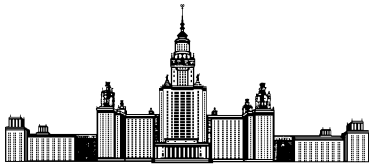
Московский государственный университет имени М. В. Ломоносова



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

Заведующий кафедрой

Математических Методов

Прогнозирования, академик РАН _____ Ю. И. Журавлёв

К защите допускаю

«_____» _____ 2020 г.

К защите рекомендую

«_____» _____ 2020 г.

Москва, 2020

Содержание

Аннотация

Todo

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.

1.2 Обзор литературы

todo, напишу после основной части

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 сорти-

руются, 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 логистиче-

ская 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 размером 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.

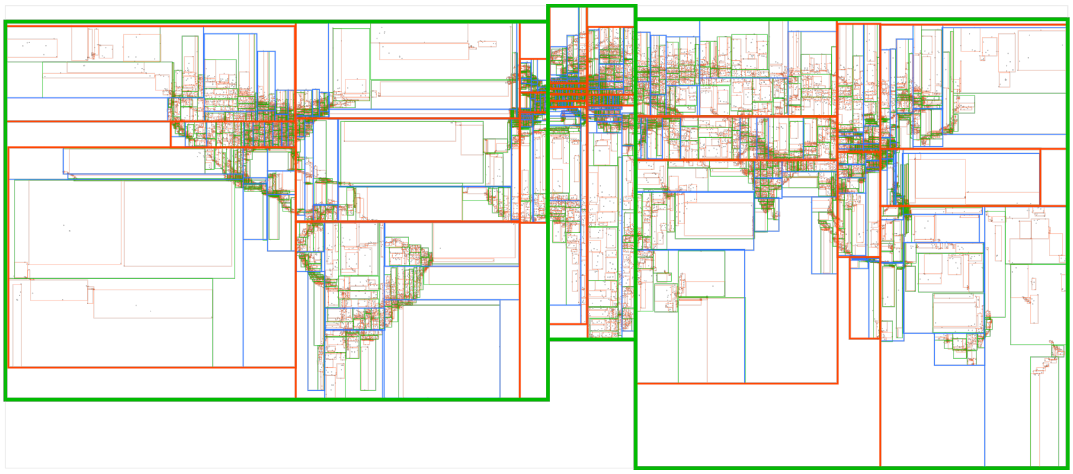


Рис. 1: Пример разбиения данных на плоскости с помощью R-Tree

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 решаемой 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
- Ball - Tree
- R - Tree
- BSP - Tree
- Quadtree
- 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.

---

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

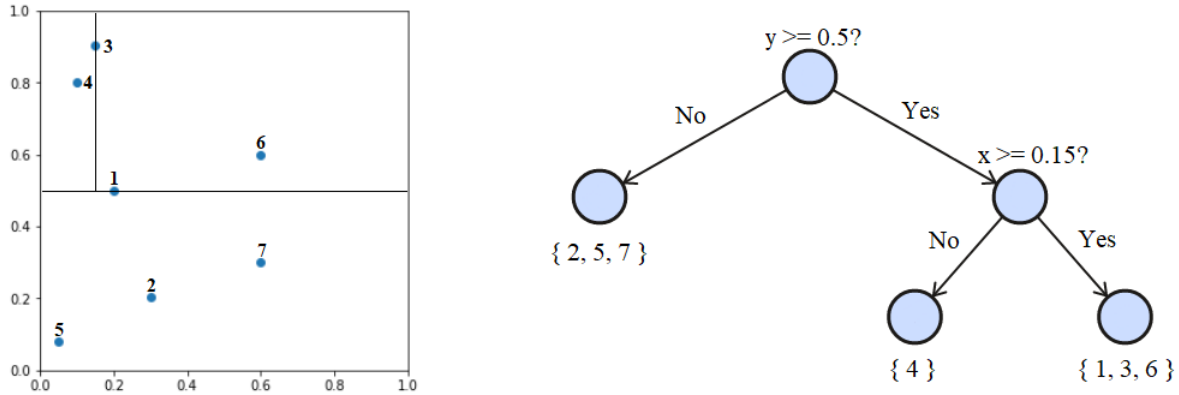


Рис. 2: Пример построенного KD-Tree, $n_{min} = 3$

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.

