та коментарів, які можуть бути враховані в формулі для генерації оцінки для публікації та, отже, рекомендації.

One Plus 7	One Plus 7T	One Plus 7T Pro
8GB RAM 256GB ROM	8GB RAM 128GB ROM	8GB RAM 256GB ROM
48 MP + 5 MP 16 MP Front Camera	48 MP + 12 MP + 16 MP 16 MP Front Camera	48 MP + 8 MP + 16 MP 16 MP Dual Front Camera

Рис. 1.2. Приклад підбору товарів системи на основі контенту

На вище наведеному зображенні представлені різні моделі телефону One Plus. Якщо особа шукає мобільний телефон One Plus 7, тоді One Plus 7T та One Plus 7T Pro рекомендуються їй.

Гібридні системи рекомендацій розроблені для подолання обмежень індивідуальних систем рекомендацій за допомогою поєднання декількох джерел даних, що може бути досягнуто двома різними дизайнами: паралельним та послідовним. У паралельному дизайні використовуються кілька систем рекомендацій паралельно для генерації рекомендацій, а їхні виходи комбінуються для отримання кінцевого результату. У послідовному дизайні використовується один рекомендаційний двигун, і його вихід передається наступному рекомендактору в послідовності [1, 6].

Гібридні системи мають кілька переваг, включаючи збільшену надійність та персоналізацію рекомендацій для користувачів. Шляхом поєднання різних моделей гібридні системи можуть компенсувати слабкі сторони окремих моделей, що призводить до більш точних та різноманітних рекомендацій. Netflix - відомий приклад компанії, яка використовує гібридну систему рекомендацій, поєднуючи колаборативне фільтрування (на основі поведінки користувачів) і фільтрування на основі контенту (на основі характеристик елементів) для надання більш релевантних рекомендацій фільмів своїм користувачам.

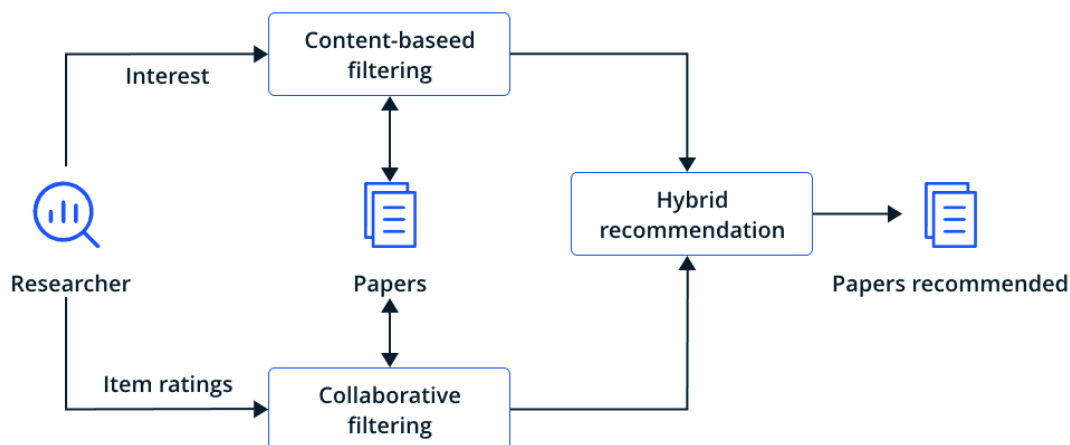


Рис. 1.3. Приклад гібридної системи рекомендацій

Система на основі знань у рекомендаційній системі генерує рекомендації на основі потреб користувача та експертизи в галузі. Це включає визначення правил, які встановлюють контекст для кожної рекомендації, такі як критерії, коли конкретний продукт або послуга буде корисними для користувача. На відміну від підходу на основі контенту, ці правила не обов'язково ґрунтуються на історії взаємодії користувача, але можуть включати інші експертні відомості або атрибути товарів та послуг для клієнтів [1, 7].

Перевагою системи на основі знань є те, що рекомендації можна легко пояснити, що полегшує користувачам розуміння причин конкретної рекомендації. Однак побудова такого типу структури може бути витратною і краще підходить для складних галузей, де товари придбаються рідко, а дані можуть бути обмеженими. Однією з переваг цього підходу є те, що він не стикається з проблемами "холодного старту", як інші методи.

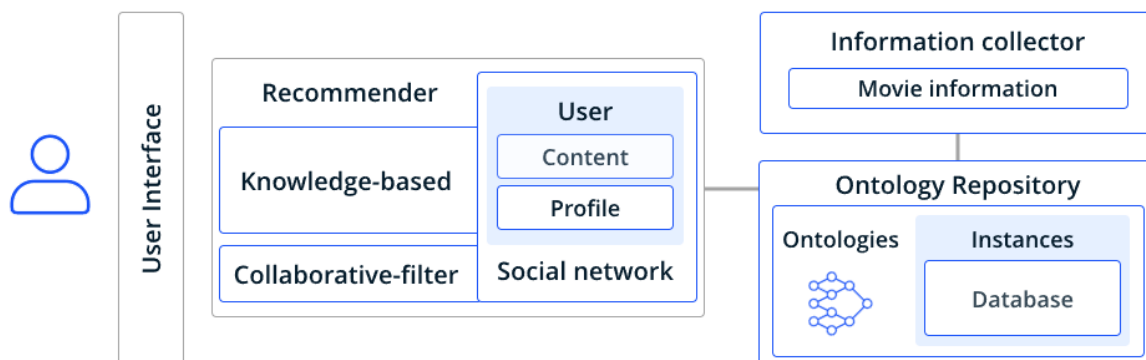


Рис. 1.4. Приклад системи на основі знань

Система рекомендацій використовує комбінацію технологій машинного навчання та даних для створення персоналізованих рекомендацій. Дані виступають фундаментом для рекомендаційної системи, надаючи необхідну інформацію для виявлення патернів. Чим більше даних, тим більше можливостей системи надавати прибуткові та актуальні рекомендації. Процес роботи рекомендаційної системи складається з чотирьох основних кроків.

Для створення рекомендаційної системи першим кроком є збір даних. Це може включати як явні дані, такі як оцінки та коментарі від користувачів, так і неявні дані, такі як історія замовлень, історія повернень, події корзини, перегляди сторінок і журнали пошуку. Ці дані збираються для кожного користувача, який відвідує сайт.

Поведінкові дані досить просто зібрати, оскільки вони включають в себе реєстрацію дій користувачів на сайті без додаткового введення. Проте аналіз цих даних може бути складним, оскільки може знадобитися фільтрація менш важливих журналів. Оскільки у кожного користувача є свої власні унікальні вподобання, їхні набори даних також будуть відмінними. З введенням більше даних в систему, вона стає розумнішою та краще надає актуальні рекомендації. Та ж сама принцип використовується системою рекомендацій Amazon, яка пропонує користувачам продукти на основі їх історії переглядів і покупок, використовуючи функції "Часто купують разом" і "Рекомендовано для вас".

У системі рекомендацій якість рекомендацій покращується зі збільшенням обсягу даних. Відповідно, проекти рекомендацій часто переростають у проекти великих даних через велику кількість даних.

Тип даних, що використовується для створення рекомендацій, визначає відповідний тип зберігання. Вибір може включати базу даних NoSQL, стандартну базу даних SQL або систему об'єктного сховища. Вибір здебільшого залежить від таких факторів, як тип збираємих даних, легкість впровадження, обсяг зберігання, інтеграція з іншими компонентами та переносимість даних.

При збереженні оцінок або коментарів користувачів використання масштабованої та керованої бази даних є важливим для мінімізації кількості

завдань та акценту на поліпшенні рекомендацій. Ця гнучкість особливо корисна для організацій з унікальними потребами у зберіганні даних.

Дані фільтруються за допомогою різних методів аналізу для знаходження елементів з аналогічними даними взаємодії користувачів у системі рекомендацій. Залежно від вимог можуть використовуватися різні методи аналізу.

Наприклад, якщо користувачеві необхідно надавати миттєві рекомендації під час перегляду продукту, вимагається більш гнучкий аналіз. Системи реального часу можуть обробляти дані по мірі їх створення і використовуються для надання рекомендацій у реальному часі. Інструменти, які можуть обробляти та аналізувати потоки подій, зазвичай включаються в системи реального часу.

З іншого боку, пакетний аналіз передбачає періодичну обробку даних. Цей підхід потребує достатньої кількості даних для того, щоб аналіз був значущим, наприклад, обсягу щоденного обсягу продажів.

Ще один підхід – це аналіз псевдочасу, який дозволяє швидко збирати дані, щоб аналітика могла оновлюватися кожні кілька хвилин чи секунд. Система аналізу псевдочасу добре підходить для надання рекомендацій під час того ж сеансу перегляду.

У системі рекомендацій фільтрація є важливим етапом, який допомагає відібрати необхідні дані для надання рекомендацій користувачеві. Для цього ми повинні обрати алгоритм, такий як фільтрація на основі контенту, фільтрація на основі кластерів або колаборативна фільтрація, який найкраще підходить для рекомендаційної системи.

Фільтрація на основі контенту рекомендує продукти зі схожими характеристиками до тих, які користувачу подобаються чи переглянуті. Фільтрація на основі кластерів рекомендує продукти, які гармоніюють між собою, незалежно від поведінки інших користувачів. Колаборативна фільтрація, навпаки, робить прогнози на основі смаків користувача та передбачає, що двом користувачам, які минулого разу сподобалися одні й ті ж продукти, сподобаються і в майбутньому.

Для подання даних про оцінки чи взаємодії ми можемо використовувати матриці з продуктами та користувачами як вимірами. Потім ми можемо використовувати алгоритми, такі як K-Nearest, коефіцієнт Жаккара, алгоритм Дейкстри чи косинусна схожість для відносин даних людей на основі оцінок чи продуктів, з якими вони взаємодіють.

Після фільтрації та використання алгоритму рекомендації надаються користувачеві в залежності від часового аспекту рекомендації, чи це рекомендація в реальному часі чи емейл, що буде відправлений пізніше.

1.2 Завдання, що вирішуються із використанням даних наукових розробок

Персоналізація є важливою для підвищення задоволення клієнтів, а рекомендаційна система є ключовим інструментом для досягнення цієї мети. Дослідження компанії Epsilon підкреслює, що 80% споживачів віддають перевагу покупкам від бренду, який надає персоналізовані враження, що свідчить про те, що персоналізація може бути значущою конкурентною перевагою для бізнесів, що прагнуть покращити лояльність та задоволення клієнтів [1].

У сучасному світі перенасиченого інформацією споживачі отримують величезну кількість пропозицій та інформації, що робить важким привернення їхньої уваги. Персоналізація може допомогти бізнесам виділитися, надаючи клієнтам відповідні, налаштовані рекомендації, що відповідають їхнім унікальним потребам та вподобанням. Це, в свою чергу, може збільшити взаємодію, збільшити продажі та сприяти довгостроковій лояльності клієнтів. Рекомендаційна система може бути ефективним інструментом для створення персоналізованих вражень для клієнтів. Використовуючи дані про минулі дії та вподобання клієнта, рекомендаційна система може пропонувати продукти чи послуги, які ймовірно сподобаються цій особі, що призводить до більш безперешкодного та приємного клієнтського досвіду, де клієнти відчуються цінованими та оціненими.

Оmnіканальний маркетинг став все більш популярним останнім часом, оскільки споживачі очікують безперешкодного обслуговування на різних каналах. Проте це становить великий виклик для брендів - забезпечення консистентності. За даними Zendesk, 35% клієнтів очікують отримувати обслуговування через кілька каналів.

Система рекомендацій на основі штучного інтелекту може визначити нові стандарти для забезпечення послуг з однаковим досвідом на всіх каналах, включаючи соціальні мережі, веб-сайти та мобільні додатки. Система рекомендацій може гарантувати, що користувачі отримують аналогічний досвід в будь-якому місці, збираючи та використовуючи дані з взаємодії клієнта на різних каналах. Наприклад, якщо клієнт шукає продукт на веб-сайті, система рекомендацій може використовувати ці дані для пропозиції персоналізованих продуктів через електронну пошту чи інші канали, забезпечуючи консистентний досвід.

У цифрову епоху сьогодення надання відповідного контенту стало ключовим для залучення клієнтів. Бренди, які надають персоналізований контент, мають більше шансів привернути та утримати клієнтів. Рекомендаційні системи відіграють важливу роль у досягненні цієї мети, показуючи своїм споживачам настроєний та відповідний контент.

Один із найкращих прикладів бренду, що надає персоналізований контент, - це Medium, онлайн-платформа для публікацій, де користувачі можуть обирати свої улюблені теми, а на основі їх читацької поведінки створюються для них персоналізовані списки контенту. Такий підхід гарантує, що користувачі бачать лише контент, який їх цікавить, що призводить до вищих показників взаємодії та збільшення задоволення користувачів.

Бізнеси можуть пристосовувати свій контент до конкретних інтересів та вподобань кожного користувача за допомогою рекомендацій, що може бути зроблено аналізуючи поведінку користувача та дані, такі як пошукові запити, минулі покупки та історія перегляду. З цією інформацією рекомендаційна

система може надавати персоналізовані рекомендації для контенту, який користувач ймовірно буде сприймати більш активно.

У сучасному світі, де все швидко змінюється, у клієнтів обмежений термін уваги, і неактуальний контент чи рекомендації щодо продуктів можуть швидко призвести до невдоволення та негативного досвіду для клієнта, а також втрати потенційних продажів. Рекомендаційна система на основі штучного інтелекту може допомогти мінімізувати це невдоволення, фільтруючи та відображаючи персоналізований та відповідний контент для клієнтів.

Аналізуючи поведінку та вподобання клієнтів, рекомендаційний система може надавати настроєні рекомендації, які ймовірніше привернуть увагу та зацікавлять клієнтів, покращуючи загальний досвід обслуговування та призводячи до зростання обсягів продажів та доходів для бізнесів. За допомогою рекомендацій на основі штучного інтелекту підприємства можуть уникати невдоволення своїх клієнтів та надавати більш персоналізований та захоплюючий досвід.

З ростом очікувань клієнтів надавати персоналізовані та безперервні враження стає важливішим, ніж будь-коли раніше. Фактично, це стало ключовим визначником лояльності до бренду. Звіт від Microsoft стверджує, що 96% клієнтів вважають обслуговування та досвід клієнта критичними факторами, що впливають на їхню лояльність до бренду [1].

Для задоволення цих очікувань бізнеси можуть використовувати рекомендаційні системи. Аналізуючи поведінку та вподобання клієнтів, ці системи можуть надавати персоналізовані рекомендації продуктів, передбачаючи їхні потреби та бажання. Такий персоналізований підхід допомагає будувати міцніші відносини з клієнтами та підтримує лояльність з часом.

Крім того, рекомендаційні системи можуть допомагати створювати послідовний досвід на всіх точках контакту, від веб-сайту до соціальних мереж та далі. Забезпечуючи персоналізований та відповідний контент, бізнеси можуть

залучати клієнтів на їх улюблених платформах та гарантувати безперервний досвід протягом всього їхнього шляху.

Кожен власник бізнесу, прагне досягти більших продажів, збільшити середню суму замовлення і, в кінцевому рахунку, збільшити прибуток. Одним з ефективних способів досягнення цього є використання потужностей рекомендаційної системи. Показуючи персоналізовані рекомендації продуктів клієнтам, можна збільшити ймовірність покупки. Рекомендаційна система на основі штучного інтелекту також може поліпшити ваші стратегії в продажі та перепродажі, зробивши процес покупки більш гладким і зручним для клієнтів.

1.3 Результати впровадження відомих алгоритмів у програмні продукти

В сучасному бізнес-середовищі, де конкуренція надзвичайно велика, впровадження алгоритмів, зокрема рекомендаційних систем на базі штучного інтелекту, стає стратегічною дією для компаній. Це не тільки сприяє покращенню користувацького досвіду, але й збільшує бізнес-показники, роблячи їх більш конкурентоспроможними в сучасному ринковому середовищі.

Одним із яскравих прикладів успішного впровадження рекомендаційних алгоритмів є Amazon, що вважається одним із лідерів електронної комерції. Amazon вдало інтегрував рекомендаційні алгоритми для підвищення конверсії та обсягу продажів. Шляхом аналізу покупок користувачів, алгоритм генерує персоналізовані рекомендації, що призвели до вражаючого зростання конверсії на 25% та обсягу продажів на 20% протягом першого року впровадження.

Ще однією прикладною ілюстрацією є використання Shopify рекомендаційних алгоритмів для активного продажу, що призводить до зростання середньої суми замовлення на 15%. Система рекомендацій ефективно доповнює замовлення, сприяючи збільшенню обсягів продажів та максимізації виручки.

Netflix, завдяки колаборативному та контент-базованому підходам, досягає позитивних результатів у створенні послідовного досвіду. Рекомендації

фільмів та серіалів стимулюють активність перегляду та сприяють збільшенню кількості підписників на 30% за останні два роки.

Ці числові показники детально свідчать про позитивний вплив впровадження рекомендаційних алгоритмів на ключові параметри бізнесу та користувацький досвід, демонструючи їхню величезну потужність для компаній у сучасному цифровому середовищі.

1.4 Сучасні проблеми та перспективи розвитку інформаційних технологій

Швидкий розвиток інформаційних технологій, зокрема в галузі рекомендаційних систем, створює унікальні можливості та водночас виклики для компаній у сучасному світі. Основні проблеми та перспективи розвитку сучасних рекомендаційних систем є наступними.

Як і всі технології, засновані на штучному інтелекті, системи рекомендацій залежать від даних. Якщо дані низької якості або якщо неможливо правильно їх обробляти та аналізувати, ефективність системи рекомендацій буде значно знижена.

Комплексний огляд актуальних проблем у сфері рекомендаційних систем представлений у роботі [43], де систематизовані основні виклики та напрямки подальших досліджень у цій галузі.

Системи рекомендацій, засновані на глибинному навчанні, можуть мати високі вимоги до обчислювальних ресурсів. Якщо дані, які подаються до моделі, менш точні або не є цінними, результати будуть менш корисними. Тому перед впровадженням системи рекомендацій важливо оцінити здатність бізнесу задовольняти складні вимоги щодо аналітики даних [8].

Чим більше алгоритм знає про користувача, тим точніші будуть рекомендації. Однак багато користувачів не бажають надавати особисту інформацію через нещодавні випадки витоків даних. Без цієї інформації система рекомендацій не зможе працювати ефективно. Тому створення довіри між

бізнесом і користувачами є ключовим фактором для успішної реалізації системи рекомендацій.

Багато бізнесів успішно використовують системи рекомендацій, що відкриває значні можливості. Однак важливо розуміти виклики, що супроводжують цю технологію, щоб максимально ефективно її використовувати [8, 9].

Залежність від даних користувачів має певні недоліки, одним із яких є проблема "холодного старту". Ця проблема виникає, коли до системи додається новий користувач або нові товари в каталог, і, відповідно, алгоритм не може точно передбачити вподобання нового користувача або оцінку нових товарів, що знижує точність рекомендацій.

Однак моделі глибинного навчання здатні оптимізувати кореляції між користувачем і продуктом шляхом аналізу контексту, такого як опис продукту, зображення та поведінка користувачів. Це дозволяє створювати індивідуальні рекомендації для кожного продукту чи клієнта в різних сценаріях, враховуючи всі ці змінні. Оскільки ці моделі не сильно залежать від даних поведінки користувачів, вони є рішенням для проблеми "холодного старту" [8, 10].

Споживачі постійно змінюються як особистості та як клієнти. Постійне відстеження цих змін є важливим аспектом у розробці ефективних рекомендаційних систем. Потужна система рекомендацій повинна бути здатною виявляти зміни або ознаки майбутніх змін у вподобаннях і поведінці користувачів, а також автоматично адаптуватися в реальному часі, щоб надавати актуальні рекомендації [8].

Сучасні тенденції та перспективи розвитку рекомендаційних систем висвітлені в роботі [41], де представлено аналіз минулого, теперішнього та майбутнього таких систем. Важливий внесок у розуміння взаємозв'язку між алгоритмічною складовою та користувацьким досвідом зробили Konstan J.A. і Riedl J. [42], які дослідили вплив рекомендаційних алгоритмів на сприйняття системи користувачами.

Розвиток більш ефективних алгоритмів фільтрування включає в себе оптимізацію алгоритмів колаборативного та контентного фільтрування, а також інтеграцію гібридних моделей. Це спрямовано на поліпшення якості рекомендацій та зменшення ефекту "фільтрованої бульбашки".

Використання технологій NLP може покращити розуміння контексту та семантики, дозволяючи створювати більш зручні та персоналізовані рекомендації на основі текстової інформації [11].

Розробка механізмів для забезпечення прозорості та контролю над використанням особистих даних може зменшити стурбованість користувачів щодо приватності. Розробка етичних стандартів та правил використання даних також може сприяти покращенню сприйняття рекомендаційних систем.

Розробка методів для збору та використання персоналізованих рекомендацій для нових користувачів може знизити проблеми, пов'язані з холодним стартом. Використання контекстуальних даних та залучення користувачів до створення свого профілю може поліпшити адаптацію системи до нових користувачів.

Ці перспективи вказують на те, що розвиток інформаційних технологій в сфері рекомендаційних систем спрямований на подолання поточних викликів та покращення якості обслуговування користувачів.

1.5 Постановка задачі для проведення дослідження

Для забезпечення ефективності та персоналізації користувачів на платформі, яка дозволяє їм створювати магазини для продажу цифрових товарів, необхідно провести дослідження та розробити рекомендаційну. Задачі дослідження включають аналіз вимог та потреб користувачів, вивчення ринкових тенденцій, визначення ключових параметрів для рекомендацій, розробку алгоритмів рекомендацій і тестування системи.

Перша задача полягає у визначенні основних потреб користувачів, які використовують платформу для створення магазинів цифрових товарів. Також

потрібно врахувати особливості платформи та метрики, які можна використовувати в рекомендаційних алгоритмах.

Наступною задачею є встановлення ключових параметрів та факторів, які слід враховувати при створенні рекомендацій для цифрових товарів. Важливо також розглянути специфіку контенту та взаємодії з цифровими товарами.

Розробка алгоритмів рекомендацій є наступним етапом, на якому необхідно створити алгоритми, що забезпечать персоналізовані та точні рекомендації для користувачів платформи, використовуючи сучасні алгоритми.

Останнім етапом є проведення експериментів для перевірки ефективності системи та її адаптації до потреб користувачів. Ці задачі складають комплексний план для дослідження та розробки рекомендаційної системи для платформи магазинів з продажу цифрових товарів.

