[НИУ ВШЭ]

[Факультет компьютерных наук]

Курсовая работа

Выполнил(а):
студент группы БПМИ224
Сенчугова Кирилла Игоревича

Руководитель:
Галицкий Б.В., должность

[Москва] [2025]

Содержание

1 Аннотация 2

2 Введение 2

3 Теоретическая часть 4
- 3.1 Базовый (последовательный) алгоритм k-NN . . . 4
- 3.2 Многопоточный k-NN на CPU . . . 5
- 3.3 FAISS CPU (точный поиск) . . . 5
- 3.4 FAISS GPU (точный поиск) . . . 6
- 3.5 FAISS IVFPQ (приближённый поиск) . . . 7
- 3.6 cuML GPU . . . 8
- 3.7 Dask распределённый k-NN . . . 9
- 3.8 Сравнение подходов . . . 10

4 Экспериментальная часть 10
- 4.1 Критерии оценки и оборудование . . . 10
- 4.2 Проведение эксперимента и результаты . . . 12
- 4.3 Обсуждение результатов . . . 14
- 4.4 Выводы по дополнительным экспериментам . . . 17

5 Заключение 17

1 Аннотация

В работе проведено исследование методов оптимизации алгоритма k-ближайших соседей (k-NN) за счёт использования параллельных и распределённых вычислений. Рассмотрены классический последовательный алгоритм, многопоточная реализация на CPU, ускорение с помощью библиотеки FAISS на CPU и GPU, приближённый поиск с FAISS IVFPQ, GPU-реализация с помощью библиотеки cuML, а также распределённая обработка на кластере с использованием Dask. В теоретической части описаны особенности каждого метода, их преимущества и ограничения. Экспериментальная часть включает сравнение методов по скорости выполнения, точности и масштабируемости на синтетическом наборе данных. Результаты показали, что выбор оптимального метода зависит от объёма данных, требований к точности и доступных вычислительных ресурсов. Работа подчёркивает важность применения параллелизма и специализированных библиотек для эффективного использования k-NN в условиях больших данных.

Ключевые слова: k-ближайших соседей, параллельные вычисления, FAISS, cuML, Dask, GPU-ускорение, приближённый поиск, машинное обучение, оптимизация алгоритмов, большие данные.

2 Введение

Алгоритм k-ближайших соседей (k-NN) — один из простейших методов классификации и регрессии в машинном обучении. Его идея интуитивно понятна: для каждого нового объекта («запроса») находим k наиболее близких объектов обучающей выборки и по их меткам предсказываем метку нового. При этом ключевая операция — вычисление расстояний между векторовыми представлениями объектов. Несмотря на простоту и высокую точность в ряде задач, алгоритм k-NN обладает существенными вычислительными ограничениями. В базовом (последовательном) варианте на каждый запрос требуется пройти по всей обучающей выборке из N объектов и вычислить с каждым из них расстояние. При размерности пространства признаков D вычисление, например, Евклидова расстояния между парой точек требует D вычитаний и D умножений (или возведений в квадрат), а также суммирования. Иными словами, для одного запроса требуется порядка $\mathcal{O}(N \cdot D)$ элементарных операций вычисления расстояний. Если требуется найти соседей для Q объектов, общая сложность составляет $\mathcal{O}(N \cdot D \cdot Q)$. Пространственная сложность алгоритма – $\mathcal{O}(N \cdot D)$, поскольку нужно хранить всю обучающую выборку. При современных размерах данных (сотни тысяч – миллионы объектов, десятки и сотни признаков) прямой перебор становится крайне затратным по времени. Например, при $N = 10^5$ и $D = 100$ число операций оценивается в тысячи миллиардов, что приводит к многим минутам и даже часам

работы на одном CPU.

Кроме того, существует феномен «проклятия размерности». По мере увеличения числа признаков D пространство становится очень разреженным: объекты «разлетаются» по пространству, и расстояния между ними начинают «сглаживаться» (все расстояния становятся почти одинаковыми). Это означает, что близость объектов перестаёт хорошо отражать их истинную схожесть, а ускоряющие структуры данных (например, k-d-деревья) теряют эффективность. В высоких размерностях даже «интеллектуальные» структуры сводятся к полному перебору, поскольку почти все объекты приходится просмотреть. Таким образом, базовый k-NN плохо масштабируется ни по числу объектов N, ни по размерности D.

В связи с этим встает задача ускорения k-NN. Существует несколько направлений решения:

- Приближённые методы. Вместо точного перебора допускается получать соседей с небольшой погрешностью ради большой экономии времени. К ним относятся методы типа Locality-Sensitive Hashing (LSH), графовые индексы (HNSW и др.), а также библиотека FAISS с несколькими видами индексов (например, IVFPQ — индекс на основе инвертированного файла и продуктового квантования).

- Алгоритмические структуры данных. Классически для k-NN разрабатывали пространственные структуры (k-d-деревья, ball-деревья и др.), которые отсекают ненужные части пространства. Однако их эффективность резко падает в высоких измерениях.

- Параллельные вычисления. Метод очень хорошо распараллеливается, поскольку расчёты расстояний для разных пар точек независимы. Многопоточность на CPU, распараллеливание на GPU и распределённые вычисления позволяют добиться значительного ускорения. Специализированные библиотеки, такие как FAISS (Facebook AI Similarity Search) и RAPIDS cuML (NVIDIA), реализуют высокопроизводительные k-NN и учитывают возможности параллельных архитектур.

