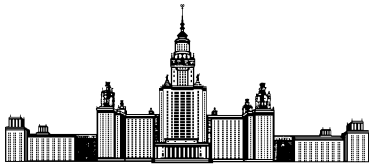
Московский государственный университет имени М. В. Ломоносова



Факультет Вычислительной Математики и Кибернетики

Кафедра Математических Методов Прогнозирования

КУРСОВАЯ РАБОТА СТУДЕНТА 317 ГРУППЫ

«Сравнительный анализ методов быстрого поиска ближайших соседей»

Выполнил:

студент 3 курса 317 группы

Федоров Илья Сергеевич

Научный руководитель:

д.ф-м.н., профессор

Дьяконов Александр Геннадьевич

Москва, 2020

Содержание

Аннотация

В данной работе приводится подробный обзор классических (древовидные структуры данных, LSH) и наиболее современных (Product Quantization, HNSW) подходов к быстрому поиску ближайших соседей, включая так называемые «приближенные методы». Проведен сравнительный анализ данных алгоритмов, для каждого из них указаны преимущества и недостатки. Установлен практически значимый порог размерности признакового пространства, при котором древовидные структуры перестают быть эффективными. Проведены эксперименты с библиотекой FAISS с использованием GPU.

1 Введение

Одним из наиболее простых и естественных методов машинного обучения является метод ближайшего соседа. Имея набор данных, представленных в виде точек в некотором многомерном пространстве, целевая величина (будь то класс или вещественное число) прогнозируется по значениям отклика на k ближайших к запросу точках из исходного набора данных. В то время как существует множество различных подходов к усреднению данных k значений, наиболее вычислительно затратной частью алгоритмов подобного типа является именно поиск ближайших соседей. Действительно, в современных задачах объемы данных достигают колоссальных размеров, что делает алгоритмы, основанные на полном переборе, неэффективными. Задача поиска ближайших к запросу точек в некотором наборе данных встречается не только в задачах прогнозирования. Примерами приложений также могут служить задачи поиска дубликатов в больших объемах данных (или «почти» дубликатов), поиска похожих изображений и текстов. Целью данной работы является обзор современных подходов к решению задачи поиска ближайших соседей и её вариаций, а также сравнительный анализ эффективности тех или иных методов её решения в зависимости от особенностей пространства, в котором расположены данных. В исследовании представлены как классические подходы, основанные на формировании некоторых дополнительных структур данных, так и наиболее современные «приближенные» методы.

1.1 Определения и обозначения

Формализуем постановку задачи. Основным объектом нашего изучения будет пространство признаков вместе с функцией расстояния $\mathbb{X} = (\mathbb{R}^n, d)$. Важно заметить, что функция d в приложениях довольно часто может не удовлетворять формальному определению метрики, однако даже в этом случае в данной работе подобные функции, допуская некоторую вольность, будут называться метриками. К примеру, широко используемое в анализе текстов косинусное расстояние $d(x, y) = 1 - \frac{\langle x,y \rangle}{\|x\|\|y\|}$ не удовлетворяет неравенству треугольника. В данной работе в большинстве случаев будет использоваться евклидовая метрика и описанное выше косинусное расстояние.

Будем обозначать $X \in \mathbb{R}^{l \times n}$ матрицу для выборки точек из $\mathbb{X}$, где строки соответствуют объектам, а столбцы признаками. Формально задача поиска k ближайших соседей ставится следующим образом: имея множество объектов X из пространства $\mathbb{X}$ и запрос $q \in \mathbb{X}$, нужно найти в X k ближайших к q точек по метрике d. Более подробно, если посчитать расстояния между q и всеми объектами из X, а потом расположить их в отсортированном порядке

$$d(x_{i_1}, q) \leq d(x_{i_2}, q) \leq \ldots \leq d(x_{i_l}, q),$$

то алгоритм должен выдать k объектов с минимальными расстояниями: $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$.

Как мы увидим далее, большинство современных методов быстрого поиска ближайших соседей на самом деле решают описанную выше задачу с некоторыми ослаблениями, что в сущности приводит к задаче «приближенного» поиска ближайших соседей. Эта задача не имеет общепризнанной формальной постановки, разные авторы могут по-разному понимать её. К примеру, довольно распространенной постановкой задачи является следующая формулировка: имея набор точек X из $\mathbb{X}$, запрос $q \in \mathbb{X}$, а также параметр $c \geq 1$, если существует точка $x \in X$, такая что $d(x, q) \leq r$, алгоритм должен вернуть точку $x^* \in X$, такую что $d(x^*, q) \leq cr$. В дальнейшем при описании конкретных методов, мы будем уточнять, какую именно задачу приближенного поиска ближайших соседей они решают.

2 Обзор существующих методов поиска ближайших соседей

2.1 Прямое вычисление матрицы попарных расстояний

Самым простым и распространенным способом поиска ближайших соседей является прямой перебор. Имея набор данных X и запрос q, мы вычисляем расстояния между каждым объектом $x \in X$ и q. После этого полученные расстояния сортируются, и алгоритм выдает k объектов из X, имеющих наименьшие расстояния до q.

Оценим вычислительную сложность такого алгоритма. Заметим, что для вычисления евклидового или косинусного расстояния между двумя объектами размерности

n нужно совершить порядка n операций. Отсюда получаем, что сложность вычисления расстояний $\mathcal{O}(nl)$ (в наших обозначениях l – число объектов). Далее требуется отсортировать полученный массив, что займет $\mathcal{O}(l \log(l))$ операций. Наконец, останется совершить $\mathcal{O}(k)$ операций для выдачи результата. Учитывая, что $k \leq l$, получим итоговую сложность: $\mathcal{O}(nl + l \log(l))$. Однако на практике данную сложность можно улучшить. Дело в том, что обычно $k \ll l$, а значит бóльшая часть информации из отсортированного массива нам не нужна. Поэтому вместо сортировки можно использовать более эффективные алгоритмы для поиска k наименьших чисел в массиве. Например, это можно сделать с помощью структуры данных под названием куча. Имея массив из l элементов, можно построить кучу за $\mathcal{O}(l)$, а далее вытащить из неё k минимальных элементов за $\mathcal{O}(\log(l))$ каждый. Получаем сложность поиска k минимальных чисел в массиве $\mathcal{O}(l + k \log(l))$. Учитывая, что $k \ll l$, вторым слагаемым можно пренебречь, и оценить итоговую сложность всего алгоритма в $\mathcal{O}(nl)$.

Данный алгоритм вполне эффективен и применим, если объем данных и запросов не слишком велик. К примеру, в соревнованиях по машинному обучению довольно часто встречаются наборы данных размером порядка $10^5 - 10^6$ размерности около 100. Имея обучающую выборку размером 10^6 размерности 100, а также тестовую выборку размером 2×10^6, алгоритм прямого перебора будет работать около 13 часов на 8-ядерном процессоре. Это вполне приемлимо, если требуется решить задачу для конкретной тестовой выборки. Однако данный алгоритм обладает рядом существенных недостатков. Во-первых, если поиск ближайших соседей проводится для решения какой-то задачи машинного обучения, то все вычисления проводятся непосредственно в момент предсказания целевой величины. Поскольку алгоритм никак не обучается, его ценность с точки зрения производительности существенно падает, так как при каждом предсказании производится набор вычислений, сопоставимый по объему с обучением какого-то другого алгоритма, который, обучившись лишь единожды, может очень быстро выдавать ответы (например, линейная или логистическая регрессия). Во-вторых, если данных становится действительно много (скажем, больше 10^{10}), то для хоть сколько-то большой тестовой выборки уже требуется колоссальное количество времени для вычислений. Современные компьютеры могут выполнять примерно 10^8 операций в секунду, поэтому для обучающей выборки раз-

мером 10^{10} (вполне реальная цифра для больших компаний), тестовой выборки размером 10^3, размерности пространства 10 такое вычисление займет $\frac{10^{10} \times 10^3 \times 10}{10^8}$ секунд $\approx$ 278 часов $\approx$ 12 дней. Безусловно, эти цифры можно сократить, используя специализированные архитектуры компьютеров и многопоточность, однако представленные два недостатка данного алгоритма в совокупности ставят под сомнение его использование в промышленных масштабах.

2.2 Древовидные структуры данных

Большой класс алгоритмов для быстрого поиска ближайших соседов основан на идее разбиения признакового пространства на области, которые объединяются в различные структуры данных, позволяющие выполнять поиск ближайших соседей для новых запросов быстрее.

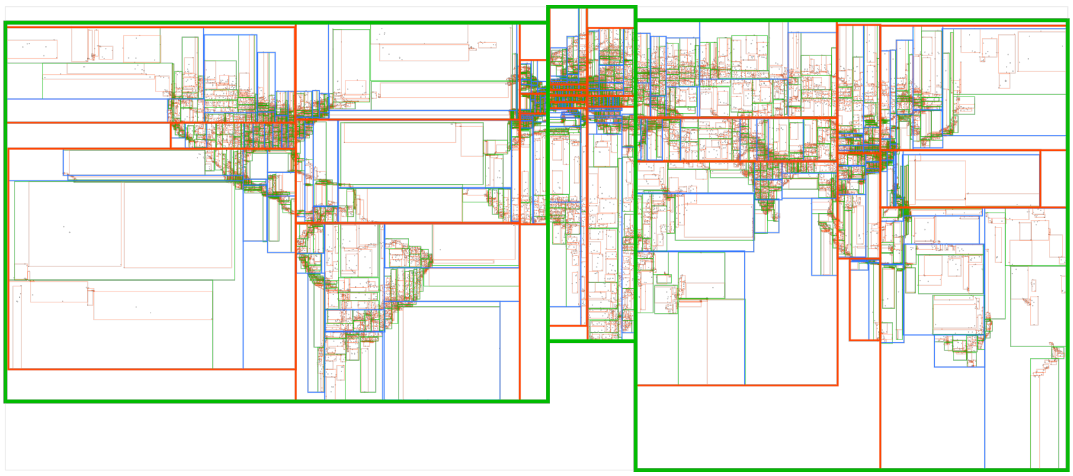


Рис. 1: Пример разбиения данных на плоскости с помощью R-Tree

Большинство алгоритмов из данного класса сначала осуществляют предварительную обработку исходного набора данных, строя древовидные структуры, состоящие из областей разбиений. Далее, при поступлении очередного запроса, используется информация, полученная на первом этапе. Таким образом, важное отличие таких алгоритмов от метода прямого перебора заключается в том, что теперь мы имеем некоторый разделенный интерфейс, состоящий из двух методов: построение дерева и запрос. Это позволяет нам по отдельности оценивать вычислительные сложности для этих двух операций. Это может иметь важную роль, к примеру, если в решае-

мой практической задаче не так существенно, сколько займет первичная обработка данных, но требуется высокая скорость обработки новых запросов.