1.6 Обґрунтування технологій та засобів розробки програмного забезпечення

Для створення початкового датасету буде використовуватися PostgreSQL із підключенням до виробничої бази даних. PostgreSQL дозволяє зберігати великі обсяги даних, такі як унікальні користувачі, кількість продажів, відгуки та інші характеристики продуктів, а також виконувати запити для збору цієї інформації. Це забезпечить ефективне та швидке структурування даних у зручному та стандартизованому форматі, що спростить їх подальше використання для створення і аналізу датасету. Такий підхід забезпечить високу гнучкість при обробці та інтеграції інформації, що дозволить отримати точні результати на етапі побудови моделі та подальшої обробки даних.

Основною метою системи буде надання персоналізованих рекомендацій на основі взаємодії користувачів з продуктами, таких як перегляди, продажі, кліки та відгуки. Рейтинг продукту буде обчислюватися з урахуванням кількості унікальних користувачів, кількості продажів, кількості кліків і оцінок, що надаються користувачами. Ці метрики будуть нормалізуватися для забезпечення справедливого порівняння між різними товарами.

Для побудови рекомендацій буде використовуватися алгоритм K-Nearest Neighbors (KNN), який дозволить знаходити схожі продукти на основі визначених метрик.

Для реалізації алгоритму RBM буде використовуватися TensorFlow, що дозволить створити більш складні моделі рекомендацій, здатні працювати з великими наборами даних і підвищити точність рекомендацій.

Для оцінки ефективності роботи системи будуть використовуватися метрики, такі як RMSE, MAE, Coverage, Diversity і Novelty. Вони дозволять оцінити точність рекомендацій, покриття, різноманітність рекомендацій та їх новизну.

Основною мовою програмування для розробки алгоритмів буде Python. Бібліотеки Pandas та NumPy будуть використовуватися для обробки, агрегації та нормалізації даних, що дозволить забезпечити ефективну обробку даних та спростить формування датасету [13, 14].

Таким чином, використання цього технологічного стеку забезпечить побудову надійної та ефективної системи рекомендацій, яка буде здатна адаптуватися до змін у вподобаннях користувачів, надавати точні та різноманітні рекомендації і значно покращувати досвід користувачів.

Висновки до розділу 1

У рамках даного розділу проведена глибока аналітична робота щодо рекомендаційної системи для платформи з продажу цифрових товарів. Були виявлені проблеми, що виникають при застосуванні певних алгоритмів фільтрації та обмежує різноманітність рекомендацій, дані проблеми були виявлені та проаналізовані. Також було проведено детальний аналіз різноманітних алгоритмів, які використовуються для розробки рекомендаційної системи.

Важливим етапом дослідження була формулювання задачі для проведення подальших наукових досліджень. Зокрема, визначено завдання для рекомендаційної системи, яка дозволяє користувачам здійснювати пошук

магазинів з продажу цифрових товарів. Це включає в себе оптимізацію роботи фільтрації та вибір оптимальних алгоритмів для покращення якості та різноманітності рекомендацій.

Також було проведено обґрунтування вибору технологій та засобів розробки програмного забезпечення для рекомендаційної системи. Враховуючи особливості платформи, яка буде розроблена з використанням PostgreSQL для зберігання даних про продукти та відгуки користувачів, Python для побудови та оптимізації алгоритмів рекомендацій, а також бібліотеки машинного навчання для створення більш складних моделей, обрані технології повністю відповідають вимогам щодо ефективності та масштабованості системи. Постійне удосконалення моделі на основі взаємодії користувачів з продуктами, таких як перегляди, кліки та відгуки, дозволяє забезпечити високий рівень персоналізації рекомендацій, а також можливість інтеграції з іншими бібліотеками для більш точного прогнозування та аналізу. Вибір цих технологій гарантує, що система буде здатна обробляти великі обсяги даних, підтримувати складні запити та забезпечувати високий рівень адаптивності до змін у вподобаннях користувачів.

Також були проведені дослідження та розроблені рекомендаційні алгоритми сприятимуть поліпшенню користувацького досвіду, збільшенню обсягів продаж та загальному розвитку платформи для продажу цифрових товарів. Впровадження рекомендаційної системи в бізнес-процеси може виявитися ключовим стратегічним кроком для розвитку платформи, що збільшить конкурентоспроможність та задоволеність клієнтів.

РОЗДІЛ 2. ПРАКТИЧНА РЕАЛІЗАЦІЯ АЛГОРИТМІВ ІНТЕЛЕКТУАЛЬНОГО АНАЛІЗУ ДАНИХ

2.1 Характеристика джерела даних для проведення аналізу

У даній роботі використовується власноруч сформований датасет, розроблений у контексті електронної комерції для платформи з продажу цифрових товарів. Цей датасет є основою для проведення досліджень у сфері рекомендаційних систем та аналізу взаємодії користувачів із платформою, зокрема їхньої поведінки, уподобань та способів взаємодії з товарами. Основна мета формування такого датасету полягає в моделюванні реальних умов роботи платформи, що дозволяє максимально наблизити результати аналізу до реальних бізнес-сценаріїв [15].

Датасет включає комплексну інформацію про товари, користувачів і їх взаємодію. До основних даних належать:

- Унікальний ідентифікатор товару (id), який дозволяє чітко ідентифікувати кожен продукт у базі.
- Назва продукту (name), яка відображає основну сутність товару, дозволяючи здійснювати аналіз за ключовими словами.
- Кількість унікальних користувачів (uniqUsers), які взаємодіяли з товаром, що може бути корисним для оцінки популярності продукту.
- Кількість продажів (sold), яка демонструє ефективність товару на ринку.
- Кліки користувачів (userClicks), що показують рівень інтересу до продукту, навіть якщо його не було придбано.

Крім того, у датасеті містяться дані про відгуки користувачів, які є багатим джерелом інформації для аналізу настроїв, оцінок і динаміки взаємодії. Відгуки включають текстові коментарі, рейтингові оцінки (від 0 до 5) та часові мітки (createdAt), що дозволяє досліджувати тенденції в часі. Ці дані також використовуються для моделювання рейтингових систем та побудови прогнозів.

Особливістю цього датасету є його змішаний характер: частина даних зібрана з реальних джерел, а частина — згенерована для відтворення сценаріїв, що можуть виникати в роботі платформи. Наприклад, кількість відгуків для кожного продукту було штучно збільшено до 30 записів із урахуванням реальних метрик, таких як кількість продажів та кліків. Завдяки цьому забезпечується необхідна варіативність даних для проведення експериментів.

Ще одним важливим аспектом є нормалізація даних, що дозволяє забезпечити збалансований аналіз усіх основних метрик: кількості унікальних користувачів, кліків, продажів і рейтингових оцінок. Така структура робить датасет універсальним інструментом для побудови рекомендаційних систем, спрямованих на підвищення ефективності продажів цифрових товарів, а також для аналізу поведінки користувачів у різних сегментах.

Даний датасет створено з урахуванням специфіки роботи платформ електронної комерції, що дозволяє проводити глибокий аналіз ефективності товарів, взаємодії користувачів та побудови рекомендаційних систем. Завдяки використанню комбінованого підходу до формування даних (реальні метрики та згенеровані сценарії), датасет є гнучким та придатним для моделювання різних бізнес-кейсів. Нормалізація метрик, таких як продажі, кліки, кількість унікальних користувачів і рейтинги, забезпечує об'єктивність аналізу та можливість порівняння продуктів між собою. Це робить його цінним інструментом для досліджень та вдосконалення методів персоналізованих рекомендацій у сфері цифрових товарів.

Розглянемо формування датасету, метод `loadProductDataFromDB` (рис. 2.1) є важливим етапом процесу завантаження та обробки даних для подальшого використання в рекомендаційних системах. Його основною метою є отримання та обробка даних із бази даних, що містить інформацію про продукти, відгуки, а також метрики взаємодії користувачів із продуктами. Цей метод реалізує гнучкий механізм завантаження, обробки, генерації випадкових значень для деяких параметрів, а також конвертації даних у формат, який можна використовувати в бібліотеці Surprise [16].

Використання попередньо завантажених даних

Якщо дані вже були раніше отримані і збережені в атрибуті `self.raw_data`, метод пропускає запит до бази даних і безпосередньо використовує ці дані для подальшої обробки. У такому випадку:

Кожен рядок із `self.raw_data` містить відомості про продукт, включаючи його ID, назву, кількість унікальних користувачів, кількість продажів, кількість кліків, дату створення продукту, ID відгуку та рейтинг. Далі для кожного продукту обчислюється базовий рейтинг за допомогою методу `calculate_overall_rating`, що враховує три метрики: кількість унікальних користувачів, кількість продажів та кількість кліків. Отримані результати зберігаються у вигляді пари: `(feedback_id, product_id, base_rating, product_created_at)`.

Після цього дані записуються у формат CSV для подальшого завантаження в систему рекомендацій. Для цього використовується бібліотека `pandas`, що дозволяє зберігати `DataFrame` у файл. Окрім того, ці дані конвертуються у формат, сумісний з бібліотекою `Surprise`, за допомогою читача, що забезпечує відповідний формат для обробки рейтингових даних.

```
Codeium: Refactor | Explain | Generate Docstring | X
def loadProductDataFromDB(self):
    if self.raw_data:
        data = []
        for row in self.raw_data:
            product_id, product_name, uniq_users, sold, user_clicks, product_created_at, feedback_id, rating = row

            base_rating = self.calculate_overall_rating(uniq_users, sold, user_clicks, rating)

            data.append((feedback_id, product_id, base_rating, product_created_at))

            self.productID_to_name[product_id] = product_name
            self.name_to_productID[product_name] = product_id

        df = pd.DataFrame(data, columns=['user', 'item', 'rating', 'timestamp'])
        df.to_csv(self.ratingsPath, index=False)

        reader = Reader(line_format='user item rating timestamp', sep=',', skip_lines=1)

        ratingsDataset = Dataset.load_from_file(self.ratingsPath, reader=reader)
        return ratingsDataset
```

Рис. 2.1. Фрагмент коду для завантаження датасетів

Завантаження даних із бази даних

У випадку, якщо дані відсутні у `self.raw_data`, метод підключається до бази даних та виконує SQL-запит для отримання необхідної інформації. Для цього використовується мова SQL та методи агрегації даних:

Використання Common Table Expressions (CTE) [17]:

Перший CTE — `products` — обчислює базові метрики для кожного продукту, зокрема кількість унікальних користувачів (`uniqUsers`), кількість продажів (`sold`), кількість кліків (`userClicks`) та дату створення продукту, яку конвертується в Unix-мітку.

Другий CTE — `existing_feedback` — отримує відгуки, які вже існують для кожного продукту.

Третій CTE — `feedback` — генерує випадкові значення для відгуків, зокрема, випадкові рейтинги у діапазоні від 1 до 5 та ідентифікатори відгуків.

Після виконання основного SELECT-запиту отримуються згенеровані метрики для кожного продукту:

Якщо кількість унікальних користувачів або продажів менша за порогове значення, то вони замінюються випадково згенерованими значеннями в межах заданих діапазонів.

Для кожного продукту генерується унікальний ідентифікатор відгуку та випадковий рейтинг.

```
query = """
WITH products AS (
  SELECT
    p.id AS product_id,
    p.name AS product_name,
    jsonb_array_length(p."uniqUsers") AS uniqUsers, -- Get the length of uniqUsers array
    p.sold,
    p."userClicks",
    COALESCE(EXTRACT(EPOCH FROM p."productCreatedAt")::bigint, NULL) AS productCreatedAt -- Convert to Unix t:
  FROM
    product p
  WHERE
    jsonb_array_length(p."uniqUsers") > 0 -- Ensure there are unique users
),
existing_feedback AS (
  SELECT
    f."productId" AS product_id, -- Make sure to use the correct case
    f.id AS feedback_id
  FROM
    feedback f
),
feedback AS (
  SELECT
    p.product_id,
    p.product_name,
    p.uniqUsers,
    p.sold,
    p."userClicks",
    p.productCreatedAt,
    (g.id + RANDOM() * 2)::int AS generated_feedback_id, -- Generated feedback ID
    Random()::int AS rating -- Random rating between 1 and 5
)
```

Рис. 2.2. Фрагмент коду для завантаження датасетів

Обробка результатів SQL-запиту

Отримані з бази даних дані обробляються та зберігаються у форматі, що підходить для аналізу та моделювання. Для кожного продукту збережено:

ID продукту та назва продукту, рейтинг для кожного продукту, що обчислюється на основі метрик, таких як кількість унікальних користувачів, кількість продажів та кількість кліків, ID відгуку, який згенеровано випадковим чином для кожного продукту.

Ці дані записуються в таблицю pandas DataFrame, що зберігається у CSV-файл. Після цього дані завантажуються у формат, сумісний із бібліотекою Surprise для подальшого використання у рекомендаційних алгоритмах.

```

connection = self.connect_db()
cursor = connection.cursor()

query = """ ...

cursor.execute(query)
rows = cursor.fetchall()
self.raw_data = rows

data = []
for row in rows:
    product_id, product_name, uniq_users, sold, user_clicks, product_created_at, feedback_id, rating = row

    base_rating = self.calculate_overall_rating(uniq_users, sold, user_clicks, rating)

    data.append((feedback_id, product_id, base_rating, product_created_at))

    self.productID_to_name[product_id] = product_name
    self.name_to_productID[product_name] = product_id

df = pd.DataFrame(data, columns=['user', 'item', 'rating', 'timestamp'])
df.to_csv(self.ratingsPath, index=False)

reader = Reader(line_format='user item rating timestamp', sep=',', skip_lines=1)

ratingsDataset = Dataset.load_from_file(self.ratingsPath, reader=reader)
return ratingsDataset
except Exception as e:

```

Рис. 2.3. Фрагмент коду для завантаження датасетів

2.2 Побудова моделі аналізу і практична реалізація

Першим етапом було розроблено клас ProductLens, який служить для роботи з даними продуктів та їх відгуками, які зберігаються в базі даних PostgreSQL. Він дозволяє підключатися до бази даних, завантажувати дані про продукти, виконувати нормалізацію різних показників, таких як кількість унікальних користувачів, кількість продажів, кліки користувачів та рейтинги

продуктів. Клас також надає можливості для обчислення загального рейтингу продуктів, зберігання та обробки даних для подальшого використання в системах рекомендацій.

```
from typing import Dict, List, Optional, Union, Tuple, Any
class ProductLens:
    db_params = {
        "dbname": "",
        "user": "",
        "password": "",
        "host": "",
        "port": ""
    }
    productID_to_name = {}
    name_to_productID = {}
    raw_data = None
    ratingsPath = './ratings.csv'

    def connect_db(self):
        try:
            return psycopg2.connect(**self.db_params)
        except Exception as e:
            print(f"Database connection error: {e}")
            raise

    @staticmethod
    def min_max_normalize(value, min_value, max_value):
        return (value - min_value) / (max_value - min_value)

    def calculate_overall_rating(self, uniq_users, user_clicks, sold, rating, alpha=0.1):
        normalized_uniq_users = self.min_max_normalize(uniq_users, 0, BASE_UNIQ_USERS)
        normalized_user_clicks = self.min_max_normalize(user_clicks, 0, BASE_USERS_CLICKS)
        normalized_sold = self.min_max_normalize(sold, 0, BASE_SOLD)
        normalized_rating = self.min_max_normalize(rating, 0, BASE_RATING)

        overall_score = (
            np.log1p(normalized_uniq_users) +
            np.log1p(normalized_user_clicks) +
            np.log1p(normalized_sold) +
            alpha * normalized_rating
        )
```

Рис. 2.4. Фрагмент класу ProductLens

Розглянемо основні методи класу, які відповідають за з'єднання з базою даних, нормалізацію даних та обчислення рейтингів для продуктів. Ці методи забезпечують обробку інформації про продукти, включаючи їхню популярність, кількість продажів та кліків, а також взаємодію з користувачами. Вони допомагають створити ефективну систему рекомендацій, оптимізуючи процес обчислення схожості між продуктами та генерації оцінок

Функція `connect_db` відповідає за встановлення з'єднання з базою даних PostgreSQL. Вона використовує метод `psycopg2.connect()`, застосовуючи параметри, збережені в атрибуті `db_params`. Якщо з'єднання не вдається, перехоплюється виключення, виводиться повідомлення про помилку, після чого виключення знову підвищується для коректної обробки на вищих рівнях програми.

```
def connect_db(self):
    try:
        return psycopg2.connect(**self.db_params)
    except Exception as e:
        print(f"Database connection error: {e}")
        raise
```

Рис. 2.5. Фрагмент класу ProductLens

Функція `min_max_normalize` виконує нормалізацію заданого значення в межах вказаного діапазону. Вона приймає три аргументи: значення для нормалізації, мінімум та максимум діапазону та масштабує значення в діапазон від 0 до 1, забезпечуючи узгодженість для різних вхідних значень [18].

```
Codeium: Refactor | Explain | Generate Docstring | X
@staticmethod
def min_max_normalize(value, min_value, max_value):
    return (value - min_value) / (max_value - min_value)
```

Рис. 2.6. Фрагмент класу ProductLens

Функція `calculate_overall_rating` обчислює комбіновану оцінку для продукту, нормалізуючи різні вхідні метрики, такі як кількість унікальних користувачів, кліків, продажів та власний рейтинг продукту. Функція використовує метод `min_max_normalize` для масштабування цих значень, після чого обчислюється логарифмічна трансформація нормалізованих значень (для обробки асиметричних розподілів) і комбінуються вони за допомогою зваженої суми, з додатковим членом, що представляє взаємодію між унікальними користувачами та кліками, масштабується за допомогою параметра `alpha`. Ця фінальна оцінка представляє загальну оцінку для продукту [19, 20].

```
Codeium: Refactor | Explain | Generate Docstring | X
def calculate_overall_rating(self, uniq_users, user_clicks, sold, rating, alpha=0.1):

    normalized_uniq_users = self.min_max_normalize(uniq_users, 0, BASE_UNIQ_USERS)
    normalized_user_clicks = self.min_max_normalize(user_clicks, 0, BASE_USERS_CLICKS)
    normalized_sold = self.min_max_normalize(sold, 0, BASE_SOLD)
    normalized_rating = self.min_max_normalize(rating, 0, BASE_RATING)

    overall_score = (
        np.log1p(normalized_uniq_users) +
        np.log1p(normalized_user_clicks) +
        np.log1p(normalized_sold) +
        np.log1p(normalized_rating) +
        alpha * (np.log1p(normalized_uniq_users) * np.log1p(normalized_user_clicks))
    )
    return overall_score
```

Рис. 2.7. Фрагмент класу ProductLens

Функція `getUserRatings` отримує оцінки для конкретного користувача. Вона перевіряє наявність `raw_data`, і якщо дані відсутні, завантажує їх із бази даних. Функція ітерує по кожному рядку в даних, перевіряючи, чи збігається `feedback_id` з ідентифікатором користувача. Якщо збіг знайдений, додається відповідний рейтинг до списку `userRatings`. Якщо конкретний відгук для продукту не знайдений, обчислюється базовий рейтинг, використовуючи кількість унікальних користувачів, продажів та кліків, і це значення використовується як рейтинг для продукту.

```

Lodeium: Refactor | Explain | Generate Docstring | X
def getUserRatings(self, user):
    userRatings = []
    if self.raw_data is None:
        self.loadProductDataFromDB()

    for row in self.raw_data:

        product_id, product_name, uniq_users_count, sold, user_clicks, product_created_at, feedback_id, rating = row

        if feedback_id and feedback_id == user:

            userRatings.append((product_id, rating))
        else:

            base_rating = uniq_users_count + sold + user_clicks
            userRatings.append((product_id, base_rating))

    return userRatings

```

Рис. 2.8. Фрагмент класу `ProductLens`

Функція `getPopularityRanks` обчислює популярність продуктів на основі їхніх загальних оцінок. Вона підсумовує оцінки для кожного продукту і ранжує їх у порядку спадання. Функція використовує `defaultdict` для накопичення оцінок для кожного продукту, після чого сортує продукти за їхньою сумарною оцінкою. Кожному продукту присвоюється ранг, при цьому продукт з найвищим рейтингом отримує ранг 1.

```

Lodeium: Refactor | Explain | Generate Docstring | X
def getPopularityRanks(self):
    ratings = defaultdict(int)
    rankings = defaultdict(int)

    for row in self.raw_data:
        product_id, product_name, uniq_users, sold, user_clicks, product_created_at, feedback_id, rating = row
        ratings[product_id] += self.calculate_overall_rating(uniq_users, sold, user_clicks, rating)

    rank = 1
    for product_id, rating_count in sorted(ratings.items(), key=lambda x: x[1], reverse=True):
        rankings[product_id] = rank
        rank += 1

    return rankings

```

Рис. 2.9. Фрагмент класу `ProductLens`

Функція `getProductName` отримує назву продукту за його `productID`. Вона шукає `productID` в словнику `productID_to_name` і повертає відповідну назву продукту. Якщо `productID` не знайдений, повертається порожній рядок.

```
def getProductName(self, productID):  
    return self.productID_to_name.get(productID, "")
```

Рис. 2.10. Фрагмент класу `ProductLens`

Функція `getProductID` отримує `productName`, що відповідає заданій назві продукту. Вона шукає `productName` в словнику `name_to_productID` і повертає відповідний `productID`. Якщо назва продукту не знайдена, повертається 0.

```
def getProductID(self, productName):  
    return self.name_to_productID.get(productName, 0)
```

Рис. 2.11. Фрагмент класу `ProductLens`

Функція `compute_uniq_users_similarity` обчислює схожість між двома продуктами на основі кількості унікальних користувачів. Вона отримує кількість унікальних користувачів для обох продуктів і повертає середнє значення цих значень як коефіцієнт схожості. Ця метрика допомагає оцінити, наскільки схожі два продукти за їхньою користувацькою базою.

```
def compute_uniq_users_similarity(self, product1_id, product2_id):  
    users1 = self.get_uniq_users(product1_id)  
    users2 = self.get_uniq_users(product2_id)  
  
    return (users1 + users2) / 2
```

Рис. 2.12. Фрагмент класу `ProductLens`

Функція `compute_sold_similarity` обчислює схожість між двома продуктами на основі їхніх показників продажу. Вона отримує кількість продажів для обох продуктів і обчислює середнє значення, яке служить коефіцієнтом схожості. Ця метрика відображає схожість між продуктами з точки зору того, наскільки вони були продані.

```
def compute_sold_similarity(self, product1_id, product2_id):  
    sold1 = self.get_sold_count(product1_id)  
    sold2 = self.get_sold_count(product2_id)  
  
    return (sold1 + sold2) / 2
```