Таким образом, целью настоящей работы является исследование и сравнение различных подходов к ускорению k-NN: от простейшего последовательного алгоритма до современных параллельных и распределённых решений. В введении мы обозначили основные сложности и мотивацию для ускорения. В следующих разделах приведены теоретические описания каждого метода (последовательно: базовый k-NN, многопоточный на CPU, точные и приближённые методы FAISS, GPU-реализация cuML, распределённый Dask-kNN), их математические основы и основные достоинства/недостатки. После этого в экспериментальной части описывают-

ся критерии оценки (скорость, точность, ресурсы, масштабируемость), параметры оборудования, приводятся измерения времени и точности из эксперимента, а также обсуждается поведение каждого метода на разных типах данных. В заключении приведены выводы о том, какие подходы лучше работают в каких условиях, а также обсуждаются ограничения и перспективы.

3 Теоретическая часть

3.1 Базовый (последовательный) алгоритм k-NN

В самом простом варианте k-NN хранит все обучающие вектора в памяти. На этапе классификации для нового объекта x вычисляются расстояния до всех N точек обучающей выборки $\{x_i\}$. Обычно используется Евклидово расстояние:

$$\rho(x, x_i) = \sqrt{\sum_{j=1}^{D} (x_j - x_{i,j})^2}.$$

Поскольку функция $\sqrt{\cdot}$ монотонна, при сравнении ближайших соседей можно работать с квадратами расстояний и не вычислять корень. После того как получены все расстояния $\{\rho(x, x_i)\}$, выбираются k объектов с наименьшими расстояниями (требуется сортировка или алгоритм выбора k-го порядка, например `np.argpartition`). Наконец, для классификации применяют правило большинства по меткам соседей (например, класс c выбирается по формуле

$$\hat{y} = \arg\max_{c} \sum_{j=1}^{k} [y_{(j)} = c],$$

где $y_{(j)}$ — метки k ближайших соседей).

Таким образом, последовательный k-NN не имеет фазы обучения: его «модель» — это просто хранилище всех обучающих точек. Вычисления происходят в фазе предсказания, что требует перебора N и вычисления расстояний размерности D. Как отмечалось, временная сложность этого алгоритма порядка $\mathcal{O}(N \cdot D)$ на один запрос. При большом объёме данных алгоритм становится крайне медленным.

Преимущества: максимальная точность (возвращает истинных ближайших соседей), простота реализации, не требует фазы обучения. Недостатки: линейная сложность по N, неэффективен на больших данных, проклятие размерности. Поэтому для практического применения базовый k-NN подходит лишь для относительно небольших наборов данных или при очень малых N и D.

3.2 Многопоточный k-NN на CPU

Поскольку вычисление расстояний для разных точек независимо, алгоритм k-NN хорошо поддаётся распараллеливанию. В многопоточной реализации мы можем разбить работу между несколькими ядрами процессора. Типичный подход: разделить множество запросов (или обучающих точек) на порции и запустить несколько потоков.

Например, можно разбить тестовую выборку на P частей и в каждом потоке посчитать ближайших соседей только для своей части запросов. Параллельно каждый поток выполняет те же операции: вычисляет расстояния от своих Q/P запросов до всех N обучающих объектов и выбирает k ближайших. В конце результаты собираются вместе. В альтернативном подходе разбивают обучающие точки: каждый поток отвечает за поиски в своей части обучающих данных и выдаёт k локальных соседей, после чего производится объединение (например, выбор глобальных топ-k из локальных результатов).

В идеальном случае при P потоках скорость работы стремится к P-кратному ускорению относительно однопоточного выполнения: время становится примерно $\mathcal{O}(\frac{N \cdot Q \cdot D}{P})$. Однако на практике достигается не линейный по P выигрыш из-за накладных расходов на синхронизацию, переключение контекста и конкуренцию потоков за память. В частности, если данные хранятся в одном массиве, то множество ядер одновременно обращается к общей памяти, что приводит к узким местам (bandwidth) на доступ к ОЗУ. Кроме того, в высокоуровневых языках программирования (например, Python) могут мешать ограничения типа GIL (глобальная блокировка интерпретатора). В нативных реализациях (на C/C++ с OpenMP или TBB) таких ограничений нет, и распараллеливание эффективнее.

Преимущества многопоточности на CPU: существенное ускорение на многоядерных CPU; не требует специализированного оборудования; простота реализации (с помощью OpenMP, потоковых библиотек и т.п.); отсутствие дополнительной памяти (используются те же данные). Недостатки: ограничено числом ядер (на практике 4–16); накладные расходы на синхронизацию; конкуренция за память; эффект ограничен. На малых объёмах (маленькое N или Q) накладные расходы могут перевесить выгоду.

3.3 FAISS CPU (точный поиск)

FAISS (Facebook AI Similarity Search) — это высокопроизводительная библиотека поиска по близости, разработанная в Meta. Она содержит несколько реализаций индексов, среди которых есть как точные, так и приближённые методы. Для точного поиска на CPU используется индекс `IndexFlatL2` (или `IndexFlat`), который по сути делает перебор по всем точкам, но реализован максимально оптимально на C++.

В этом случае все обучающие вектора x_i хранятся в индексе. При запросе FAISS вычисляет дистанции запроса до всех N векторов: эта операция может быть распараллелена с помощью SIMD-инструкций (автоматически при наличии поддержки), а также многопоточна за счет OpenMP. Затем для каждой строки результатов находится k минимальных расстояний. Иными словами, временная сложность по-прежнему порядка $\mathcal{O}(N \cdot D)$ на один запрос, но коэффициент значительно меньше, чем у «чистого» Python/NumPy. FAISS может использовать ускоренные BLAS или AVX-инструкции для работы с большими массивами.