Перечислим наиболее популярные алгоритмы поиска ближайших соседей, основанные на древовидных структурах данных:

- KD - Tree
- BSP - Tree
- Ball - Tree
- Quadtree
- R - Tree
- B - Tree

Стоит отметить, что некоторые из этих структур данных предназначенны для работы с данными какой-то фиксированной размерности. Например, B - Tree работает для одномерных данных, а Quadtree – для двумерных. На практике наиболее часто встречаются алгоритмы KD - Tree и Ball - Tree, поскольку они включены в самые известные библиотеки для машинного обучения. Рассмотрим в качестве примера подробную реализацию KD - Tree.

Приведем возможную реализацию KD - Tree на псевдокоде. Сначала рассмотрим операцию построения дерева.

1: **procedure** BUILDNODE(Ω)
2: **if** $|\Omega| \leq n_{min}$ **then**
3: self.objects = Ω
4: **else**
5: self.pivot_feature_idx = $\underset{1 \leq i \leq n}{\operatorname{argmax}} \mathbb{D}[x^i]$
6: self.threshold = median($x^{self.pivot\_feature\_idx}$)
7: self.left = BuildNode($\{\ x_k \in \Omega \mid x_k^{self.pivot\_feature\_idx} < self.threshold\ \}$)
8: self.right = BuildNode($\{\ x_k \in \Omega \mid x_k^{self.pivot\_feature\_idx} \geq self.threshold\ \}$)
9: **end if**
10: **end procedure**

Таким образом, метод разбивает набор данных по медиане признака с наибольшей дисперсией на две части и рекурсивно применяется к каждой из них. Полученные деревья считаются сыновьями данной вершины. Рекурсия прекращается, когда набор данных становится достаточно небольшим по размеру. Отметим, что различных

источниках можно найти немного отличающиеся реализации данной операции, однако приведенный вариант хорош тем, что приводит к достаточно сбалансированному дереву.

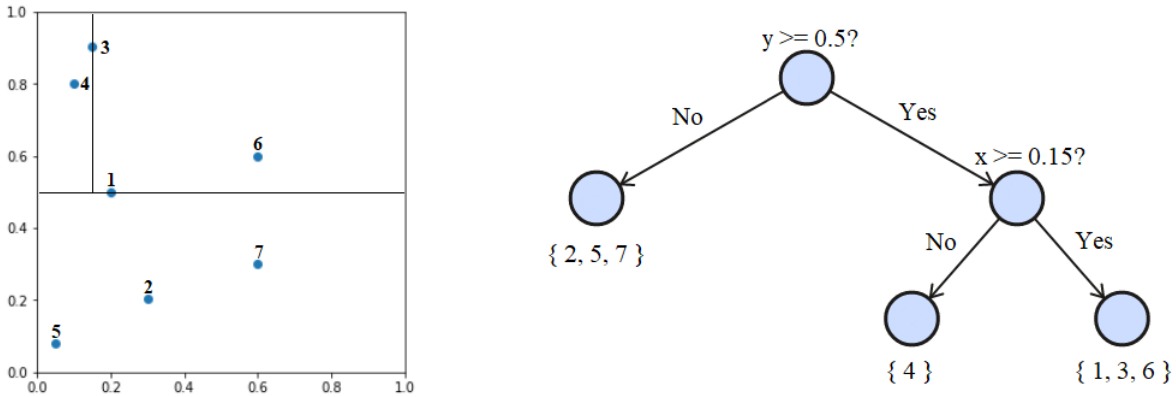


Рис. 2: Пример построенного KD-Tree, $n_{min} = 3$

Операция запроса в KD - деревьях работает по следующему принципу: сначала устанавливается лист, соответствующей области разбиения, содержащей запрос. Вычисляется ближайший к запросу сосед среди точек в этом листе. Далее начинается восходящий по структуре дерева поиск ближайших соседей в соседних областях. А именно, если расстояние от запроса к прямоугольнику, который является другим сыном родителя листа, в котором находится запрос, меньше, чем расстояние до текущего ближайшего соседа, то алгоритм проверяет эту область на наличие ещё более близких соседей. Далее алгоритм поднимается на одну вершину вверх по дереву и выполняет те же действия. На рисунке 3 представлена иллюстрация к описанному алгоритму. Красная и синияя области – это листья в KD - дереве, черная область – их родитель, зеленые точки – исходный набор данных, черная точка - запрос. Ближайшей точкой в красной области является точка под номером 1, однако точка 3 находится ближе к запросу, поэтому алгоритм проверяет «братские» области к тем, в которых расположен запрос.

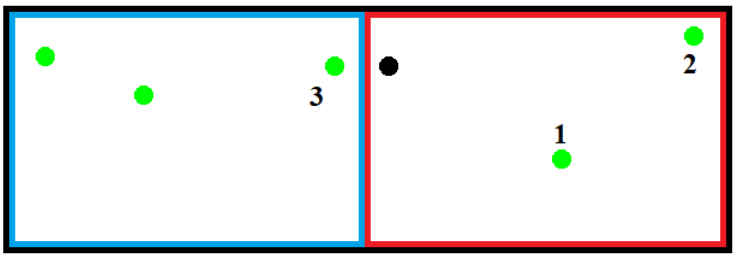


Рис. 3: Иллюстрация к операции запроса в KD - дереве

Приведем реализацию операциии запроса на псевдокоде. Данный код был взят из [1]

```
1: procedure MakeQuery(Tree, Query)
2:     // Поиск листа
3:     CURRENT_NODE = root node of Tree
4:     while CURRENT_NODE is not leaf node do
5:         pivot = Query^{CURRENT_NODE.pivot_feature_idx}
6:         μ = CURRENT_NODE.threshold
7:         if pivot ≤ μ then
8:             CURRENT_NODE = CURRENT_NODE.left
9:         else
10:            CURRENT_NODE = CURRENT_NODE.right
11:        end if
12:        ascendant_search(CURRENT_NODE)
13:    end while
14: end procedure
```

```
1: procedure ASCENDANT_SEARCH(CURRENT_NODE)
2:     // Восходящий поиск
3:     mark CURRENT_NODE as checked
4:     while not all nodes of Tree checked do
5:         SIBLING_NODE = brother node of CURRENT_NODE
6:         RECT_DIST = distance from Query to rectangle, associated with
   SIBLING_NODE
7:         if RECT_DIST ≥ NN_DIST then
8:             mark SIBLING_NODE and all its descendants as checked
9:         else
10:            NN,NN_DIST = check_tree(SIBLING_NODE)
11:        end if
12:        mark SIBLING_NODE and PARRENT_NODE as checked
13:        set CURRENT_NODE to PARENT_NODE
14:    end while
15: end procedure
```

```
1: procedure CHECK_TREE(CurrentNode, x, NN, NN_Dist)
2:     if CURRENT_NODE is leaf node then
3:         CURRENT_NN = closest object to x from all objects associated with
   CURRENT_NODE
4:         CURRENT_NN_DIST = distance from x to CURRENT_NN
5:         if CURRENT_NN_DIST < NN_DIST then
6:             NN = CURRENT_NN
7:             NN_DIST = CURRENT_NN_DIST
8:         end if
9:         return NN, NN_DIST
10:    else
11:        for each node NODE from children of CURRENT_NODE do
12:            DIST = distance from x to rectangle of CURRENT_NODE
13:            if NN_DIST ≥ DIST then
14:                mark NODE and all its descendants as checked
15:            else
16:                NN,NN_DIST = check_tree(NODE,x,NN,NN_DIST)
17:            end if
18:        end for
19:    end if
20: end procedure
```

2.3 Неэффективность деревьев в пространствах высокой размерности

В предыдущей секции были приведены примеры древовидных структур данных, которые разбивают признаковое пространство на некоторые области, с помощью его ускоряется поиск ближайших соседей. Стоит отметить, что таких структур данных на практике встречается существует огромное количество. К примеру, существующие методы можно оптимизировать, если выбирать правило для разбиения пространства не по одному конкретному признаку, а в направлении первой главной компоненты,

что приведет к более сбалансированным деревьям и ускорению операции запроса [6]. Однако было установлено, что все подобные структуры данных перестают давать преимущство в скорости выполнения запроса с ростом размерности признакового пространства. Более того, этот вопрос был детально исследован, а предыдущее утверждение было строго доказано в [13]. Приведем некоторые ключевые наблюдения из данной статьи (которые в совокупности принято называть проклятием размерности), а также основные результаты.

Наблюдение 1 (Число разбиений). *Наиболее простая схема разбиения пространства делит его по каждой размерности на две части. Имея d-мерное пространство, будет существовать 2^d областей разбиения. Если $d \leq 10$ и число объектов имеет порядок около 10^6, то в разбиениях будет смысл. Однако если d растет, скажем, до 100, то число разбиений будет порядка 10^{30} для числа объектов 10^6, то подавляющее большинство областей будет пустыми.*

Наблюдение 2 (Разреженность данных в пространствах высокой размерности). *Рассмотрим d мерный единичный гиперкуб Ω в признаковом пространстве. Рассмотрим запрос получения данных из гиперкуба со стороной l. Тогда вероятность того, что равномерно распределенная по единичную кубу точка попадет в наш запрос равна*

$$P^d[s] = s^d$$

При $d = 100$, $l = 0.95$ эта вероятность будет равна 0.59%. Отметим, что меньший гиперкуб может быть расположен где угодно в Ω. Отсюда можно сделать вывод, что нам сложно найти точки в Ω, пространство является разреженным.