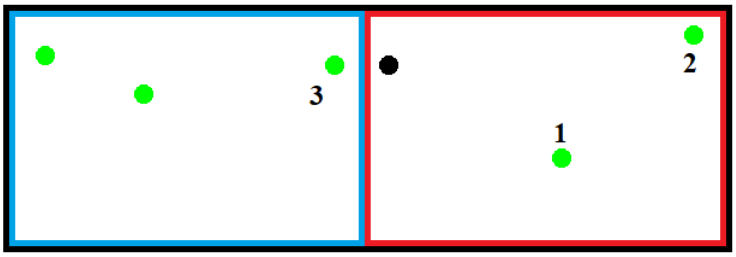


Рис. 3: Иллюстрация к операции запроса в KD - дереве

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 [3]. 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 [6]. 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 $\mathcal{H}$. 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 [5].

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 $\mathbb{R}^2$.

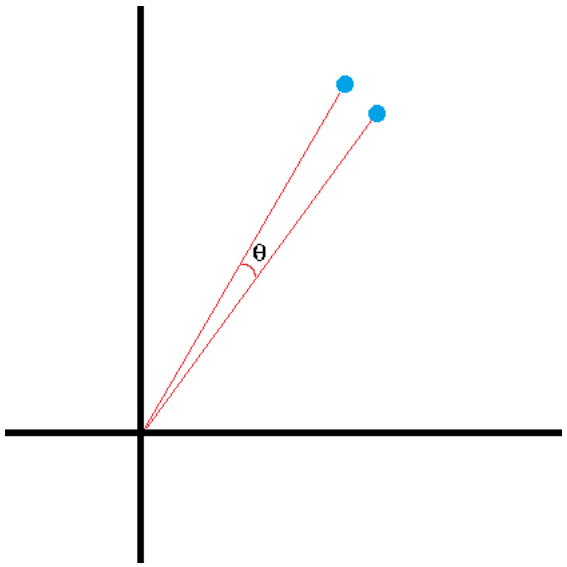


Рис. 4: Угол между двумя точками

На рисунке 4 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 случайным 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:

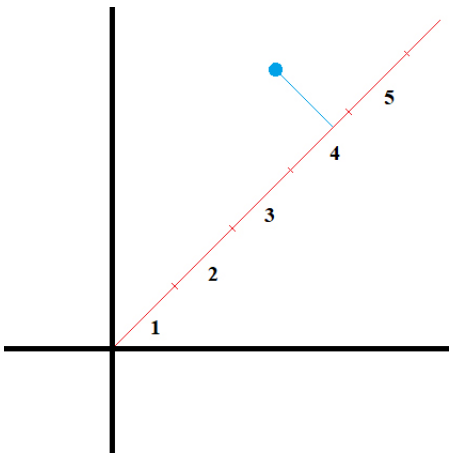


Рис. 5: Хэш для евклидового расстояния

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:

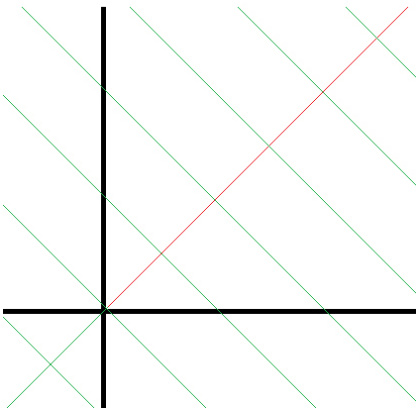


Рис. 6: Области разбиения плоскости одной хэш функцией

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:

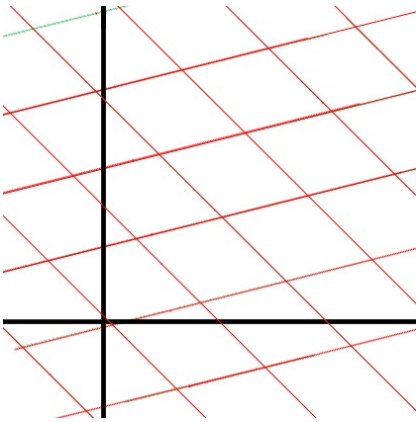


Рис. 7: Области разбиения плоскости двумя хэшами

Интуитивно понятно, семейство $\mathcal{H}$ является локально чувствительным. Строгое доказательство этого факта можно найти в [5].

