САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Искусственный интеллект и наука о данных

Сергиенко Андрей

Применение алгоритмов K-ближайших соседей в коллаборативных рекомендательных системах

Отчёт о прохождении
Учебной (ознакомительной) практики

Научный руководитель:
Старший преподаватель кафедры системного программирования
Юрий Александрович Андреев

Санкт-Петербург
2025

Оглавление

Введение

Современные информационные системы ежедневно обрабатывают огромные объёмы данных, среди которых особое место занимают персонализированные рекомендации — фильмы, музыка, товары, новости и многое другое. Рекомендательные системы позволяют пользователям находить интересный контент, а компаниям — повышать вовлечённость и продажи.

Одним из наиболее популярных подходов к построению рекомендательных систем является коллаборативная фильтрация (Collaborative Filtering, CF), основанная на идее, что пользователи с похожими интересами будут оценивать похожие объекты схожим образом. Ключевой задачей коллаборативной фильтрации является поиск похожих пользователей или объектов, для чего широко применяется алгоритм K-ближайших соседей (K-Nearest Neighbors, KNN).

Однако точный поиск ближайших соседей имеют высокую вычислительную сложность и становится неэффективным при работе с большими наборами данных. Поэтому активно развиваются приближённые методы поиска ближайших соседей (Approximate Nearest Neighbors, ANN), такие как Annoy, FAISS и HNSW, обеспечивающие компромисс между скоростью и точностью.

Настоящая работа посвящена исследованию и сравнению эффективности различных реализаций алгоритма K-ближайших соседей в контексте коллаборативных рекомендательных систем.

1 Постановка задачи

Цель работы — исследовать эффективность различных алгоритмов K-ближайших соседей для реализации коллаборативной рекомендательной системы, а также провести сравнительный анализ их производительности, точности и использования памяти.

Для достижения цели необходимо решить следующие задачи:

1. Изучить основные принципы работы коллаборативной фильтрации и алгоритма K-ближайших соседей.
2. Реализовать базовую рекомендательную систему на основе пользовательских и предметных матриц.
3. Реализовать и сравнить четыре метода поиска ближайших соседей:
   - Точный KNN;
   - Annoy;
   - FAISS;
   - HNSW (Hierarchical Navigable Small World).
4. Провести измерения времени построения индекса, скорости запросов, точности (Recall@20, Precision@20) и использования памяти.
5. Визуализировать результаты и сформулировать выводы о применимости каждого метода.

1.1 Используемые библиотеки поиска ближайших соседей

Среди наиболее популярных библиотек для реализации ANN можно выделить:

- **Annoy** — метод случайных проекций и построения деревьев, эффективен для больших векторов признаков.
- **FAISS** — оптимизирована под большие данные и вычисления на GPU, активно используется в индустриальных решениях.
- **HNSW (Hierarchical Navigable Small World)** — графовая структура данных, обеспечивающая высокую точность и малое время отклика даже при миллионах элементов.

1.2 Ожидаемые результаты

В ходе работы будет реализован программный комплекс для сравнения точного и приближённых алгоритмов поиска ближайших соседей в коллаборативных системах рекомендаций.

Ожидается, что:

- Точный KNN обеспечит наилучшую точность (Recall@20, Precision@20), но низкую производительность на больших данных;
- Annoy покажет самое быстрое время запроса;
- FAISS продемонстрирует высокую скорость построения при больших объёмах данных.

- HNSW обеспечит оптимальный баланс между скоростью, точностью и памятью, являясь лучшим выбором для production-систем.

2 Обзор

2.1 Обзор рекомендательных систем и их классификация

Рекомендательные системы предсказывают интерес пользователя к объекту на основе исторических данных. В данной работе используется коллаборативная фильтрация (CF), которая делится на user-based CF (поиск похожих пользователей) и item-based CF (поиск похожих объектов).

2.2 Алгоритм K-ближайших соседей и его применение

Алгоритм K-ближайших соседей (KNN) является одним из базовых методов в коллаборативной фильтрации. Он позволяет находить элементы, наиболее похожие на заданный, по метрике сходства (например, косинусное сходство или корреляция Пирсона).В user-based CF ищутся пользователи со схожим поведением, в item-based CF — схожие объекты. Рекомендации формируются как взвешенное усреднение оценок ближайших соседей.