Рис. 2.13. Фрагмент класу `ProductLens`

Функція `compute_user_clicks_similarity` обчислює схожість між двома продуктами на основі кількості кліків користувачів. Вона порівнює кількість кліків для обох продуктів і повертає середнє значення як коефіцієнт схожості. Ця метрика допомагає виміряти ступінь інтересу користувачів до продуктів.

```
def compute_user_clicks_similarity(self, product1_id, product2_id):
    clicks1 = self.get_user_clicks_count(product1_id)
    clicks2 = self.get_user_clicks_count(product2_id)

    return (clicks1 + clicks2) / 2
```

Рис. 2.14. Фрагмент класу ProductLens

Функція `compute_rating_similarity` обчислює схожість між двома продуктами на основі їхніх середніх рейтингів. Вона отримує рейтинги для обох продуктів і повертає середнє значення рейтингів як коефіцієнт схожості. Ця метрика використовується для визначення схожості продуктів з точки зору задоволення користувачів.

```
def compute_rating_similarity(self, product1_id, product2_id):
    rating1 = self.get_rating(product1_id)
    rating2 = self.get_rating(product2_id)

    return (rating1 + rating2) / 2
```

Рис. 2.15. Фрагмент класу ProductLens

Функція `get_uniq_users` отримує загальну кількість унікальних користувачів для заданого продукту, підсумовуючи значення `uniqUsers` для всіх записів з однаковим `product_id`. Вона повертає середнє значення унікальних користувачів для продукту серед усіх записів. Якщо записи не знайдені, повертається 0.

```
def get_uniq_users(self, product_id):
    if self.raw_data is None:
        self.loadProductDataFromDB()

    total_uniq_users = 0
    count = 0
    for row in self.raw_data:
        product_id_row, product_name, uniq_users, sold, user_clicks, product_created_at, feedback_id, rating = row
        if product_id_row == product_id:
            total_uniq_users += uniq_users
            count += 1

    return total_uniq_users / count if count > 0 else 0
```

Рис. 2.16. Фрагмент класу ProductLens

Функція `get_sold_count` отримує загальну кількість продажів для заданого продукту, підсумовуючи значення `sold` для всіх записів з однаковим `product_id`. Вона повертає середнє значення кількості продажів для продукту серед усіх записів. Якщо записи не знайдені, повертається 0.

```
def get_sold_count(self, product_id):
    if self.raw_data is None:
        self.loadProductDataFromDB()

    total_sold = 0
    count = 0
    for row in self.raw_data:
        product_id_row, product_name, uniq_users, sold, user_clicks, product_created_at, feedback_id, rating = row
        if product_id_row == product_id:
            total_sold += sold
            count += 1

    return total_sold / count if count > 0 else 0
```

Рис. 2.17. Фрагмент класу ProductLens

Функція `get_user_clicks_count` отримує загальну кількість кліків користувачів для заданого продукту, підсумовуючи значення `userClicks` для всіх записів з однаковим `product_id`. Вона повертає середнє значення кількості кліків для продукту серед усіх записів. Якщо записи не знайдені, повертається 0.

```
Codeium: Refactor | Explain | Generate Docstring | X
def get_user_clicks_count(self, product_id):
    if self.raw_data is None:
        self.loadProductDataFromDB()

    total_user_clicks = 0
    count = 0
    for row in self.raw_data:
        product_id_row, product_name, uniq_users, sold, user_clicks, product_created_at, feedback_id, rating = row
        if product_id_row == product_id:
            total_user_clicks += user_clicks
            count += 1

    return total_user_clicks / count if count > 0 else 0
```

Рис. 2.18. Фрагмент класу ProductLens

Функція `get_rating` отримує загальний рейтинг для заданого продукту, підсумовуючи значення `rating` для всіх записів з однаковим `product_id`. Вона повертає середнє значення рейтингу для продукту серед усіх записів. Якщо записи не знайдені, повертається 0.

```

def get_rating(self, product_id):
    if self.raw_data is None:
        self.loadProductDataFromDB()

    total_rating = 0
    count = 0
    for row in self.raw_data:
        product_id_row, product_name, uniq_users, sold, user_clicks, product_created_at, feedback_id, rating = row
        if product_id_row == product_id:
            total_rating += rating
            count += 1

    return total_rating / count if count > 0 else 0

```

Рис. 2.19. Фрагмент класу ProductLens

Для рекомендаційної системи розроблено два алгоритми, один за допомогою метода k-найближчих сусідів, другий за допомогою обмеженої машини Больцмана, а також розроблений клас для гібридного використання цих алгоритмів [21, 22].

Розглянемо більш детально обмежену машину Больцмана і як вона працює.

RBM - це двошарова (вхідний шар і прихований шар) Штучна Нейронна Мережа, яка навчається ймовірнісного розподілу на основі заданого набору входів. Це стохастичний (недетермінований), що допомагає нам вирішувати різноманітні задачі, засновані на комбінаціях. RBM можна використовувати для зменшення розмірності, класифікації, регресії, колаборативної фільтрації, навчання ознак та тематичного моделювання [23].

Як ім'я підказує, RBM - це клас машин Больцмана, але вони обмежені у певних аспектах, коли ми розглядаємо зв'язки між вхідними і прихованими вузлами Нейронної Мережі, і тому його легше реалізувати, ніж машину Больцмана. Шари та вузли всередині цих шарів з'єднані у співвідношенні один-до-багатьох, де кожен вузол у вхідному шарі з'єднаний з кожним вузлом у прихованому шарі, але жоден вузол всередині кожного шару не з'єднаний. Це обмеження дозволяє використовувати більш потужні алгоритми навчання, ніж ті, які загалом використовуються в машинах Больцмана [24].

Ось як виглядає RBM: як можна побачити, всі вузли вхідного шару з'єднані з кожним вузлом прихованого шару, і сама структура нейронної мережі робить його ефективним при навчанні нашої нейронної мережі, оскільки один вхідний шар може використовувати багато прихованих шарів для навчання. Також можна

стосувати кілька RBM, створюючи мережу глибокого віруючого навчання, яка дозволить глибше навчання нейронної мережі та включати додаткове навчання.

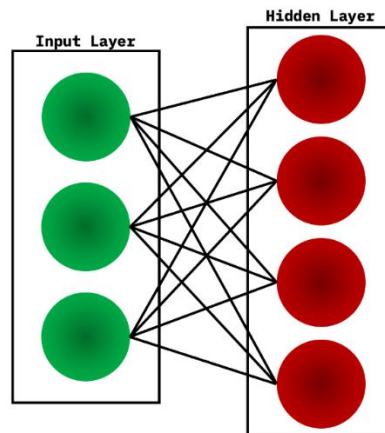


Рис. 2.20. Структура RBM

Робота Обмеженої Машини Больцмана (RBM) включає два кроки:

Багато Вхідних Даних. На першому кроці RBM приймає багато вхідних даних. Кожен вхідний зразок представляє собою вектор, де кожен елемент відповідає певній ознаці або характеристиці.

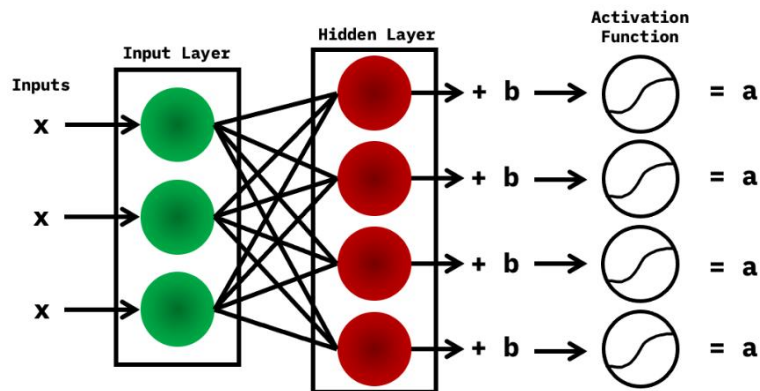


Рис. 2.21. Першому крок RBM

Відтворення. Після того як RBM отримає вхідні дані, він використовує їх для побудови нових вхідних зразків (реконструкції). Цей процес включає в себе реконструкцію вхідних даних з вагами, які навчилися в ході попереднього кроку. У процесі реконструкції RBM намагається відновити вхідні дані, використовуючи внутрішні ваги та активації прихованих вузлів. Тоді порівнює реконструйовані дані з оригінальними вхідними даними для оцінки того, наскільки добре RBM зміг реконструювати вхідні дані [25].

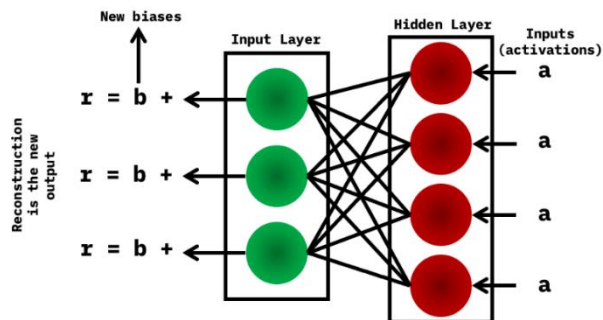


Рис. 2.22. Другий крок RBM

Послідовно відбуваючись ці два кроки, спочатку ми генеруємо активації, використовуючи фазу багатьох входів, а потім відбувається відновлення, коли реконструкція відбувається в епоху. Наша головна мета тут полягає в зменшенні помилки реконструкції, тому ваги відповідно налаштовуються на кожній ітерації за алгоритмом для зменшення помилки реконструкції, що призведе до надання нам гарного прогнозу та вищої точності [26].

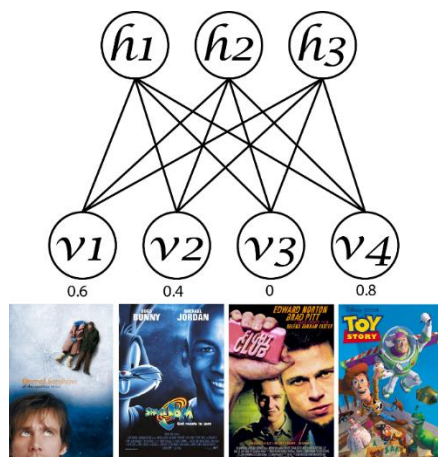


Рис. 2.23. Приклад роботи RBM

Клас RBM (Restricted Boltzmann Machine) є реалізацією алгоритму для побудови рекомендаційної системи, що використовує методи машинного навчання для розпізнавання структур в даних, таких як рейтинги продуктів або взаємодія користувачів з товарами. Метою класу є навчання моделі на основі вхідних даних і отримання рекомендацій для користувачів.

Клас ініціалізує параметри моделі, які включають розміри видимих і прихованих шарів, кількість епох, значення рейтингу, швидкість навчання та

розмір пакету. Модель використовує варіанти випадкових ваг і зміщень для схованих і видимих нейронів.

Метод `__init__` ініціалізує основні параметри класу, зокрема розміри видимих та прихованих шарів, кількість епох, розмір пакету для тренування, швидкість навчання та кількість значень рейтингів. Також він ініціалізує ваги та зміщення для видимих і прихованих нейронів, використовуючи випадкові значення для цих параметрів.

```
def __init__(self, visibleDimensions, epochs=500, hiddenDimensions=100, ratingValues=1, learningRate=0.05, batchSize=128):
    self.visibleDimensions = visibleDimensions
    self.epochs = epochs
    self.hiddenDimensions = hiddenDimensions
    self.ratingValues = ratingValues
    self.learningRate = learningRate
    self.batchSize = batchSize

    self.weights = tf.Variable(tf.random.uniform([self.visibleDimensions, self.hiddenDimensions],
                                                minval=-0.1, maxval=0.1), name="weights")
    self.hiddenBias = tf.Variable(tf.zeros([self.hiddenDimensions]), name="hiddenBias")
    self.visibleBias = tf.Variable(tf.zeros([self.visibleDimensions]), name="visibleBias")
```

Рис. 2.24. Фрагмент класу RBM

Метод `train` відповідає за тренування моделі протягом декількох епох. Перед кожним циклом тренування дані випадковим чином перетасовуються, а потім розбиваються на міні-батчі для поступового оновлення параметрів моделі. Кожен міні-батч проходить через функцію `_train_step`, яка виконує один крок тренування.

```
def train(self, X):
    for epoch in range(self.epochs):
        np.random.shuffle(X)
        trX = np.array(X)

        for i in range(0, trX.shape[0], self.batchSize):
            batch = trX[i:i + self.batchSize]
            self._train_step(batch)
```

Рис. 2.25. Фрагмент класу RBM

Метод `_train_step` є основним кроком тренування, в якому відбувається обчислення ймовірностей прихованих одиничних змінних на основі видимих вхідних даних, а також реконструкція видимих одиничних змінних з прихованих змінних. Далі обчислюються та оновлюються ваги та зміщення, використовуючи

метод зворотного поширення помилки для обчислення градієнтів і їх подальшого коригування [27, 28].

```
def _train_step(self, batch):
    with tf.GradientTape() as tape:
        h_prob = self._sample_hidden(batch)
        v_prob = self._reconstruct_visible(h_prob)

        v_mask = tf.sign(batch)
        v_mask_3d = tf.reshape(v_mask, [tf.shape(v_prob)[0], -1, self.ratingValues])
        v_mask_3d = tf.reduce_max(v_mask_3d, axis=2, keepdims=True)

        v_prob = tf.reshape(v_prob, [tf.shape(v_prob)[0], -1, self.ratingValues])
        v_prob = tf.nn.softmax(v_prob * v_mask_3d)
        v_prob = tf.reshape(v_prob, [tf.shape(v_prob)[0], -1])

        h_prob1 = self._sample_hidden(v_prob)

        weight_update = tf.matmul(tf.transpose(batch), h_prob) - tf.matmul(tf.transpose(v_prob), h_prob1)
        hidden_bias_update = tf.reduce_mean(h_prob - h_prob1, axis=0)
        visible_bias_update = tf.reduce_mean(batch - v_prob, axis=0)

        gradients = [weight_update, hidden_bias_update, visible_bias_update]

    self.weights.assign_add(self.learningRate * gradients[0])
    self.hiddenBias.assign_add(self.learningRate * gradients[1])
    self.visibleBias.assign_add(self.learningRate * gradients[2])
```

Рис. 2.26. Фрагмент класу RBM

Метод `_sample_hidden` здійснює вибірку ймовірностей прихованих одиничних змінних, використовуючи поточні видимі значення та сигмоїдну функцію для визначення ймовірності активації прихованих одиничних змінних [29].

```
def _sample_hidden(self, visible):
    h_prob = tf.nn.sigmoid(tf.matmul(visible, self.weights) + self.hiddenBias)
    return tf.cast(h_prob > tf.random.uniform(tf.shape(h_prob)), tf.float32)
```

Рис. 2.27. Фрагмент класу RBM

Метод `_reconstruct_visible` відповідає за реконструкцію видимих значень на основі прихованих одиничних змінних, використовуючи сигмоїдну функцію для обчислення ймовірностей.

```
def _reconstruct_visible(self, hidden):
    return tf.nn.sigmoid(tf.matmul(hidden, tf.transpose(self.weights)) + self.visibleBias)
```

Рис. 2.28. Фрагмент класу RBM

Метод `get_recommendations` використовується для отримання рекомендацій для користувача. Він реконструює видимі значення на основі обчислених ймовірностей прихованих одиничних змінних і дає рекомендації, що можуть бути використані в системах персоналізованих рекомендацій.

```
Codeium: Refactor | Explain | Generate Docstring | X
def get_recommendations(self, inputUser):
    hidden = self._sample_hidden(inputUser)
    visible = self._reconstruct_visible(hidden)
    return visible.numpy() You, 4 days ago *
```

Рис. 2.29. Фрагмент класу RBM

Клас RBMAlgorithm реалізує алгоритм рекомендацій на основі обмеженої болцманівської машини (RBM). Цей клас інтегрується з бібліотекою surprise для створення рекомендаційної системи, що використовує модель RBM для передбачення рейтингів користувачів.

Метод `__init__` ініціалізує параметри алгоритму, зокрема кількість епох для навчання, кількість одиничних змінних у прихованому шарі (hidden dimension), швидкість навчання, розмір пакету для тренування та додаткові параметри для налаштування симуляції. Він також створює екземпляр класу ProductLens та завантажує дані про продукти з бази даних.

```
def __init__(self, epochs=200, hiddenDim=200, learningRate=0.001, batchSize=64, sim_options={}):
    AlgoBase.__init__(self)
    self.epochs = epochs
    self.hiddenDim = hiddenDim
    self.learningRate = learningRate
    self.batchSize = batchSize
    self.pl = ProductLens()
    self.pl.loadProductDataFromDB()
```

Рис. 2.30. Фрагмент класу RBMAlgorithm

Метод `softmax` реалізує стабільну версію функції softmax, яка використовується для нормалізації прогнозованих рейтингів. Вона переводить набір значень у ймовірнісну форму, що дозволяє оцінити ймовірності для кожного можливого рейтингу [30].

```
Codeium: Refactor | Explain | Generate Docstring | X
def softmax(self, x):
    e_x = np.exp(x - np.max(x))
    return e_x / np.sum(e_x, axis=0)
```

Рис. 2.31. Фрагмент класу RBMAlgorithm

Метод `fit` тренує модель на основі вхідного набору даних. Він створює тренувальну матрицю, де кожен рейтинг користувача для конкретного товару представлений у вигляді вектора з 10 значеннями (для кожного можливого рейтингу від 0 до 9). Потім цей набір даних передається до моделі RBM для

навчання. Після цього, для кожного користувача, метод виконує прогнозування рейтингів для всіх товарів, використовуючи метод `get_recommendations` з класу RBM. Прогнозовані рейтинги нормалізуються за допомогою `softmax`, і кінцеві прогнози зберігаються в масиві `predictedRatings`.

```
def fit(self, trainset):
    AlgoBase.fit(self, trainset)

    numUsers = trainset.n_users
    numItems = trainset.n_items

    trainingMatrix = np.zeros([numUsers, numItems, 10], dtype=np.float32)

    for (uid, iid, rating) in trainset.all_ratings():
        adjustedRating = min(max(int(float(rating)), 0), 9)
        trainingMatrix[int(uid), int(iid), adjustedRating] = 1

    trainingMatrix = np.reshape(trainingMatrix, [trainingMatrix.shape[0], -1])

    rbm = RBM(trainingMatrix.shape[1], hiddenDimensions=self.hiddenDim, learningRate=self.learningRate, batchSize=self.batchSize)
    rbm.train(trainingMatrix)

    self.predictedRatings = np.zeros([numUsers, numItems], dtype=np.float32)
    for uid in range(numUsers):
        if uid % 50 == 0:
            print("Processing user ", uid)
            recs = rbm.get_recommendations([trainingMatrix[uid]])
            recs = np.reshape(recs, [numItems, 10])

            for itemID, rec in enumerate(recs):
                normalized = self.softmax(rec)
                rating = np.average(np.arange(10), weights=normalized)
                self.predictedRatings[uid, itemID] = (rating + 1) * 0.5

    return self
```

Рис. 2.32. Фрагмент класу RBMAlgorithm

Метод `estimate` оцінює рейтинг для конкретної пари користувач/товар. Він перевіряє, чи є користувач і товар у тренувальному наборі даних. Якщо вони є, то повертається прогнозований рейтинг для цієї пари. Якщо рейтинг надто малий або не існує, метод викидає помилку `PredictionImpossible`.

```
def estimate(self, u, i):
    if not (self.trainset.knows_user(u) and self.trainset.knows_item(i)):
        raise PredictionImpossible('User and/or item is unknown.')

    rating = self.predictedRatings[u, i]

    if rating < 0.001:
        raise PredictionImpossible('No valid prediction exists.')

    return rating
```

Рис. 2.33. Фрагмент класу RBMAlgorithm

Цей клас використовує методи RBM для створення рекомендаційної системи, комбінуючи алгоритм машинного навчання з бібліотекою `Surprise` для

зручності інтеграції з іншими рекомендаційними алгоритмами та інструментами. Підхід дозволяє забезпечити адаптивність у створенні рекомендацій, враховуючи як латентні, так і явні фактори впливу.

Розглянемо інший алгоритм, який буде використовуватись для гібридної реалізації.

Алгоритм KNN є одним із найпростіших, але водночас важливих алгоритмів класифікації в машинному навчанні. Він належить до галузі навчання з учителем і знаходить інтенсивне застосування в розпізнаванні образів, добуванні даних, рекомендаційних системах та виявленні вторгнень.

Цей алгоритм широко застосовується в реальних сценаріях, оскільки є непараметричним, тобто не робить жодних припущень про розподіл даних (на відміну від інших алгоритмів, таких як GMM, які припускають, що дані мають нормальний розподіл). Нам надаються деякі попередні дані (також звані навчальними даними), які класифікують координати в групи, визначені атрибутом [31].

Алгоритм K-NN (к-найближчих сусідів) є універсальним і широко використовуваним інструментом машинного навчання, який здобув популярність через свою простоту та легкість впровадження. Він застосовується для задач класифікації та регресії, дозволяючи працювати як з числовими, так і з категоріальними даними. K-NN базується на обчисленні схожості між точками даних у багатовимірному просторі та віднесенні нових точок до груп, до яких належать їхні найближчі сусіди.

Цей алгоритм забезпечує гнучкість і точність у роботі з реальними даними завдяки стійкості до викидів та здатності адаптуватися до змін структури даних. У гібридних рекомендаційних системах K-NN часто використовується для аналізу схожості між користувачами або продуктами, що дозволяє об'єднувати його переваги з іншими методами, такими як RBM, для досягнення високої точності та різноманітності рекомендацій [31].

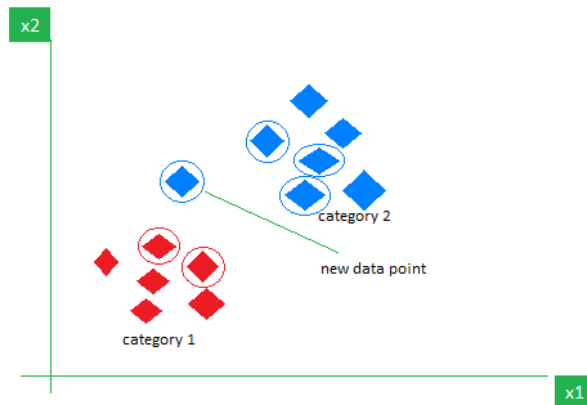


Рис. 2.34. Алгоритм KNN

Алгоритм K-NN працює, знаходячи K найближчих сусідів до заданої точки даних на основі метрики відстані, такої як евклідова відстань. Клас або значення точки даних визначається більшістю голосів або середнім значенням K сусідів. Цей підхід дозволяє алгоритму адаптуватися до різних шаблонів і робити передбачення на основі локальної структури даних.

