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 (scikit-learn);
   - Annoy (Spotify);
   - FAISS (Meta);
   - 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.

2 Обзор

2.1 Обзор рекомендательных систем и их классификация

Рекомендательные системы (RS) представляют собой программные системы, задача которых — предсказать интерес пользователя к объекту на основе анализа исторических данных. Существует три основных подхода к построению RS:

1. **Контентная фильтрация** (Content-Based Filtering) — анализирует характеристики объектов и строит рекомендации, исходя из сходства контента.
2. **Коллаборативная фильтрация** (Collaborative Filtering, CF) — основывается на взаимодействиях пользователей и объектов (рейтинги, покупки, клики).
3. **Гибридные методы** (Hybrid Methods) — объединяют оба подхода для повышения качества рекомендаций.

Коллаборативная фильтрация, в свою очередь, делится на:

- **User-based CF** — рекомендации формируются на основе схожих пользователей;
- **Item-based CF** — рекомендации формируются на основе схожих 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 контексте рекомендательных систем:

- В user-based CF ищутся пользователи, схожие по поведению с данным пользователем;
- В item-based CF — объекты, схожие с теми, что пользователь уже оценил.

Рекомендации формируются как взвешенное усреднение оценок ближайших соседей, что позволяет эффективно предсказывать предпочтения даже при отсутствии контентной информации.

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.

2.4 Существующие библиотеки поиска ближайших соседей

Среди наиболее популярных библиотек для реализации ANN можно выделить:

- **Annoy (Spotify)** — метод случайных проекций и построения деревьев, эффективен для больших векторов признаков.
- **FAISS (Meta)** — 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.

2.5 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 компромисс между скоростью и точностью, потребляя умеренные ресурсы;
- FAISS продемонстрирует высокую скорость при больших объёмах данных, особенно с GPU;
- HNSW обеспечит оптимальный баланс между скоростью, точностью и памятью, являясь лучшим выбором для production-систем.

Результаты исследования будут представлены в виде таблиц и графиков, отражающих:

- время построения индекса;
- среднее время запроса;

- использование оперативной памяти;
- показатели точности (Precision@20, Recall@20).

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 загрузки и предварительной обработки данных

- Загрузка датасета MovieLens (файлы movies.csv и ratings.csv).
- Удаление избыточных признаков (жанры фильмов, временные метки) для концентрации на данных взаимодействий.
- Первичная валидация и проверка целостности данных.

2. Модуль фильтрации и индексации данных

- Отбор активных пользователей (оставлены пользователи с более чем 50 оценок).
- Отбор популярных фильмов (оставлены фильмы с более чем 10 оценками).
- Создание непрерывных отображений (маппингов) для идентификаторов пользователей и фильмов в индексы матрицы, что позволяет избежать «пробелов» в индексации.

3. Модуль формирования разреженной матрицы взаимодействий

- Прямое формирование разреженной матрицы «пользователь-фильм» в формате CSR (Compressed Sparse Row) из отфильтрованных данных.
- Использование векторов оценок, индексов пользователей и фильмов для эффективного построения матрицы без создания промежуточных плотных структур.

4. Модуль матричной факторизации и генерации эмбеддингов

- Применение алгоритма TruncatedSVD из библиотеки Scikit-learn для выполнения матричной факторизации.
- Снижение размерности исходной разреженной матрицы до латентного пространства заданной размерности (128 компонентов).
- Генерация векторных представлений (эмбеддингов) для пользователей и фильмов как результатов разложения матрицы.

5. Модуль сохранения результатов и модели

- Сериализация полученных эмбеддингов в удобные для использования форматы (CSV для интерпретации, NumPy для быстрой загрузки).
- Сохранение созданных маппингов ID в индексы для последующего использования в сервисе рекомендаций.
- Сохранение обученной модели SVD для возможности ее дальнейшего использования или дообучения.