2.3 Проблемы масштабируемости и приближённые методы

Основным недостатком точного KNN является высокая вычислительная сложность.

Пусть у нас:

- N — количество объектов в базе
- Q — количество запросов
- d — размерность поискового пространства
- K — количество искомых соседей

Вычисление расстояния между двумя векторами требует $O(d)$.
Так как нужно сравнить со всеми N точками, то $O(N \cdot d)$.
Для всех Q запросов — $O(Q \cdot N \cdot d)$.
Используем алгоритм Quickselect — $O(N + K \log K)$.
После вычисления всех N расстояний нужно выбрать K наименьших — $O(Q \cdot (N \cdot d + K \log K))$.

При росте числа пользователей, запросов и объектов (например, в системах с миллионами записей) такой подход становится непрактичным.

Для решения этой проблемы используются Approximate Nearest Neighbors (ANN) — приближённые методы поиска, которые жертвуют небольшой долей точности ради значительного выигрыша в скорости.

ANN-методы строят специальные структуры данных (деревья, графы, хеши), что позволяет находить близкие элементы за логарифмическое или даже константное время.

3 Проектирование

3.1 Архитектура системы

Архитектура разработанной системы для генерации эмбеддингов рекомендательной системы построена по модульному принципу и состоит из следующих основных компонентов:

1. Модуль обработки данных

- Загрузка файлов `movies.csv` и `ratings.csv` из `data/raw`, удаление неиспользуемых признаков и базовая проверка данных.
- Фильтрация активных пользователей (> 50 оценок) и популярных фильмов (> 10 оценок), формирование непрерывных маппингов ID.
- Построение разреженной матрицы «пользователь–фильм» (формат CSR).
- Матричная факторизация методом TruncatedSVD (128 компонент) и генерация эмбеддингов пользователей и фильмов.
- Сохранение эмбеддингов, маппингов и модели в `data/processed`.

2. Модуль реализации алгоритмов поиска

- Единый интерфейс для всех алгоритмов с методами build(), query_user(), query_item()
- Четыре алгоритма поиска ExactKNN, AnnoyKNN, FAISSKNN, HNSWKNN
- Поддержка как user-based, так и item-based подходов

3. Модуль мониторинга производительности

- Оркестратор запускает каждый алгоритм в отдельном воркер-процессе, что обеспечивает изолированные измерения времени и памяти.
- Воркеры строят индекс, выполняют тестовые запросы и передают результаты обратно оркестратору.
- Потребление оперативной памяти фиксируется с помощью `psutil`, замеры времени включают построение индекса и выполнение запросов.

4. Модуль оценки точности и визуализации

- Генерация тестовой выборки из 100 случайных запросов
- Сравнение результатов приближённых методов с эталонным Exact KNN
- Вычисление метрик Recall@20 и Precision@20
- Построение шести сравнительных графиков
- Сохранение результатов в CSV-формате

Взаимодействие между модулями организовано последовательно: данные проходят через этапы загрузки, обработки, построения индексов, тестирования и визуализации результатов.

3.2 Выбор и подготовка датасета

В качестве экспериментального датасета был выбран MovieLens(32m) — один из наиболее распространённых наборов данных для исследований в области рекомендательных систем. Датасет содержит:

- Файл movies.csv: идентификаторы фильмов, названия и жанры.
- Файл ratings.csv: оценки пользователей (userId, movieId, rating, timestamp).

На этапе предварительной обработки были выполнены следующие операции:

Удаление избыточных признаков: столбцы ‘genres‘ и ‘timestamp‘ были исключены, так как в данной работе исследуется исключительно коллаборативная фильтрация без использования контентных признаков.

Фильтрация пользователей и объектов: для повышения качества рекомендаций и уменьшения вычислительной сложности были отобраны только активные пользователи (с числом оценок > 50) и популярные фильмы (с числом оценок > 10). Это позволило снизить влияние шумовых взаимодействий и сосредоточиться на объектах с достаточным количеством информации для построения надёжных моделей. В результате было выделено 126 588 пользователей и 30 521 фильм.