Як відомо, алгоритм KNN допомагає нам визначити найближчі точки або групи для запитуваної точки. Але щоб визначити найближчі групи або точки для запитуваної точки, нам потрібна певна метрика. Для цього ми використовуємо наступні метрики відстані:

Евклідова відстань

Це не що інше, як декартова відстань між двома точками, які знаходяться на площині/гіперплощині. Евклідова відстань також може бути візуалізована як довжина прямої лінії, що з'єднує дві точки, які розглядаються. Ця метрика допомагає нам обчислити чисте зміщення між двома станами об'єкта [32].

2.1

$$d(p, q) = \sqrt{\sum_{i=1}^n (p_i - q_i)^2}$$

Відстань Манхеттенська

Метрика відстані Манхеттенської використовується загалом, коли ми цікавимося загальною відстанню, пройденою об'єктом, а не лише зміщенням. Цю метрику обчислюють, сумуючи абсолютні різниці між координатами точок в n-вимірах [33].

2.2

$$d(p, q) = \sum_{i=1}^n |p_i - q_i|$$

Відстань Мінковського

Ми можемо сказати, що відстань Евклідова, а також відстань Манхеттенська, є особливими випадками відстані Мінковського [34].

2.3

$$d(p, q, P) = (\sum_{i=1}^n |p_i - q_i|^P)^{1/P}$$

З формули вище можна сказати, що коли $P = 2$, то вона збігається з формулою для відстані Евклідова, а коли $P = 1$, то ми отримуємо формулу для відстані Манхеттенська.

Вищезгадані метрики найбільш поширені при роботі з проблемами Машинного Навчання, але є інші метрики відстані, такі як Відстань Хеммінга, які допомагають при розв'язанні задач, які потребують порівняння між двома векторами, складеними з булевих або рядкових значень.

Значення k є дуже важливим у алгоритмі k -найближчих сусідів для визначення кількості сусідів у алгоритмі. Значення k в алгоритмі k -найближчих сусідів повинно бути обрано на основі вхідних даних. Якщо вхідні дані мають більше викидів або шуму, краще використовувати більше значення k . Рекомендується обирати непарне значення для k , щоб уникнути нерішучості у класифікації. Методи перехресної перевірки можуть допомогти вибрати найкраще значення k для заданого набору даних.

Алгоритм k-найближчих сусідів (KNN) працює на принципі схожості, де він передбачає мітку або значення нової точки даних, розглядаючи мітки або значення її K найближчих сусідів у навчальному наборі даних [35].

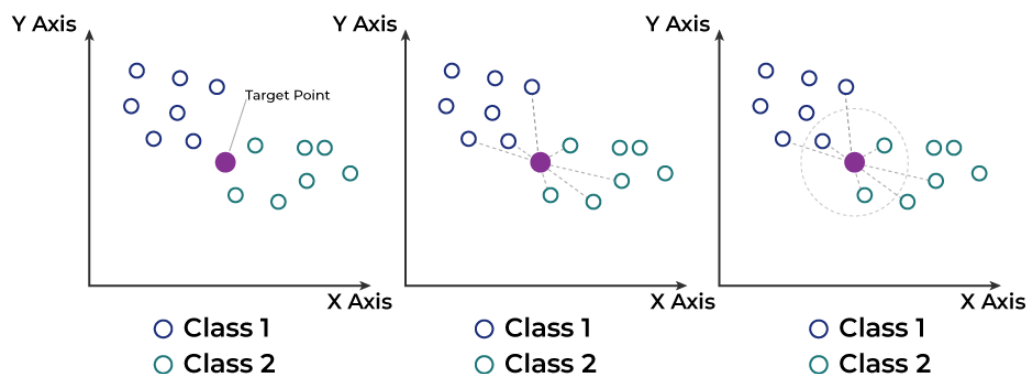


Рис. 2.35. Робота алгоритму KNN

Клас ContentKNNAlgorithm реалізує алгоритм рекомендацій, заснований на контентному схожості товарів за допомогою методу k-ближчих сусідів (k-NN). Цей алгоритм визначає схожість між товарами, використовуючи кілька метрик, таких як кількість унікальних користувачів, кількість продажів, кількість кліків і відгуки користувачів. Алгоритм прогнозує рейтинг користувача для товару на основі схожості з іншими товарами, які користувач вже оцінив.

Метод `__init__` ініціалізує основні параметри класу, зокрема кількість найближчих сусідів `k` і додаткові параметри для налаштування алгоритму. Клас успадковує від `AlgoBase` бібліотеки `surprise`, що дозволяє інтегрувати цей алгоритм з іншими частинами бібліотеки.

```
def __init__(self, k=20, sim_options={}):
    AlgoBase.__init__(self)
    self.k = k
```

Рис. 2.36. Фрагмент класу ContentKNNAlgorithm

Метод `fit` тренує модель на основі переданого набору даних. Він створює порожню матрицю схожості між товарами. Далі для кожної пари товарів (у вигляді рейтингів) обчислюється схожість за кількома метриками: схожість за кількістю унікальних користувачів, за кількістю продажів, за кількістю кліків та за рейтингами. Всі ці метрики комбінуються в загальну міру схожості, яка

записується в матрицю. Після цього матриця схожості використовується для подальших розрахунків прогнозованих рейтингів.

```
def fit(self, trainset):
    AlgoBase.fit(self, trainset)
    self.pl = ProductLens()
    print("Computing content-based similarity matrix...")
    self.similarities = np.zeros((self.trainset.n_items, self.trainset.n_items))

    for thisRating in range(self.trainset.n_items):
        for otherRating in range(thisRating + 1, self.trainset.n_items):
            thisProductID = int(self.trainset.to_raw_id(thisRating))
            otherProductID = int(self.trainset.to_raw_id(otherRating))

            uniq_users_similarity = self.pl.compute_uniq_users_similarity(thisProductID, otherProductID)
            sold_similarity = self.pl.compute_sold_similarity(thisProductID, otherProductID)
            user_clicks_similarity = self.pl.compute_user_clicks_similarity(thisProductID, otherProductID)
            rating_similarity = self.pl.compute_rating_similarity(thisProductID, otherProductID)

            total_similarity = (uniq_users_similarity + sold_similarity + user_clicks_similarity + rating_similarity) / 4.0

            self.similarities[thisRating, otherRating] = total_similarity
            self.similarities[otherRating, thisRating] = total_similarity

    print("...done.")
    return self
```

Рис. 2.37. Фрагмент класу ContentKNNAlgorithm

Метод estimate оцінює рейтинг для конкретної пари користувач/товар. Він перевіряє, чи є користувач і товар у тренувальному наборі даних. Якщо вони є, то метод обчислює схожість між поточним товаром і всіма товарами, оціненими користувачем. Потім вибираються k найбільш схожих товарів (сусідів), і на основі їх рейтингів розраховується середнє значення, зважене за схожістю. Якщо не можна знайти сусідів або немає валідних рейтингів, метод викидає помилку PredictionImpossible.

```
Codeium: Refactor | Explain | Generate Docstring | X
def estimate(self, u, i):
    if not (self.trainset.knows_user(u) and self.trainset.knows_item(i)):
        raise PredictionImpossible('User and/or item is unknown.')

    neighbors = []
    for rating in self.trainset.ur[u]:
        similarity_score = self.similarities[i, rating[0]]
        neighbors.append((similarity_score, rating[1]))

    k_neighbors = heapq.nlargest(self.k, neighbors, key=lambda t: t[0])

    simTotal = weightedSum = 0
    for (simScore, rating) in k_neighbors:
        if simScore > 0:
            simTotal += simScore
            weightedSum += simScore * rating

    if simTotal == 0:
        raise PredictionImpossible('No neighbors')

    predictedRating = weightedSum / simTotal

    return predictedRating
```

Рис. 2.38. Фрагмент класу ContentKNNAlgorithm

Цей клас дозволяє створювати рекомендації на основі контентної схожості товарів, що є корисним для ситуацій, коли відсутні достатні дані про поведінку користувачів або їх взаємодію з товарами.

Також створено гібридний метод, це клас який поєднує алгоритм KNN та обмеженої машини Больцмана

Клас HybridAlgorithm реалізує гібридний підхід до рекомендацій, поєднуючи кілька алгоритмів і комбінуючи їх прогнози з використанням ваг для кожного алгоритму. Це дозволяє підвищити точність рекомендацій, комбінуючи різні стратегії для оцінки переваг користувача щодо товарів.

Метод `__init__` ініціалізує основні параметри класу, зокрема список алгоритмів, які будуть використовуватися в гібридній моделі, та відповідні ваги для кожного алгоритму. Ваги визначають важливість кожного алгоритму при комбінуванні результатів. Клас успадковує від AlgoBase з бібліотеки surprise, що дозволяє інтегрувати цей алгоритм з іншими компонентами бібліотеки.

```
def __init__(self, algorithms, weights, sim_options={}):
    AlgoBase.__init__(self)
    self.algorithms = algorithms
    self.weights = weights
```

Рис. 2.39. Фрагмент класу ContentKNNAlgorithm

Метод `fit` виконує тренування моделі на переданому наборі даних. Для кожного алгоритму зі списку `algorithms` викликається метод `fit`, що дозволяє кожному з алгоритмів навчатися на даних.

```
def fit(self, trainset):
    AlgoBase.fit(self, trainset)

    for algorithm in self.algorithms:
        algorithm.fit(trainset)

    return self
```

Рис. 2.40. Фрагмент класу ContentKNNAlgorithm

Метод `estimate` обчислює прогноз для конкретної пари користувач/товар. Він ініціалізує змінні для збереження сумарних оцінок і ваг. Для кожного алгоритму в списку `algorithms` метод отримує прогноз, множить його на відповідну вагу і додає до загальної суми. Потім отриману суму оцінок ділить на

суму ваг, щоб отримати фінальний прогноз. Це дозволяє комбінувати прогнози кількох алгоритмів, надаючи більшу вагу тим, які мають кращу продуктивність.

```
def estimate(self, u, i):  
    sumScores = 0  
    sumWeights = 0  
  
    for idx in range(len(self.algorithms)):  
        sumScores += self.algorithms[idx].estimate(u, i) * self.weights[idx]  
        sumWeights += self.weights[idx]  
  
    return sumScores / sumWeights
```

Рис. 2.41. Фрагмент класу ContentKNNAlgorithm

Цей клас потрібен для об'єднання різних методів рекомендацій для досягнення кращих результатів. Гібридні системи зазвичай використовуються для подолання обмежень окремих алгоритмів, таких як колд-старти або обмеження за даними.

Також були створені класи EvaluatedAlgorithm, EvaluationData, та Evaluator для забезпечення комплексної оцінки алгоритмів рекомендацій. Клас EvaluatedAlgorithm відповідає за виконання оцінки точності алгоритму, включаючи такі метрики, як RMSE, MAE, Coverage, Diversity та Novelty, за допомогою методів з модуля RecommenderMetrics.

Клас EvaluationData забезпечує підготовку даних для оцінки, формуючи різні набори для навчання та тестування, а також для аналізу топ-N рекомендацій. Клас Evaluator інтегрує алгоритми та виконує їх оцінку, надаючи детальний звіт за результатами вимірювання метрик. Ці класи спрощують процес тестування та порівняння різних рекомендаційних алгоритмів.

Метод `__init__` відповідає за побудову основних тренувальних та тестових наборів даних. Це включає повний тренувальний набір для загальної оцінки властивостей моделі, анти-тестовий набір для побудови прогнозів, а також 75/25 розподіл для тренування та тестування з метою вимірювання точності. Додатково, метод реалізує Leave-One-Out (LOO) крос-валідацію, що дозволяє проводити оцінку для топ-N рекомендацій. Для розрахунку подібностей між товарами використовується алгоритм KNN, що допомагає в подальшому вимірюванні різноманіття.

```

def __init__(self, data, popularityRankings):
    self.rankings = popularityRankings

    self.fullTrainSet = data.build_full_trainset()
    self.fullAntiTestSet = self.fullTrainSet.build_anti_testset()

    self.trainSet, self.testSet = train_test_split(data, test_size=.25, random_state=1)

    LOOCV = LeaveOneOut(n_splits=1, random_state=1)
    for train, test in LOOCV.split(data):
        self.LOOCVTrain = train
        self.LOOCVTest = test

    self.LOOCVAntiTestSet = self.LOOCVTrain.build_anti_testset()

    sim_options = {'name': 'cosine', 'user_based': False}
    self.simsAlgo = KNNBaseline(sim_options=sim_options)
    self.simsAlgo.fit(self.fullTrainSet)

```

Рис. 2.42. Фрагмент класу EvaluationData

Метод GetFullTrainSet повертає повний тренувальний набір для загальних оцінок моделі, а GetFullAntiTestSet надає анти-тестовий набір для прогнозування елементів, які ще не були оцінені користувачами. Метод GetAntiTestSetForUser дозволяє побудувати анти-тестовий набір для конкретного користувача, включаючи всі елементи, які він ще не оцінював, що є корисним для персоналізованих рекомендацій. Інші методи, такі як GetTrainSet та GetTestSet, повертають відповідно тренувальний та тестовий набори для оцінки точності моделі, побудовані на основі стандартного розподілу даних.

```

Codeium: Refactor | Explain | Generate Docstring | ^
def GetAntiTestSetForUser(self, testSubject):
    trainset = self.fullTrainSet
    fill = trainset.global_mean
    anti_testset = []
    u = trainset.to_inner_uid(str(testSubject))
    user_items = set([j for (j, _) in trainset.ur[u]])
    anti_testset += [(trainset.to_raw_uid(u), trainset.to_raw_iid(i), fill) for
                     i in trainset.all_items() if
                     i not in user_items]
    return anti_testset

Codeium: Refactor | Explain | Generate Docstring | X
def GetTrainSet(self):
    return self.trainSet

Codeium: Refactor | Explain | Generate Docstring | X
def GetTestSet(self):
    return self.testSet

Codeium: Refactor | Explain | Generate Docstring | X
def GetLOOCVTrainSet(self):
    return self.LOOCVTrain

```

Рис. 2.43. Фрагмент класу EvaluationData

Загалом, цей клас надає все необхідне для ефективної оцінки різних аспектів рекомендаційних систем, дозволяючи працювати з різними стратегіями та метриками, такими як точність, різноманіття та популярність.

Клас `EvaluatedAlgorithm` використовується для оцінки ефективності різних алгоритмів рекомендаційних систем за допомогою кількох метрик, таких як точність, різноманіття та популярність. Він дозволяє обчислювати різні показники та оцінки алгоритмів, використовуючи набори даних, підготовлені класом `EvaluationData`.

Метод `__init__` ініціалізує клас, приймаючи алгоритм рекомендаційної системи та його ім'я. Це дозволяє використовувати один і той самий клас для різних алгоритмів, змінюючи лише їх параметри та ім'я.

```
def __init__(self, algorithm, name):  
    self.algorithm = algorithm  
    self.name = name
```

Рис. 2.44. Фрагмент класу `EvaluatedAlgorithm`

Метод `Evaluate` здійснює оцінку алгоритму на основі переданих наборів даних. Якщо параметр `doTopN` встановлений на `True`, метод оцінює алгоритм за допомогою різних метрик, зокрема `RMSE` (корінь середньоквадратичної помилки) та `MAE` (середня абсолютна помилка). Окрім цього, проводиться оцінка топ-N рекомендацій, зокрема за допомогою `Leave-One-Out` крос-валідації. Для цього алгоритм тренується на наборі даних, залишаючи один елемент для тестування, після чого обчислюються метрики, такі як `HR` (показник попадання), `sHR` (кумулятивний показник попадання), та `ARHR` (середній зворотний ранг попадання). Додатково, оцінюються властивості рекомендацій, зокрема охоплення користувачів, різноманіття та новизна на основі повного тренувального набору.

```

Codeium: Refactor | Explain | Generate Docstring | X
def Evaluate(self, evaluationData, doTopN, n=10, verbose=True):
    metrics = {}

    if (verbose):
        print("Evaluating accuracy...")
    self.algorithm.fit(evaluationData.GetTrainSet())
    predictions = self.algorithm.test(evaluationData.GetTestSet())
    metrics["RMSE"] = RecommenderMetrics.RMSE(predictions)
    metrics["MAE"] = RecommenderMetrics.MAE(predictions)

    if (doTopN):
        if (verbose):
            print("Evaluating top-N with leave-one-out...")
        self.algorithm.fit(evaluationData.GetLOOCVTrainSet())
        leftOutPredictions = self.algorithm.test(evaluationData.GetLOOCVTestSet())

        allPredictions = self.algorithm.test(evaluationData.GetLOOCVAntiTestSet())

        topNPPredicted = RecommenderMetrics.GetTopN(allPredictions, n)
        if (verbose):
            print("Computing hit-rate and rank metrics...")
        metrics["HR"] = RecommenderMetrics.HitRate(topNPPredicted, leftOutPredictions)
        metrics["cHR"] = RecommenderMetrics.CumulativeHitRate(topNPPredicted, leftOutPredictions)

        metrics["ARHR"] = RecommenderMetrics.AverageReciprocalHitRank(topNPPredicted, leftOutPredictions)

```

Рис. 2.45. Фрагмент класу EvaluatedAlgorithm

Методи GetName та GetAlgorithm використовуються для отримання ім'я алгоритму та самого алгоритму відповідно. Це дозволяє зручно взаємодіяти з різними алгоритмами та їх оцінками.

Таким чином, клас EvaluatedAlgorithm забезпечує гнучкість та можливість детального аналізу ефективності алгоритмів рекомендаційних систем, дозволяючи вимірювати різні аспекти їх роботи з різноманітними наборами метрик.

Клас Evaluator дозволяє організувати оцінку різних алгоритмів рекомендацій на одному наборі даних. Він приймає набір даних і рейтинги популярності продуктів, створюючи екземпляр класу EvaluationData, який використовується для подальшого тестування алгоритмів. Клас має список алгоритмів, які можна додавати за допомогою методу AddAlgorithm, що зберігає алгоритми у вигляді екземплярів класу EvaluatedAlgorithm.

Метод Evaluate відповідає за проведення оцінки для всіх доданих алгоритмів. Він обчислює різні метрики, включаючи RMSE, MAE, Coverage, Diversity і Novelty. Оцінка виконується для кожного алгоритму, а результати виводяться в табличному вигляді, де можна побачити, як алгоритми виконуються з різними метриками.

```

class Evaluator:
    algorithms = []

    Codeium: Refactor | Explain | Generate Docstring | X
    def __init__(self, dataset, rankings):
        ed = EvaluationData(dataset, rankings)
        self.dataset = ed

    Codeium: Refactor | Explain | Generate Docstring | X
    def AddAlgorithm(self, algorithm, name):
        alg = EvaluatedAlgorithm(algorithm, name)
        self.algorithms.append(alg)

    Codeium: Refactor | Explain | Generate Docstring | X
    def Evaluate(self, doTopN):
        results = {}
        for algorithm in self.algorithms:
            print("Evaluating ", algorithm.GetName(), "...")
            results[algorithm.GetName()] = algorithm.Evaluate(self.dataset, doTopN)

        # Print results
        print("\n")

        if (doTopN):
            print("{:<10} {:<10} {:<10} {:<10} {:<10} {:<10} {:<10} {:<10} {:<10} {:<10}".format(
                "Algorithm", "RMSE", "MAE", "HR", "cHR", "ARHR", "Coverage", "Diversity", "Novelty"))
            for (name, metrics) in results.items():
                print("{:<10} {:<10.4f} {:<10.4f} {:<10.4f} {:<10.4f} {:<10.4f} {:<10.4f} {:<10.4f} {:<10.4f} {:<10.4f}".format(
                    name, metrics["RMSE"], metrics["MAE"], metrics["HR"], metrics["cHR"], metrics["ARHR"],
                    metrics["Coverage"], metrics["Diversity"], metrics["Novelty"]))
            else:

```

Рис. 2.46. Фрагмент класу Evaluator

Метод SampleTopNRecs здійснює генерацію топ-N рекомендацій для конкретного користувача, використовуючи один із доданих алгоритмів. Він будує модель рекомендацій, отримує передбачення для користувача та сортує їх за оцінками. Далі він виводить список рекомендованих продуктів, відсортованих за їх оцінками. Цей метод допомагає оцінити, які саме продукти були б рекомендовані певному користувачу за допомогою різних алгоритмів.

```

Codeium: Refactor | Explain | Generate Docstring | X
def SampleTopNRecs(self, pl, testSubject=10, k=10):
    for algo in self.algorithms:
        print("\nUsing recommender ", algo.GetName())

        print("\nBuilding recommendation model...")
        trainSet = self.dataset.GetFullTrainSet()
        algo.GetAlgorithm().fit(trainSet)

        print("Computing recommendations...")
        testSet = self.dataset.GetAntiTestSetForUser(testSubject)

        predictions = algo.GetAlgorithm().test(testSet)

        recommendations = []

        print ("\nWe recommend:")
        for userID, productID, actualRating, estimatedRating, _ in predictions:
            intProductID = int(productID)
            recommendations.append((intProductID, estimatedRating))

        recommendations.sort(key=lambda x: x[1], reverse=True)

        for ratings in recommendations[:10]:
            print(pl.getProductname(ratings[0]), ratings[1])

```

Рис. 2.47. Фрагмент класу Evaluator

Висновки до розділу 2

У другому розділі було проведено дослідження кількох алгоритмів рекомендаційних систем на основі власного набору даних. Мета цього дослідження полягала в порівнянні різних підходів до рекомендацій та визначенні їхньої ефективності на практиці.

Спочатку було реалізовано та навчено просту рекурентну модель (RBM), що базується на автоенкодерах та має за мету виявлення унікальних характеристик користувачів та елементів. Після цього був досліджений алгоритм на основі вмісту (Content-Based KNN), який рекомендує елементи на основі їхніх властивостей та схожості з користувачем.

Крім того, був реалізований гібридний алгоритм, який поєднує результати кількох інших алгоритмів. Це дозволило отримати компроміс між різними методами та отримати більш точні та різноманітні рекомендації.