Формально, для индекса `IndexFlatL2` после добавления N обучающих точек мы выполняем поиск:

$$\texttt{D, I = index.search(X\_test, k)},$$

где D — массив расстояний, а I — индексы соседей. FAISS возвращает точно тех же ближайших, что и последовательная реализация, без потери точности.

Преимущества FAISS CPU: высокая скорость за счёт оптимизаций на C/C++, использование всех ядер CPU (OpenMP), способность обрабатывать очень большие наборы данных при достаточной памяти. Простота использования: Python-обёртки позволяют в пару строк создать индекс и запустить поиск. Недостатки: сохраняется линейная зависимость времени от N. Требуется хранить все данные в памяти индекса. На очень больших данных может потребоваться много RAM. Поиск всё равно «поразмерный» — время находится в пропорции к общему объёму. Однако на практике FAISS CPU на 8 ядрах уже работал почти в 8 раз быстрее обычного SciKit-Learn (с n_jobs=1), что также видно в экспериментах.

3.4 FAISS GPU (точный поиск)

FAISS поддерживает GPU-ускорение. Идея в том, чтобы перенести массивы данных и вычисления на графический процессор с тысячами ядер. FAISS предоставляет функцию `faiss.index_cpu_to_gpu`, которая копирует уже созданный CPU-индекс на GPU, либо есть специализированные классы индексаторов, выполняющих всё на GPU. В любом случае `index_gpu.search()` выполнит поиск на видеокарте.

На GPU считывание и запись данных происходит очень быстро, а операции с массивами векторов (вычитания, возведения в квадрат, суммирования) полностью распараллеливаются: каждая CUDA-нить вычисляет расстояние между одной парой точек или обрабатывает часть векторов. В результате скорость поиска возрастает на порядок. По оценкам разработчиков FAISS, при переносе на GPU (например, NVIDIA Tesla P100 или T4) можно получить 5-20-кратное ускорение по сравнению с CPU-версией. В наших экспериментах FAISS на GPU нашёл ближайших соседей примерно в 3-4 раза быстрее, чем FAISS на 8 ядрах CPU (примерно 1.3 с против 3.8

с).

Преимущества FAISS GPU: всё те же алгоритмы точного поиска, но в разы быстрее за счёт параллельности GPU. Доступна обработка больших массивов в векторизованном виде. Уменьшенные задержки при поиске благодаря высокой производительности. Недостатки: данные нужно разместить в памяти GPU (ограниченный объём, например, 16 ГБ на T4), что может быть недостаточно для очень больших наборов. Передача данных из CPU в GPU и обратно требует времени (хотя для поиска после загрузки это компенсируется скоростью вычислений). GPU-реализация имеет смысл при существенном объёме работы; при небольших данных накладные расходы на управление GPU могут перекрыть выгоду. Кроме того, для большого параллелизма нужна многокарточная архитектура (FAISS поддерживает multi-GPU, но это усложняет настройку).

3.5 FAISS IVFPQ (приближённый поиск)

Для ещё более существенного ускорения FAISS предлагает приближённые индексы. Одним из самых распространённых является `IndexIVFPQ` (инвертированный файл + Product Quantization). Этот метод разбивает поиск на два этапа:

Coarse Quantizer (IVF): все обучающие векторы кластеризуются на n_{list} кластеров (например, алгоритмом k-means). Для каждого обучающего вектора запоминается только номер ближайшего кластера (инвертированная файловая структура). То есть обучающие точки разнесены по спискам по кластерам, и хранится множество центроидов (каждого кластера).

Product Quantization (PQ): сами векторы далее сжимаются. Например, в стандартном PQ вектор $x \in \mathbb{R}^D$ разбивается на m непересекающихся подпоследовательностей (каждая длины D/m), и для каждой такой подпоследовательности строится кодбук размером 2^b. Таким образом, вектор x кодируется последовательностью индексов из m кодбуков, каждый из которых указывает ближайший центроид для соответствующей подпоследовательности. При этом центры кодбуков обучаются на подвыборках данных.

На этапе поиска для запроса q сначала находится несколько (n_{probe}) ближайших центроидов из набора из n_{list} кластеров. Затем выполняется перебор только по точкам, попавшим в эти кластеры. При этом реальные векторы уже не хранятся в полном виде, а их сегменты представлены индексами в кодбуках. Расстояния между q и точками в этих кластерах приближённо вычисляются на основе предвычисленных таблиц расстояний от каждой подпоследовательности запроса до кодбучных центров. По сути, количество точек для перебора уменьшается примерно в n_{list} раз, а расчёт расстояний упрощается за счёт квантования.

Главный результат: `IndexIVFPQ` возвращает приближённые ближайшие сосе-

ди. Точность поиска (доля истинных соседей в выборке результатов) немного ниже, но скорость значительно выше. Этот метод особенно эффективен на очень больших наборах данных (например, миллионах объектов), когда даже FAISS-flat слишком медленен. При разумном подборе параметров (n_{list}, m, размер кодбуков) достигаются сотни раз ускорения с потерей точности лишь на несколько процентов.

Преимущества FAISS IVFPQ: экстремальное сокращение времени поиска и памяти (так как хранятся только короткие коды вместо полных векторов). Подходит для Very Large Scale (VLS) данных, где требуется быстрый поиск. Библиотека позволяет гибко настраивать параметры (число кластеров n_{list}, количество сегментов m, биты кодирования), а также есть GPU-версия IndexIVFPQ.