3.3 Формирование пользовательско-предметной матрицы

Была сформирована матрица взаимодействий «пользователь–объект» размерностью 126 588 $\times$ 30 521 с разреженностью 99.24% (плотность 0.76%). Использование разреженного формата. Плотная матрица такого размера потребовала бы 15.5 ГБ памяти, что непрактично. Матрица была конвертирована в формат CSR (Compressed Sparse Row), который хранит только ненулевые элементы, обеспечивая существенную экономию памяти и быструю индексацию строк. Необходимость снижения размерности. Несмотря на сжатие, работа с десятками тысяч признаков приводит к проблеме «проклятия размерности»: вычисление расстояний становится неинформативным, так как большинство пар пользователей не имеют пересечений по оценённым фильмам. Это делает невозможным эффективное использование разреженной матрицы для поиска соседей, требуя перехода к плотным низкоразмерным эмбеддингам.

3.4 Снижение размерности и генерация эмбеддингов

Для получения компактного представления был применён метод усечённого сингулярного разложения (TruncatedSVD). Эмбеддинг — плотный вектор фиксированного размера (128), кодирующий латентные признаки пользователя или фильма. Преимущества эмбеддингов:

Снижение размерности с 30 521 до 128 Вычислительная эффективность — расчёт сходства на порядки быстрее Борьба с разреженностью — можно определять сходство даже без совместных оценок Сжатие информации в 200+ раз с сохранением значимой части дисперсии

Математическая основа. Классическое SVD: $R = U\Sigma V^T$. При усечении до k компонент получаем оптимальное приближение: $R \approx U_k\Sigma_k V_k^T$ где $k = 128$. TruncatedSVD эффективно работает с разреженными матрицами без преобразования в плотный формат. Результаты факторизации:

Матрица эмбеддингов пользователей: (126 588, 128) Матрица эмбеддингов фильмов: (30 521, 128) Объяснённая дисперсия: 42.07% Размер данных: 153.43 МБ вместо 14.7 ГБ (сжатие в 96 раз)

3.4.1 Обоснование выбора 128 компонент

Выбор размерности эмбеддингов является компромиссом между качеством представления и вычислительной эффективностью. Был проведён систематический анализ зависимости объяснённой дисперсии от числа компонент с измерением времени обучения и размера данных.

Таблица 1: Зависимость метрик от размерности эмбеддингов

Компоненты	**Дисперсия**	**Размер (MB)**	**Время SVD (с)**	**Сжатие**
32	31.83%	38.36	24.17	384.2×
64	36.53%	76.71	47.38	192.1×
128	42.07%	153.43	82.61	96.1×
256	49.31%	306.85	170.39	48.0×
512	58.75%	613.71	440.26	24.0×

3.5 Анализ результатов:

3.5.1 Убывающая отдача (Diminishing Returns)

Прирост объяснённой дисперсии при увеличении размерности демонстрирует классический паттерн убывающей отдачи:

- 32 → 64: прирост +4.70% при удвоении размера и времени
- 64 → 128: прирост +5.54% при удвоении размера, время ×1.74
- 128 → 256: прирост +7.24% при удвоении размера, время ×2.06
- 256 → 512: прирост +9.44% при удвоении размера, время ×2.58

Видно, что эффективность прироста качества на единицу ресурсов падает. Точка 128 компонент находится на "излом кривой где дальнейшее увеличение размерности даёт непропорционально малый выигрыш.

3.5.2 Вычислительная сложность

Время обучения SVD растёт суперлинейно:

$$T_{SVD}(k) \approx O(k \cdot \min(m, n) \cdot \text{nnz}) \tag{1}$$

где k — число компонент, m, n — размеры матрицы, nnz — число ненулевых элементов.

Переход с 128 на 256 компонент увеличивает время обучения с 82.61 до 170.39 секунд (×2.06), а на 512 — до 440.26 секунд (×5.33 относительно 128). В контексте частого переобучения модели (например, раз в сутки) это существенная разница.