6. Модуль реализации алгоритмов поиска

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

7. Модуль мониторинга производительности

- Класс MemoryTracker для измерения потребления оперативной памяти
- Замеры времени построения индексов и выполнения запросов
- Использование библиотеки psutil для точного профилирования

8. Модуль оценки точности

- Генерация тестовой выборки из 100 случайных запросов
- Сравнение результатов приближённых методов с эталонным Exact KNN
- Вычисление метрик Recall@20 и Precision@20

9. Модуль визуализации и анализа

- Построение шести сравнительных графиков
- Сохранение результатов в CSV-формате
- Генерация текстового отчёта с рекомендациями

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).

Преимущества выбранного датасета:

- Реальные данные от пользователей сервиса MovieLens, что обеспечивает высокую внешнюю валидность результатов.
- Высокая разреженность, типичная для коллаборативных систем и представляющая ключевую вычислительную проблему.

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

Центральным элементом коллаборативной фильтрации является матрица взаимодействий «пользователь–объект». В данной работе была сформирована матрица размерностью $126\,588 \times 30\,521$, где строки соответствуют пользователям, столбцы — фильмам, а значения — оценкам от 0.5 до 5.0. Показатель разреженности матрицы составил 99.24% (плотность 0.76%).

Проблема плотного представления

Теоретически можно построить плотную матрицу размера $126\,588 \times 30\,521$, заполнив отсутствующие оценки нулями или специальными значениями. Однако такой подход имеет ряд критических недостатков:

- **Чрезмерные требования к памяти**: плотная матрица такого размера содержит более 3.86 млрд элементов. При хранении в формате `float32` это потребовало бы порядка 15.5 ГБ памяти, что значительно превышает объём оперативной памяти большинства рабочих станций.
- **Шум из-за заполнения пропусков**: подавляющее большинство взаимодействий отсутствует, и заполнение нулями создаёт искусственные связи «пользователь не любит этот фильм», хотя отсутствие оценки не означает отрицательное отношение.
- **Неэффективные вычисления**: большая часть операций (умножение матриц, вычисление расстояний) будет выполняться на бесполезных нулевых элементах.

Преобразование в разреженный формат

Для эффективной работы с высокоразреженными данными матрица была конвертирована в формат CSR (Compressed Sparse Row) с использованием библиотеки SciPy. Формат CSR хранит только ненулевые элементы и их индексы, что обеспечивает:

- Существенную экономию памяти (в данном случае хранится лишь 0.76% от полного объёма).
- Быструю индексацию строк и эффективное выполнение матричных операций, критически важных для последующих этапов.

Почему разреженного формата недостаточно

Несмотря на существенное снижение объёма данных, разреженная матрица остаётся крайне высокой размерности (десятки тысяч признаков на пользователя). Это приводит к следующим ограничениям:

- **Проблема «проклятия размерности»**: вычисление расстояний или сходства между разреженными векторами с десятками тысяч измерений становится неинформативным — большинство пар пользователей не имеют пересечения по оценённым фильмам.
- **Невозможность обобщения**: если два пользователя не оценили общие фильмы, методы на основе расстояний не смогут определить их сходство.
- **Высокие вычислительные затраты**: операции со столь широкими матрицами требуют значительных ресурсов, даже при хранении их в CSR-формате.

Эти ограничения делают невозможным эффективное использование разреженной матрицы напрямую для поиска соседей или построения рекомендаций. Поэтому применяется следующий этап: снижение размерности и генерация эмбеддингов.

3.4 Снижение размерности и генерация эмбеддингов

Чтобы получить компактное и информативное представление пользователей и фильмов, был применён подход матричной факторизации, позволяющий перейти от разреженного пространства исходных оценок к плотным низкоразмерным векторным представлениям — эмбеддингам.

Концепция эмбеддингов. Эмбеддинг представляет собой плотный вектор фиксированного размера (в данном случае 128), который кодирует сущность (пользователя или фильм) в непрерывном векторном пространстве. В отличие от разреженного представления, эмбеддинги не просто указывают на наличие взаимодействия, а захватывают его *латентные (скрытые) признаки*. Для фильма это могут быть абстрактные характеристики, такие как «уровень драматизма», «научно-фантастичность» или «динамичность экшена». Для пользователя эмбеддинг отражает его предпочтения по этим же латентным осям.