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 [4])

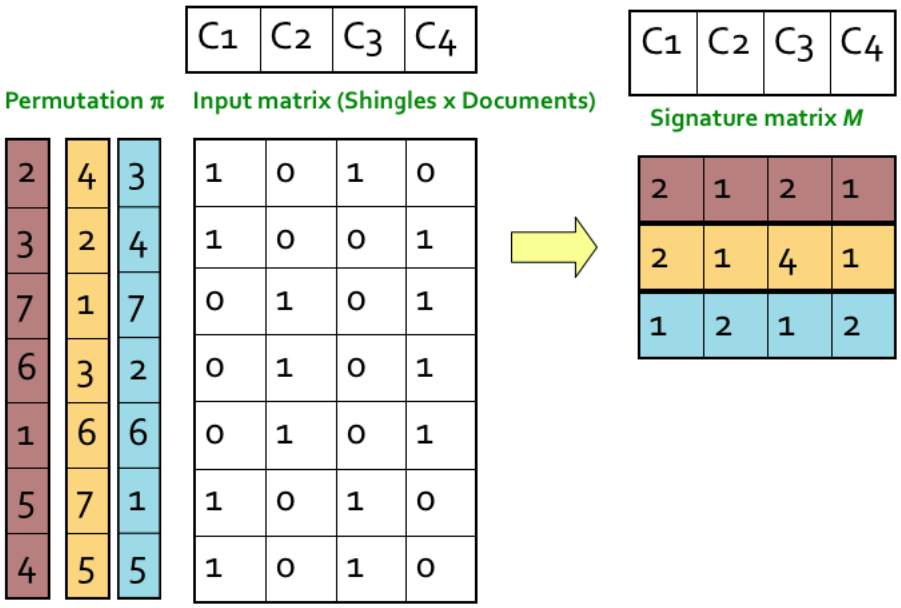


Рис. 8: MinHash [4]

2.6 Приближенные методы: FAISS

2.7 Приближенные методы: HNSW

3 Вычислительные эксперименты

Цель данного раздела: продемонстрировать, что предложенная теория работает на практике; показать границы её применимости; рассказать о новых экспериментальных фактах.

Чисто теоретические работы могут вообще не содержать раздела экспериментов (не работает, ну и не надо — зато теория красивая). Кстати, теоретики имеют право не догадываться, где, кому и когда их теории пригодятся.

3.1 Исходные данные и условия эксперимента

Описывается прикладная задача, параметры анализируемых данных (например, сколько объектов, сколько признаков, каких они типов), параметры эксперимента (например, как производился скользящий контроль).

3.2 Результаты эксперимента

Результаты экспериментов представляются в виде таблиц и графиков. Объясняется точный смысл всех обозначений на графиках, строк и столбцов в таблицах.

3.3 Обсуждение и выводы

Приводятся выводы: в какой степени результаты экспериментов согласуются с теорией? Достигнут ли желаемый результат? Обнаружены ли какие-либо факты, не нашедшие объяснения, и которые нельзя списать на «грязный» эксперимент?

Обсуждаются основные отличия предложенных методов от известных ранее. В чем их преимущества? Каковы границы их применимости? Какие проблемы удалось решить, а какие остались открытыми? Какие возникли новые постановки задач?

4 Заключение

В квалификационных работах последний раздел нужен для того, чтобы конспективно перечислить основные результаты, полученные лично автором.

Результатами, в частности, являются:

- Предложен новый подход к...
- Разработан новый метод..., позволяющий...
- Доказан ряд теорем, подтверждающих (опровергающих), что...
- Проведены вычислительные эксперименты..., которые подтвердили / опровергли / привели к новым постановкам задач.

Цель данного раздела: доказать квалификацию автора. Даже беглого взгляда на заключение должно быть достаточно, чтобы стало ясно: автору удалось решить актуальную, трудную, ранее не решённую задачу, предложенные автором решения обоснованы и проверены.

Иногда в Заключении приводится список направлений дальнейших исследований.

Список литературы необходим в любой научной публикации. В дипломной работе он обязателен. Дурным тоном считается: ссылаться на работы только одного-двух авторов (например, себя или шефа); ссылаться на слишком малое число работ; ссылаться только на очень старые работы; ссылаться на работы, которых автор ни разу не видел; ссылаться на работы, которые не упоминаются в тексте или которые не имеют отношения к данному тексту.

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

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

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

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