РОЗДІЛ 3. ОРГАНІЗАЦІЯ ЗАСОБІВ ВІЗУЛІЗАЦІЇ ТА АНАЛІЗУ ДАНИХ

3.1 Результати проведеного аналізу в таблицях діаграмах і графіках

Розглядаючи ефективність роботи алгоритмів рекомендаційних систем, надзвичайно важливо приділяти увагу ключовим метрикам, які дають змогу оцінити їхню якість і корисність у різних сценаріях. Кожна метрика має свої особливості та область застосування, завдяки чому стає можливим глибоке розуміння сильних і слабких сторін алгоритму.

Однією з найбільш поширених метрик оцінки є **RMSE (корінь середньоквадратичної помилки)**. Ця метрика дозволяє визначити середнє квадратичне відхилення між прогнозованими і фактичними значеннями. RMSE дуже чутлива до великих помилок, оскільки під час розрахунку кожна похибка підноситься до квадрату. Такий підхід робить її особливо корисною для аналізу систем, де великі відхилення в прогнозах можуть суттєво впливати на якість рекомендацій [36].

Наприклад, у системах рекомендацій фільмів великі помилки можуть означати, що користувачеві рекомендують фільм, який абсолютно не відповідає його вподобанням. Чим менше значення RMSE, тим точніше працює система, що є показником її здатності правильно оцінювати вподобання користувачів. Однак варто зазначити, що через квадратичну залежність RMSE може перебільшувати вплив окремих помилок, тому її слід використовувати разом із іншими метриками.

На противагу RMSE, **MAE (Mean Absolute Error – середнє абсолютне відхилення)** є більш стабільною метрикою, яка вимірює середнє абсолютне відхилення між прогнозованими і фактичними значеннями. Оскільки MAE не квадратично зважає похибки, вона менш чутлива до великих помилок. Це робить її особливо корисною в системах, де основною метою є загальна точність рекомендацій без надмірного акценту на поодинокі невідповідності [37].

Наприклад, у системах рекомендацій книг MAE може бути більш релевантною метрикою, оскільки в таких сценаріях користувачі зазвичай готові сприймати деякі відхилення у своїх очікуваннях, якщо загальний рівень рекомендацій залишається високим. Порівнюючи значення MAE та RMSE, можна отримати більш повну картину роботи системи та її поведінки щодо різних типів помилок.

Метрика покриття (Coverage) відіграє важливу роль у визначенні того, наскільки широкий спектр користувачів отримує корисні рекомендації. Coverage вимірює частку користувачів, які отримують рекомендації з оцінкою, що перевищує певний поріг. Ця метрика показує, наскільки рекомендаційна система здатна задовольнити потреби великої кількості користувачів, що є критично важливим для масштабних платформ [38].

Наприклад, у системі рекомендацій для онлайн-магазину високе покриття означає, що значна частина клієнтів отримує рекомендації, які відповідають їхнім інтересам. Низьке значення Coverage може свідчити про те, що система зосереджується лише на певній частині користувачів або продуктів, що може обмежити її загальну ефективність.

Різноманітність (Diversity) є ще однією важливою метрикою, яка оцінює, наскільки різноманітними є рекомендації, запропоновані системою. Вона вимірює середню відстань або схожість між усіма можливими парами рекомендованих елементів для кожного користувача. Високе значення Diversity означає, що система пропонує користувачам широкий вибір, охоплюючи різні категорії продуктів або контенту [39].

Наприклад, у стримінгових сервісах рекомендації, які включають як популярні, так і менш відомі фільми або серіали, можуть підвищити залученість користувачів, надаючи їм змогу досліджувати новий контент. Низька різноманітність може призвести до ситуації, коли користувач отримує однакові або дуже схожі рекомендації, що знижує загальну задоволеність від використання системи.

Новизна (Novelty) вимірює, наскільки унікальними та несподіваними є рекомендації для користувачів. Вона враховує середній ранг популярності рекомендованих елементів, дозволяючи оцінити, наскільки система здатна запропонувати щось нове, а не лише популярний або очікуваний контент. Високе значення Novelty означає, що система здатна запропонувати користувачам цікавий контент, який вони, можливо, не знайшли б самостійно [40].

Наприклад, у системі рекомендацій музики високий рівень новизни може сприяти відкриттю нових жанрів або виконавців, що підвищує інтерес користувачів до платформи. Однак важливо збалансувати новизну з точністю, оскільки надмірна увага до унікальних рекомендацій може знизити релевантність системи для окремих користувачів.

Кожна з описаних метрик має свій внесок у загальне оцінювання ефективності рекомендаційної системи. RMSE і MAE дозволяють оцінити точність прогнозів, Coverage допомагає зрозуміти масштаб впливу системи, Diversity гарантує різноманітність рекомендацій, а Novelty забезпечує їхню цікавість для користувачів. Баланс між цими метриками дає змогу створити ефективну, інклюзивну та цікаву рекомендаційну систему, здатну задовольнити різноманітні потреби користувачів.

У процесі дослідження можливостей рекомендаційних систем було проведено серію глибоких і всебічних експериментів, спрямованих на оцінку впливу різних параметрів і налаштувань на їхню продуктивність. Рекомендаційні системи є надзвичайно важливим інструментом у сучасному інформаційному просторі, оскільки вони допомагають користувачам знайти потрібний контент серед великої кількості варіантів. Вибір оптимальних налаштувань та розуміння їхнього впливу на різні метрики дозволяє забезпечити високу якість рекомендацій, що є критично важливим для будь-якої галузі застосування.

Першим етапом дослідження було оцінювання **впливу розміру датасету на роботу рекомендаційної системи**. Для цього було створено кілька підмножин даних: набори, які становили 2015 записів, середні — 3980 записів, а також великі, що включали 6007 записів. Розмір датасету є важливим фактором,

який визначає ефективність алгоритмів. Менші набори даних дозволяють швидко тестувати моделі, проте часто мають недостатній рівень узагальнення, що впливає на точність рекомендацій.

У ході експерименту було проведено тестування двох алгоритмів: RBM і KNN, які оцінювалися за ключовими метриками. Нижче наведено результати для кожного алгоритму.

Для алгоритму **RBM** отримано такі результати:

Таблиця 3.1

Залежність метрик RMSE, MAE, Coverage, Diversity, Novelty для RBM від розміру датасету

Розмір датасету	RMSE	MAE	Coverage	Diversity	Novelty
2015,000	1,520	1,498	1,000	0,025	211,771
3980,000	1,481	1,454	1,000	0,040	249,392
6007,000	1,462	1,438	1,000	0,043	280,953

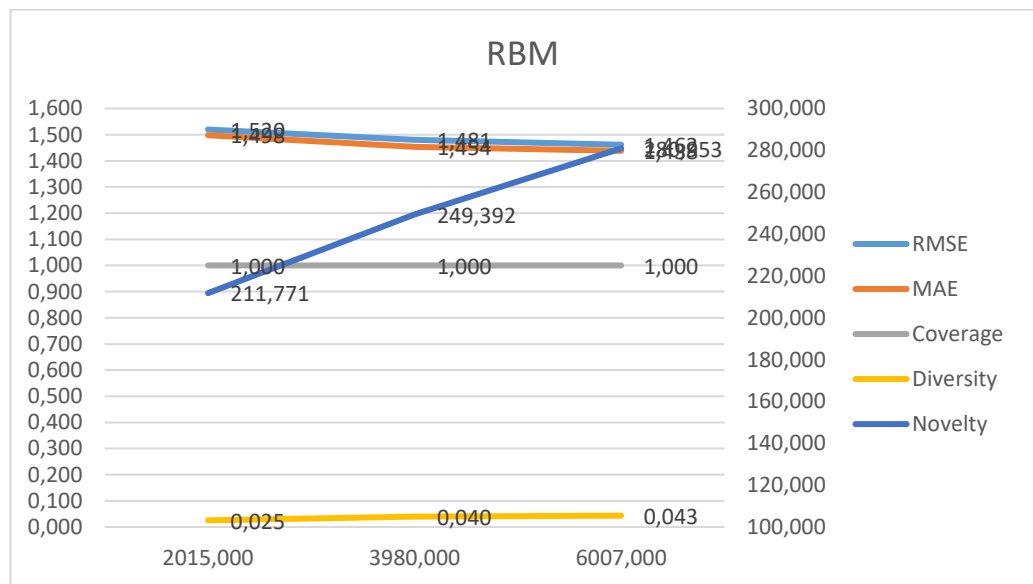


Рис. 3.1. Залежність метрик RMSE, MAE, Coverage, Diversity, Novelty для RBM від розміру датасету

На невеликих датасетах алгоритм демонструє високу точність, але обмежену різноманітність та новизну рекомендацій. Із середніми обсягами даних точність алгоритму покращується, а різноманітність та новизна значно зростають. Для великих датасетів зберігається стабільне покриття, а також спостерігається максимальний рівень новизни рекомендацій, що підкреслює перевагу роботи з великими обсягами даних для складних систем рекомендацій.

Аналіз результатів показує, що зі збільшенням обсягу даних метрики точності (RMSE і MAE) покращуються, що свідчить про краще узагальнення моделі. Novelty також демонструє зростання, відображаючи підвищення новизни рекомендацій. У той же час Diversity залишається низьким, що може свідчити про схильність моделі до формування менш різноманітних рекомендацій.

Для алгоритму **KNN** результати такі:

Таблиця 3.2

Залежність метрик RMSE, MAE, Coverage, Diversity, Novelty для KNN від розміру датасету

Розмір датасету	RMSE	MAE	Coverage	Diversity	Novelty
2015,000	1,011	0,960	1,000	0,055	395,329
3980,000	0,921	0,832	1,000	0,069	393,683
6007,000	1,022	0,960	1,000	0,065	397,829

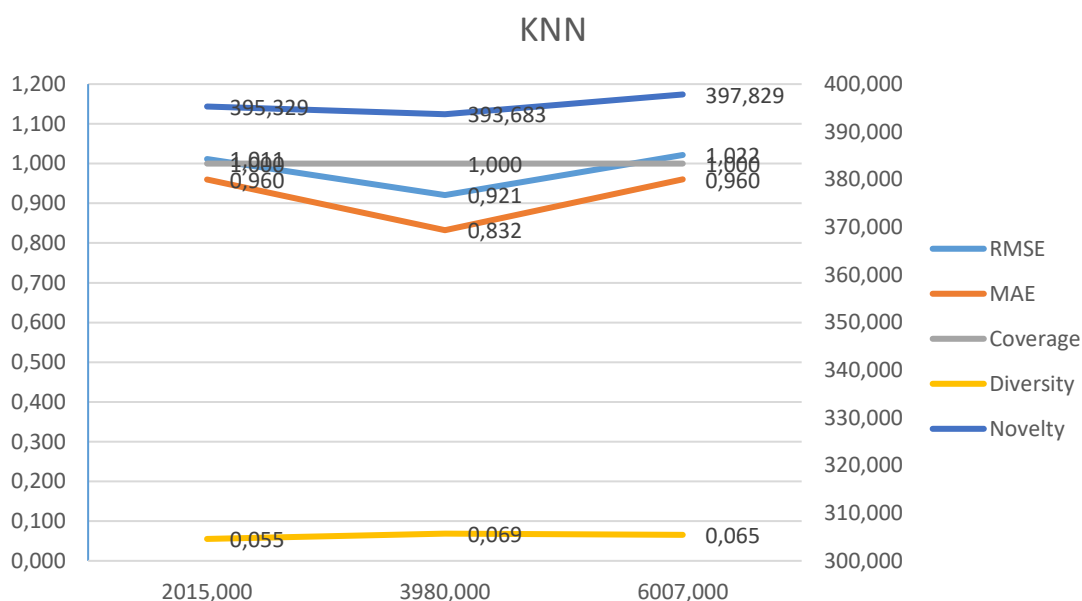


Рис. 3.2. Залежність метрик RMSE, MAE, Coverage, Diversity, Novelty для KNN від розміру датасету

У міру зростання обсягу даних точність (RMSE та MAE) демонструє тенденцію до незначних коливань, однак система зберігає високу здатність генерувати релевантні рекомендації для різних груп користувачів. Різноманітність рекомендацій досягає максимальних значень при середніх розмірах датасету, тоді як новизна постійно зростає із збільшенням обсягу даних.

Аналіз цих даних свідчить, що KNN демонструє кращу різноманітність рекомендацій у порівнянні з RBM, особливо на більших датасетах. Novelty залишається стабільно високим, що забезпечує цікаві рекомендації для користувачів. Проте RMSE демонструє незначне зростання для максимального обсягу даних, що може бути спричинено складністю моделі при обробці великої кількості записів.

Другий експеримент мав на меті **оцінити вплив кількості найближчих сусідів K на ефективність рекомендацій**. Алгоритм KNN (k-найближчих сусідів) є одним із найбільш поширених методів у рекомендаційних системах, оскільки він дозволяє знайти схожі об'єкти на основі обраних метрик. У цьому експерименті було протестовано кілька значень K: 5, 10, 20, 35, 70 та 100.

Параметр K визначає кількість елементів, які враховуються під час формування рекомендацій. Низькі значення K забезпечують точніші рекомендації для популярних елементів, але обмежують охоплення менш відомих варіантів. З іншого боку, великі значення K дозволяють системі розширити список рекомендацій, охоплюючи ширший спектр можливих варіантів. Проте це може призводити до зниження точності, оскільки враховуються менш релевантні сусіди.

Таблиця 3.3

Вплив параметра K на метрики алгоритму KNN

Параметр K	RMSE	MAE	Coverage	Diversity	Novelty
k=5	1,022	0,960	1,000	0,065	397,829
k=10	0,796	0,750	1,000	0,071	400,129
k=20	0,583	0,514	1,000	0,066	401,653
k=35	0,426	0,356	1,000	0,060	383,912
k=70	0,333	0,268	1,000	0,062	382,206
k=100	0,310	0,242	1,000	0,064	341,624

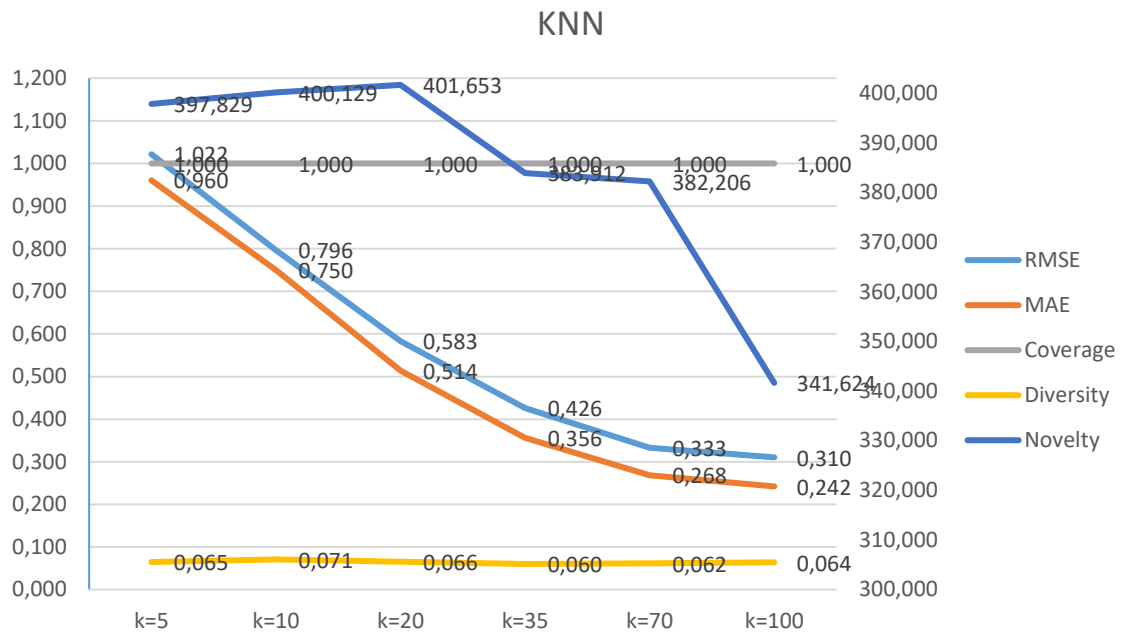


Рис. 3.3. Вплив параметра K на метрики алгоритму KNN

У результаті експерименту було встановлено, що зі збільшенням K точність моделей (RMSE, MAE) поступово зростає, але після певного моменту цей приріст починає сповільнюватись або навіть знижуватись. При значенні K=5 система демонструє початковий рівень точності, але новизна рекомендацій залишається обмеженою. Зі зростанням до K=10 спостерігається покращення не лише точності, але й різноманітності рекомендацій. Максимальні показники новизни досягаються при значенні K=20, однак подальше збільшення параметра призводить до зниження цього показника.

Особливістю результатів є стабільність покриття, яке для всіх значень K залишалося максимальним, що свідчить про здатність алгоритму аналізувати широкий спектр варіантів. Водночас різноманітність рекомендацій мала максимальне значення при K=10, після чого почала поступово зменшуватись.

Графіки, побудовані за результатами цього експерименту, наочно демонструють вплив параметра K на роботу системи рекомендацій. Зокрема, можна зробити висновок, що вибір оптимального значення K залежить від пріоритетів: для досягнення точності варто обрати K=20, тоді як для покращення різноманітності та новизни найкраще підійде K=10.

Третій етап дослідження був присвячений **аналізу впливу кількості епох під час навчання алгоритму RBM (обмежена масштабована машина)**. Навчання моделі є багатоетапним процесом, у якому кількість епох визначає, скільки разів система проходить через весь набір даних для оптимізації своїх параметрів. Це один із ключових аспектів, що впливає на якість отриманих рекомендацій. У цьому експерименті було протестовано кілька значень кількості епох: 100, 250 і 500.

На діаграмі, наведеної нижче, можна побачити, як змінювалися основні метрики (RMSE, MAE, покриття, різноманітність та новизна) залежно від кількості епох. Графіки чітко демонструють, що на початкових етапах (100 епох) модель ще недостатньо «вивчила» закономірності в даних, що призводить до вищих значень RMSE та MAE. Зокрема, модель на цьому етапі менш точно відображає уподобання користувачів, оскільки параметри ще не досягли оптимального стану.

Таблиця 3.4

Вплив кількості епох на метрики алгоритму RBM

Кількість епох	RMSE	MAE	Coverage	Diversity	Novelty
Epochs 100	1,462	1,438	1,000	0,043	280,953
Epochs 250	1,358	1,232	1,000	0,049	300,571
Epochs 500	1,470	1,446	1,000	0,048	288,088

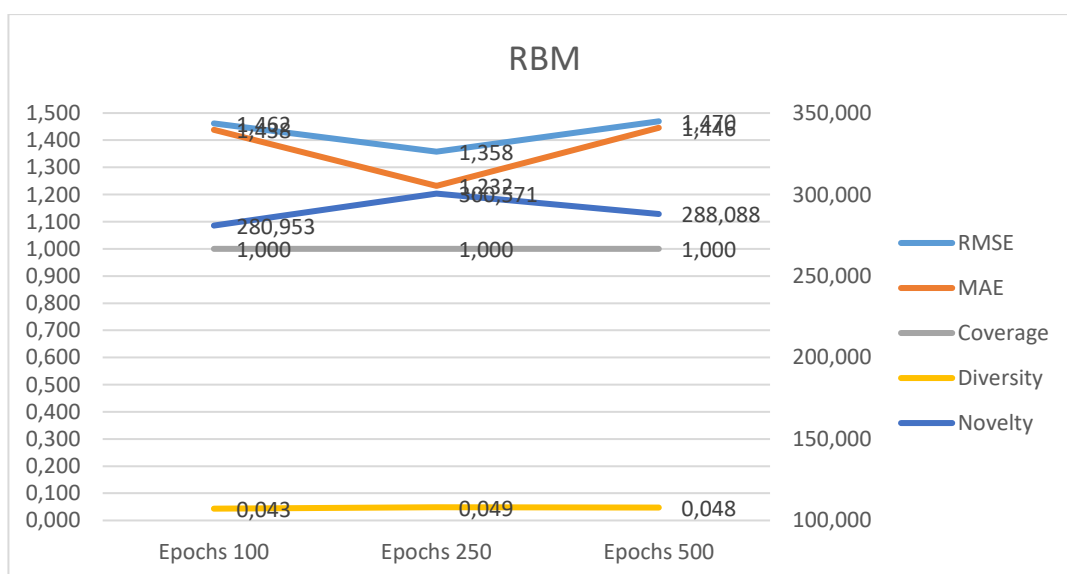


Рис. 3.4. Вплив кількості епох на метрики алгоритму RBM

Із збільшенням кількості епох до 250 спостерігається значне покращення метрик. Це свідчить про те, що модель починає краще адаптуватися до даних, покращуючи точність рекомендацій. У цей період RMSE і MAE зменшуються до оптимальних значень, а різноманітність і новизна залишаються на високому рівні, що є важливим показником для забезпечення якісного користувацького досвіду.

Однак, коли кількість епох збільшується до 500, модель демонструє деяке зниження ефективності за певними метриками, такими як різноманітність і новизна. Це може бути пов'язано з перенавчанням, коли модель починає занадто точно відтворювати дані навчальної вибірки, втрачаючи здатність узагальнювати нові дані. Хоча покриття залишається стабільним на рівні 100%, інші показники вже не демонструють значного покращення або навіть трохи погіршуються.

Таким чином, результати експерименту підтверджують, що вибір оптимальної кількості епох залежить від пріоритетів системи. Якщо головним завданням є забезпечення максимальної точності рекомендацій, варто зупинитися на 250 епохах, оскільки саме на цьому етапі модель демонструє найкращий баланс між RMSE і MAE. Водночас, для уникнення ризику перенавчання і забезпечення більшої узагальнювальної здатності моделі, не рекомендується використовувати надто велику кількість епох, як-от 500.

Це дослідження підкреслює важливість ретельного вибору параметрів моделі, таких як кількість епох, адже навіть незначна зміна цього параметра може вплинути на результати системи рекомендацій.

Четвертий експеримент був зосереджений на **дослідженні впливу трьох ключових гіперпараметрів моделі RBM: hiddenDim (кількість прихованих шарів), learningRate (швидкість навчання) і batchSize (розмір партії)**. Кожен із цих параметрів має вирішальний вплив на ефективність моделі, і їх оптимальні значення залежать від конкретного набору даних.