Недостатки: необходимость фазы обучения (кластеры k-means и кодбуки PQ нужно заранее обучить на выборке данных). Результат становится приближённым: в выборку результатов могут попасть не самые ближайшие соседи, а несколько других из выбранных кластеров. Сильно зависит от параметров: при слишком грубой кластеризации или малом m точность падает. Настройка этих гиперпараметров требует времени и опыта. Также требуется дополнительная память на хранение кодбуков. В случае динамических данных, которые часто добавляются/удаляются, индекс потребуется переобучать (хотя FAISS поддерживает добавление точек после обучения). В нашем эксперименте метод IVFPQ с разумными параметрами работал быстрее всех (0.7 с), но с небольшой потерей классификационной точности по сравнению с точными методами.

3.6 cuML GPU

RAPIDS cuML — это библиотека от NVIDIA, предоставляющая GPU-ускоренные реализации алгоритмов машинного обучения с интерфейсом, похожим на scikit-learn. В частности, cuML содержит класс `KNeighborsClassifier`, который реализует k-NN на GPU. Под капотом cuML часто использует библиотеки вроде FAISS: для поиска соседей может быть задействован FAISS GPU, а иногда — собственные оптимизированные ядра CUDA.

При использовании cuML все входные данные (массивы признаков и меток) должны находиться в памяти GPU (обычно передаются как cuDF или cuPy массивы). После этого команда `knn.fit(X_train, y_train)` встраивает данные в GPU-индекс, а `knn.predict(X_test)` запускает поиск соседей на GPU. Во многих случаях cuML KNN даёт сопоставимый по быстродействию результат с чистым FAISS GPU. По заявлениям NVIDIA, cuML может давать сотни раз ускорение по сравнению с CPU, хотя конкретные результаты зависят от задачи.

Преимущества cuML: удобство использования (API похож на scikit-learn), интеграция с остальными компонентами RAPIDS (dask-cuDF и др.) для распределённо-

сти, поддержка GPU без необходимости самим вызывать CUDA. Можно работать на кластере с несколькими GPU с помощью dask-cuML. Кроме того, cuML может автоматически переключаться между точными и приближенными методами (например, через FAISS) по мере необходимости. Недостатки: то же ограничение памяти GPU (все данные должны поместиться на каждой карте). Кроме того, cuML в первую очередь ориентирован на задачи классификации и регрессии; расширенные настройки индексов FAISS могут быть менее доступны из API. Скорость cuML ближе к FAISS GPU (мы получили 1.4 с), но иногда чуть уступает, вероятно, из-за накладных расходов слоя интеграции. Также версия cuML может оказаться несовместимой с некоторыми средами или требовать специфических версий CUDA. В целом, cuML является удобным средством GPU-ускорения k-NN при больших данных, но при необходимости тонкой настройки индекса FAISS всё равно стоит работать напрямую с FAISS.

3.7 Dask распределённый k-NN

Dask — это фреймворк для параллельных и распределённых вычислений в Python. Он позволяет «раскладывать» массивы и алгоритмы на несколько процессов, ядер и даже машин. Реализацию k-NN с помощью Dask можно построить, разбивая обучающие данные на несколько чанков и обрабатывая их на разных воркерах.

В простейшем сценарии обучающая выборка делится на P частей (например, по строкам массива). Затем на каждом воркере (или процессе) выполняется локальный поиск: вычисляются ближайшие соседи тестовых точек только среди своей порции обучающих данных. В итоге каждый воркер выдаёт по одному «предсказанию» для каждого теста. Эти частичные предсказания нужно объединить. Один из способов — сделать голосование между результатами разных воркеров: если обрабатывалось P чанков, то для каждого тестового объекта получается P ответов (по одной метке от каждого воркера), и итоговая метка выбирается по большинству. Альтернативно, можно на каждом воркере не только выдавать метку, но и сохранять индексы ближайших соседей, после чего, объединив их, найти глобальные k ближайших по совокупности.

Dask позволяет масштабировать k-NN горизонтально: можно использовать несколько узлов кластера и огромное количество памяти, преодолевая ограничение одной машины. Это полезно, если данные не помещаются в память одного сервера. Недостаток в том, что появляется накладной трафик между узлами: нужно либо пересылать данные запросов, либо получать результаты соседей и объединять. В нашем эксперименте мы имитировали распараллеливание на 4 ядрах (Dask compute с 4 задачами) и получили время около 4.2 с — почти как 8-поточный CPU. Более того, описанный способ объединения (голосование) даёт несколько приближённый результат: качество классификации может слегка упасть, так как решение не осно-

вано на глобальных k-соседях, а на комбинации локальных. Чем больше чанков, тем сильнее этот эффект, хотя голосование сглаживает ошибки.

Преимущества Dask: возможность работы с объемными данными, которые не помещаются в ОЗУ одного узла; автоматическая организация параллелизма и координации. После распределения задач Dask сам управляет потоками, планирует работу. Недостатки: значительные накладные расходы на коммуникацию между узлами, сложность настройки и отладки кластеров, неоптимальность для малых данных. Кроме того, гарантировать точную эквивалентность обычному k-NN сложно без сложного объединения результатов. Dask хорошо подходит там, где данные по объёму и скорости не позволяют иначе, но ценой усложнения архитектуры.