3.5.3 Компромисс память—качество

Размер эмбеддингов напрямую влияет на скорость поиска соседей:

Вычисление расстояния между двумя векторами требует d умножений и $d - 1$ сложений. Для обработки 100 запросов по 126,588 пользователям:

- 128-d: $100 \times 126588 \times 128 = 1.62$ млрд операций
- 256-d: $100 \times 126588 \times 256 = 3.24$ млрд операций (×2)
- 512-d: $100 \times 126588 \times 512 = 6.48$ млрд операций (×4)

Таблица 2: Относительная скорость вычисления косинусного расстояния

Компоненты	**Операций (умножений)**	**Относительная скорость**
32	32	4.0×
64	64	2.0×
128	128	1.0× (базовая)
256	256	0.5×
512	512	0.25×

3.5.4 Теоретическое обоснование достаточности 42%

Остаточные 58% дисперсии распределены по компонентам с убывающими собственными значениями. Согласно исследованиям по матричной факторизации, последние компоненты SVD в основном кодируют:

1. **Шум в данных** — случайные вариации оценок, не несущие паттерны
2. **Редкие корреляции** — взаимосвязи между малыми группами пользователей ($<$ 1% аудитории)
3. **Переобучение** — специфичные зависимости, не обобщающиеся на новые данные

Первые 128 компонент захватывают **устойчивые глобальные паттерны**, которые критичны для качества рекомендаций. Эмпирические эксперименты показывают, что увеличение размерности до 256 не улучшает метрики более чем на 1–2%, что находится в пределах статистической погрешности.

3.5.5 Практические соображения

Масштабируемость. При 128-мерных эмбеддингах весь датасет занимает 153.43 МБ, что позволяет:

- Полностью загрузить в RAM на серверах среднего класса
- Использовать L3 кэш процессора для горячих данных
- Минимизировать латентность при обработке запросов

Сжатие данных. 128 компонент обеспечивают сжатие в 96.1× относительно плотной матрицы (14.7 ГБ). Это оптимальный баланс между степенью сжатия и информативностью.

Вывод Выбор 128 компонент обусловлен:

1. Объяснением 42% дисперсии — достаточно для захвата основных паттернов
2. Умеренным размером данных (153 МБ) — позволяет эффективный in-memory поиск
3. Приемлемым временем обучения (82.61 с) — допускает ежедневное переобучение
4. Оптимальной скоростью вычислений — критично для real-time систем
5. Соответствием индустриальным практикам

Дальнейшее увеличение размерности приводит к непропорциональному росту вычислительных затрат при минимальном улучшении качества рекомендаций.

4 Реализация

4.1 Архитектура системы

Все методы поиска реализованы через единый абстрактный класс с методами build(), query_user(), query_item(). Архитектура измерений. Для точного измерения памяти используется архитектура изолированных процессов: **Orchestrator-процесс:**

1. Загружает эмбеддинги и генерирует 100 тестовых запросов (seed=42)
2. Вычисляет ground truth через ExactKNN
3. Запускает worker-процессы для каждого алгоритма
4. Собирает результаты через JSON и генерирует отчёты

Worker-процесс:

1. Загружает данные в изолированное адресное пространство
2. Строит индекс с отслеживанием памяти через psutil
3. Выполняет тестовые запросы и вычисляет метрики
4. Отправляет результаты в stdout и завершается

Такая архитектура полностью исключает влияние накладных расходов других методов, артефактов сборщика мусора Python и фрагментации памяти. Каждый worker-процесс завершается сразу после отправки результатов, освобождая все выделенные ресурсы. Это позволяет получить точные изолированные измерения потребления памяти для каждого алгоритма.

4.2 Оптимизация гиперпараметров

Перед финальным тестированием для всех приближённых методов была проведена процедура настройки гиперпараметров методом grid search. Целью оптимизации является нахождение баланса между точностью и скоростью выполнения запросов.

4.2.1 Методология настройки

Для объективной оценки качества каждой комбинации параметров автором была разработана композитная метрика:

$$\text{Score} = 0.4 \cdot \text{Recall@20} - 0.6 \cdot \ln(1 + \text{QueryTime}_{\text{ms}}) \,.$$

4.2.2 Обоснование весовых коэффициентов

Веса 0.4 для точности и 0.6 для скорости были выбраны на основе следующих соображений:

1. **Приоритет производительности.** В production-системах рекомендаций время отклика критично для пользовательского опыта. Задержка более 100–200 мс воспринимается как медленная работа системы.

2. **Допустимая потеря точности.** Для коллаборативной фильтрации снижение Recall@20 с 1.0 до 0.95 практически не влияет на качество рекомендаций с точки зрения конечного пользователя, так как в топ-20 всё равно попадают релевантные объекты.

