Київський національний університет

імені Тараса Шевченка

Модульна контрольна робота з курсу

«Основи Data Mining»

**ВАРІАНТ 5.**

Роботу виконав:

студент 4 курсу

факультету ФКНК

групи ТТП-42

Чебан Боглан Володимирович

**Київ 2025**

1.Практичне завдання: K-Means (60 балів)  
  
Є набір:  
synthetic\_customers.csv  
(age, income, loyalty\_score)  
  
23,20000,70  
45,50000,30  
51,60000,20  
28,25000,80  
34,35000,65  
58,75000,15  
41,65000,25  
30,40000,85  
24,21000,78  
53,62000,22

Реалізуйте власний K-Means з ініціалізацією центроїдів як k-means++ (описати логіку алгоритму).

Побудуйте Elbow графу і визначте оптимальне число кластерів.

* Ось виконання цього завдання з описом коду:

<https://github.com/xaxinotf/Test_DM_var-5_Cheban_Bogdan/tree/master>

2. Теоретичне завдання: SVD (20 балів)

В чому ризик використання SVD для динамічних рекомендаційних систем (коли користувачі та товари постійно змінюються)?

Чи можна оновити SVD без перерахунку всієї матриці?

У нас фор екзапл є матриця оцінок R розміром m×n (m – кількість користувачів, n – кількість товарів), і ми хочемо її апроксимувати розкладом SVD:

R ≈ U·Σ·Vᵀ , де

U – матриця m×k з «факторами» користувачів,

Σ – діагональна матриця k×k із сингулярними числами σ₁ ≥ σ₂ ≥ … ≥ σₖ,

Vᵀ – матриця k×n з «факторами» товарів,  
причому вибирають k значно меншим за min(m,n), щоб зберегти тільки головні компоненти.

Розклад SVD дуже корисний у рекомендаційних системах: він виокремлює латентні фактори, зводить розмірність і часто покращує якість рекомендацій. Але коли дані змінюються постійно — нові користувачі реєструються, товари додаються, оцінки змінюються — класичний підхід із повним перерахунком SVD щоразу зазнає кількох проблем.

* ризик “застаріти” == після того як нові оцінки додаються в R, старі матриці U, Σ і V вже не відображають поточні вподобання. Рекомендації на їх основі стають менш точними, бо структура латентних факторів зміщена.
* висока вартість перерахунку == повний SVD алгоритмічно коштує приблизно O(min(m·n², m²·n)). На матрицях розміром сотні тисяч на сотні тисяч це може займати години чи дні — занадто довго для систем, де потрібно швидко реагувати на зміни.
* нестабільність факторів == навіть невелика зміна у рейтингах може спричинити суттєве «переозначення» сингулярних векторів. Через це рекомендації можуть різко змінюватися, що б’є по довірі користувача до системи.
* проблема холодного старту == новий користувач чи товар приходить без історії. Якщо просто підставити їх у старі U і V, ми не отримаємо адекватних факторів і, як наслідок, погано зможемо зорієнтуватися, що порекомендувати.

Щоб не перераховувати всю матрицю щоразу, застосовують два основних підходи:

* “вписування” (fold‑in)
  + для нового користувача з вектором рейтингів r\_new розміром 1×n:

u\_new = r\_new · V · Σ⁻¹  
отримуємо його k‑вимірний вектор факторів у просторі користувачів.

* + для нового товару з вектором r\_item розміром m×1:

v\_new = Σ⁻¹ · Uᵀ · r\_item  
– це дозволяє швидко «вписати» нові елементи в існуючий розклад.  
Але при цьому сам U, Σ, V не оновлюються, тож із часом апроксимація “розбіжиться” зі справжніми даними.

* інкрементальні оновлення  
  Існують алгоритми, які дозволяють частково коригувати SVD при додаванні рядка чи стовпця:
  + **Brand’s incremental SVD** — робить rank‑1 оновлення, додаючи нові дані й перераховуючи тільки k×k підматриці;
  + **Rank‑one update** — якщо зміни ΔR мають низький ранг, можна оновити U, Σ, V за формулами з власних значень невеликої допоміжної матриці.  
    Такі методи працюють приблизно за O(k²)–O(k³) замість O(min(m·n², m²·n)), що дозволяє підтримувати розклад «свіжим» у онлайн‑режимі.