Наблюдение 3 (Сферические запросы). *Рассмотрим наибольший сферический запрос $sp^d(Q, 0.5)$, помещающийся в признаковое пространство с центром Q. Вероятность того, что произвольная точка R лежит внутри этого запроса определяется отношением объемов:*

$$P[R \in sp^d(Q, \frac{1}{2})] = \frac{Vol(sp^d(Q, \frac{1}{2}))}{Vol(\Omega)} = \frac{\sqrt{\pi^d}\left(\frac{1}{2}\right)^d}{\Gamma(\frac{d}{2}+1)}$$

Если d является четным числом, то это выражение можно упростить до

$$P[R \in sp^d(Q, \frac{1}{2})] = \frac{\sqrt{\pi^d}\left(\frac{1}{2}\right)!}{\left(\frac{d}{2}\right)!}$$

Примеры значений этой вероятности приведены во втором столбце таблицы 1.

Наблюдение 4 (Экспоненциальный рост набора данных)**.** *Из значения вероятности из наблюдения 3 можно получить размер набора данных, который необходим, чтобы хотя бы одна точка в среднем попадала в запрос:*

$$N(d) = \frac{\left(\frac{d}{2}\right)}{\sqrt{\pi^d}\left(\frac{1}{2}\right)^d}$$

Некоторые значения этого количества в зависимости от d приведены в третьем столбцы таблицы 1.

d	$P[R \in sp^d(Q, 0.5)]$	$N(d)$
2	0.785	1.273
4	0.308	3.242
10	0.002	401.5
20	2.461×10^{-8}	40631627
40	3.278×10^{-21}	3.050×10^{20}
100	1.868×10^{-70}	5.353×10^{69}

Таблица 1: Проклятие размерности

Исходы из этих наблюдений, а также проведя ряд других исследований, авторы статьи приходят к следующим заключениям.

Вывод 1 (Производительность)**.** *Для каждого метода кластеризации и разбиения, существует размерность $\widetilde{d}$, такая что на наборе данных в признаковом пространстве размерности $d > \widetilde{d}$ алгоритм прямого перебора работает быстрее.*

Вывод 2 (Сложность)**.** *Вычислительная сложность всех алгоритмов кластеризации и разбиения стремится к $\mathcal{O}(N)$ при увеличении размерности пространства d.*

Вывод 3 (Деградация)**.** *Для каждого метода кластеризации и разбиения, существует размерность $\widetilde{d}$, такая что на наборе данных в признаковом пространстве размерности $d > \widetilde{d}$ в среднем будут перебраны все области разбиения.*

В разделе «Вычислительные эксперименты» будет показано, что на практике алгоритмы поиска ближайших соседей перестают быть эффективными (работают столько же времени, как линейный поиск, или даже медленнее его) при d примерно равным 10.

2.4 Приближенные методы

Как было установлено в предыдущей секции, древовидные структуры данных перестают оптимизировать поиск ближайших соседей в пространствах высокой размерности. Однако на практике достаточно часто требуется работать с пространствами высокой размерности. Примерами таких задач могут служить задачи поиска похожих изображений и текстов. Весьма часто возникает необходимость поиска дубликатов среди документов в некотором наборе данных. Оказывается, что существенный прирост производительности в задаче поиска ближайших соседей можно получить, если отказаться от точного её решения и перейти к приближенному. Строго говоря, понятие «приближенное решение» не имеет общепризнанного определения, разные авторы в своих трудах могут уточнять, что именно они понимают под этим. Однако интуиция за этим стоит всегда одинаковая: имея набор данных X и запрос q, алгоритм имеет право выдавать не самого ближайшего соседа из X к q, а «почти ближайшего». Данное ослабление требований делается для существенного повышения скорости работы таких алгоритмов. Кроме того, можно также видеть и дополнительные возможности приближенных методов: к примеру, решая задачу поиска дубликатов среди текстов, мы можем получить «почти дубликаты», то есть тексты, которые немного отличаются, но в сущности являются почти одинаковыми. Рассмотрим классические и наиболее современные методы приближенного поиска ближайших соседей.

2.5 Приближенные методы: LSH

Большой класс алгоритмов приближенного поиска ближайших соседей основывается на отображении исходного признакового пространства в некоторое другое пространство, в котором проверку на схожесть выполнить проще. Такие отображения обычно называются хэш функциями, а сам процесс хэшированием. Аналогии данно-

му процессу можно найти в области обработки естественного языка: векторы-слова, полученные с помощью One-Hot кодирования, превращаются в вектора малой размерности с помощью некоторого отображения, которое проводится таким образом, чтобы выполнялся некоторый критерий. В рассматриваемой нами задаче поиска ближайших соседей требуется найти такое отображение, чтобы близкие в некотором смысле объекты имели похожие хэши, а дальние – достаточно разные (можно заметить, что данная идея является полной противоположность требований к хэшу в криптографии, ведь в этой области требуется, чтобы сходство хэшей не свидетельствовало о схожести исходных данных). Такое хэширование принято называть локально чувствительным хэширование (Locality-sensitive hashing, LSH). Этот алгоритм опирается на существование локально чувствительных хэшей. Приведем формальное определения этого понятия [2].

Определение 1. *Семейство $\mathcal{H}$ функций из пространства $\mathbb{X}$ в какое-то пространство $\widetilde{\mathbb{X}}$ называется (R, cR, P_1, P_2)-чувствительным, если $\forall p, q \in \mathbb{X}$*

- *если $\|p - q\| \leq R$, то $\mathbb{P}_{\mathcal{H}}[h(q) = h(p)] \geq P_1$*
- *если $\|p - q\| \geq cR$, то $\mathbb{P}_{\mathcal{H}}[h(q) = h(p)] \leq P_2$*

Чтобы такое семейство было полезным, логично потребовать также, чтобы выполнялось неравенство $P_1 > P_2$. Также обратим внимание, что вероятность берется по семейству функций $\mathcal{H}$ с равномерной вероятностной мерой.

Пример 1. *Рассмотрим случай, когда $\mathbb{X} = \{0, 1\}^d$ - пространство бинарных векторов размерности d, метрика – расстояние Хэмминга (число компонент, в которых два вектора различны). Возьмем в качестве $\mathcal{H}$ набор функций, представляющих собой проекции различных компонент вектора: $h_i(p) = p_i,\ i \in \overline{1, d}$. Выбирая равномерно функцию h_i из $\mathcal{H}$, мы будем получать случайную компоненту вектора p. Заметим, что данное семейство является локально-чувствительным: вероятность $\mathbb{P}_{\mathcal{H}}[h(q) = h(p)]$ равна доле совпадающих компонент векторов p и q. Отсюда $P_1 = 1 - \frac{R}{d}$, $P_2 = 1 - \frac{cR}{d}$. Поскольку параметр аппроксимации $c > 1$, то $P_1 > P_2$.*

После выбора семейства функций $\mathcal{H}$, итоговый хэш (тэг, эмбеддинг) для объекта из исходного пространства обычно получается путем конкатенации значений

нескольких случайным образом выбранных (но при этом фиксированных для данного алгоритма) функций из $\mathcal{H}$. Далее, имея хэши для точек из исходного набора данных, вектора распределяются по ячейкам (корзинкам), таким что вектора в одной корзине имеют равный хэш. При поступлении очередного запроса q, сначала вычисляется его хэш, а потом поиск ближайших соседей ведется прямым перебором среди векторов, лежащих с ним в одной корзине.

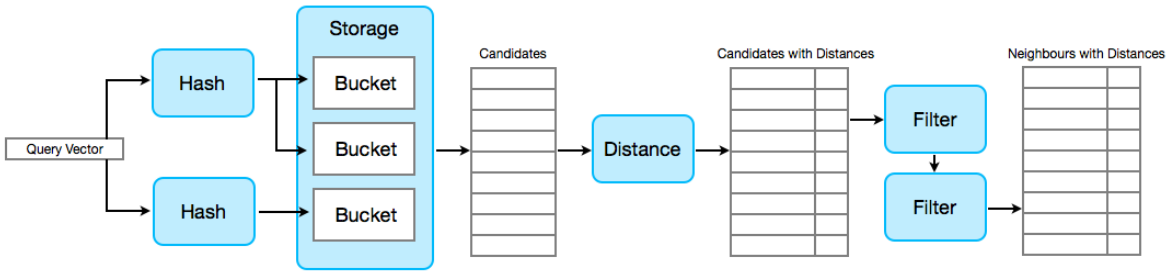


Рис. 4: Схема работы LSH

Пример 1 в сущности иллюстрирует, что обычно в качестве $\mathcal{H}$ берется набор легко вычислимых функций, удовлетворяющих определению. Чаще всего $\mathcal{H}$ выбирается исходя из метрики в пространстве $\mathbb{H}$. На практике в машинном обучении и анализе данных наиболее часто применяются евклидово расстояние и косинусное расстояние. Рассмотрим примеры семейств $\mathcal{H}$, которые используются для этих метрик [12].

Косинусное расстояние. В качестве функции $h \in \mathcal{H}$ обычно берется $h(x) = \text{sign}\langle w, x\rangle$, где w - некоторый вектор из $\mathbb{R}^n$. При этом разным функциям h соответствуют разные (но фиксированные) w, полученные случайным образом (компоненты векторов w можно получать, к примеру, из равномерного распределения). Несложно понять, почему такое семейство функций будет попадать под определение: множество $\langle w, x\rangle = 0$ представляет собой гиперплоскость в n мерном пространстве. Если две точки получили одинаковый хэш, то это значит, что они лежат по разные стороны от данной гиперплоскости. Но если угол между этими точками (мерой которого служит косинусное расстояние) достаточно мал, то вероятность того, что гиперплоскость попадет в этот небольшой промежуток, мала. Проиллюстрируем это на обычной плоскости $\mathbb{R}^2$.