Преимущества использования эмбеддингов:

- **Снижение размерности**: Переход от пространства размерностью 30 521 (число фильмов) к пространству размерностью 128 значительно упрощает вычисления.
- **Вычислительная эффективность**: Расчёт метрик сходства в 128-мерном пространстве на порядки быстрее, чем в исходном разреженном пространстве.
- **Борьба с разреженностью**: Плотные векторы позволяют определять сходство даже между пользователями, не имеющими совместно оценённых фильмов.
- **Обобщение предпочтений**: Модель улавливает скрытые паттерны в поведении пользователей и структуре фильмов, сглаживая шум и исключая влияние единичных оценок.
- **Сжатие информации**: Эмбеддинги обеспечивают уменьшение числа признаков более чем в 200 раз и при этом сохраняют значимую часть дисперсии данных, что подтверждает эффективность кодирования.

Технология реализации. Для снижения размерности матрицы взаимодействий был использован метод усечённого сингулярного разложения (`TruncatedSVD`) из библиотеки `scikit-learn`. Классическое сингулярное разложение (SVD) представляется формулой

$$R = U\Sigma V^T.$$

Такое разложение можно понимать как разбиение исходной матрицы R на набор *основных направлений*, по которым данные изменяются сильнее всего. Эти направления задаются столбцами матриц U и V, а величины изменений вдоль них — значениями диагональной матрицы Σ.

Каждое сингулярное значение показывает, насколько "важна" соответствующая компонента: чем оно больше, тем больший вклад эта компонента вносит в структуру данных. Если оставить только k наибольших сингулярных значений, то мы получим приближение

$$R \approx U_k \Sigma_k V_k^T,$$

которое содержит только самые информативные латентные признаки.

Этот подход является оптимальным в следующем смысле: приближение ранга k, полученное таким образом, сохраняет *максимально возможную* часть информации исходной матрицы среди всех возможных матриц ранга k. Поэтому такой метод хорошо подходит для задач, где нужно уменьшить размерность без существенной потери смысла данных.

Метод `TruncatedSVD` эффективно работает с разреженными матрицами и не требует преобразования в плотный формат. В итоге в данной работе были получены:

- матрица эмбеддингов пользователей U_k размером $(126\,588, 128)$;
- матрица эмбеддингов фильмов V_k размером $(30\,521, 128)$.

Каждая из 128 координат в этих эмбеддингах соответствует одному из главных скрытых признаков, которые SVD сумело выделить из исходных оценок.

Полученные векторные представления являются конечным продуктом этапа предварительной подготовки и служат основой для построения моделей рекомендаций, основанных на вычислении сходства векторов в общем латентном пространстве.

Вывод по результатам генерации эмбеддингов

В результате факторизации пользовательско-предметной матрицы методом усечённого сингулярного разложения (SVD) были получены плотные эмбеддинги пользователей и фильмов размерностью 128. Несмотря на крайне высокую разреженность исходной матрицы (0.76% заполненности), метод позволил выделить ключевые латентные структуры данных, сохранив 42.07% общей дисперсии оценок.

Созданные векторы имеют компактный размер: 153.43 МБ вместо 14.7 ГБ, которые потребовались бы для хранения плотной матрицы, что соответствует сжатию информации в 96 раз без существенной потери качества. Полученные эмбеддинги успешно используются для вычисления сходства между пользователями и фильмами в низкоразмерном пространстве и демонстрируют корректность рекомендаций на практике: для тестового пользователя были возвращены фильмы с ожидаемо высокими оценками и жанровой близостью.

Показатель «42% объяснённой дисперсии» означает, что выбранные 128 латентных признаков сохраняют 42% всей информации, содержащейся в исходной разреженной матрице оценок. Несмотря на то, что это значение меньше 100%, его оказывается достаточно для построения рекомендаций. Это объясняется тем, что в реальных данных большая часть сигналов относится к шуму: оценки пользователей неоднородны, субъективны и часто неполны. Латентные компоненты SVD автоматически отбрасывают шумовые направления и сохраняют только наиболее устойчивые и значимые зависимости между пользователями и фильмами.