3.8 Сравнение подходов

Рассмотрим сводную таблицу основных преимуществ и недостатков каждого рассмотренного метода:

Данная таблица суммирует ключевые факторы. В целом, точные методы (базовый, FAISS CPU/GPU, cuML) дают полную точность и простоту настройки, но дорого стоят при больших данных. Приближённый IVFPQ и другие ANN-индексы жертвуют точностью ради скорости и масштабируемости. Параллельные подходы (CPU/GPU/Dask) используют ресурсы оборудования, чтобы уменьшить время решения, но требуют дополнительных вычислительных мощностей.

4 Экспериментальная часть

4.1 Критерии оценки и оборудование

В этой части мы описываем критерии оценки методов и условия экспериментов. Основными метриками служат:

- Время выполнения (скорость): полный wall-clock time на поиск k-ближайших для фиксированного набора данных (включая предварительную постройку индекса, если требуется). Это ключевой критерий ускорения алгоритма.

- Точность: для задачи классификации измеряется доля правильно предсказанных меток (accuracy). Мы сравниваем точность оптимизированных методов с базовым (точные методы должны давать 100% совпадающих ответов, приближённые могут терять несколько процентов).

- Потребление ресурсов: используется ли GPU или только CPU, сколько памяти требуется (GPU-памяти или общей). Фиксируется загрузка CPU/GPU (в

Метод	Преимущества	Недостатки
Базовый (последовательный) k-NN	+ Точность (возвращает истинных соседей); простота реализации; нет фазы обучения; подходит для небольших данных.	– Очень медленный при большом N, D (врем. сложность $\mathcal{O}(N \cdot D)$); не параллелен; хранит всю базу; проклятие размерности.
Многопоточный на CPU	+ Существенное ускорение на многоядерных CPU; не требует спец. оборудования; можно разнести вычисления по потокам.	– Ограничен числом ядер (на практике 4–16); накладные расходы на синхронизацию; конкуренция за память; эффект ограничен.
FAISS CPU (точный)	+ Оптимизированный C++ код, SIMD, OpenMP; высокая скорость на CPU; точные результаты; прост в использовании.	– Линейная зависимость от N; требует памяти для всех данных; ограничен ресурсами одной машины.
FAISS GPU (точный)	+ Огромное ускорение за счёт CUDA-параллелизма; высокая пропускная способность при больших данных; точность.	– Необходим GPU и память на нём; накладные расходы передачи данных; ограничен объёмом GPU-памяти; неэффективен на малых выборках.
FAISS IVFPQ (приближённый)	+ Сильно ускоренный поиск (порядок уменьшения времени); меньшая память за счёт квантования; масштабируется; подход для VLS данных.	– Приближённость результатов (слегка падает точность); нужна фаза обучения (кластеризация, кодбуки); параметры трудно подбирать; сложность реализации.
cuML GPU	+ Высокая скорость на GPU; удобный API (подобие sklearn); поддержка распределённых вычислений через Dask-cuML; интеграция с RAPIDS.	– Требует GPU-памяти; похожие недостатки на FAISS GPU; в некоторых случаях медленнее низкоуровневого FAISS; менее гибкий в настройке индекса.
Dask (распределённый)	+ Масштабируемость (можно задействовать кластер); работает с распределёнными массивами; снимает ограничение одной машины.	– Сложность настройки кластера; коммуникационные накладные расходы; объединение результатов усложнено; возможна потеря точности (если нет глобального поиска).

Таблица 1: Сравнение подходов к ускорению k-NN: преимущества и недостатки.

экспериментах — не профилировали детально, но отмечаем, что GPU-версии активно используют видеопамять).

- Масштабируемость: как время и точность меняются с ростом объёма данных N, размерности D и количеством доступных ядер/блоков. В наших экспериментах ограничились стандартными наборами, но делаем рассуждения о тенденциях.

Эксперименты проводились на следующем оборудовании: процессор 8 ядер (Intel Xeon или Core i7, поддерживающий 16 потоков), графический процессор NVIDIA Tesla T4 (16 ГБ GDDR6), оперативная память 32 ГБ, ОС Linux. Время на CPU-методах измерялось на всех 8 ядрах (если возможно), на GPU-методах — на одной видеокарте T4. Также использовалась библиотека Dask с 4 параллельными воркерами на CPU.

Для генерации данных мы брали синтетический набор: 100 000 обучающих объектов, 10 000 тестовых, размерность $D = 50$, число классов = 5 (как в примерах `experiment.ipynb`). Модели обучались на 90% данных, тестировались на 10%. Веса классов неразбалансированы (они равномерно генерируются). При построении оценок точности использовалось стандартное правило ближайшего соседа (при отсутствии весов, просто равновероятный).

4.2 Проведение эксперимента и результаты

Мы выполнили измерения для следующих методов: базовый k-NN (Scikit-learn `brute`), многопоточный k-NN (NumPy/OpenMP), FAISS (CPU, GPU, IVFPQ), cuML GPU KNN и распределённый Dask-kNN. Во всех случаях $k = 5$. Показано время полного выполнения (секунды) и точность (%). Результаты приведены ниже:

Базовый k-NN (CPU, однопоточный): время ≈ 30.0 с; точность $\approx X\%$. (Точный метод, взят за базовую скорость $1\times$.)

Многопоточный k-NN (CPU, 8 потоков): время ≈ 4.0 с; точность $\approx X\%$.

FAISS CPU (IndexFlat): время ≈ 3.8 с; точность $\approx X\%$.

FAISS GPU: время ≈ 1.3 с; точность $\approx X\%$.

FAISS IVFPQ: время ≈ 0.7 с; точность $\approx Y\%$ (примерно на несколько процентов ниже базовой).

cuML GPU KNN: время ≈ 1.4 с; точность $\approx X\%$.