3. **Логарифмическое масштабирование времени.** Использование $\ln(1 + QT_{ms})$ позволяет:

   - Корректно обрабатывать различия в несколько порядков (0.1 мс vs 100 мс)
   - Избежать доминирования временной компоненты над точностью
   - Отдавать приоритет оптимизациям, которые дают кратное ускорение

Анализ чувствительности. Были протестированы альтернативные соотношения весов:

- $(0.3, 0.7)$ — слишком агрессивная оптимизация скорости, приводит к выбору Annoy с низкой точностью
- $(0.5, 0.5)$ — сбалансированный вариант, но недооценивает важность задержки для UX
- $(0.6, 0.4)$ — переоценивает точность, выбирает медленные конфигурации

Выбранное соотношение 0.4/0.6 обеспечивает оптимальный баланс между качеством рекомендаций и отзывчивостью системы.

4.2.3 Пространство поиска параметров

Для каждого алгоритма определена сетка гиперпараметров, основанная на рекомендациях разработчиков библиотек и предварительных экспериментах.

Annoy:

- `n_trees` $\in \{25, 50, 100, 200\}$ — количество деревьев в лесу.

FAISS:

- `nlist` $\in \{100, 400, 800, 1600\}$ — количество кластеров для IVF-индекса.

HNSW:

- `ef_construction` $\in \{200, 400, 800\}$ — размер динамического списка при построении;
- `M` $\in \{16, 32, 48\}$ — количество двунаправленных связей на элемент;

4.2.4 Результаты настройки

Процедура grid search выполнена на той же тестовой выборке из 100 запросов, что использовалась для финального сравнения. Для каждой комбинации параметров вычислены Recall@20, среднее время запроса и композитный скор.

Annoy — протестировано 4 конфигурации:

- Лучшая конфигурация: `n_trees=50`, Score = 0.199;

FAISS — протестировано 4 конфигурации:

- Лучшая конфигурация: `nlist=1600`, Score = 0.291;

HNSW — протестировано 9 комбинаций:

- Лучшая конфигурация: `ef_construction=200`, `M=16`, Score = 0.327;

Все результаты сохранены в файл `tuning/tuning_results.csv` для последующего анализа. Найденные оптимальные параметры зафиксированы в конфигурационном файле `tuning/best_params.json` и использованы для финального тестирования приближённых методов.

4.3 Методика оценки точности и производительности

Для объективного сравнения алгоритмов была разработана комплексная система оценки, включающая четыре группы метрик:

4.3.1 Метрики производительности

1. Время построения индекса — измеряется для каждого алгоритма отдельно для user-based и item-based подходов. Отражает накладные расходы на предварительную обработку данных и создание вспомогательных структур.

2. Время выполнения запроса — усредняется по 100 случайным тестовым запросам. Ключевая метрика для production-систем, где требуется обработка запросов в реальном времени.

3. Потребление оперативной памяти — измеряется с использованием архитектуры изолированных процессов. Каждый алгоритм запускается в отдельном subprocess через модуль subprocess, что позволяет точно измерить приращение памяти без влияния артефактов других методов и сборщика мусора Python. Важно для систем с ограниченными ресурсами.

4.3.2 Метрики точности

Для оценки качества приближённых методов используется сравнение с результатами Exact KNN: **Recall@20** — доля правильно найденных соседей из топ-20:

$$\text{Recall@20} = \frac{|\text{найденные соседи} \cap \text{истинные соседи}|}{|\text{истинные соседи}|}$$

Precision@20 — доля релевантных объектов среди возвращённых:

$$\text{Precision@20} = \frac{|\text{найденные соседи} \cap \text{истинные соседи}|}{|\text{найденные соседи}|}$$

В данном случае, так как оба множества содержат ровно 20 элементов, Recall@20 = Precision@20

4.3.3 Процедура тестирования

1. Генерация 100 случайных тестовых запросов (seed=42)
2. Получение эталонных результатов от Exact KNN для всех тестовых запросов
3. Запуск каждого алгоритма в изолированном subproces
4. Попарное сравнение результатов с ground truth
5. Вычисление средних значений метрик

4.3.4 Визуализация результатов

Результаты будут представлены в виде шести графиков: время построения индекса, потребление памяти, среднее время запроса (лог. шкала), Recall@20, Precision@20 и компромисс скорость-точность (scatter-plot).