Таким образом, применение SVD позволило эффективно преобразовать исходные данные в информативное и вычислительно удобное представление, являющееся надёжной основой для построения рекомендательной системы.

4 Реализация

4.1 Архитектура системы

Все методы поиска ближайших соседей реализованы как наследники единого абстрактного базового класса, обеспечивающего унифицированный интерфейс для построения индексов и выполнения запросов. Такой подход гарантирует консистентность измерений и упрощает добавление новых алгоритмов.

Для обеспечения корректного измерения потребления памяти применена архитектура изолированных процессов, состоящая из двух типов компонентов: orchestrator (координатор) и worker (исполнитель). **Orchestrator-процесс** выполняет следующие функции:

1. Подготовка общих данных: загрузка эмбеддингов, генерация тестовой выборки из 100 случайных пользователей и фильмов (seed=42 для воспроизводимости)
2. Вычисление ground truth с использованием ExactKNN для всех тестовых запросов
3. Сериализация подготовленных данных (тестовые индексы и ground truth) во временные pickle-файлы для передачи worker-процессам
4. Последовательный запуск изолированных worker-процессов для каждого алгоритма через модуль subprocess
5. Сбор результатов из stdout каждого worker-процесса путем парсинга JSON-данных между маркерами `__RESULT_START__` и `__RESULT_END__`
6. Агрегация результатов в pandas DataFrame и генерация итоговых отчетов
7. Очистка временных файлов после завершения всех измерений

Worker-процесс работает в полностью изолированном окружении:

1. Загрузка собственной копии эмбеддингов в чистое адресное пространство процесса
2. Десериализация тестовых индексов и ground truth из временных файлов
3. Инициализация конкретного алгоритма с оптимизированными параметрами из конфигурационного файла
4. Построение индекса с одновременным отслеживанием потребления памяти через класс MemoryTracker (использует psutil для мониторинга RSS)
5. Выполнение тестовых запросов с измерением времени и вычислением метрик точности
6. Сериализация результатов в JSON и вывод в stdout для захвата orchestrator-процессом

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 была разработана композитная метрика:

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

Эта формула балансирует два ключевых требования:

- **Точность** (вес 0.4): более высокий Recall@20 увеличивает итоговый скор;
- **Скорость** (вес 0.6): логарифм времени запроса вычитается, поэтому более быстрые алгоритмы получают преимущество.

Использование логарифма времени позволяет корректно обрабатывать различия в несколько порядков и избежать доминирования временной компоненты над точностью. Веса 0.4/0.6 были выбраны эмпирически с приоритетом на производительность, что соответствует требованиям production-систем рекомендаций.

4.2.2 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;
- Общее количество комбинаций: $3 \times 3 = 9$.

4.2.3 Результаты настройки

Процедура grid search выполнена на той же тестовой выборке из 100 запросов, что использовалась для финального benchmarking. 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;
- Увеличение количества деревьев до 200 повышает точность с 0.76 до 0.95, но увеличивает время запросов с 0.127 мс до 0.669 мс, что снижает композитный скор;

- Оптимум смещён к минимальному значению `n_trees` из-за высокого веса скорости.

FAISS — протестировано 4 конфигурации:

- Лучшая конфигурация: `nlist=1600`, Score = 0.291;
- `nlist=800` обеспечивает баланс: точность 0.966 при времени 0.249 мс;
- `nlist=1600` увеличивает точность до 0.99, но замедляет запросы до 0.510 мс.

HNSW — протестировано 9 комбинаций:

- Лучшая конфигурация: `ef_construction=200`, `M=16`, Score = 0.327;
- Параметр `M=16` показал стабильно высокие результаты во всех настройках;
- Конфигурация (200, 16) достигла точности 0.9735 при времени запроса 0.11 мс;
- Максимальная точность 0.994 (800, 48) достигается за счёт увеличения времени до 0.201 мс, что снижает итоговый скор до 0.201.