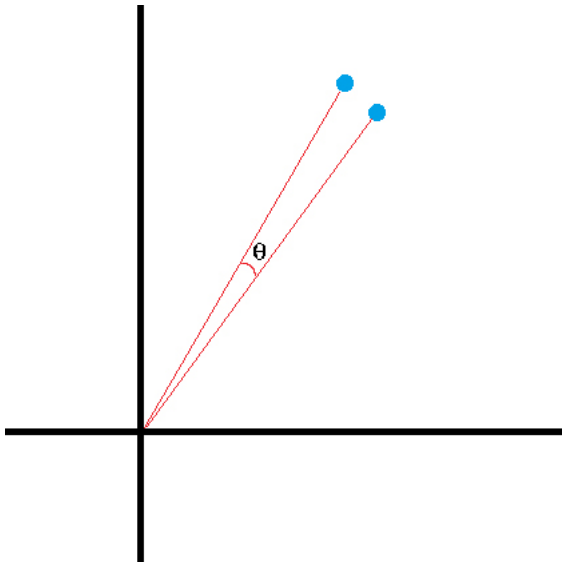


Рис. 5: Угол между двумя точками

На рисунке 5 изображены 2 синие точки. Угол между данными точками равен θ. Найдем вероятность того, данные точки будут располагаться по разные стороны от случайным образом (коэффициенты выбираются из равномерного распределения) проведенной прямой. Поскольку прямая определяется углом с осью абсцисс, то, очевидно эта вероятность равна $\mathbb{P} = \frac{\theta}{\pi}$. Поскольку косинусное расстояние монотонным образом зависит от угла между двумя точками, то, применив некоторое монотонное преобразование к косинусному расстоянию, мы можем узнать точную вероятность того, что случайнным образом определенная прямая разделит точки x_1 и x_2, таких что $d_{cos}(x_1, x_2) = R$. В то же время, если между точками большой угол, то и вероятность того, что данные точки получат разные теги функцией $h(x)$, будет велика. Стоит отметить, что в этих рассуждениях мы выбирали случайным образом прямые, а не функции $h \in \mathcal{H}$, хотя в определении локально чувствительного семейства фигурирует вероятность по $\mathcal{H}$. Однако это не является ошибкой, потому что семейство $\mathcal{H}$ мы выбираем сами, случайным образом добавляя в него функции, соответствующие различным разделяющим гиперплоскостям.

Евклидово расстояние. Для евклидова расстояния каждая функция $h(x) \in \mathcal{H}$ соответствует прямой в многомерном пространстве $\mathbb{R}$. При этом каждая такая прямая разбивается на отрезки равной длины a. Для получения хэша точка x сначала

проецируется на данную прямую, а после этого происходит поиск номера отрезка, в который она попала. Иллюстрация:

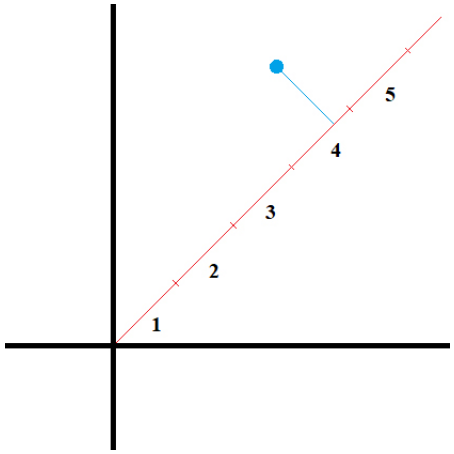


Рис. 6: Хэш для евклидового расстояния

Функция $h(x)$, которая соответствует данной прямой, выдаст значение 4 для объекта x, который изображен синей точкой. Выполнять данное хэширование оказывается тоже достаточно просто: для получения проекции точки $x \in \mathbb{R}^n$ на направление $d \in \mathbb{R}^n$, где $\|d\| = 1$, достаточно взять посчитать их скалярное произведение: $p = \langle x, d \rangle$. Для получения отрезка разбиения, в который попадет проекция, достаточно взять (допуская волность речи) остаток от деления p на a, где a – длина отрезков разбиений, гиперпараметр модели. Заметим, что одна такая хэш функция разбивает все пространство на бесконечные полосы равной ширины:

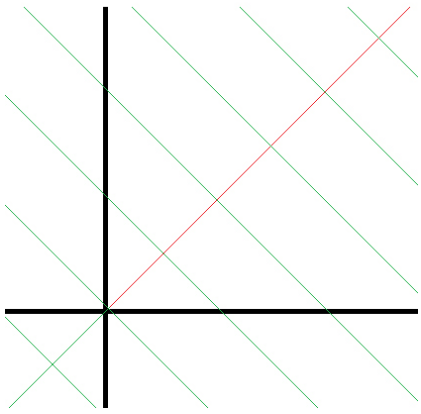


Рис. 7: Области разбиения плоскости одной хэш функцией

Равные значения хэшей получат те объекты x, которые лежат в одной такой полосе. Однако поскольку итоговая хэш функция получается конкатенацией нескольких элементов из $\mathcal{H}$, то вместе они будут разбивать плоскость на некоторые многоугольники:

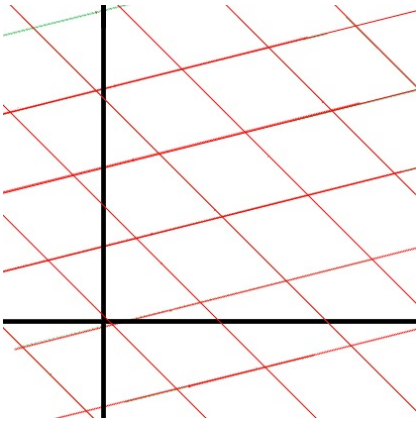


Рис. 8: Области разбиения плоскости двумя хэшами

Интуитивно понятно, семейство $\mathcal{H}$ является локально чувствительным. Строгое доказательство этого факта можно найти в [12].

В заключение разговора о LSH стоит также описать известный метод хэширования **MinHash** для поиска дубликоватов в наборе текстов. Прежде чем использовать данный алгоритм, необходимо некоторым образом представить текст в виде вектора. Наиболее простым и распространенным способом сделать применительно к нашей задаче является метод разбиения документа на цепочки из k последовательно идущих слов и их one-hot кодирования (shingling). Для задачи поиска дубликатов обычно используются значения k от 5 и больше.

Пример 2. *Кодирование строк A = «ab ba ba ab» и B = «ca ab ba» при длине цепочки k=2 происходит следующим образом: составляются пары соседних слов [«ab ba», «ba ba», «ba ab»], [«ca ab», «ab ba»]. Каждой уникальной паре сопоставляется некоторая позиция в векторе (таким образом, финальная размерность кодов будет равна числу уникальных пар слов во всем наборе документов). Сделаем следующее сопоставление: {«ab ba» = 1, «ba ba» = 2, «ba ab» = 3, «ca ab» = 4}. Тогда код первой строчки [1, 1, 1, 0], а второй [1, 0, 0, 1].*

Таким образом, мы представляем каждый текст в виде множества шинглов и описываем бинарными векторами. В качестве меры сходства между такими множествами можно взять коэффициент Жаккара

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Легко заметить, что размерность векторов кодов для документов может быть колоссально большой, поэтому в случае, когда документов достаточно много, прямое вычисление коэффициента Жаккара между всеми парами документов оказывается очень неэффективным.

Альтернативой является хэширование данных бинарных векторов с помощью алгоритма MinHash. Кратко опишем его реализацию.

Шаг 1. Из кодов документов составляется матрицу Shingles × Documents.

Шаг 2. Строки матрицы переставляются случайным образом.

Шаг 3. В каждом столбце происходит поиск номера первой строки, значение в которой равно единице.

Шаг 4. Собрать полученные номера в вектор, добавить его к результирующей матрице H. Если число строк H меньше c, повторить шаги 2 и 3.

Натуральное число c является гиперпараметром алгоритма. Итоговые хэши для каждого документа будут располагаться по столбцам матрицы H. Именно они и будут сравниваться для выявления дубликатов.

Важным свойством данного алгоритма (благодаря которому он так популярен) является тот факт, что вероятность совпадения MinHash для случайной перестановки элементов двух множеств равна коэффициенту Жаккара этих множеств. Таким образом, при увеличении параметра c, мы все точнее оцениваем данную вероятность, а тем самым и J(A, B). На практике этот параметр логично выбирать таким, чтобы оценка как можно более точной, но при этом время вычисления оставалось приемлимым. Возможно также организовывать иерархию из нескольких хэшей: сначала применить MinHash, а потом какой-нибудь другой метод приближенного поиска ближайших соседей, например, LSH для евклидового расстояния. Иллюстрации к MinHash (источник [7])

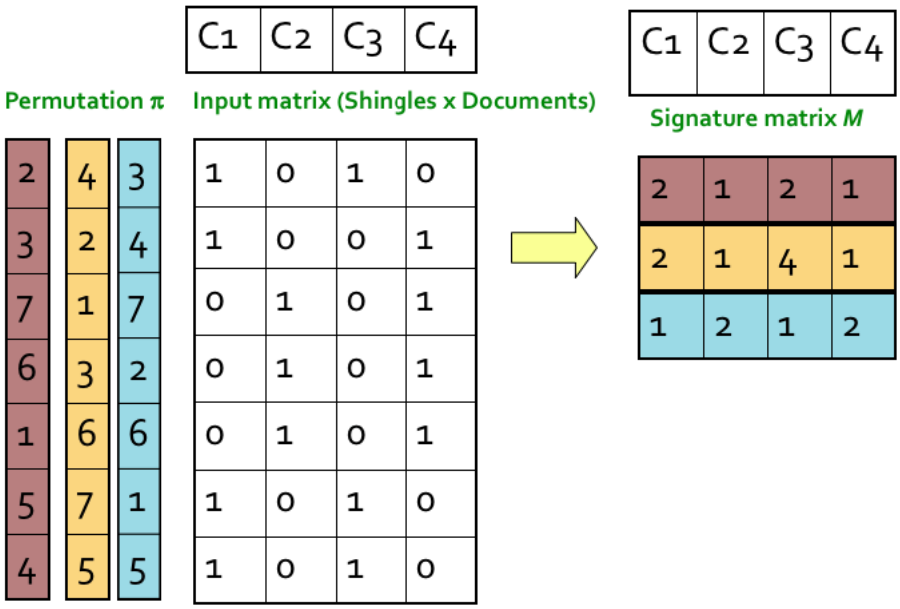


Рис. 9: MinHash [7]

Отметим, что представленные методы хэширования не исчерпывают все существующие. К примеру, достаточно популярным является метод FlyHash[5], основная идея которого была получена из области биологии, а именно из способа распозавания запахов мухой «Дрозофила фруктовая».

2.6 Приближенные методы: FAISS

В то время как приближенные методы поиска ближайших соседей с помощью LSH активно развивались уже долго время, наиболее эффективными на данный момент являются алгоритмы, разработанные в ближайшие 5 лет. Среди них библиотека от исследовательской группы Facebook под названием FAISS (Facebook AI Similarity Search) и графовый метод поиска ближайших соседей HNSW (Hierarchical Navigable Small World), разработанный нашими соотечественниками. В данной секции будет изложен подробный обзор библиотеки от Facebook, а в следующей будет рассмотрен HNSW.