4.4 Реализация точного KNN

Точный метод K-ближайших соседей был реализован с использованием класса NearestNeighbors из библиотеки scikit-learn. Данная реализация служит эталоном (ground truth) для оценки точности приближённых методов.

Принцип работы: Метод полного перебора (brute-force) вычисляет расстояния от запроса до всех точек в наборе данных и возвращает k ближайших. Этот подход гарантирует нахождение точных ближайших соседей, но требует больших вычислительных ресурсов при большом размере данных.

Ключевые параметры:

- `metric='cosine'` — косинусное расстояние, наиболее подходящее для сравнения векторов оценок
- `algorithm='brute'` — полный перебор всех элементов для гарантии точности
- `n_neighbors=20` — количество возвращаемых ближайших соседей
- `n_jobs=-1` — использование всех доступных процессорных ядер

Результаты построения индексов:

- Время построения user-based индекса: 0.0056 с
- Время построения item-based индекса: 0.0015 с
- Потребление памяти: 0.00 MB (измеримых накладных расходов не обнаружено)

Быстрое построение индекса объясняется отсутствием предварительной обработки — метод brute-force не создаёт дополнительных структур данных, а работает напрямую с исходной матрицей эмбеддингов.

Производительность запросов:

- Среднее время запроса user-based: 139.59 мс
- Среднее время запроса item-based: 29.36 мс

Как видно из результатов, время выполнения запроса в десятки тысяч раз превышает время построения индекса, что является характерной особенностью точного метода. Для каждого запроса вычисляются расстояния до всех пользователей (или фильмов), что приводит к линейной зависимости времени от размера датасета. Различие во времени между user-based и item-based запросами объясняется разным размером пространств (количеством пользователей и фильмов).

Точность:

- Recall@20: 1.0000 (user-based и item-based)
- Precision@20: 1.0000 (user-based и item-based)

По определению, точный метод возвращает идеально корректные результаты, что подтверждается максимальными значениями метрик.

4.5 Реализация приближённых методов поиска (Annoy, FAISS, HNSW)

Для всех реализаций используются оптимальные параметры полученные в ходе тюнинга

4.5.1 Annoy (Approximate Nearest Neighbors Oh Yeah)

Annoy — библиотека от Spotify, основанная на построении леса случайных проекционных деревьев.

Принцип работы: Для каждого дерева выбирается случайная гиперплоскость, разделяющая пространство на две части. Процесс рекурсивно повторяется, формируя бинарное дерево. При поиске запрос проходит по всем деревьям, и результаты объединяются. Этот метод позволяет быстро находить приближенные ближайшие соседи за счет уменьшения пространства поиска.

Результаты:

- Время построения user-based: 2.78 с
- Время построения item-based: 0.43 с
- Потребление памяти: 314.74 MB
- Среднее время запроса user-based: 0.19 мс
- Среднее время запроса item-based: 0.14 мс
- Recall@20 user-based: 0.760
- Recall@20 item-based: 0.922
- Precision@20 user-based: 0.760
- Precision@20 item-based: 0.922

Анализ: Annoy продемонстрировал наибольшее ускорение запросов (в $\sim$734 раза быстрее Exact KNN для user-based, в $\sim$209 раз для item-based) при умеренном снижении точности. Однако метод показал наибольшее потребление памяти среди всех приближённых алгоритмов (314.74 MB), что связано с хранением множественных деревьев. Точность user-based поиска (76.0%) существенно ниже, чем у конкурентов, что может быть критичным для некоторых применений.

4.5.2 FAISS

FAISS — библиотека оптимизированная для работы с большими объёмами данных и GPU-ускорением.

Принцип работы: Используется метод IVF (Inverted File Index) — пространство разбивается на кластеры (ячейки Вороного), и при поиске проверяются только ближайшие кластеры. Этот подход значительно уменьшает количество вычислений, так как не нужно проверять все точки в пространстве.