Все результаты настройки 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 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.

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

4.4.1 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.

Параметры реализации:

- `n_trees=50` — количество деревьев в лесу
- `metric='angular'` — эквивалент косинусного расстояния

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 ∼734 раза быстрее Exact KNN для user-based, в ∼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.

4.4.2 FAISS (Facebook AI Similarity Search)

FAISS — библиотека от Meta, 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.

Параметры реализации:

- `nlist=1600` — целевое количество кластеров
- `nprobe=20` — количество кластеров, проверяемых при поиске
- L2-нормализация векторов для корректного вычисления косинусного сходства

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.

4.4.3 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.

Параметры реализации:

- `ef_construction=200` — размер динамического списка при построении
- `M=16` — количество двунаправленных связей на элемент

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.

4.5 Анализ разброса показателей производительности и потребления ресурсов

В эксперименте метрики качества (Recall@K, Precision@K) полностью совпадали между разными запусками при фиксированном `random seed`. Однако время построения индекса (Build Time), время запроса (Avg Querry Time) и максимальное потребление памяти (Peak Memory Usage) немного менялись. Это нормальное поведение для современных вычислительных систем и объясняется следующими факторами.

4.5.1 Работа планировщика задач операционной системы

Обычные ОС не гарантируют строго одинаковое время работы программ. Планировщик распределяет процессорное время между многими процессами.

- **Переключение контекста.** Программа периодически прерывается системными процессами и прерываниями. Это добавляет случайные задержки, поэтому итоговое время можно представить как

$$T_{\text{total}} = T_{\text{compute}} + T_{\text{wait}}.$$

- **Состояние файлового кэша.** Быстрота загрузки данных зависит от того, находятся ли файлы уже в кэше ОС или читаются с диска. Это меняет время функции `load_embeddings`.

4.5.2 Особенности работы процессора

Современные процессоры используют динамическое изменение частоты (DVFS). Например, при повышении температуры или увеличении нагрузки частота может снижаться. Это влияет на время построения индексов, особенно вычислительно сложных (HNSW, FAISS).

4.5.3 Аллокация памяти и особенности измерений

- **Фрагментация памяти.** Аллокаторы Python и C++ распределяют память блоками. Из-за различий в порядке выделения могут возникать небольшие расхождения в объёме занятой памяти.

- **Дискретность измерений.** Профайлер памяти делает замеры с определённой периодичностью. Настоящий пик мог произойти между измерениями, поэтому зафиксированное значение может немного отличаться от реального.

4.6 Методика оценки точности и производительности

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

4.6.1 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.

4.6.2 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

4.6.3 Процедура тестирования

1. Генерация тестовой выборки из 100 случайных пользователей и 100 случайных фильмов (seed=42 для воспроизводимости)
2. Получение эталонных результатов от Exact KNN для всех тестовых запросов
3. Сохранение ground truth и тестовых индексов во временные файлы для передачи worker-процессам
4. Последовательный запуск каждого алгоритма в изолированном subprocess
5. Выполнение тех же запросов для каждого приближённого метода в чистом окружении
6. Попарное сравнение результатов с использованием множественных операций
7. Вычисление средних значений метрик по всем тестовым запросам
8. Сбор результатов через JSON-сериализацию в stdout worker-процессов

4.6.4 Визуализация результатов

Результаты представлены в виде шести графиков:

1. Время построения индекса — сравнение накладных расходов
2. Потребление памяти — оценка ресурсоёмкости
3. Среднее время запроса (логарифмическая шкала) — демонстрация различий в производительности
4. Recall@20 — оценка полноты результатов
5. Precision@20 — оценка точности результатов
6. Компромисс скорость-точность — scatter-plot для выявления оптимальных алгоритмов

Такой комплексный подход позволяет оценить каждый алгоритм с разных сторон и выбрать оптимальное решение в зависимости от приоритетов конкретной задачи.

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