Прежде всего стоит упомянуть статью [9], выпущенную авторами библиотеки, в которой излагаются подробности реализации алгоритма, а также его особенности, связанные с его выполнением на GPU. Одной из важных особенностей FAISS является тот факт, что большинство её методов могут выполняться именно на графических

процессорах, которые уже достаточно давно весьма активно используются в области машинного обучения для вычислений, связанных с обучением глубоких нейронных сетей. GPU «заточены» под однородные параллельные вычисления, работу с тензорами и матрицами. Возможность выполнения алгоритмов FAISS на GPU существенно повышает эффективность их работы.

Библиотека включает в себя эффективную реализацию нескольких методов поиска ближайших соседей (как приближенных, так и точных). Среди них, к примеру, уже упомянутый HNSW. Однако наибольшую популярность FAISS завоевала благодаря эффективной реализации алгоритма IVFADC [8] с использованием Product quantization (IVFPQ). Опишем его подробнее.

Основной идеей данного алгоритма является так называемая квантизация (слово широко используется в обработке сигналов), а именно представление вектора в некотором дискретном пространстве. В качестве простейшего аналога можно привести следующий пример. Имея набор данных X, мы можем откластеризовать эти точки на k кластеров, используя некоторый алгоритм кластеризации (например, K-Means). После этого мы можем заменить каждый вектор из X на центроид его кластера (или его индекс). После этого, при поступлении запроса q, мы можем посчитать расстояния до всех центроидов этих кластеров, и выдать из них минимальный. Проблема такого подхода заключается в trade-off между точностью результата и количеством вычислений. Очевидно, что если исходный набор данных велик (а именно таким наборам и посвящена данная работа), то, выбрав малое число кластеров, мы получим низкую точность. Однако если выбрать слишком большое число кластеров, то возрастает сложность кластеризации и время поиска ближайшего центроида для запроса. Именно эту проблему пытается решить структура IVFADC.

Все векторы в исходном наборе данных предлагается приблизить следующим образом:

$$y \approx q(y) = q_1(y) + q_2(y - q_1(y))$$

Где $q_1 : \mathbb{R}^n \to \mathcal{C}_1 \subset \mathbb{R}^n$ и $q_2 : \mathbb{R}^n \to \mathcal{C}_2 \subset \mathbb{R}^n$ – это квантизаторы, то есть функции, переводящие вектора в конечные множества. При этом q_1 является квантизатором первого уровня и называется грубым, а q_2 – квантизатор второго уровня, более точный.

Имея данные приближения, метод Asymmetric Distance Computation (ADC) выполняет поиск приближенного результата (x - запрос):

$$L_{ADC} = \text{k-argmin}_{i=0:l} \|x - q(y_i)\|_2$$

Однако в IVFADC мы пытаемся избежать слишком большого перебора. Для этого предварительно вычисляется набор «центроидов» кластеров из множества $\mathcal{C}_1$, среди элементов которых будет происходить поиск ближайших соседей:

$$L_{IVF} = \tau\text{-argmin}_{c \in \mathcal{C}_1} \|x - c\|_2$$

В контексте приведенного выше примера, мы ищем, к каким τ центроидам кластеров из существующих ближе всех наш запрос. При этом натуральное число τ является гиперпараметром алгоритма – он влияет на то, как много центроидов мы хотим рассматривать. После этого выполняется поиск ближайших соседей к x среди тех векторов, которые относятся к кластерам, найденным на прошлом шаге:

$$L_{IVFADC} = \text{k-argmin}_{i=0:ls.t.q_1(y_i) \in L_{IVF}} \|x - q(y_i)\|$$

Стоит отметить, что быстрый поиск принадлежащих данному кластеру точек выполняется с помощью «инвертированного файла»: для каждого кластера составляется список индексов тех векторов, которые ему принадлежат.

Как было отмечено выше, основой данного алгоритма являются идея квантизации. Грубый квантизатор q_1 должен иметь сравнительно небольшое число выходных значений (авторами статьи рекомендуется использвать $|\mathcal{C}_1| \approx \sqrt{l}$), полученных с помощью K-Means. Однако q_2 предлагается устроить несколько более сложным образом. Проведем следующее наблюдение: поскольку в данному алгоритме мы работаем с евклидовым расстоянием $d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)}$, а функция $f(x) = \sqrt{x}$ является монотонной, то

$$\text{k-argmin}_{i=0:l} \|x - y_i\|_2 = \text{k-argmin}_{i=0:l} \sqrt{\sum_{j=1}^{n}(x_j - y_{ij})_2} = \text{k-argmin}_{i=0:l} \sum_{j=1}^{n}(x_j - y_{ij})_2$$

Разобьем теперь размерность n нашего признакового пространства на несколько равных частей: $y = [y^0 \dots y^{b-1}]$. К примеру, если мы имели вектор размерности 16, то мы можем разбить его на 4 компоненты размерности 4: $[x_1\ x_2\ x_3 \dots x_{15}\ x_{16}] \rightarrow$

$[x_1\ x_2\ x_3\ x_4], [x_5\ x_6\ x_7\ x_8], [x_9\ x_{10}\ x_{11}\ x_{12}], [x_{13}\ x_{14}\ x_{15}\ x_{16}]$. Тогда указанное выше выражение можно переписать как

$$\text{k-argmin}_{i=0:l} \sum_{j=1}^{n} (x_j - y_{ij})_2 = \text{k-argmin}_{i=0:l} \sum_{c=1}^{C} \sum_{z=1}^{Z} (x_{cz} - y_{icz})_2$$

Где C - число компонент, на которые мы разбили вектор, а Z - число «размерностей» в каждой из компонент. Далее, каждый из полученных подвекторов мы квантизируем с помощью отдельной (для данного номера компоненты) функции, получая кортеж $(q^0(y^0), \ldots, q^{b-1}(y^{b-1}))$. Каждая из функций q^i на самом деле представляет собой алгоритм K-Means с 256 кластерами (чтобы помещаться в один байт). Итоговое значение квантизации представимо в виде $q_2(y) = q^0(y^0) + 256 \times q^1(y^1) + \cdots + 256^{b-1} \times q^{b-1}(y^{b-1})$, что в сущности является конкатенацией байтов, полученных под-квантизаторами q^i. Описанный алгоритм называется Product quantizer.

В предыдущем абзаце был подробно изложен алгоритм кодирования исходных векторов в наборе данных с помощью квантизации. Опишем его ещё раз в более общих чертах:

1. Набор исходных данных кластеризуется с помощью K-Means на $|\mathcal{C}_1| \approx \sqrt{l}$ кластеров

2. Для каждого вектора вычисляется разность $y - q_1(y)$ – «невязка» между вектором и центроидом его кластера

3. Матрица полученных невязок разбивается на подпространства одинаковой размерности, и каждое из полученных подпространств кластеризуется отдельно с помощью K-Means

4. Каждый вектор из матрицы невязок кодируется последовательностью из байтов, каждый из которых соответствует индексу кластера, к которому относится соответствующая часть невязки

Отдельно отметим, что на втором этапе квантизации используется именно невязка $y - q_1(y)$, а не сам вектор y, с целью повышения точности: модуль вектора $y - q_1(y)$ будет меньше, чем модуль вектора y, поэтому их проще и устойчивее кластеризовать.

Опишем теперь процедуру поиска ближайшего соседа. Как было указано выше, для запроса x сначала нужно вычислить τ ближайших к нему центроидов кластеров. Далее мы ведем поиск ближайшего соседа среди только тех точек из X, которые принадлежат этому набору кластеров. Для каждого центроида c_i мы вычисляем невязку: $\delta = x - c_i$ и должны найти среди элементов данного кластера чья невязка с c_i наиболее похожа δ. Однако вспомним, что все невязки закодированы последовательностью байтов - результат второго уровня квантизации. Для эффективного поиска среди этих векторов мы также разобьем запрос x на подпространства и для каждого из них посчитаем расстояния до соответствующих 256 центроидов кластеров. Получим матрицу $D = C \times 256$, где C - число компонент (подпространств) на которые были разбиты невязки. Теперь же, для вычисления приближенного расстояния между невязками $y_i - q_1(y_i)$ и $x - q_1(x)$, достаточно просуммировать значения матрицы D, взятыми по индексам столбцов, равным байтам второго уровня квантизации $q_2(y_i - q_1(y_i))$ и соответствующих им строк. То есть, для каждого номера байта b к сумме добавится значение $D[b, q^b(y_i - q_1(y_i))]$. Строго говоря, таким образом мы получим оценку квадрата расстояния между $y_i - q_1(y_i)$ и $x - q_1(x)$, но выше было доказано, что точка минимума от этого не изменится.

Подведем некоторые итоги. Алгоритм IVFPQ, реализованный в библиотеке FAISS, в сущности представляет собой иерархическую систему квантизации, использующую идею разделения пространства на подпространства. Важной особенность данного алгоритма является то, что он может быть эффективно выполнен на GPU. Кроме того, в памяти хранятся сжатые представления векторов, что экономит ресурсы.

2.7 Приближенные методы: HNSW

Финальным методом, рассматриваемым в данной работе, является Hierarchical Navigable Small World (HNSW). Данный алгоритм был разработан российскими учеными из Нижнего Новгорода. Ими были опубликованы две статьи: первая [10] описывает базовую модель поиска ближайших соседей, используя графовую модель «Маленького мира», а вторая [11] предлагает её усовершенственную версию, с использованием иерархии из нескольких слоев.

В основе данного алгоритма лежит идея представления пространства объектов виде графа: каждый объект x из набора данных X представляется вершиной $v \in V$ некоторого графа $G = (V, E)$. При этом между некоторыми объектами существуют связи – ребра E в графе G. Предположим, что нам удалось некоторым образом построить такой граф. Тогда при поступлении запроса q, из какой-то точки графа (в простейшем случае случайной) начинается жадный поиск: на каждой итерации среди всех вершин, смежных данной, выбирается вершина, расстояния от которого до q минимально. После этого эта вершина выбирается в качестве текущей. В тот момент, когда все расстояния от всех смежных вершин до q будут больше, чем текущее, алгоритм прекращает работу.