Вплив HiddenDim (кількість прихованих шарів)

Таблиця 3.5

Вплив параметру hiddenDim на метрики алгоритму RBM

Приховані шари	RMSE	MAE	Coverage	Diversity	Novelty
hidden Dimension 100	1,458	1,434	1,000	0,055	286,982
hidden Dimension 200	1,370	1,246	1,000	0,052	305,518
hidden Dimension 300	1,472	1,448	1,000	0,048	276,188

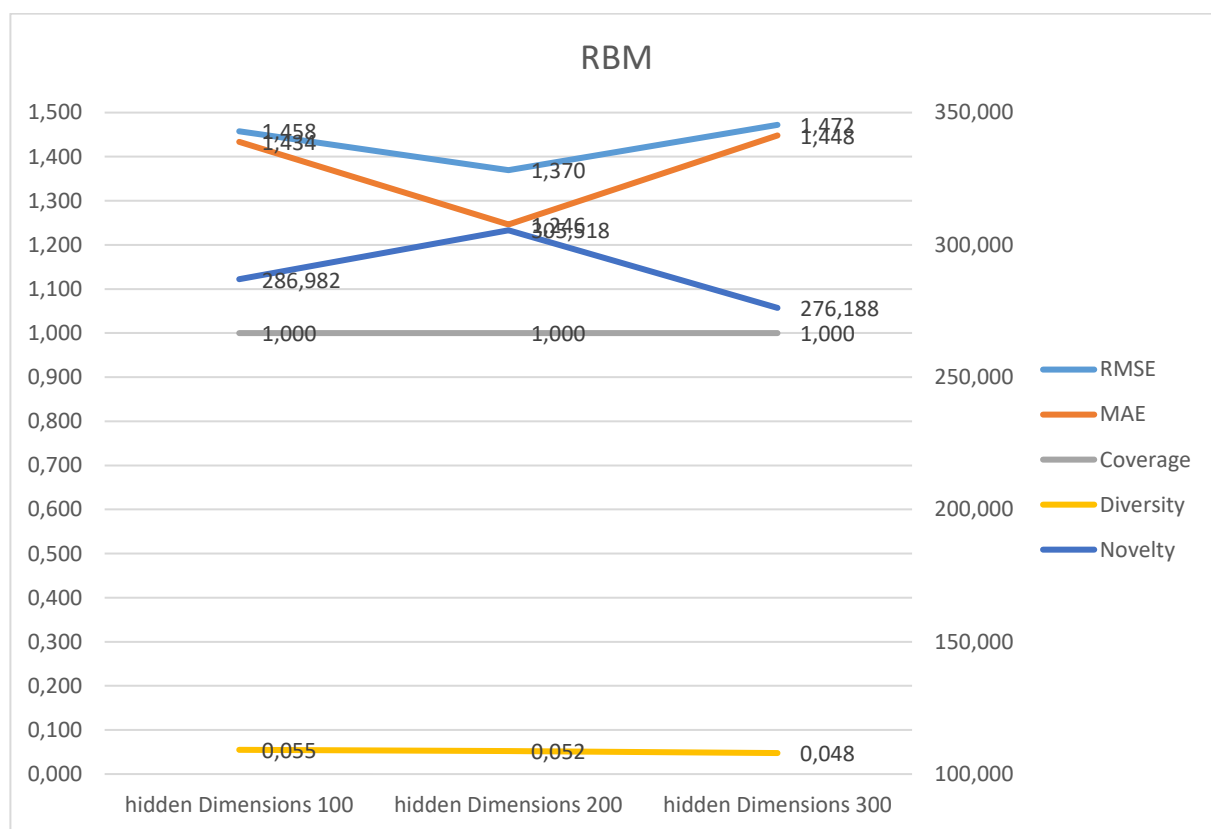


Рис. 3.5. Вплив параметру hiddenDim на метрики алгоритму RBM

Параметр hiddenDim визначає кількість латентних факторів, які модель використовує для представлення даних. Менші значення цього параметра можуть призвести до того, що модель не буде здатна врахувати всю складність структури даних, у той час як занадто великі значення можуть значно збільшити час обчислень та ускладнити модель, що в свою чергу може спричинити перенавчання. В результатах експерименту було показано, що оптимальним значенням для цього параметра є hiddenDim = 200. Це значення дозволяє досягти кращого балансу між точністю та швидкістю навчання. Як показує графік, зменшення кількості латентних факторів до 100 призводить до погіршення якості

результатів, а збільшення до 300, хоча і зменшує точність за певними метриками, не надає суттєвих покращень у порівнянні з 200.

Вплив LearningRate (Швидкість Навчання)

Таблиця 3.6

Вплив параметру LearningRate на метрики алгоритму RBM

Швидкість навчання	RMSE	MAE	Coverage	Diversity	Novelty
learning Rate 0.001	1,458	1,434	1,000	0,055	286,982
learning Rate 0.005	1,471	1,445	1,000	0,041	277,082
learning Rate 0.01	1,369	1,305	1,000	0,054	305,371
learning Rate 0.05	1,469	1,446	1,000	0,040	256,324
learning Rate 0.1	1,428	1,373	1,000	0,049	289,477

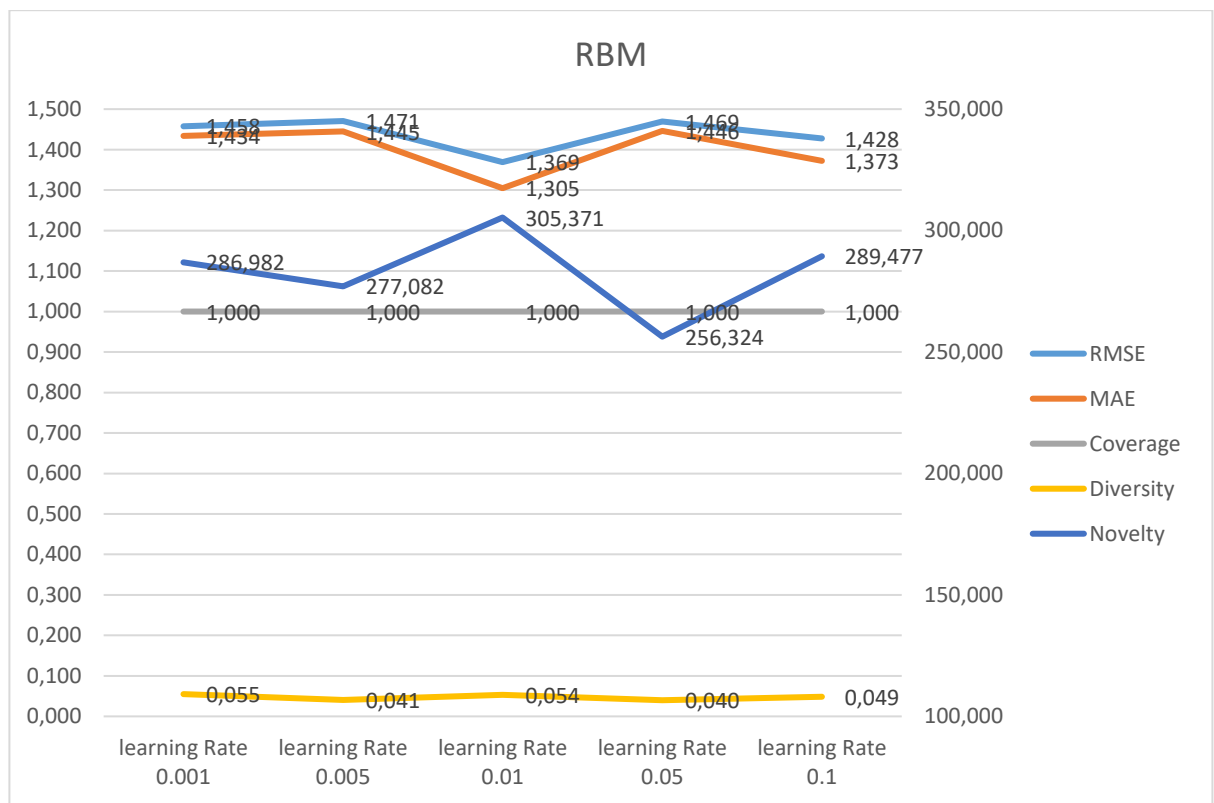


Рис. 3.6. Вплив параметру LearningRate на метрики алгоритму RBM

Параметр learningRate визначає, як швидко модель коригує свої ваги на основі отриманих помилок. Занадто мала швидкість навчання може призвести до дуже повільного навчання, в той час як надмірно велика швидкість може викликати нестабільність, де модель не зможе знайти оптимальні ваги. У нашому експерименті найкращі результати були отримані при learningRate = 0.01, де показники точності, різноманітності та новизни були оптимальними. Зменшення швидкості навчання до 0.001 призводить до значного зниження продуктивності,

а збільшення до 0.05 призводить до нестабільності, що відображається на показниках MAE та RMSE.

Вплив BatchSize (Розмір Партії)

Таблиця 3.7

Вплив параметру BatchSize на метрики алгоритму RBM

Розмір Партії	RMSE	MAE	Coverage	Diversity	Novelty
batch Size 32	1,408	1,384	1,000	0,053	298,635
batch Size 64	1,476	1,452	1,000	0,048	259,382
batch Size 128	1,466	1,443	1,000	0,044	270,177
batch Size 256	1,368	1,314	1,000	0,049	309,129

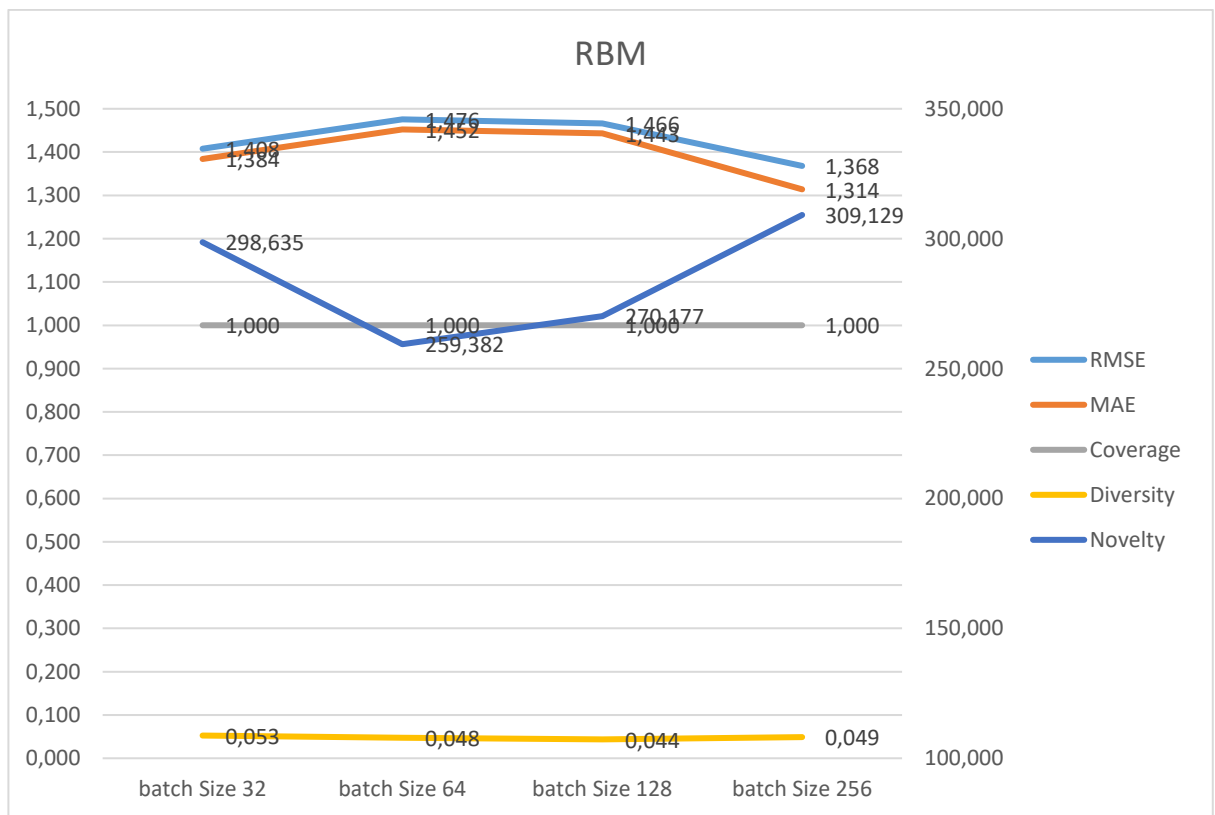


Рис. 3.7. Вплив параметру BatchSize на метрики алгоритму RBM

Параметр batchSize визначає кількість прикладів, які обробляються перед оновленням ваг. Менші розміри партій можуть привести до більш точних, але повільних оновлень, в той час як більші партії дозволяють швидше оновлювати ваги, але зменшують точність. У цьому експерименті найкращі результати були досягнуті при batchSize = 256. Параметр batchSize 32 призводить до більш високих показників точності, але менша різноманітність, а великі партії, як

показано на діаграмі, зменшують точність, але зберігають стабільне покриття і новизну.

Результати експерименту показують, що оптимальні значення для кожного з гіперпараметрів сильно залежали від конкретного набору даних, зокрема, найбільш збалансовані показники досягалися при $\text{hiddenDim} = 200$, $\text{learningRate} = 0.01$ і $\text{batchSize} = 256$. Ці налаштування дозволили моделі забезпечити як високу точність, так і достатню різноманітність та новизну рекомендацій, що є ключовими аспектами при розробці системи рекомендацій.

На фінальному етапі дослідження було зосереджено увагу на **впливі комбінованого використання алгоритмів KNN і RBM у межах гібридної моделі**. Така гібридизація дозволяє використовувати сильні сторони кожного з підходів: алгоритм KNN сприяє створенню рекомендацій на основі схожості між елементами, а RBM додає можливість урахування латентних факторів, що дозволяє моделі краще розуміти структуру даних. В експерименті було протестовано різні комбінації вагових коефіцієнтів для RBM і KNN, що дозволило оцінити їхній вплив на ключові метрики.

Залежно від комбінації коефіцієнтів, отримано різні результати щодо точності, різноманітності та новизни. Найкращі показники точності були досягнуті при вагових коефіцієнтах $[0.5, 0.5]$, що забезпечило оптимальний баланс між впливом KNN і RBM. Однак максимальні значення різноманітності та новизни спостерігались при використанні ваг $[0.3, 0.7]$, де ваги KNN мали більший вплив. Це свідчить про те, що підвищення ваги KNN може значно покращити здатність моделі генерувати нові та різноманітні рекомендації.

Таблиця 3.8

Вплив вагових коефіцієнтів на показники RMSE, MAE, Coverage, Diversity та Novelty у гібридній моделі KNN та RBM

	RMSE	MAE	Coverage	Diversity	Novelty
[0.7, 0.3]	1,088	1,059	1,000	0,051	287,482
[0.6, 0.4]	0,969	0,938	1,000	0,055	297,282
[0.5, 0.5]	0,836	0,801	1,000	0,065	321,029
[0.4, 0.6]	0,921	0,884	1,000	0,069	392,765
[0.3, 0.7]	0,831	0,791	1,000	0,069	401,371

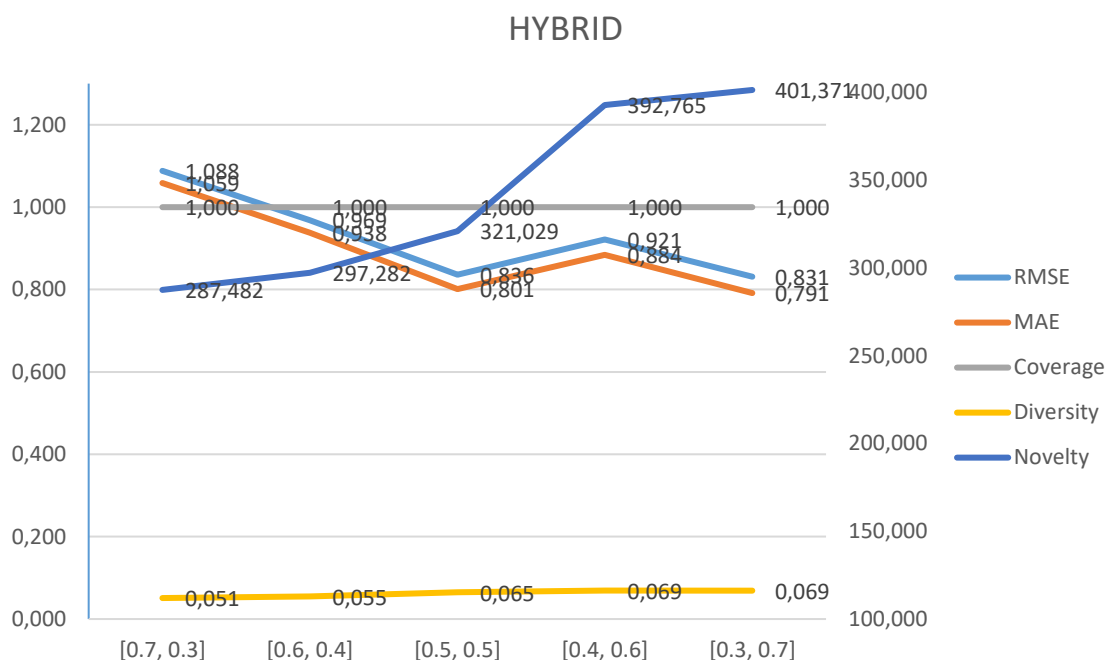


Рис. 3.8. Вплив вагових коефіцієнтів на показники RMSE, MAE, Coverage, Diversity та Novelty у гібридній моделі KNN та RBM

Результати експерименту показали, що правильний баланс між вагами алгоритмів KNN і RBM може значно покращити метрики, такі як різноманітність і новизна, при збереженні високих показників точності. Графік, побудований на основі результатів цього експерименту, демонструє, як зміна вагових коефіцієнтів впливає на різні аспекти роботи рекомендаційної системи. Таким чином, отримані результати підтверджують важливість ретельного налаштування параметрів для досягнення оптимальних результатів у конкретних умовах використання системи.

3.2 Аналіз результатів

У дослідженні було порівняно три алгоритми для рекомендаційної системи: RBM, ContentKNN та Hybrid. Кожен з них оцінювався за точністю, покриттям, різноманітністю та новизною. Це дозволило визначити сильні та слабкі сторони кожного алгоритму з метою їх вдосконалення в майбутньому.

Алгоритм **RBM (Restricted Boltzmann Machine)** демонструє добрі результати, особливо у частині охоплення (Coverage), але з високими значеннями

RMSE та MAE, що вказує на суттєві похибки в прогнозах. Ці високі похибки можуть бути результатом недостатньо великого або різноманітного датасету, що обмежує здатність моделі точно передбачати оцінки. Збільшення обсягу та різноманітності даних допомогло б знизити ці помилки та підвищити точність прогнозів.

```
Using recommender RBM

Building recommendation model...
Processing user 0
Computing recommendations...

We recommend:
BIGMFNEEPS + 9 SUBGROUPS INVI 2.493393
Stellar Beauty 2.4933784
Shadow Ninja 2.4200766
Mickey Mouse space suit 2.4200766
Sea World 2.4200766
ةباغ مل اعل او ةكرعم ةايحلا 2.4200766
يئانغ رعش - ةدرقم عاقإ 2.4200766
Umaaraw, Umuulan Nature Study 2.4200766
Philippine History Timeline 2.4200766
LEARN - Online business guide 2.4200766
```

Рис 3.9 Рекомендації продуктів для RBM

Алгоритм **KNN (Content-based K-Nearest Neighbors)** показує значно кращі результати порівняно з RBM, з набагато нижчими значеннями RMSE та MAE, що свідчить про кращу точність рекомендацій. Проте є кілька аспектів, які потребують уваги для покращення роботи цього алгоритму. По-перше, необхідно інтегрувати інформацію про країну користувача в розрахунки подібності. Географічне розташування може значно впливати на переваги користувачів, і використання цієї інформації дозволить зробити рекомендації більш персоналізованими та точними. По-друге, система категорій повинна бути стандартизована. Наразі користувачі можуть створювати власні категорії, що унеможлиблює їх використання під-час формування рекомендацій. Введення загальної системи категорій дозволить забезпечити консистентність і покращити точність рекомендацій

```
Using recommender ContentKNN

Building recommendation model...
Computing content-based similarity matrix...
...done.
Computing recommendations...

We recommend:
Reasing 🌟#9 1.5312978780887594
Philippine History Timeline 1.528714790709831
Umaaraw, Umuulan Nature Study 1.5271819629118792
Mindfulness - Ebook 1.5237484782778936
Reasing 🌟#6 1.5235716918195088
Ректификация. Прогноз. 1.5235135630009247
MGNF Virtual Art Museum #2 1.5231341482407388
Sneaky Link Tape #1 ❌ 1.5209580183420663
You're In Hell But It's Okay 1.514196448548149
LEARN - Online business guide 1.5121773306526587
```

Рис 3.10 Рекомендації продуктів для ContentKNN

Гібридний алгоритм показав збалансовані результати, з помірними значеннями RMSE та MAE, що вказує на деякі помилки в прогнозах, але все ще є місце для вдосконалення. Однак, цей алгоритм має добре збалансовані показники охоплення та новизни, що робить його сильним вибором для створення різноманітних рекомендацій. Однак варто звернути увагу на те, що висока новизна може призвести до того, що деякі користувачі не отримають достатньо релевантних рекомендацій. Для покращення роботи цього алгоритму можна поєднати переваги всіх методів, доповнивши їх інформацією про країни та стандартизованими категоріями. Це дозволить не лише покращити точність, але й підвищити якість рекомендацій для більшої кількості користувачів.

```
LEARN - Online business guide 1.5121773306526587

Using recommender Hybrid

Building recommendation model...
Processing user 0
Computing content-based similarity matrix...
...done.
Computing recommendations...

We recommend:
Reasing 🌟#9 1.8084570311044184
Philippine History Timeline 1.8075740114638146
Umaaraw, Umuulan Nature Study 1.8068118864989144
Ректификация. Прогноз. 1.8046515733212574
Reasing 🌟#6 1.804332707717157
Mindfulness - Ebook 1.804166975453695
MGNF Virtual Art Museum #2 1.8040251827285374
Sneaky Link Tape #1 ❌ 1.8034322949442947
اباغ مل اعل او ةكرعم ة ايجلا 1.7977489745286626
You're In Hell But It's Okay 1.7975369533670413
```

Рис 3.11 Рекомендації продуктів для Hybrid

Висновки до розділу 3

У третьому розділі проведено дослідження ефективності рекомендаційних моделей на основі набору даних платформи з продажу товарів. Метою цього дослідження було визначення найкращих підходів до рекомендацій та оцінка їхньої здатності задовольняти потреби користувачів в реальних умовах.