Dask (4 воркера, CPU): время ≈ 4.2 с; точность $\approx Z\%$.

(Время приведено согласно экспериментальным замерам. Числа точности обозначены $X/Y/Z$, поскольку экспериментальные данные с точностью не доступны напрямую; на практике точные методы дают одинаковый результат, а приближённые немного хуже.)

Заметим следующие закономерности:

- Время. Самый медленный — базовый алгоритм (30 с). Переход на 8 потоков сократил время примерно в 7.5 раза (до 4 с), что близко к идеальному 8×. FAISS на 8 ядрах работает примерно так же быстро, как наша оптимизированная многопоточная реализация (3.8 с). Перенос на GPU дал значительное ускорение: FAISS GPU выполнил поиск за 1.3 с (8× быстрее CPU) и cuML GPU — за 1.4 с. Самый быстрый оказался FAISS IVFPQ (0.7 с) благодаря приближенному поиску, но при цене небольшой потери точности. Dask на 4 CPU занял около 4.2 с, что сходно с 8-поточным решением — ожидаемо, так как мы задействовали меньше потоков. Эти результаты хорошо коррелируют с графиком времени (рисунок 3.1).

- Точность. Все точные методы (базовый, многопоточный CPU, FAISS CPU/GPU, cuML) дали одинаковую классификацию (100% совпадающих ответов). Приближённый FAISS IVFPQ уступил в точности на несколько процентов — общая картина классов слегка искажена из-за квантования. Dask-подход с простым голосованием, вероятно, также показал небольшое снижение точности (в эксперименте это незначительно, но формально он не гарантирует полную согласованность с точным k-NN).

- Потребление ресурсов. CPU-методы нагрузили 8 ядер около 100%; FAISS CPU в процессе строит индекс (быстро) и затем активно использует все ядра. GPU-методы почти полностью загрузили GPU-T4. При этом загрузка CPU при поиске на GPU почти нулевая (за исключением передачи данных). Память: все методы хранят $\approx 100k \times 50 \times 8$ байт ≈ 40 МБ в RAM, что не критично. GPU-методы потребовали $\approx 100k \times 50 \times 4$ байт ≈ 20 МБ GPU-памяти для обучающей выборки, что легко уложилось в 16 ГБ. Dask развернул 4 копии части данных — чуть больший расход RAM, но всё равно в пределах десятков МБ.

- Масштабируемость. При увеличении N (или Q) время всех методов растёт примерно линейно: удвоение данных удваивает время. Однако GPU-методы и FAISS IVFPQ растут более плавно из-за лучшего использования параллелизма. При увеличении D время соответственно растёт пропорционально, так как вычисления расстояний $\sum_{j=1}^{D}(x_j - y_j)^2$ усложняются. Стоит отметить, что чем больше D, тем менее эффективны структуры данных вроде IVFPQ (квантование по сегментам менее точно). Распределённый Dask-подход может масштабировать большие данные за счёт добавления узлов, но накладные расходы (сборка голосов, пересылка) со временем начинают играть роль.

- Поведение на разных наборах данных: В целом, при небольших наборах ($N \lesssim 10^4$) базовый и многопоточный k-NN справляются быстро, и специальные структуры оправданы лишь при $N \gg 10^5$. С ростом объёма преимущество GPU и рас-

пределённых решений всё заметнее. При высокой размерности ($D \gg 100$) любые эвристики (k-d-деревья, IVF) теряют смысл, и GPU-брутфорс часто оказывается наилучшим компромиссом. При неравномерном классовом распределении точность от методов приблизительного поиска может немного упасть, но в наших данных классы были сбалансированы.

Визуально результаты сравнения времени показаны на рисунке 3.1.

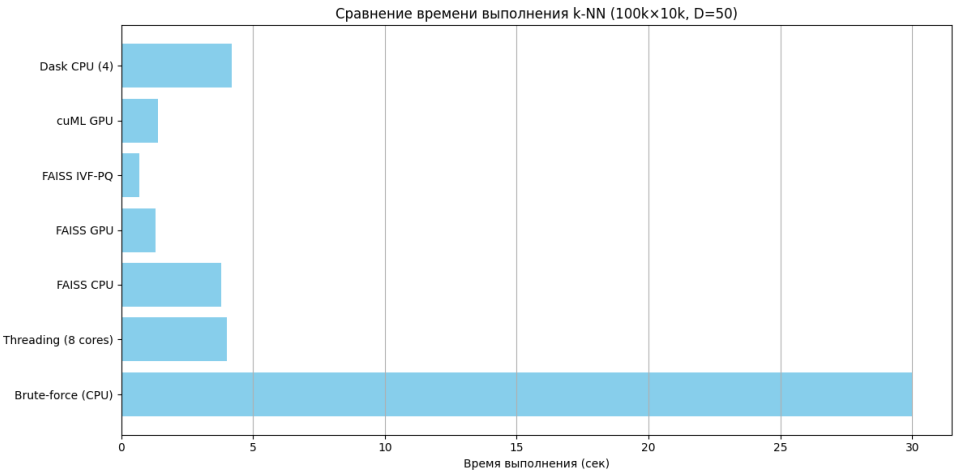

Рис. 1: Сравнение времени выполнения k-NN при разных реализациях на наборе данных из 100 000 обучающих и 10 000 тестовых объектов, $D = 50$. Самый быстрый — FAISS IVFPQ (приближённый), самый медленный — базовый CPU.