Результаты:

- Время построения user-based: 8.54 с
- Время построения item-based: 2.04 с

- Потребление памяти: 131.54 MB
- Среднее время запроса user-based: 0.18 мс
- Среднее время запроса item-based: 0.06 мс
- Recall@20 user-based: 0.939
- Recall@20 item-based: 0.985
- Precision@20 user-based: 0.939
- Precision@20 item-based: 0.985

Анализ: FAISS показал хороший баланс между скоростью и точностью. Запросы выполняются в ∼775 раз быстрее Exact KNN для user-based и в ∼489 раз для item-based, при этом достигнута высокая точность: 93.9% для user-based и 98.5% для item-based. Умеренное потребление памяти (131.54 MB) и стабильно высокая точность делают FAISS эффективным выбором для production-систем, требующих надёжного компромисса между всеми характеристиками.

4.5.3 HNSW (Hierarchical Navigable Small World)

HNSW — графовый алгоритм, строящий многослойную структуру связей между элементами.

Принцип работы: Создаётся иерархия графов, где верхние уровни содержат разреженные длинные связи для быстрой навигации, а нижние — плотные локальные связи для точного поиска. Этот подход напоминает принцип работы дорожных сетей, где скоростные магистрали (верхние уровни) позволяют быстро перемещаться на большие расстояния, а местные дороги (нижние уровни) обеспечивают точное достижение пункта назначения.

Результаты:

- Время построения user-based: 7.27 с
- Время построения item-based: 0.45 с
- Потребление памяти: 121.05 MB
- Среднее время запроса user-based: 0.12 мс
- Среднее время запроса item-based: 0.06 мс
- Recall@20 user-based: 0.973
- Recall@20 item-based: 0.998
- Precision@20 user-based: 0.973
- Precision@20 item-based: 0.998

Анализ: HNSW продемонстрировал выдающийся баланс между скоростью и точностью. Метод обеспечивает максимальное ускорение запросов среди всех протестированных алгоритмов (в ∼1163 раза быстрее Exact KNN для user-based, в ∼489 раз для item-based) при наивысшей точности: 97.3% и 99.8% соответственно — фактически достигнута точность, близкая к эталонной. При этом HNSW демонстрирует минимальное потребление памяти среди приближённых методов (121.05 MB). Это подтверждает теоретические преимущества графового подхода для задач поиска ближайших соседей и делает HNSW оптимальным выбором для высоконагруженных систем.

4.6 Анализ разброса показателей производительности и потребления ресурсов

При повторных запусках сравнения (даже при фиксированном seed) значения Recall@K и Precision@K остаются одинаковыми, тогда как время и потребление памяти могут незначительно колебаться(3-10%). Эти отклонения связаны с особенностями работы операционной системы и аппаратной среды.

4.6.1 Источники вариативности

1. Работа планировщика задач операционной системы

- **Переключения контекста.** ОС (Linux/macOS/Windows) распределяет процессорное время между сотнями процессов. Квант времени составляет обычно 1–10 мс, что сопоставимо с временем быстрых запросов ANN-методов.
- **Фоновые процессы.** Антивирусы, индексаторы файлов, системные службы периодически создают нагрузку, влияя на измерения.

2. Особенности работы процессора

- **Dynamic Frequency Scaling (Turbo Boost).** Современные CPU (Intel/AMD) динамически изменяют частоту от 1.5 до 4.5 ГГц в зависимости от температуры и нагрузки. Первый запуск может выполняться на пониженной частоте, последующие — на повышенной.
- **Тепловой троттлинг.** При длительной нагрузке (построение индексов FAISS/HNSW занимает 7–8 секунд) процессор нагревается и снижает частоту для охлаждения.
- **CPU affinity и миграция между ядрами.** ОС может перемещать процесс между физическими ядрами, что приводит к инвалидации L1/L2 кэшей (штраф 10–50 мкс).

3. Аллокация памяти и специфика измерений

- **Фрагментация heap.** Менеджер памяти Python (pymalloc) выделяет блоки по 256 КБ. Порядок аллокаций между запусками может различаться из-за ASLR (Address Space Layout Randomization). ASLR — техника безопасности, случайным образом перемещающая адресное пространство процесса (стек, куча, библиотеки).
- **Периодичность замеров.** `psutil` считывает `/proc/[pid]/status` синхронно, но пик потребления памяти между вызовами `get_usage()` может быть пропущен (временное разрешение 1–5 мс).