**Підсумок**

Повний SVD дає якісні латентні фактори, але занадто повільний і нетривкий для динамічних систем.Fold‑in підходить для швидкого «вписування» нових користувачів чи товарів, але не оновлює самі основні матриці.Інкрементальні алгоритми (Brand, rank‑1 updates) дозволяють коригувати розклад локально з відчутно меншою обчислювальною вартістю.Удало поєднуючи ці підходи та зрідка роблячи повний SVD, можна отримати баланс між актуальністю рекомендацій і швидкістю обробки.

3. Аналітичне завдання: Apriori (20 балів)

Вам дали 50000 транзакцій.

Опишіть оптимальну стратегію, щоб знайти правила з мінімальною підтримкою 1% і довірою 80%, якщо обмеження на пам’ять дуже жорстке.

Є файл і там крч із 50 000 транзакцій – кожна транзакція містить перелік куплених товарів. Потрібно знайти всі асоціативні правила з мінімальною підтримкою 1 % (тобто товарна комбінація має зустрічатися принаймні в 500 транзакціях) та довірою не менше за 80 %. Однак оперативна пам’ять дуже обмежена: не вийде завантажити всі транзакції й усі кандидат-списки одночасно. Ось як я би підійшов, степ бай степ:

– **1. Перший прохід – лічильник одиниць.**  
• Зчитуємо транзакції блоками (наприклад, по 5000 рядків), не зберігаючи їх повністю в пам’яті.  
• Для кожного блоку рахуємо, скільки разів зустрічається кожен окремий товар (1‑айтемсет).  
• Після обробки всіх блоків маємо загальні лічильники count(A) для кожного товару A.  
• Відкидаємо всі товари з count(A) < 0.01·50 000 = 500, — це наші часті одиничні множини L₁.

– **2. Другий прохід – генерація та відсів кандидатів.**  
• Знову читаємо дані блоками.  
• Генеруємо кандидати 2‑айтемсетів C₂ з усіх пар товарів із L₁ (розмір L₁ зазвичай невеликий, тож C₂ поміщається в пам’ять).  
• Для кожної транзакції в блоці проходимо по всіх парах АВ, що входять і в транзакцію, і в C₂, і збільшуємо їхній лічильник.  
• Після проходу по всіх блоках формуємо часті пари L₂ = {AB ∈ C₂ | count(AB) ≥ 500}.  
• Далі аналогічно для k ≥ 3:  
– Cₖ = усі об’єднання двох елементів із Lₖ₋₁, які мають k−1 спільних елемент;  
– Під час сканування блоку оновлюємо лічильники тільки для тих кандидатів, що входять у поточну транзакцію.  
– Після обробки блоку звільняємо пам’ять від транзакції.

– **3. Зберігання та компресія кандидатів.**  
• Щоб не тримати в пам’яті весь список Cₖ, використовуйте хеш-таблицю на диску або сплітку candidate‑файлів, завантажуючи в пам’ять тільки по частині.  
• Можна скористатися bit‑vector-репрезентацією для кожного кандидата (одна біт‑послідовність на блок транзакцій), що значно зменшить обсяг оперативної пам’яті під кожен підрахунок.

– **4. Правила й довіра.**  
• Як тільки у вас є часті множини Lₖ для всіх k, переходьте до генерації правил.  
• Для кожної частої множини X і кожного непустого підмножини A ⊂ X обчислюємо  
– support(X) = count(X) / 50 000  
– support(A) = count(A) / 50 000  
– confidence(A → X \ A) = support(X) / support(A)  
• Відбираємо тільки ті правила, де confidence ≥ 0.8. Всі необхідні лічильники (count(X) і count(A)) вже зібрані під час Apriori.

**Чому це працює оптим. за жорсткого обмеження пам’яті?**

Ми ніколи не тримаємо в пам’яті більше одного блоку транзакцій.Кандидати зберігаємо окремо в компактній хеш‑структурі або на диску, підвантажуючи партіями.На кожному рівні k генеруємо та відфільтровуємо кандидат‑множини *до* підрахунку, тому їх кількість залишається мінімальною.Використовуємо bit‑vectors або компресовані структури, щоб зменшити розмір лічильників.

Таким чином, за два-три проходи по даних і мінімум об’єм пам’яті під транзакції й часткові кандидати, ми знаходимо всі правила з підтримкою ≥1 % і довірою ≥80 %, не перевантажуючи оперативну пам’ять.