Основной проблемой в предложенной конструкции является выбор эффективного набора ребер. Очевидно, что если выбрать граф, в котором все вершины связаны друг с другом (полный граф), то алгоритм будет работать корректно (всегда выдавать точный ответ). Однако такой подход ничем не будет отличаться от прямого перебора. Оказывается [4], что для корректной работы граф должен содержать (и этого достаточно) в качестве подграфа граф Делоне. В двумерном случае он соответствует триангуляции Делоне, конструкции, двойственной диаграмме Вороного.

Определение 2. *Имея некоторый набор точек P на плоскости, ячейкой Вороного для точки $p \in P$ называется геометрическое место точек на плоскости, которые расположены к p ближе, чем к любой другой точке из P. Совокупность ячеек Вороного для всех точек из P называется диаграммой Вороного для P.*

Определение 3. *Триангуляцией Делоне для набора точек P на плоскости называется триангуляция, которая получается из диаграммы Вороного соединением каждой точки $p \in P$ с теми точками, которые соответствуют граничным ячейкам Вороного для данной точки.*

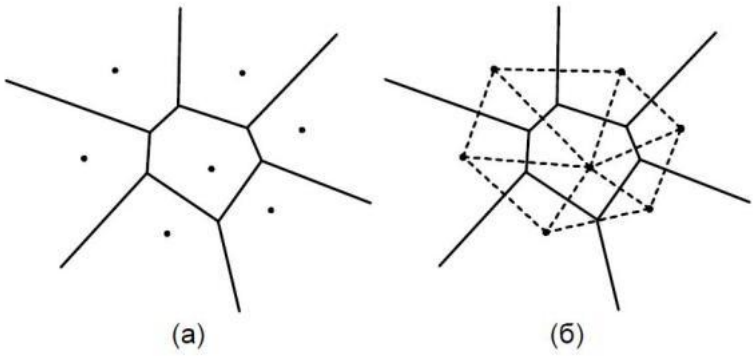


Рис. 10: Диаграмма Вороного (а), Триангуляция Делоне (б, пунктир)

Однако для эффективного построения графа Делоне требуется априорная информация о внутренней структуре данных. На практике же обычно строят некоторые его приближения.

В [10] авторы предлагают концепцию графа NWS (Navigable Small World), то есть графа, в котором число итераций жадного алгоритма в среднем логарифмическое (или полилогарифмическое). Алгоритм построения прост: граф строится итеративно, при добавлении очередной точки, она связывается двунаправленными ребрами с M ближайшими вершинами в уже построенном графе.

Главное отличие HNSW от NSW заключается в системе слоев. Для повышения эффективности поиска, граф представляется в виде нескольких слоев: чем выше по уровню слой, тем меньше на на нем вершин и длиннее ребра, при этом все точки, которые есть на уровне n+1, также есть и на уровне n. Процедура поиска устроена следующим образом. Поиск начинается с произвольной точки в самом верхнем слое, в нем же ведется жадный поиск ближайшего соседа. После того, как жадный алгоритм остановится, мы переходим на уровень ниже, и запускаем снова жадный поиск с той же точки, в которой мы остановились на верхнем слое.

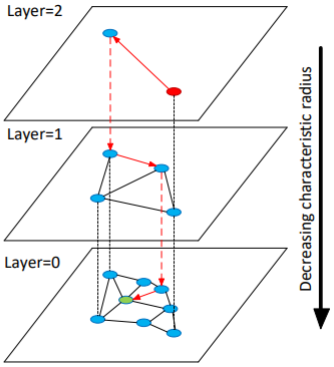


Рис. 11: Слои в HSNW [11]

Одним из способов построения такой иерархической структуры является рандомизация: для каждого элемента $x \in X$ мы выбираем номер l наивысшего слоя, в котором он должен присутствовать. Если выбирать l из экспоненциально убывающего распределения (например, из геометрического), то ожидаемое число слоев будет логарифмическим. Здесь прослеживается важное отличие NSW от HNSW: в отличие от NSW, HNSW не требует предварительного перемешивания элементов в наборе данных (поскольку структура графа зависит от порядка вставки элементов), это достигается из-за рандомизации при выборе слоев. Подробности эффективной реализации слоев можно найти в [11].

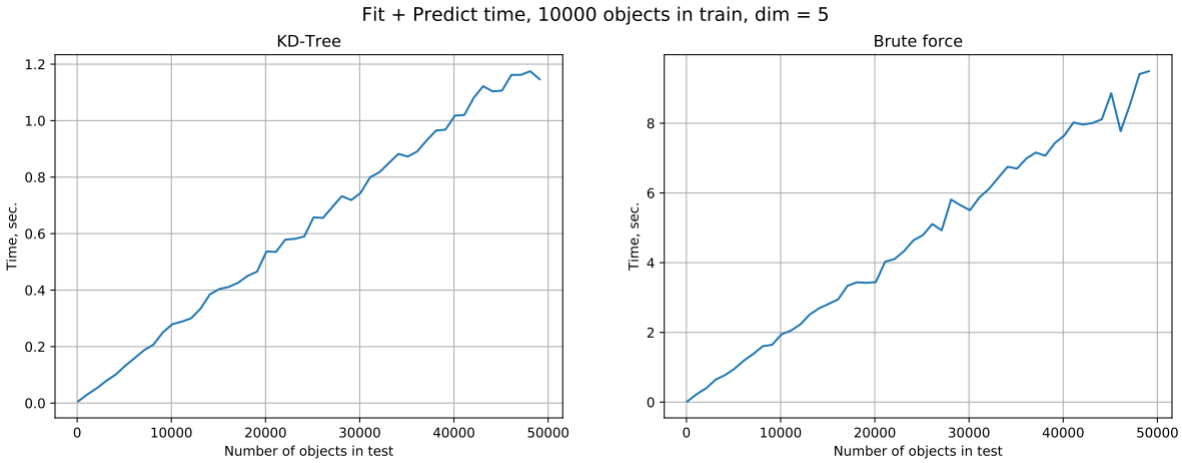
3 Вычислительные эксперименты

Проведем серию экспериментов для сравнения описанных в предыдущем разделе методов поиска ближайших соседей. Поскольку мы планируем исследовать время работы алгоритмов в зависимости от объема данных и их размерности, будем использовать данные, которые генерирует для классификации библиотека sklearn (а именно sklearn.datasets.make_classification с параметром random_state = 42 для воспроизводимости экспериментов).

3.1 Прямой перебор и деревья

Для начала сравним наиболее классические методы: прямой перебор и древовидные структуры данных на примере kd-tree. Число желаемых соседей k=5. Сравним

графики времени поиска ближайших соседей в зависимости от числа объектов в запросе. Число объектов в исходном наборе данных равно 10000. Размерность признакового пространства 5.



Поскольку запросы выполняются независимо, то графики получаются линейными. При этом структура kd-tree дает выигрыш в скорости примерно в 10 раз. Попробуем увеличить размерность признакового пространства.

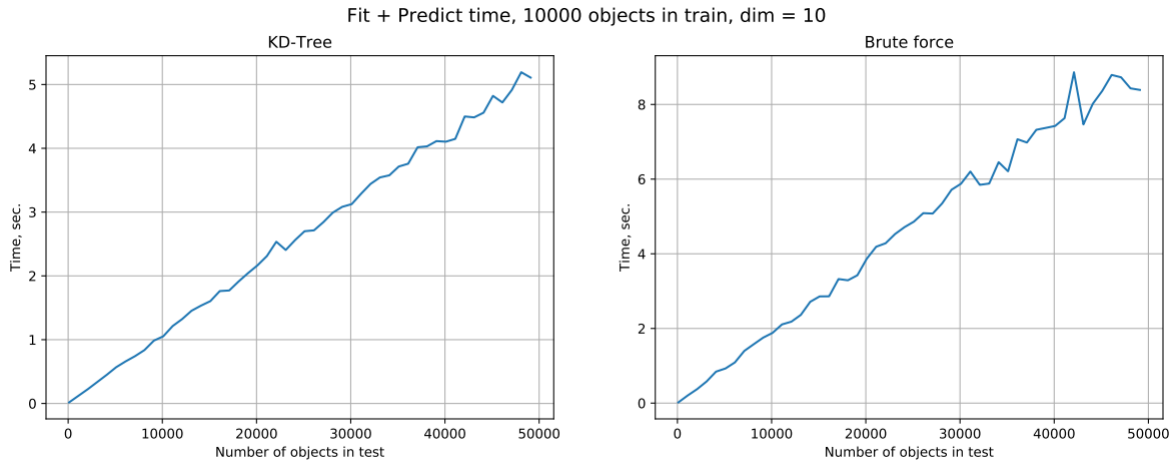
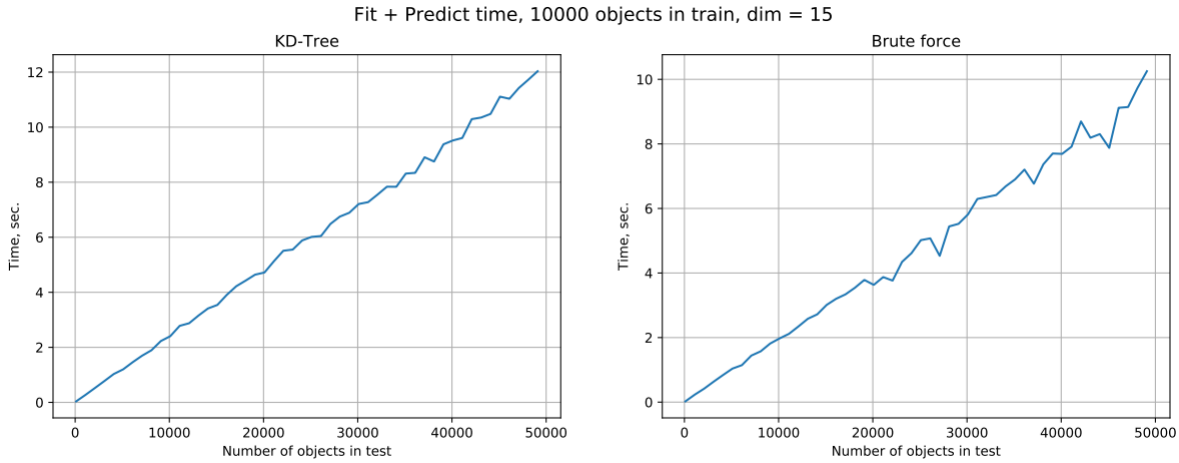
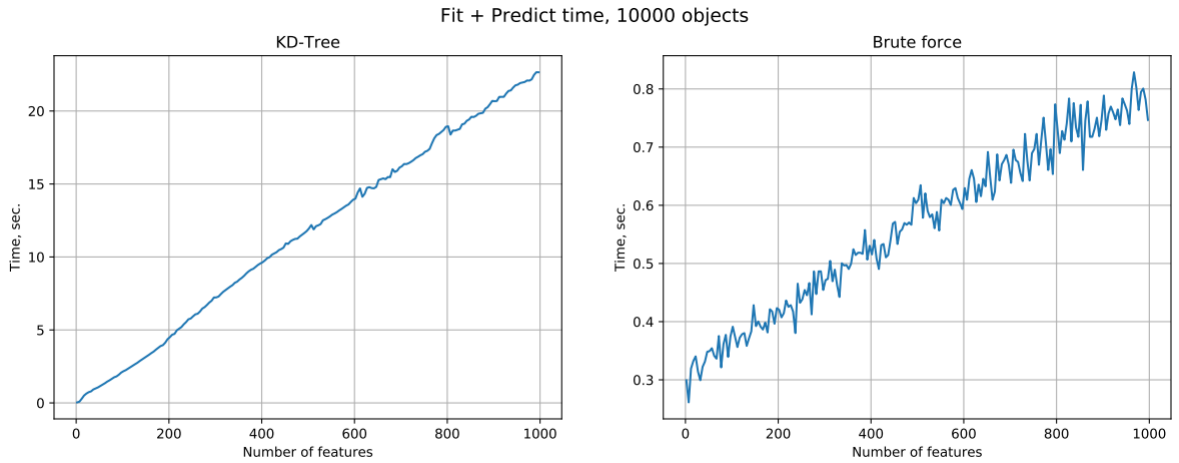