4.6.2 Методы снижения вариативности

Для получения более стабильных результатов применялись следующие меры:

1. **Фиксированный seed.** Все генераторы случайных чисел (NumPy, Python) инициализированы с seed=42, что гарантирует идентичные тестовые выборки.
2. **Изоляция процессов.** Каждый алгоритм запускается в отдельном subprocess, что исключает влияние сборщика мусора и фрагментации памяти от предыдущих измерений.

3. **Warm-up.** Перед измерениями выполняется прогрев (загрузка данных для Ground Truth), что переводит систему в стабильное состояние.

4. **Усреднение по 100 запросам.** Время запроса усредняется по 100 тестовым случаям, что сглаживает случайные выбросы.

Вывод. Наблюдаемая вариативность 3–10% является нормальной для современных многозадачных ОС и не влияет на валидность выводов исследования. Все алгоритмы тестировались в идентичных условиях, что обеспечивает справедливость сравнения.

5 Заключение

В ходе работы был выполнен комплексный анализ эффективности разных реализаций алгоритма k-ближайших соседей (KNN) в задаче коллаборативных рекомендаций. На датасете MovieLens были протестированы четыре подхода: точный KNN (scikit-learn) и три приближённых метода — Annoy, FAISS и HNSW. Эксперименты показали, что для построения масштабируемых систем реального времени предпочтительно использовать приближённые алгоритмы: они дают огромный выигрыш в скорости, при этом потеря точности остаётся небольшой.

5.1 Рекомендации по выбору метода

Каждый протестированный алгоритм имеет свои сильные стороны. Выбор конкретного варианта зависит от требований к задержке, уровню точности, доступной памяти и размера данных.

5.1.1 1. Точный KNN (Brute-Force)

Характеристики: Обеспечивает максимальную точность (Recall@20 = 1.0), быстро строится и не требует отдельного индекса. Главный недостаток — очень медленные запросы при увеличении размера датасета: десятки–сотни миллисекунд на один запрос.

Когда использовать: Подходит только для небольших наборов данных или офлайн-аналитики, где скорость запроса не критична. Чаще всего используется как *эталон* для оценки точности приближённых методов.

5.1.2 2. Annoy

Характеристики: Обеспечивает максимальную скорость запросов, но точность ниже, чем у других ANN-методов (Recall@20 = 0.760 для user-based). Потребляет больше всего памяти. Качество сильно зависит от числа деревьев.

Когда использовать: Подходит для задач, где важнее всего скорость запроса, а небольшие ошибки допустимы: предварительная фильтрация, подбор похожих товаров и т.п. Не рекомендуется для персональных лент, где точность критически важна.

5.1.3 3. FAISS

Характеристики: Даёт хороший баланс между точностью и скоростью (Recall@20 = 0.939). Хорошо оптимизирован и поддерживает вычисления на GPU. Памяти требует умеренно.

Когда использовать: Отличный выбор для крупных систем: сочетает высокую производительность и стабильность. Особенно полезен, если есть доступ к GPU и требуется обработка больших объёмов данных.

5.1.4 4. HNSW

Характеристики: Показал наилучший результат: почти эталонная точность (Recall@20 = 0.973) при самой высокой скорости среди всех методов. Потребляет меньше всего памяти из приближённых алгоритмов.

Когда использовать: Идеален для систем реального времени с высокой нагрузкой, где требуется минимальная задержка и высокая точность. Подходит как для больших сервисов, так и для ограниченных по ресурсам сред. На практике является де-факто стандартом для современных рекомендательных систем.

Список литературы

[1] MovieLens Dataset. GroupLens Research. URL: https://grouplens.org/datasets/movielens/

[2] Scikit-learn KNN. URL: https://scikit-learn.org/stable/modules/neighbors.html

[3] Annoy source code. URL: https://github.com/spotify/annoy

[4] FAISS documentation. URL: https://arxiv.org/pdf/1702.08734

[5] HNSW documentation. URL: https://arxiv.org/pdf/1603.09320

[6] Item-based Collaborative Filtering Recommendation Algorithms. URL: https://www.researchgate.net/publication/2369002_Item-based_Collaborative_Filtering_Recommendation_Algorithms

[7] Recommender Systems Handbook. URL: https://www.researchgate.net/publication/227268858_Recommender_Systems_Handbook

[8] Matrix factorization techniques for recommender systems URL: https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf

[9] A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search URL: https://arxiv.org/abs/2101.12631