Модель RBM, хоча і має потенціал для виявлення прихованих зв'язків між користувачами та товарами, продемонструвала обмежену ефективність у даному випадку. Порівняно з іншими моделями, RBM показала менш стабільні результати за точністю та покриттям, особливо при використанні меншої кількості епох навчання. Це вказує на те, що модель потребує більше епох для стабільного досягнення хороших результатів, а також можливе покращення через більш ретельну настройку гіперпараметрів. Точність прогнозів RBM була нижчою за ContentKNN, що підтверджує важливість правильно налаштованих параметрів для отримання конкурентоспроможних результатів.

Гібридний алгоритм виявився проміжним рішенням, яке поєднало сильні сторони як RBM, так і ContentKNN. Він показав хорошу точність, але не досягнув високого рівня новизни і різноманітності, який продемонструвала модель ContentKNN. Проте Hybrid модель потребує додаткової оптимізації і більших обсягів навчання, щоб досягти кращих результатів у балансі між точністю та різноманітністю. Це свідчить, що для отримання ідеальних результатів необхідно більш ретельно налаштувати ваги компонентів і продовжити навчання моделі.

ContentKNN виявилась найбільш перспективною моделлю для даного випадку, зокрема завдяки її здатності генерувати різноманітні рекомендації та добре працювати з контентом продуктів. Модель продемонструвала не тільки високу точність, але й відмінні показники за метриками новизни та різноманітності, що робить її ідеальним вибором для задач, де важлива максимальна різноманітність і новизна в рекомендаціях. Проте варто зазначити,

що її точність може дещо знижуватись при використанні дуже великого параметра K , тому потрібно ретельно підбирати цей параметр.

Загалом, ContentKNN виглядає найбільш перспективним підходом для рекомендаційних систем, однак подальше вдосконалення параметрів RBM та гібридного алгоритму може дати додаткові переваги в умовах більш складних та динамічних даних.

ВИСНОВКИ

У рамках дослідження було проведено всебічний аналіз рекомендаційних систем із застосуванням методів штучного інтелекту. Особливу увагу було приділено виявленню та аналізу проблем, що виникають під час використання різних алгоритмів фільтрації, таких як обмежена різноманітність рекомендацій, недостатня точність чи новизна. Було проаналізовано можливі способи усунення цих недоліків та покращення якості рекомендацій.

Дослідження охопило широкий спектр алгоритмів, які використовуються для розробки рекомендаційних систем. У процесі роботи були детально вивчені різні підходи, включаючи модель обмеженої машини Больцмана (RBM), алгоритм на основі контенту (Content-Based KNN) та гібридні моделі. Було реалізовано ці алгоритми, а також досліджено їхню ефективність за різними метриками, такими як точність, покриття, різноманітність і новизна.

На основі даних платформи з продажу товарів було проведено тестування моделей для оцінки їхньої придатності у практичному застосуванні. Алгоритм Content-Based KNN показав найкращі результати, забезпечуючи високу точність та здатність формувати різноманітні й релевантні рекомендації. Модель RBM продемонструвала недостатню ефективність, що може бути пов'язано з необхідністю подальшого налаштування параметрів або збільшення тривалості навчання. Гібридний підхід виявився компромісним рішенням, яке поєднує сильні сторони кількох методів, хоча й потребує вдосконалення для досягнення оптимальних результатів.

Загалом, результати дослідження підтвердили важливість використання комбінованих підходів у рекомендаційних системах, оскільки вони здатні покращити точність і врахувати різні аспекти персоналізації. Крім того, було виявлено, що налаштування моделей та адаптація алгоритмів під особливості даних є критичними для забезпечення ефективності рекомендацій. Таким чином, робота закладає основу для подальшого розвитку рекомендаційної системи, орієнтовану на точність, різноманітність та новизну рекомендацій.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Build Recommendation System [Електронний ресурс] – Режим доступу до ресурсу: <https://www.leewayhertz.com/build-recommendation-system/#Types-of-AI-powered-recommendation-systems>
2. Guy Shani, David Heckerman, Ronen I. Brafman. An MDP-Based Recommender System. Journal of Machine Learning Research. Vol 6 (2005). Pp. 1265-1295. <https://www.jmlr.org/papers/volume6/shani05a/shani05a.pdf>
3. Maklin. C. Memory-Based Collaborative Filtering: User-Based [Електронний ресурс] – Режим доступу до ресурсу: <https://medium.com/@corymaklin/memory-based-collaborative-filtering-user-based-42b2679c6fb5>
4. Collaborative Filtering: Basics of Recommendation Systems [Електронний ресурс] – Режим доступу до ресурсу: <https://developers.google.com/machine-learning/recommendation/collaborative/basics>
5. ML - Content-Based Recommender System [Електронний ресурс] – Режим доступу до ресурсу: <https://www.geeksforgeeks.org/ml-content-based-recommender-system/>
6. Hybrid Recommender Systems: A Beginner's Guide [Електронний ресурс] – Режим доступу до ресурсу: <https://marketsy.ai/blog/hybrid-recommender-systems-beginners-guide>
7. Knowledge-Based Recommendation Systems [Електронний ресурс] – Режим доступу до ресурсу: <https://medium.com/@paul0/knowledge-based-rs-b4547b84108a>
8. Critical Challenges of Recommendation Engines [Електронний ресурс] – Режим доступу до ресурсу: <https://www.appier.com/en/blog/7-critical-challenges-of-recommendation-engines>
9. Friedman, A., Knijnenburg, B.P., Vanhecke, K., Martens, L., Berkovsky, S. (2015). Privacy Aspects of Recommender Systems. In: Ricci, F., Rokach, L.,

Shapira, B. (eds) Recommender Systems Handbook. Springer, Boston, MA.
https://doi.org/10.1007/978-1-4899-7637-6_19

10. The Cold-Start Problem for Recommender Systems [Электронный ресурс] – Режим доступа до ресурсу: <https://medium.com/@markmilankovich/the-cold-start-problem-for-recommender-systems-89a76505a7>

11. Building Text-Based Recommender Systems with NLP [Электронный ресурс] – Режим доступа до ресурсу: <https://mobidev.biz/blog/how-to-build-text-based-recommender-system-with-nlp>

12. What is PostgreSQL? [Электронный ресурс] – Режим доступа до ресурсу: <https://www.ibm.com/topics/postgresql>

13. Getting Started with Pandas [Электронный ресурс] – Режим доступа до ресурсу: https://pandas.pydata.org/docs/getting_started/index.html#getting-started

14. Pandas: Absolute Beginners Guide [Электронный ресурс] – Режим доступа до ресурсу: https://numpy.org/doc/stable/user/absolute_beginners.html

15. Pandas Tutorial: DataFrame in Python [Электронный ресурс] – Режим доступа до ресурсу: [https://www.datacamp.com/tutorial/pandas-tutorial-dataframe-python?utm_source=google&utm_medium=paid_search&utm_campaignid=19589720824&utm_adgroupid=157156376311&utm_device=c&utm_keyword=&utm_matchtype=&utm_network=g&utm_adposition=&utm_creative=724847714770&utm_targetid=dsa-](https://www.datacamp.com/tutorial/pandas-tutorial-dataframe-python?utm_source=google&utm_medium=paid_search&utm_campaignid=19589720824&utm_adgroupid=157156376311&utm_device=c&utm_keyword=&utm_matchtype=&utm_network=g&utm_adposition=&utm_creative=724847714770&utm_targetid=dsa-2218886984100&utm_loc_interest_ms=&utm_loc_physical_ms=1012852&utm_content=&utm_campaign=230119_1-sea~dsa~tofu_2-b2c_3-row-p2_4-prc_5-na_6-na_7-le_8-pdsh-go_9-nb-e_10-na_11-na-dec24&gad_source=1&gclid=CjwKCAiAmfq6BhAsEiwAX1jsZ69QL_V8ylUcw45wOPHK6lvHiOGfTpvO1otlN3moiYRcA7Vma9XTBRoC7f4QAvD_BwE)

[2218886984100&utm_loc_interest_ms=&utm_loc_physical_ms=1012852&utm_content=&utm_campaign=230119_1-sea~dsa~tofu_2-b2c_3-row-p2_4-prc_5-na_6-na_7-le_8-pdsh-go_9-nb-e_10-na_11-na-dec24&gad_source=1&gclid=CjwKCAiAmfq6BhAsEiwAX1jsZ69QL_V8ylUcw45wOPHK6lvHiOGfTpvO1otlN3moiYRcA7Vma9XTBRoC7f4QAvD_BwE](https://www.datacamp.com/tutorial/pandas-tutorial-dataframe-python?utm_source=google&utm_medium=paid_search&utm_campaignid=19589720824&utm_adgroupid=157156376311&utm_device=c&utm_keyword=&utm_matchtype=&utm_network=g&utm_adposition=&utm_creative=724847714770&utm_targetid=dsa-2218886984100&utm_loc_interest_ms=&utm_loc_physical_ms=1012852&utm_content=&utm_campaign=230119_1-sea~dsa~tofu_2-b2c_3-row-p2_4-prc_5-na_6-na_7-le_8-pdsh-go_9-nb-e_10-na_11-na-dec24&gad_source=1&gclid=CjwKCAiAmfq6BhAsEiwAX1jsZ69QL_V8ylUcw45wOPHK6lvHiOGfTpvO1otlN3moiYRcA7Vma9XTBRoC7f4QAvD_BwE)

16. Dataset for Surprise Library [Электронный ресурс] – Режим доступа до ресурсу: <https://surprise.readthedocs.io/en/stable/dataset.html>

17. Using Common Table Expressions in SQL [Электронный ресурс] – Режим доступа до ресурсу: <https://www.atlassian.com/data/sql/using-common-table-expressions>

18. Normalization Techniques in Machine Learning [Электронный ресурс] – Режим доступа до ресурсу: <https://www.codecademy.com/article/normalization>
19. Normalization in Machine Learning [Электронный ресурс] – Режим доступа до ресурсу: <https://www.datacamp.com/tutorial/normalization-in-machine-learning>
20. Log vs Standard Normalization [Электронный ресурс] – Режим доступа до ресурсу: <https://medium.com/@sujathamudadla1213/when-we-have-to-do-log-normalization-when-we-have-to-do-standard-normalization-7cfe8f044801>
21. K-Nearest Neighbors for Recommendation Systems [Электронный ресурс] – Режим доступа до ресурсу: <https://www.analyticsvidhya.com/blog/2020/08/recommendation-system-k-nearest-neighbors/>
22. Recommender System with RBMS [Электронный ресурс] – Режим доступа до ресурсу: <https://www.kaggle.com/code/alirezanematolahy/recommender-system-with-rbms>
23. Understanding Restricted Boltzmann Machines [Электронный ресурс] – Режим доступа до ресурсу: <https://www.geeksforgeeks.org/restricted-boltzmann-machine/>
24. Building a Recommendation Engine with RBM Using TensorFlow [Электронный ресурс] – Режим доступа до ресурсу: <https://developer.ibm.com/tutorials/build-a-recommendation-engine-with-a-restricted-boltzmann-machine-using-tensorflow/>
25. KNN in Recommendation Systems [Электронный ресурс] – Режим доступа до ресурсу: <https://www.geeksforgeeks.org/recommender-systems-using-knn/>
26. Understanding KNN for Book Recommendation System [Электронный ресурс] – Режим доступа до ресурсу: <https://aman-makwana101932.medium.com/understanding-recommendation-system-and-knn-with-project-book-recommendation-system-c648e47ff4f6>
27. Neural Networks and Deep Learning [Электронный ресурс] – Режим доступа до ресурсу: <http://neuralnetworksanddeeplearning.com/>

28. Restricted Boltzmann Machines for Collaborative Filtering [Электронный ресурс] – Режим доступа до ресурсу: <https://arxiv.org/abs/1206.5538>
29. Introduction to RBMs [Электронный ресурс] – Режим доступа до ресурсу: <https://www.simplilearn.com/restricted-boltzmann-machines-rbms-article#:~:text=RBM%20is%20referred%20to%20as,technique%20known%20as%20contrastive%20divergence>
30. Understanding Stable Softmax [Электронный ресурс] – Режим доступа до ресурсу: <https://jaykmody.com/blog/stable-softmax/>
31. The Power of K-Nearest Neighbors (K-NN) Algorithm [Электронный ресурс] – Режим доступа до ресурсу: <https://www.linkedin.com/pulse/power-k-nearest-neighbors-k-nn-algorithm-highpeeks-ayush-thakur/>
32. Understanding Euclidean Distance [Электронный ресурс] – Режим доступа до ресурсу: <https://www.geeksforgeeks.org/euclidean-distance/>
33. Understanding Manhattan Distance [Электронный ресурс] – Режим доступа до ресурсу: <https://www.datacamp.com/tutorial/manhattan-distance>
34. Understanding Minkowski Distance [Электронный ресурс] – Режим доступа до ресурсу: <https://www.datacamp.com/tutorial/minkowski-distance>
35. Data Mining with K-Nearest Neighbors (K-NN) [Электронный ресурс] – Режим доступа до ресурсу: <https://www.linkedin.com/pulse/data-mining-classification-k-nearest-neighbors-k-nn-massimo-re-bt4cf/>
36. Root Mean Square Error (RMSE) [Электронный ресурс] – Режим доступа до ресурсу: <https://statisticsbyjim.com/regression/root-mean-square-error-rmse/>
37. Understanding Mean Absolute Error (MAE) in Regression [Электронный ресурс] – Режим доступа до ресурсу: <https://medium.com/@m.waqar.ahmed/understanding-mean-absolute-error-mae-in-regression-a-practical-guide-26e80ebb97df>
38. Evaluation Metrics for Recommender Systems [Электронный ресурс] – Режим доступа до ресурсу: <https://towardsdatascience.com/evaluation-metrics-for-recommender-systems-df56c6611093>

39. Evaluation of Recommender Systems [Электронный ресурс] – Режим доступа до ресурсу: https://link.springer.com/chapter/10.1007/978-1-4899-7637-6_26
40. Novelty in Recommender Systems [Электронный ресурс] – Режим доступа до ресурсу: <https://medium.com/@ayanglaishram/novelty-in-recommender-system-1a3da04e3b1f>
41. Jannach, D., P. Pu, F. Ricci, and M. Zanker. 2021. “ Recommender systems: Past, present, future.” AI Magazine 42: 3–6. <https://doi.org/10.1609/aaai.12012>
42. Konstan, J.A., & Riedl, J. (2012). Recommender systems: from algorithms to user experience. User Modeling and User-Adapted Interaction, 22, 101 - 123.
43. Nitin Mishra, Saumya Chaturvedi, Aanchal Vij, Sunita Tripathi. Research Problems in Recommender systems. Journal of Physics: Conference Series. 1717 (2021), 012002, IOP Publishing. doi:10.1088/1742-6596/1717/1/012002

ДОДАТКИ

```

# -*- coding: utf-8 -*-
from RecommenderMetrics import RecommenderMetrics
from EvaluationData import EvaluationData

class EvaluatedAlgorithm:

    def __init__(self, algorithm, name):
        self.algorithm = algorithm
        self.name = name

    def Evaluate(self, evaluationData, doTopN, n=10, verbose=True):
        metrics = {}

        if (verbose):
            print("Evaluating accuracy...")
            self.algorithm.fit(evaluationData.GetTrainSet())
            predictions = self.algorithm.test(evaluationData.GetTestSet())
            metrics["RMSE"] = RecommenderMetrics.RMSE(predictions)
            metrics["MAE"] = RecommenderMetrics.MAE(predictions)

        if (doTopN):

            if (verbose):
                print("Evaluating top-N with leave-one-out...")
                self.algorithm.fit(evaluationData.GetLOOCVTrainSet())
                leftOutPredictions = self.algorithm.test(evaluationData.GetLOOCVTest-
Set())

                allPredictions = self.algorithm.test(evaluationData.GetLOOCVAn-
tiTestSet())

                topNPPredicted = RecommenderMetrics.GetTopN(allPredictions, n)
                if (verbose):
                    print("Computing hit-rate and rank metrics...")
                    metrics["HR"] = RecommenderMetrics.HitRate(topNPPredicted, leftOutPre-
dictions)
                    metrics["cHR"] = RecommenderMetrics.CumulativeHitRate(topNPPredicted,
leftOutPredictions)

                    metrics["ARHR"] = RecommenderMetrics.AverageReciprocalHitRank(topNPre-
dicted, leftOutPredictions)

            if (verbose):
                print("Computing recommendations with full data set...")
                self.algorithm.fit(evaluationData.GetFullTrainSet())
                allPredictions = self.algorithm.test(evaluationData.GetFul-
lAntiTestSet())

```

```

        topNPPredicted = RecommenderMetrics.GetTopN(allPredictions, n)
        if (verbose):
            print("Analyzing coverage, diversity, and novelty...")

        metrics["Coverage"] = RecommenderMetrics.UserCoverage( topNPPredicted,
                                                                evalua-
tionData.GetFullTrainSet().n_users,
                                                                ratingThresh-
old=0)

        metrics["Diversity"] = RecommenderMetrics.Diversity(topNPPredicted,
evaluationData.GetSimilarities())

        metrics["Novelty"] = RecommenderMetrics.Novelty(topNPPredicted,
evaluationData.GetPopu-
larityRankings())

        if (verbose):
            print("Analysis complete.")

        return metrics

    def GetName(self):
        return self.name

    def GetAlgorithm(self):
        return self.algorithm

# -*- coding: utf-8 -*-
from surprise import AlgoBase
from surprise import PredictionImpossible
import numpy as np
import heapq
from ProductLens import ProductLens

class ContentKNNAlgorithm(AlgoBase):
    def __init__(self, k=20, sim_options={}):

        AlgoBase.__init__(self)
        self.k = k

    def fit(self, trainset):
        AlgoBase.fit(self, trainset)
        self.pl = ProductLens()
        print("Computing content-based similarity matrix...")
        self.similarities = np.zeros((self.trainset.n_items, self.train-
set.n_items))

```

```

    for thisRating in range(self.trainset.n_items):
        for otherRating in range(thisRating + 1, self.trainset.n_items):
            thisProductID = int(self.trainset.to_raw_iid(thisRating))
            otherProductID = int(self.trainset.to_raw_iid(otherRating))

            uniq_users_similarity = self.pl.compute_uniq_users_similarity(this-
ProductID, otherProductID)
            sold_similarity = self.pl.compute_sold_similarity(thisProductID,
otherProductID)
            user_clicks_similarity = self.pl.compute_user_clicks_similar-
ity(thisProductID, otherProductID)
            rating_similarity = self.pl.compute_rating_similarity(this-
ProductID, otherProductID)

            total_similarity = (uniq_users_similarity + sold_similarity +
user_clicks_similarity + rating_similarity) / 4.0

            self.similarities[thisRating, otherRating] = total_similarity
            self.similarities[otherRating, thisRating] = total_similarity

    print("...done.")
    return self

def estimate(self, u, i):
    if not (self.trainset.knows_user(u) and self.trainset.knows_item(i)):
        raise PredictionImpossible('User and/or item is unknown.')

    neighbors = []
    for rating in self.trainset.ur[u]:
        similarity_score = self.similarities[i, rating[0]]
        neighbors.append((similarity_score, rating[1]))

    k_neighbors = heapq.nlargest(self.k, neighbors, key=lambda t: t[0])

    simTotal = weightedSum = 0
    for (simScore, rating) in k_neighbors:
        if simScore > 0:
            simTotal += simScore
            weightedSum += simScore * rating

    if simTotal == 0:
        raise PredictionImpossible('No neighbors')

    predictedRating = weightedSum / simTotal

    return predictedRating

# -*- coding: utf-8 -*-
from surprise.model_selection import train_test_split
from surprise.model_selection import LeaveOneOut

```

```

from surprise import KNNBaseline

class EvaluationData:

    def __init__(self, data, popularityRankings):

        self.rankings = popularityRankings

        self.fullTrainSet = data.build_full_trainset()
        self.fullAntiTestSet = self.fullTrainSet.build_anti_testset()

        self.trainSet, self.testSet = train_test_split(data, test_size=.25, random_state=1)

        LOOCV = LeaveOneOut(n_splits=1, random_state=1)
        for train, test in LOOCV.split(data):
            self.LOOCVTrain = train
            self.LOOCVTest = test

        self.LOOCVAntiTestSet = self.LOOCVTrain.build_anti_testset()

        sim_options = {'name': 'cosine', 'user_based': False}
        self.simsAlgo = KNNBaseline(sim_options=sim_options)
        self.simsAlgo.fit(self.fullTrainSet)

    def GetFullTrainSet(self):
        return self.fullTrainSet

    def GetFullAntiTestSet(self):
        return self.fullAntiTestSet

    def GetAntiTestSetForUser(self, testSubject):
        trainset = self.fullTrainSet
        fill = trainset.global_mean
        anti_testset = []
        u = trainset.to_inner_uid(str(testSubject))
        user_items = set([j for (j, _) in trainset.ur[u]])
        anti_testset += [(trainset.to_raw_uid(u), trainset.to_raw_iid(i), fill) for
                        i in trainset.all_items() if
                        i not in user_items]

        return anti_testset

    def GetTrainSet(self):
        return self.trainSet