График времени работы алгоритма прямого перебора почти не, однако эффективность kd-дерева упала примерно в 5 раз. Увеличим размерность до 15.

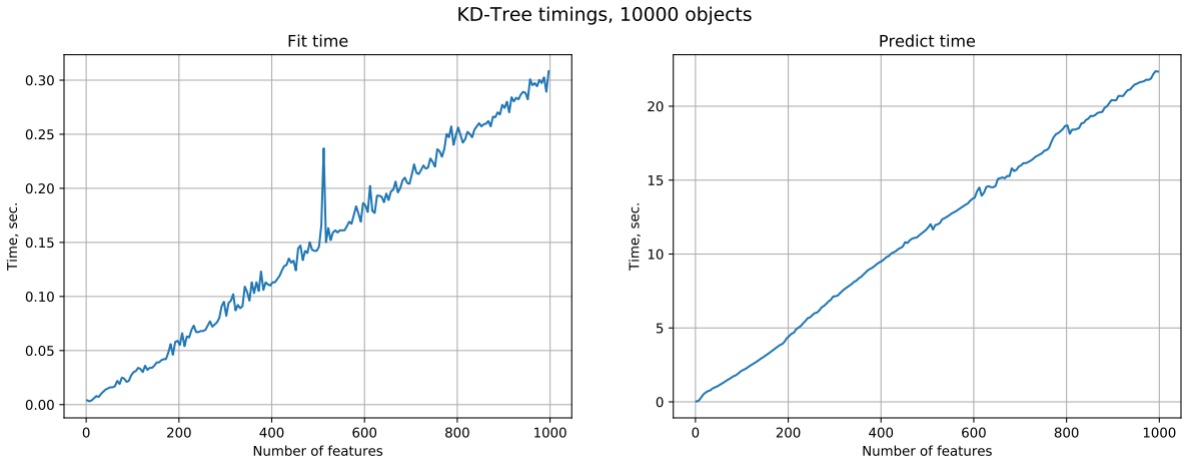


Как видим, при размерности пространства dim=15, прямой перебор становится более эффективным, чем использование деревьев. Стоит отметить, что на самом деле этот порог зависит от количества точек в исходном наборе данных, однако, как говорит нам проклятие размерности, количество данных придется увеличивать экспоненциальным образом, чтобы поддерживать высокую эффективность. Зафиксируем число объектов в тестовом наборе (2000), и посмотрим, как зависит время работы алгоритмов в зависимости от размерности признакового пространства.



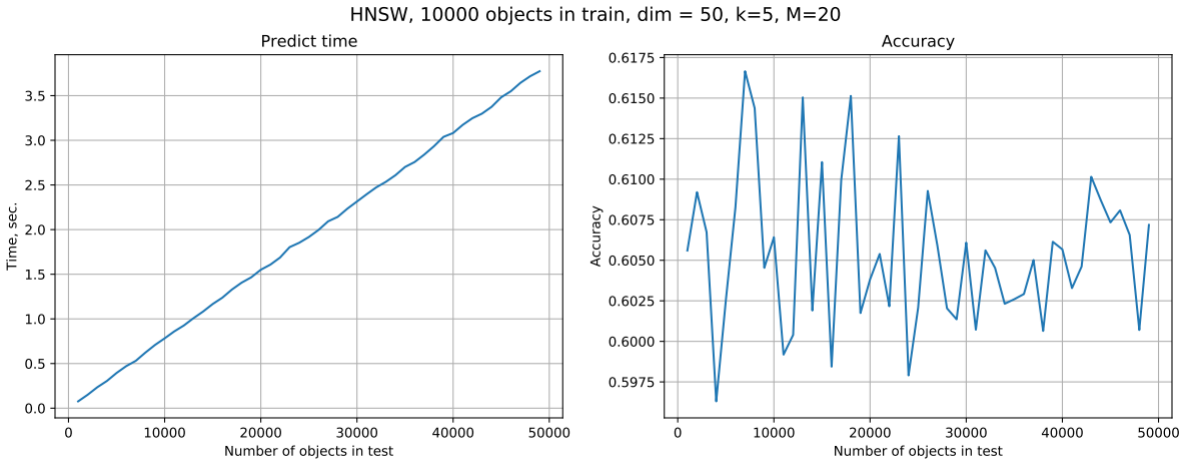
Как видим, обе зависимости являются линейными, однако в случае с kd-деревом, коэффициент наклона прямой гораздо выше. На всех изображенных выше графиках этапы построения дерева и непосредственного выполнения запроса были совмещены. Представленные ниже графики демонстрируют, что этап построения дерева занима-

ет существенно меньше времени, чем этап поиска ближайших соседей с использованием уже построенной структуры.



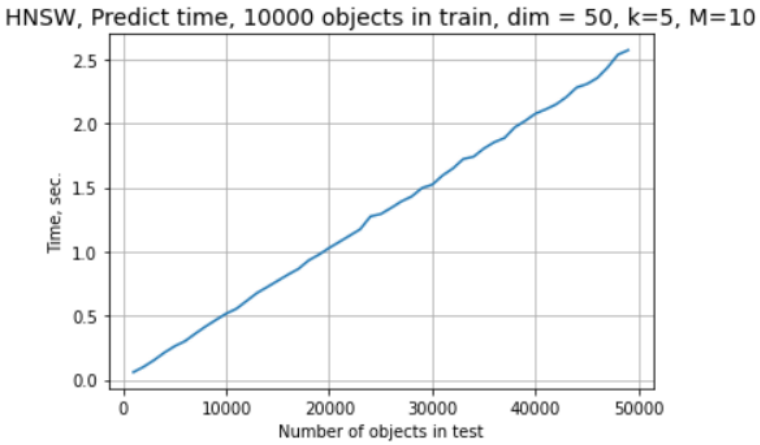
3.2 HNSW

Исследуем теперь время работы HNSW (реализация из FAISS). Как было указано в разделе 2.7, данный алгоритм имеет гиперпараметр M – число ближайших соседей, к которым проводим связи в графе при добавлении очередного элемента. Чем выше значение M, тем точнее выполняется поиск ближайших соседей и дольше время выполнения. Посмотрим на время работы алгоритма в завимости от количества объектов в запросе. Число объектов в исходном наборе 10000, размерность пространства 50, M=20, k=5. Поскольку метод является приближенным, также введем в рассмотрение дополнительную метрику: долю правильных ответов. А именно, если X_Pred - множество индексов выданных алгоритмом ближайших соседей, а X_True - индексы истинных ближайших соседей, то наша метрика равна $\frac{|X\_Pred \cap X\_True|}{|X\_true|}$.

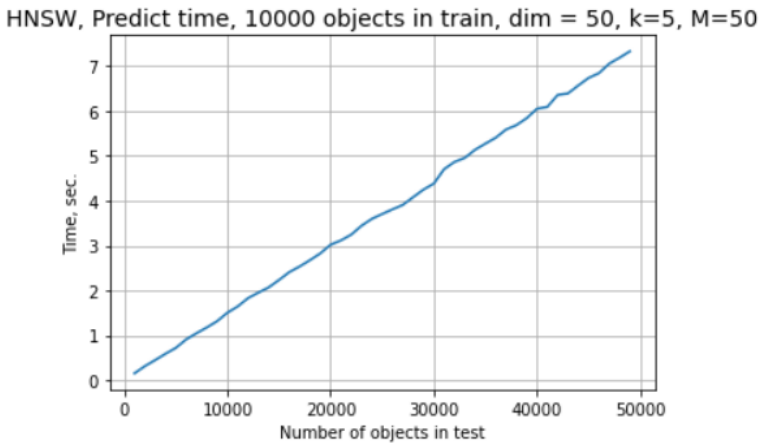


Как видим, точность колеблется около значения 0.6075 некоторым случайным образом, поэтому в дальнейшем мы будем высчитывать это значение лишь один раз, и считать, что оно мало отклоняется от среднего.