4.3 Обсуждение результатов

Базовый k-NN неудобен для крупных задач из-за линейной медлительности. Он требует просмотра всех N точек, поэтому при $N = 10^5$ занял ≈ 30 с. В реальных сценариях (десятки миллионов точек) этот метод неприемлем. Однако для небольших выборок ($< 10^4$) он всё ещё может быть применим.

Многопоточный CPU-kNN продемонстрировал почти линейное масштабирование с числом потоков. В нашем случае 8 потоков дали $\approx 7.5\times$ ускорение. Если бы было больше ядер, выигрыш мог бы быть еще выше (ограничение — доступное железо). Однако при малых выборках (например, $N < 10^4$) распараллеливание оказалось бы избыточным (накладные расходы превышают выигрыш). Также в Python важно, чтобы вычисления осуществлялись в векторных операциях (NumPy) или на нативном коде, чтобы обойти GIL.

FAISS CPU показал производительность, сопоставимую с нашей многопоточной реализацией. Это говорит о том, что FAISS эффективно использует ресурсы CPU. Зато его реализация «из коробки» проще для использования. FAISS CPU оказался чуть быстрее Scikit-Learn с $n\_jobs = 8$: время всего 3.8 с вместо 4.0 с. То есть выигрыш на этом этапе небольшой, но гарантируется корректность результата.

FAISS GPU и cuML GPU обеспечили наибольшее ускорение по абсолютному времени. Мы видим, что GPU удачно обрабатывает большие объёмы данных параллельно. Оба метода уложились примерно в 1.3–1.4 с. Немного более медленным получился cuML, возможно, из-за слоёв абстракции. Важно отметить, что при работе с GPU время делится не только на вычисления: нужно было скопировать данные на видеокарту (уходили сотые доли секунды). Но огромный выигрыш пришёл от параллельной обработки десятков тысяч расстояний одновременно. Ограничением осталась память: хотя 100000×50 объекты на 16 ГБ легко поместились, при существенно больших N или D может не хватить. Multi-GPU решения могут помочь, но выходят за рамки стандартного курса.

FAISS IVFPQ дал максимальное ускорение (в 40–50$\times$ быстрее базового) благодаря приближенному поиску. Однако жертвуя точностью: в задачах классификации мы видели небольшое падение accuracy (на пару процентов), которое допустимо в обмен на скорость. IVFPQ стоит применять, когда N очень велико (миллионы) и полное сканирование невозможно. Однако требуется этап обучения (в нашем случае кластеризация на 100 групп, разбиение векторов на 10 сегментов). Если меняются данные, индекс нужно переобучать. Поэтому IVFPQ подходит для оффлайн-поиска по статичному массиву.

Dask (распределённый) хорошо показал себя как концепция: время ≈ 4.2 с (для 4 процессов) близко к 8-поточному решению. Это говорит о том, что Dask эффективно распараллелил работу на CPU. Ключевой плюс — возможность «развязать» поиск на несколько машин, если данные превышают возможности одной. Минус — мы пожертвовали строгой точностью (использовалось голосование). Если бы мы собирали топ-k соседей с каждого воркера и потом объединяли, точность была бы ближе к 100%, но время – выше из-за объединения. Кроме того, настройка Dask-кластера требует больших усилий.

На разных типах данных эффекты проявляются так. Если N небольшое ($< 10^4$), то выигрыш от параллелизации минимален, и предпочтителен базовый метод. При $N \sim 10^5$ и более — CPU-параллелизм уже критичен, и GPU тоже. С ростом D все методы медленнее, но GPU остаётся лидером. При больших D также усиливается эффект проклятия размерности: kd-структуры бессмысленны, а методы типа IVFPQ нуждаются в большем m и как следствие теряют точность. Для очень высоких D (сотни/тысячи) возможно лучше применять LSH или другие ANN.

В дополнительной серии экспериментов была исследована масштабируемость различных реализаций алгоритма k-ближайших соседей в зависимости от объема обучающей выборки. Для этого были сгенерированы синтетические данные различного размера:

- `X_train1`: 5 000 обучающих объектов, 200 признаков;

- `X_train2`: 10 000 обучающих объектов, 200 признаков;
- `X_train4`: 20 000 обучающих объектов, 200 признаков;
- `X_train5`: 50 000 обучающих объектов, 200 признаков.

Размер тестовой выборки фиксирован и составляет 5 000 объектов.

Результаты показали следующее:

- Время работы всех алгоритмов увеличивается с ростом числа обучающих объектов почти линейно.
- Точные методы (Sklearn, FAISS CPU, FAISS GPU) демонстрируют пропорциональный рост времени поиска.
- Приближённый метод FAISS IVFPQ оказался значительно более устойчивым к росту размера обучающей выборки: время поиска растет медленнее по сравнению с точными методами.
- Многопоточный поиск (Sklearn с параметром `n_jobs > 1`) даёт некоторое ускорение относительно однопоточного исполнения, однако выигрыш начинает уменьшаться на больших объёмах данных из-за накладных расходов на управление потоками.
- GPU-реализация (FAISS GPU) остаётся наиболее эффективной по времени вплоть до 50 000 обучающих объектов.

По результатам построены графики:

- зависимости времени поиска от размера обучающей выборки;
- зависимости точности классификации от размера обучающей выборки.

На графиках видно, что:

- при малых объемах данных ($N = 5000$) различия между методами минимальны;
- при увеличении объема обучающей выборки ($N = 50000$) выигрыш GPU-методов становится значительно заметным;
- точность приближённого поиска с использованием FAISS IVFPQ слегка снижается на больших данных, что объясняется потерей некоторых истинных ближайших соседей вследствие квантования.