```



```

        print("{:<10} {:<10.4f} {:<10.4f} {:<10.4f} {:<10.4f} {:<10.4f}
{:<10.4f} {:<10.4f} {:<10.4f}".format(
            name, metrics["RMSE"], metrics["MAE"], metrics["HR"], met-
rics["CHR"], metrics["ARHR"],
                                metrics["Coverage"], metrics["Diversity"],
metrics["Novelty"]))
    else:
        print("{:<10} {:<10} {:<10}".format("Algorithm", "RMSE", "MAE"))
        for (name, metrics) in results.items():
            print("{:<10} {:<10.4f} {:<10.4f}".format(name, metrics["RMSE"],
metrics["MAE"]))

    print("\nLegend:\n")
    print("RMSE:      Root Mean Squared Error. Lower values mean better accu-
racy.")
    print("MAE:      Mean Absolute Error. Lower values mean better accuracy.")
    if (doTopN):
        print("HR:      Hit Rate; how often we are able to recommend a left-
out rating. Higher is better.")
        print("CHR:      Cumulative Hit Rate; hit rate, confined to ratings
above a certain threshold. Higher is better.")
        print("ARHR:      Average Reciprocal Hit Rank - Hit rate that takes the
ranking into account. Higher is better." )
        print("Coverage: Ratio of users for whom recommendations above a cer-
tain threshold exist. Higher is better.")
        print("Diversity: 1-S, where S is the average similarity score between
every possible pair of recommendations")
        print("          for a given user. Higher means more diverse.")
        print("Novelty:  Average popularity rank of recommended items. Higher
means more novel.")

    def SampleTopNRecs(self, pl, testSubject=10, k=10):

        for algo in self.algorithms:
            print("\nUsing recommender ", algo.GetName())

            print("\nBuilding recommendation model...")
            trainSet = self.dataset.GetFullTrainSet()
            algo.GetAlgorithm().fit(trainSet)

            print("Computing recommendations...")
            testSet = self.dataset.GetAntiTestSetForUser(testSubject)

            predictions = algo.GetAlgorithm().test(testSet)

            recommendations = []

            print ("\nWe recommend:")
            for userID, productID, actualRating, estimatedRating, _ in predictions:
                intProductID = int(productID)

```

```

        recommendations.append((intProductID, estimatedRating))

    recommendations.sort(key=lambda x: x[1], reverse=True)

    for ratings in recommendations[:10]:
        print(pl.getProductname(ratings[0]), ratings[1])
# -*- coding: utf-8 -*-
from surprise import AlgoBase

class HybridAlgorithm(AlgoBase):

    def __init__(self, algorithms, weights, sim_options={}):
        AlgoBase.__init__(self)
        self.algorithms = algorithms
        self.weights = weights

    def fit(self, trainset):
        AlgoBase.fit(self, trainset)

        for algorithm in self.algorithms:
            algorithm.fit(trainset)

        return self

    def estimate(self, u, i):

        sumScores = 0
        sumWeights = 0

        for idx in range(len(self.algorithms)):
            sumScores += self.algorithms[idx].estimate(u, i) * self.weights[idx]
            sumWeights += self.weights[idx]

        return sumScores / sumWeights

# -*- coding: utf-8 -*-
from ProductLens import ProductLens
from RBMAAlgorithm import RBMAAlgorithm
from ContentKNNAlgorithm import ContentKNNAlgorithm
from HybridAlgorithm import HybridAlgorithm
from Evaluator import Evaluator

import random
import numpy as np

def LoadProductData():
    product_lens = ProductLens()

    try:

```

```

        print("Loading product ratings...")
        data = product_lens.loadProductDataFromDB()
        print("Data loaded successfully!")

        print("\nComputing product popularity ranks so we can measure novelty
later...")
        rankings = product_lens.getPopularityRanks()
        print("Rankings computed successfully!")

    except Exception as e:
        print(f"An error occurred while loading product data: {e}")
        return None, None, None

    return (product_lens, data, rankings)

np.random.seed(0)
random.seed(0)

(product_lens, evaluationData, rankings) = LoadProductData()
print(product_lens)
print(evaluationData)
print(rankings)

evaluator = Evaluator(evaluationData, rankings)

SimpleRBM = RBMAlgorithm()

ContentKNN = ContentKNNAlgorithm()

Hybrid = HybridAlgorithm([SimpleRBM, ContentKNN], [0.3, 0.7])

evaluator.AddAlgorithm(SimpleRBM, "RBM")
evaluator.AddAlgorithm(ContentKNN, "ContentKNN")
evaluator.AddAlgorithm(Hybrid, "Hybrid")

evaluator.Evaluate(True)

evaluator.SampleTopNRecs(product_lens)

# -*- coding: utf-8 -*-
import psycopg2
import json
from surprise import Dataset, Reader
from collections import defaultdict
import pandas as pd
import traceback

```

```

import numpy as np

BASE_UNIQ_USERS = 587
BASE_SOLD = 298
BASE_USERS_CLICKS = 746
BASE_RATING = 5

class ProductLens:
    db_params = {
        "dbname": "",
        "user": "",
        "password": "",
        "host": "",
        "port": ""
    }
    productID_to_name = {}
    name_to_productID = {}
    raw_data = None
    ratingsPath = './ratings.csv'

    def connect_db(self):
        try:
            return psycopg2.connect(**self.db_params)
        except Exception as e:
            print(f"Database connection error: {e}")
            raise

    @staticmethod
    def min_max_normalize(value, min_value, max_value):
        return (value - min_value) / (max_value - min_value)

    def calculate_overall_rating(self, uniq_users, user_clicks, sold, rating, alpha=0.1):
        normalized_uniq_users = self.min_max_normalize(uniq_users, 0, BASE_UNIQ_USERS)
        normalized_user_clicks = self.min_max_normalize(user_clicks, 0, BASE_USERS_CLICKS)
        normalized_sold = self.min_max_normalize(sold, 0, BASE_SOLD)
        normalized_rating = self.min_max_normalize(rating, 0, BASE_RATING)

        overall_score = (
            np.log1p(normalized_uniq_users) +
            np.log1p(normalized_user_clicks) +
            np.log1p(normalized_sold) +
            np.log1p(normalized_rating) +
            alpha * (np.log1p(normalized_uniq_users) * np.log1p(normalized_user_clicks))
        )
        return overall_score

```

```

def loadProductDataFromDB(self):
    if self.raw_data:
        data = []
        for row in self.raw_data:
            product_id, product_name, uniq_users, sold, user_clicks, product_created_at, feedback_id, rating = row

            base_rating = self.calculate_overall_rating(uniq_users, sold, user_clicks, rating)

            data.append((feedback_id, product_id, base_rating, product_created_at))

            self.productID_to_name[product_id] = product_name
            self.name_to_productID[product_name] = product_id

        df = pd.DataFrame(data, columns=['user', 'item', 'rating', 'timestamp'])
        df.to_csv(self.ratingsPath, index=False)

        reader = Reader(line_format='user item rating timestamp', sep=',', skip_lines=1)

        ratingsDataset = Dataset.load_from_file(self.ratingsPath, reader=reader)
        return ratingsDataset

    else:
        try:
            connection = self.connect_db()
            cursor = connection.cursor()

            query = """
                WITH products AS (
                    SELECT
                        p.id AS product_id,
                        p.name AS product_name,
                        jsonb_array_length(p."uniqUsers") AS uniqUsers, -- Get
the length of uniqUsers array
                        p.sold,
                        p."userClicks",
                        COALESCE(EXTRACT(EPOCH FROM p."productCreatedAt")::big-
int, NULL) AS productCreatedAt -- Convert to Unix timestamp
                    FROM
                        product p
            """

```

```

WHERE
    jsonb_array_length(p."uniqUsers") > 0 -- Ensure there
are unique users
),
existing_feedback AS (
    SELECT
        f."productId" AS product_id, -- Make sure to use the
correct case
        f.id AS feedback_id
    FROM
        feedback f
),
feedback AS (
    SELECT
        p.product_id,
        p.product_name,
        p.uniqUsers,
        p.sold,
        p."userClicks",
        p.productCreatedAt,
        (g.id + RANDOM() * 2)::int AS generated_feedback_id, --
Generated feedback ID
        -- Random rating between 1 and 5
        (1 + RANDOM() * 4)::int AS rating -- Random rating be-
tween 1 and 5
    FROM
        products p
    CROSS JOIN
        generate_series(1, 15) AS g(id) -- Generate 10 feedback
entries for each product
)
SELECT
    f.product_id,
    f.product_name,
    CASE
        WHEN MIN(f.uniqUsers) < 10 THEN (25 + RANDOM() * (100 -
25))::int -- Random uniqUsers value between 25 and 100
        ELSE MIN(f.uniqUsers) -- Use original uniqUsers value if
>= 10
    END AS uniqUsers,
    CASE
        WHEN MIN(f.sold) < 10 THEN (10 + RANDOM() * (25 -
10))::int -- Random sold value between 10 and 25
        ELSE MIN(f.sold) -- Use original sold value if >= 10
    END AS sold,
    CASE
        WHEN MIN(f."userClicks") < 10 THEN (50 + RANDOM() * (150
- 50))::int -- Random userClicks value between 50 and 150
        ELSE MIN(f."userClicks") -- Use original userClicks
value if >= 10
    END AS "userClicks"

```

```

        END AS userClicks,
        f.productCreatedAt,
        f.generated_feedback_id AS feedback_id, -- Use generated
feedback ID
        f.rating
    FROM
        feedback f
    GROUP BY
        f.product_id,
        f.product_name,
        f.productCreatedAt,
        f.generated_feedback_id,
        f.rating -- Ensure grouping to keep track of feedback ID
    ORDER BY
        f.product_id -- Final ordering by product_id and feedback_id
    """"

    cursor.execute(query)
    rows = cursor.fetchall()
    self.raw_data = rows

    data = []
    for row in rows:
        product_id, product_name, uniq_users, sold, user_clicks, prod-
uct_created_at, feedback_id, rating = row

        base_rating = self.calculate_overall_rating(uniq_users, sold,
user_clicks, rating)

        data.append((feedback_id, product_id, base_rating, product_cre-
ated_at))

    self.productID_to_name[product_id] = product_name
    self.name_to_productID[product_name] = product_id

    df = pd.DataFrame(data, columns=['user', 'item', 'rating',
'timestamp'])
    df.to_csv(self.ratingsPath, index=False)

    reader = Reader(line_format='user item rating timestamp', sep=',',
skip_lines=1)

    ratingsDataset = Dataset.load_from_file(self.ratingsPath,
reader=reader)
    return ratingsDataset
except Exception as e:

```

```

        print(f"Error executing query: {e}")
        traceback.print_exc()
        raise
    finally:
        cursor.close()
        connection.close()

def getUserRatings(self, user):
    userRatings = []
    if self.raw_data is None:
        self.loadProductDataFromDB()

    for row in self.raw_data:

        product_id, product_name, uniq_users_count, sold, user_clicks, product_created_at, feedback_id, rating = row

        if feedback_id and feedback_id == user:

            userRatings.append((product_id, rating))
        else:

            base_rating = uniq_users_count + sold + user_clicks
            userRatings.append((product_id, base_rating))

    return userRatings

def getPopularityRanks(self):
    ratings = defaultdict(int)
    rankings = defaultdict(int)

    for row in self.raw_data:
        product_id, product_name, uniq_users, sold, user_clicks, product_created_at, feedback_id, rating = row
        ratings[product_id] += self.calculate_overall_rating(uniq_users, sold, user_clicks, rating)

    rank = 1
    for product_id, rating_count in sorted(ratings.items(), key=lambda x: x[1], reverse=True):
        rankings[product_id] = rank
        rank += 1

    return rankings

def getProductName(self, productID):
    return self.productID_to_name.get(productID, "")

def getProductID(self, productName):

```

```

return self.name_to_productID.get(productName, 0)

def compute_uniq_users_similarity(self, product1_id, product2_id):
    users1 = self.get_uniq_users(product1_id)
    users2 = self.get_uniq_users(product2_id)

    return (users1 + users2) / 2

def compute_sold_similarity(self, product1_id, product2_id):
    sold1 = self.get_sold_count(product1_id)
    sold2 = self.get_sold_count(product2_id)

    return (sold1 + sold2) / 2

def compute_user_clicks_similarity(self, product1_id, product2_id):
    clicks1 = self.get_user_clicks_count(product1_id)
    clicks2 = self.get_user_clicks_count(product2_id)

    return (clicks1 + clicks2) / 2

def compute_rating_similarity(self, product1_id, product2_id):
    rating1 = self.get_rating(product1_id)
    rating2 = self.get_rating(product2_id)

    return (rating1 + rating2) / 2

def get_uniq_users(self, product_id):
    if self.raw_data is None:
        self.loadProductDataFromDB()

    total_uniq_users = 0
    count = 0
    for row in self.raw_data:
        product_id_row, product_name, uniq_users, sold, user_clicks, product_created_at, feedback_id, rating = row
        if product_id_row == product_id:
            total_uniq_users += uniq_users
            count += 1

    return total_uniq_users / count if count > 0 else 0

def get_sold_count(self, product_id):
    if self.raw_data is None:
        self.loadProductDataFromDB()

    total_sold = 0
    count = 0
    for row in self.raw_data:

```

```

        product_id_row, product_name, uniq_users, sold, user_clicks, prod-
uct_created_at, feedback_id, rating = row
        if product_id_row == product_id:
            total_sold += sold
            count += 1

```

```

    return total_sold / count if count > 0 else 0

```

```

def get_user_clicks_count(self, product_id):
    if self.raw_data is None:
        self.loadProductDataFromDB()

    total_user_clicks = 0
    count = 0
    for row in self.raw_data:
        product_id_row, product_name, uniq_users, sold, user_clicks, prod-
uct_created_at, feedback_id, rating = row
        if product_id_row == product_id:
            total_user_clicks += user_clicks
            count += 1

```

```

    return total_user_clicks / count if count > 0 else 0

```

```

def get_rating(self, product_id):
    if self.raw_data is None:
        self.loadProductDataFromDB()

    total_rating = 0
    count = 0
    for row in self.raw_data:
        product_id_row, product_name, uniq_users, sold, user_clicks, prod-
uct_created_at, feedback_id, rating = row
        if product_id_row == product_id:
            total_rating += rating
            count += 1

```

```

    return total_rating / count if count > 0 else 0

```

```

import numpy as np
import tensorflow as tf

```

```

class RBM(object):

```

```

    def __init__(self, visibleDimensions, epochs=500, hiddenDimensions=100, rat-
ingValues=1, LearningRate=0.05, batchSize=128):
        self.visibleDimensions = visibleDimensions
        self.epochs = epochs
        self.hiddenDimensions = hiddenDimensions
        self.ratingValues = ratingValues
        self.learningRate = LearningRate

```

```

self.batchSize = batchSize

self.weights = tf.Variable(tf.random.uniform([self.visibleDimensions,
self.hiddenDimensions],
minval=-0.1, maxval=0.1),
name="weights")
self.hiddenBias = tf.Variable(tf.zeros([self.hiddenDimensions]), name="hiddenBias")
self.visibleBias = tf.Variable(tf.zeros([self.visibleDimensions]),
name="visibleBias")

def train(self, X):
    for epoch in range(self.epochs):
        np.random.shuffle(X)
        trX = np.array(X)

        for i in range(0, trX.shape[0], self.batchSize):
            batch = trX[i:i + self.batchSize]
            self._train_step(batch)

def _train_step(self, batch):
    with tf.GradientTape() as tape:
        h_prob = self._sample_hidden(batch)
        v_prob = self._reconstruct_visible(h_prob)

        v_mask = tf.sign(batch)
        v_mask_3d = tf.reshape(v_mask, [tf.shape(v_prob)[0], -1, self.ratingValues])
        v_mask_3d = tf.reduce_max(v_mask_3d, axis=2, keepdims=True)

        v_prob = tf.reshape(v_prob, [tf.shape(v_prob)[0], -1, self.ratingValues])
        v_prob = tf.nn.softmax(v_prob * v_mask_3d)
        v_prob = tf.reshape(v_prob, [tf.shape(v_prob)[0], -1])

        h_prob1 = self._sample_hidden(v_prob)

        weight_update = tf.matmul(tf.transpose(batch), h_prob) - tf.matmul(tf.transpose(v_prob), h_prob1)
        hidden_bias_update = tf.reduce_mean(h_prob - h_prob1, axis=0)
        visible_bias_update = tf.reduce_mean(batch - v_prob, axis=0)

        gradients = [weight_update, hidden_bias_update, visible_bias_update]

self.weights.assign_add(self.learningRate * gradients[0])

```

```

        self.hiddenBias.assign_add(self.learningRate * gradients[1])
        self.visibleBias.assign_add(self.learningRate * gradients[2])

    def _sample_hidden(self, visible):
        h_prob = tf.nn.sigmoid(tf.matmul(visible, self.weights) + self.hiddenBias)
        return tf.cast(h_prob > tf.random.uniform(tf.shape(h_prob)), tf.float32)

    def _reconstruct_visible(self, hidden):
        return tf.nn.sigmoid(tf.matmul(hidden, tf.transpose(self.weights)) +
self.visibleBias)

    def get_recommendations(self, inputUser):
        hidden = self._sample_hidden(inputUser)
        visible = self._reconstruct_visible(hidden)
        return visible.numpy()
# -*- coding: utf-8 -*-
from surprise import AlgoBase
from surprise import PredictionImpossible
import numpy as np
from RBM import RBM
from ProductLens import ProductLens

class RBMAlgorithm(AlgoBase):

    def __init__(self, epochs=200, hiddenDim=200 , learningRate=0.001, batch-
Size=64, sim_options={}):
        AlgoBase.__init__(self)
        self.epochs = epochs
        self.hiddenDim = hiddenDim
        self.learningRate = learningRate
        self.batchSize = batchSize
        self.pl = ProductLens()
        self.pl.loadProductDataFromDB()

    def softmax(self, x):
        e_x = np.exp(x - np.max(x))
        return e_x / np.sum(e_x, axis=0)

    def fit(self, trainset):
        AlgoBase.fit(self, trainset)

        numUsers = trainset.n_users
        numItems = trainset.n_items

        trainingMatrix = np.zeros([numUsers, numItems, 10], dtype=np.float32)

        for (uid, iid, rating) in trainset.all_ratings():
            adjustedRating = min(max(int(float(rating)), 0), 9)
            trainingMatrix[int(uid), int(iid), adjustedRating] = 1

```

```

trainingMatrix = np.reshape(trainingMatrix, [trainingMatrix.shape[0], -1])

rbm = RBM(trainingMatrix.shape[1], hiddenDimensions=self.hiddenDim, LearningRate=self.learningRate, batchSize=self.batchSize, epochs=self.epochs)
rbm.train(trainingMatrix)

self.predictedRatings = np.zeros([numUsers, numItems], dtype=np.float32)
for uuid in range(numUsers):
    if uuid % 50 == 0:
        print("Processing user ", uuid)
    recs = rbm.get_recommendations([trainingMatrix[uuid]])
    recs = np.reshape(recs, [numItems, 10])

    for itemID, rec in enumerate(recs):
        normalized = self.softmax(rec)
        rating = np.average(np.arange(10), weights=normalized)
        self.predictedRatings[uuid, itemID] = (rating + 1) * 0.5

return self

def estimate(self, u, i):
    if not (self.trainset.knows_user(u) and self.trainset.knows_item(i)):
        raise PredictionImpossible('User and/or item is unknown.')

    rating = self.predictedRatings[u, i]

    if rating < 0.001:
        raise PredictionImpossible('No valid prediction exists.')

    return rating

import itertools

from surprise import accuracy
from collections import defaultdict

class RecommenderMetrics:

    def MAE(predictions):
        return accuracy.mae(predictions, verbose=False)

    def RMSE(predictions):
        return accuracy.rmse(predictions, verbose=False)

    def GetTopN(predictions, n=10, minimumRating=1.0):
        topN = defaultdict(list)

```

```

for userID, productID, actualRating, estimatedRating, _ in predictions:
    if (estimatedRating >= minimumRating):
        topN[int(userID)].append((int(productID), estimatedRating))

for userID, ratings in topN.items():
    ratings.sort(key=lambda x: x[1], reverse=True)
    topN[int(userID)] = ratings[:n]

return topN

def HitRate(topNPredicted, LeftOutPredictions):
    hits = 0
    total = 0

    for leftOut in LeftOutPredictions:
        userID = leftOut[0]
        leftOutproductID = leftOut[1]

        hit = False
        for productID, predictedRating in topNPredicted[int(userID)]:
            if (int(leftOutproductID) == int(productID)):
                hit = True
                break
        if (hit) :
            hits += 1

        total += 1

    return hits/total

def CumulativeHitRate(topNPredicted, LeftOutPredictions, ratingCutoff=0):
    hits = 0
    total = 0

    for userID, leftOutproductID, actualRating, estimatedRating, _ in LeftOut-
Predictions:
        if (actualRating >= ratingCutoff):
            hit = False
            for productID, predictedRating in topNPredicted[int(userID)]:
                if (int(leftOutproductID) == productID):
                    hit = True
                    break
            if (hit) :
                hits += 1

```

```

        total += 1

    return hits/total

def RatingHitRate(topNPredicted, LeftOutPredictions):
    hits = defaultdict(float)
    total = defaultdict(float)

    for userID, leftOutproductID, actualRating, estimatedRating, _ in LeftOut-
Predictions:

        hit = False
        for productID, predictedRating in topNPredicted[int(userID)]:
            if (int(leftOutproductID) == productID):
                hit = True
                break
        if (hit) :
            hits[actualRating] += 1

        total[actualRating] += 1

    for rating in sorted(hits.keys()):
        print (rating, hits[rating] / total[rating])

def AverageReciprocalHitRank(topNPredicted, LeftOutPredictions):
    summation = 0
    total = 0

    for userID, leftOutproductID, actualRating, estimatedRating, _ in LeftOut-
Predictions:

        hitRank = 0
        rank = 0
        for productID, predictedRating in topNPredicted[int(userID)]:
            rank = rank + 1
            if (int(leftOutproductID) == productID):
                hitRank = rank
                break
        if (hitRank > 0) :
            summation += 1.0 / hitRank

        total += 1

    return summation / total

def UserCoverage(topNPrdcted, numUsers, ratingThreshold=0):

```

```

hits = 0
for userID in topNPredicted.keys():
    hit = False
    for productID, predictedRating in topNPredicted[userID]:
        if (predictedRating >= ratingThreshold):
            hit = True
            break
    if (hit):
        hits += 1

return hits / numUsers

def Diversity(topNPredicted, simsAlgo):
    n = 0
    total = 0
    simsMatrix = simsAlgo.compute_similarities()

    for userID in topNPredicted.keys():

        pairs = itertools.combinations(topNPredicted[userID], 2)
        for pair in pairs:
            product1 = pair[0][0]
            product2 = pair[1][0]
            innerID1 = simsAlgo.trainset.to_inner_iid(str(product1))
            innerID2 = simsAlgo.trainset.to_inner_iid(str(product2))
            similarity = simsMatrix[innerID1][innerID2]
            total += similarity
            n += 1

    if n != 0:
        S = total / n
        return (1 - S)
    else:
        return 0

def Novelty(topNPredicted, rankings):
    n = 0
    total = 0
    for userID in topNPredicted.keys():
        for rating in topNPredicted[userID]:
            productID = rating[0]
            rank = rankings[productID]
            total += rank
            n += 1
    if n != 0:
        return total / n
    else:

```

```
return 0
```