Попробуем изменять параметр алгоритма M и следить, как будет изменяться график времени работы и показатель точности поиска. Для начала уменьшим M до 10. Получим точность 0.472 и следующий график:

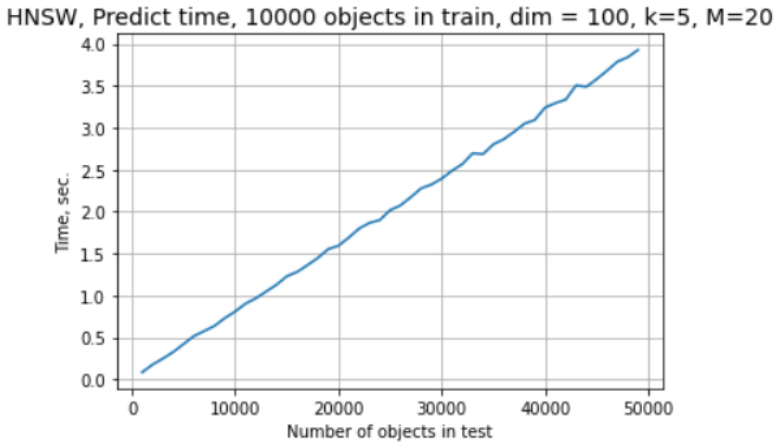


Как и ожидалось, время запроса действительно уменьшилось, а также упала точность. Увеличим теперь параметр M до 50. Как и ожидалось, получим повышение точности до 0.816 и следующий график:



Таким образом, варьируя параметр M, мы можем поддерживать нужный баланс между временем вычислений и точностью результата.

В [10] авторы утверждают, что эффективность алгоритма слабо зависит от размерности данных. Проверим это, увеличив размерность до 100.



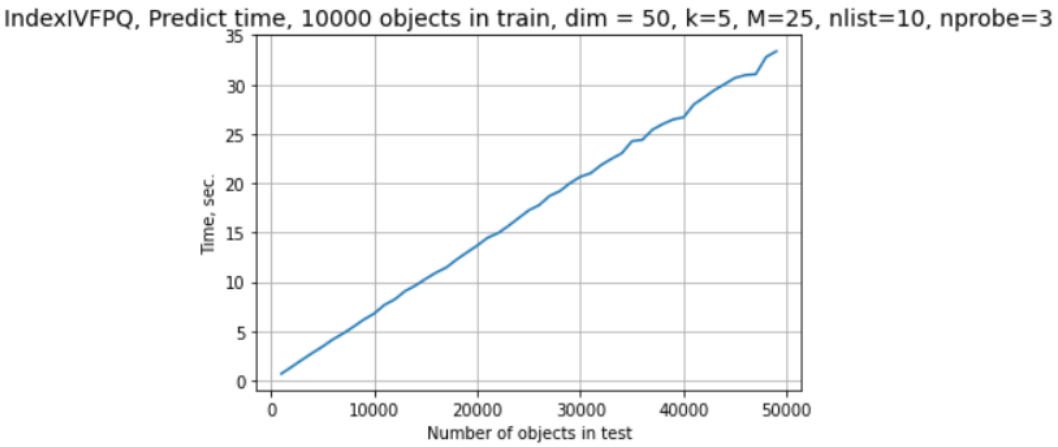
Как видим, скорость запроса увеличилась действительно незначительно.

3.3 FAISS: IVFADC

IVFADC имеет целый набор параметров: quantization - метод квантизации, nlists – число кластеров для грубой квантизации, M – число подпространств, на которые разбиваются векторы-невязки, nbits – число бит, которыми кодируются кластеры в квантизаторе второго уровня (обычно 8, что соответствует 256 кластерам), nprobe – число кластеров, в которых ведется поиск. В библиотеке есть попытка автоматизации

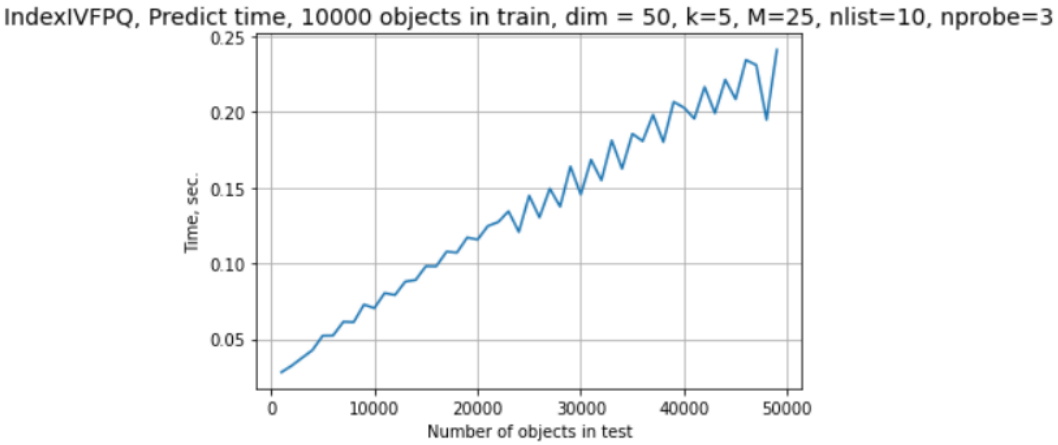
подбора параметров, однако в целях сокращения времени работы, мы ограничимся лишь несколькими экспериментами.

Важным является соотношение nlists и nprobe, поскольку если число кластеров велико, а nprobe мало, то точность такой модели оказывается достаточно низким. К примеру, при nlists=50, а nprobe=1, качество точности принимает значения меньше 0.1 (число объектов в исходном наборе 10000, в запросе 25000). Чем меньше nlists, тем больше кластеры размером, а соответственно дольше время запроса. Если выбрать nlists=5, а nproba=4, то такая модель дает более 0.8 точных ответов, но время запроса существенно увеличивается (может работать дольше, чем прямой перебор). Запустим модель на следующих параметрах: 10000 объектов в исходном наборе, размерность пространства 50, nlists = 10, nprobe = 3, M = 25, nbits = 8. До этого было установлено, что слишком низкое число подпространств, на которые разбиваются невязки, влечет за собой низкое качество (около 0.2).



Получаем такой график и точность 0.61. Как видим, время работы достаточно большое, примерно в 10 раз дольше, чем HNSW.

Попробуем запустить модель с теми же параметрами на GPU (Tesla K80).

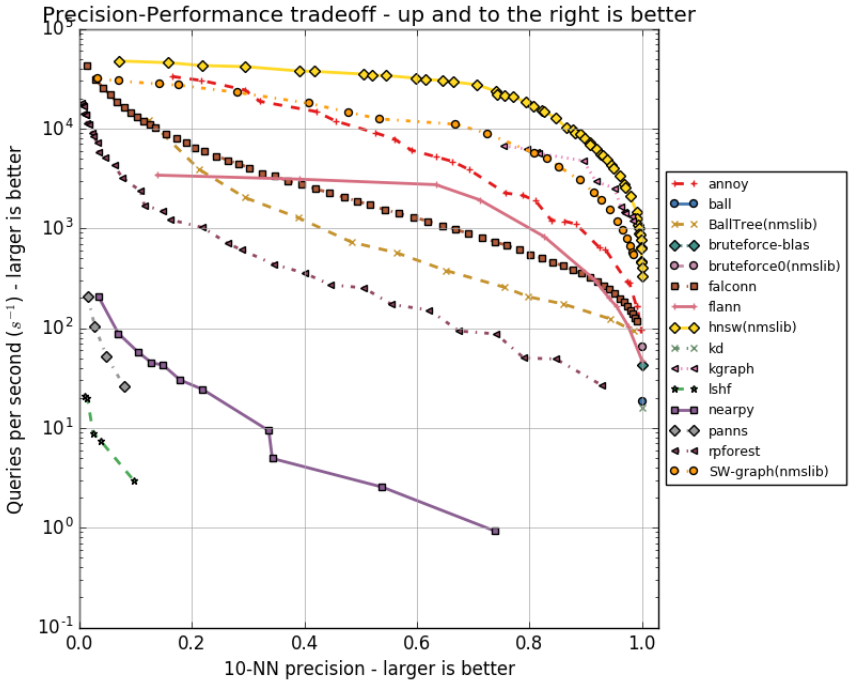


Как видим, выполнение алгоритма на GPU невероятно быстрое: запрос размером в 50 тысяч векторов обрабатывается примерно за четверть секунды. Качество при этом остается около 0.6.

Важно отметить, что все представленные графики отображают линейную зависимость, поскольку запросы выполняются независимо друг от друга. При этом важной информацией из них является лишь коэффициент наклона этой прямой: чем он ниже, тем быстрее метод. Нескольких экспериментов при этом выполняется для повышения точности оценки данного коэффициента. В качестве дальнейшего направления исследований можно также указать изучение скорости работы в зависимости от размера исходного набор данных (при фиксированном наборе запроса.)

3.4 Сторонние результаты

В [3] приведены более подробные экспериментальные сравнения приближенных алгоритмов поиска ближайших соседей. Некоторые результаты:



4 Резюме

Описав классические и наиболее современные методы поиска ближайших соседей, а также проведя ряд вычислительных экспериментов, мы можем подвести итоги: для каждого алгоритма описать его преимущества и недостатки.

Алгоритм	**Преимущества**	**Недостатки**
Прямой перебор	Простота	Неэффективность
kd-tree	Есть в стандартных библиотеках	Неэффективен в пространствах высокой размерности
LSH	Гибкость Возможность добавлять точки в трейн	Проигрывает HNSW и FAISS в скорости и затратах на память
FAISS	Гибкость (множество параметров) Высокая эффективность на GPU Сжатые представления векторов	На CPU работает медленнее, чем HNSW
HNSW	Простота Высокая эффективность	Не поддерживает сжатие векторов

5 Заключение

В данной работе был представлен подробный обзор точных и приближенных методов поиска ближайших соседей. Были указаны основные преимущества и недостатки тех или иных методов. Был проведен эксприментальное сравнение наиболее современных методов, в том числе с использованием графических процессоров.

Список литературы

[1] Виктор Китов. Лекционные слайды из курса Математические Методы Распознавания Образов.

[2] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, January 2008.

[3] Martin Aum?ller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms, 2018.

[4] Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, September 1991.

[5] Sanjoy Dasgupta, Charles F. Stevens, and Saket Navlakha. A neural algorithm for a fundamental computing problem. 2017.

[6] Mohamad Dolatshah, Ali Hadian, and Behrouz Minaei-Bidgoli. Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces. 2015.

[7] Shikhar Gupta. Locality sensitive hashing (towardsdatascience.com). 2018.

[8] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[9] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

[10] Yu Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 01 2013.

[11] Yu. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. 2016.

[12] Anand Rajaraman and Stephen Blott. Standford cs345a, winter: Data mining. 2009.

[13] Roger Weber, Hans-J. Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. 1998.