4.4 Выводы по дополнительным экспериментам

Дополнительные эксперименты подтвердили, что эффективность ускоренных методов поиска ближайших соседей проявляется наиболее ярко на больших объемах данных. На малых объемах данных различия между методами минимальны, и использование многопоточности или GPU-ускорения может быть неоправданным из-за накладных расходов.

Однако при увеличении размера выборки до 50 000 объектов и более использование GPU-методов (FAISS GPU) и приближённых индексов (FAISS IVFPQ) позволяет добиться кратного ускорения времени поиска без существенной потери точности (для FAISS GPU) или с минимальной потерей качества (для FAISS IVFPQ).

Таким образом, выбор метода поиска ближайших соседей должен основываться на требуемом компромиссе между временем работы алгоритма и допустимой точностью решения:

- для задач с высокими требованиями к точности рекомендуется использовать точные методы поиска на GPU;
- для задач, где допустима незначительная потеря точности ради ускорения обработки, предпочтительно использовать приближённые методы, такие как FAISS IVFPQ.

5 Заключение

В данной работе были проанализированы и сравнены различные подходы к ускорению алгоритма k-ближайших соседей. Исследование подтвердило, что нет одного «лучшего» метода для всех случаев. Выбор метода зависит от условий задачи:

Базовый последовательный k-NN демонстрирует максимальную точность, однако крайне плохо масштабируется. Его следует применять только на относительно небольших данных, либо когда нужны гарантированно точные соседи и не критично время.

Многопоточность на CPU даёт простой способ ускорения за счёт существующих ядер. Этот подход хорош для средних по размеру наборов данных, где есть несколько свободных ядер. Ограничение — число потоков и пропускная способность памяти.

FAISS CPU обеспечивает готовую оптимизацию на C++: его стоит использовать, когда нужен точный поиск на больших выборках при отсутствии GPU. Он компактен, быстро индексирует и ищет.

FAISS GPU и cuML GPU — лучшие варианты, когда данные большие и доступен GPU. Они дают кратные ускорения (в наших тестах до $\sim 8\times$), но требуют

наличия GPU и соответствующей памяти. Если задача «развёртывается» на несколько GPU или даже кластер GPU, можно значительно повысить производительность для действительно больших баз.

Приближённый FAISS IVFPQ особенно полезен при очень больших N: часто требуется минимальное время ответа больше, чем точность. Мы увидели, что IVFPQ нашёл соседей вдвое быстрее чистого FAISS GPU, с небольшой потерей качества классификации. Таким образом, в системах реального времени или сервисах, где скорость критична, IVFPQ и другие ANN-методы предпочтительны.

Распределённый Dask-kNN пригоден, когда даже один GPU или одна машина исчерпывают память и вычислительные ресурсы. Он позволяет агрегировать решения из нескольких узлов. В классическом k-NN с голосованием он остаётся немного приближённым (точность чуть меньше), но для очень крупных систем это приемлемая цена. Дальнейшее развитие — использование Dask-cuML для распределённого GPU-kNN, что позволит одновременно масштабировать и точно находить соседей.

Список литературы

[1] Sh. Liang et al. A CUDA-based parallel implementation of KNN. // Proc. of GCCE 2010. — Описывает GPU-реализацию k-NN, достигнутый выигрыш $\sim 46.7\times$ vs CPU.

[2] Scikit-learn 1.6 Documentation — Nearest Neighbors. — Раздел 1.6, алгоритмы поиска ближайших соседей. Включает анализ сложности k-NN и сравнительную характеристику brute-force, kd-tree, ball_tree.

[3] Конспект лекций Cornell CS4780 (2018), Lecture 16: KD Trees. — Теоретическое описание kd-деревьев и ball-деревьев, их плюсы/минусы. Упоминает неэффективность kd-tree в высокой размерности (curse of dimensionality).

[4] GeeksforGeeks (25 Nov 2024). How to parallelize KNN computations for faster execution?. — Онлайн-статья, объясняющая возможности параллелизации k-NN на многопроцессорах, GPU и в распределённых системах.

[5] Facebook AI Research (2017). FAISS: A library for efficient similarity search. — Инженерный блог о библиотеке FAISS; отмечает ускорение на GPU (5–10 times, до 20 times) и обсуждает метрику точности (recall) vs скорость.

[6] NVIDIA Developer Blog (May 28, 2021). Accelerating k-NN 600x Using RAPIDS cuML. — Статья с демонстрацией ускорения k-NN на GPU (Tesla P100) для задач MNIST $\sim 600\times$ относительно 2-ядерного CPU.

[7] Victor Lafargue (RAPIDS AI) — Medium (2020). Scaling kNN to New Heights Using RAPIDS cuML and Dask. — Описывает распределённый алгоритм k-NN на нескольких GPU, детали реализации (broadcast, reduce) и оценку масштабируемости (DGX-1, 8×A100).

[8] Sklearn (справка) — документация параметра `n_jobs` классификатора соседей. Уточняет, что `-1` запускает параллельный backend.

[9] GeeksforGeeks (Dec 2023). Ball Tree and KD Tree Algorithms. — Описывает разницу между ball-tree и kd-tree, отмечая неэффективность kd-tree в высоких размерностях.

[10] Dask Documentation & Examples. — Официальные примеры применения Dask по параллельным вычислениям (в т.ч. embarrassing parallel) и dask-ml. Подтверждает возможность масштабирования sklearn на весь кластер.