Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт
(государственный университет)»

Физтех-школа прикладной математики и информатики

Факультет управления и прикладной математики

Кафедра информатики и вычислительной математики

Сравнение алгоритмов поиска ближайших соседей в контексте определения похожих в смысле индекса Жаккара множеств

Выпускная квалификационная работа
(бакалаврская работа)

Направление подготовки: 03.03.01 Прикладные математика и физика

Выполнил:
студент 476 группы Скоморохов Филипп Алексеевич

Научный руководитель:
к.ф.-м.н. Овчинников Георгий Викторович

Москва 2018

Содержание

1 Введение **3**

2 Литературный обзор **4**

2.1 MinHash 4

2.2 R-дерево 5

2.3 BallTree 6

2.4 KDTree 9

2.5 FAISS 13

2.6 LSH 14

2.7 Annoy 16

2.8 LSH Forest 22

3 Метод **24**

4 Результаты **26**

5 Анализ результатов **28**

6 Заключение **30**

7 Список литературы **30**

1 Введение

В последние несколько десятилетий поиск похожих множеств среди большого объёма данных стал одной из наиважнейших задач. Использование структуры данных индекс помогает избежать миллионов прямых сравнений этих множеств друг с другом. Цель данной работы - представить сравнительный анализ алгоритмов поиска приблизительных ближайших соседей (approximate nearest neighbour) и нередко применяющегося в данной задаче подхода LSH Forest (locality sensitive hashing forest)

Вычисления на больших данных стали неотъемлемой частью компьютерных наук. Оперирование объёмными массивами данных требует не только наличия современных аппаратных средств, но также и математических алгоритмов, которые позволили бы сократить необходимое число вычислительных операций для компьютера. Одна из наиважнейших проблем работы с большими данными - поиск похожих множеств и определение их мощности, то есть подсчёт количества уникальных элементов в них. Один из наиболее современных методов решения этой задачи - сопоставление каждому из сравниваемых документов или множест так называемых скетчей - объектов как правило фиксированного размера, которые сравниваются между собой. А именно: для вычисления индекса Жаккара двух множеств генерируются скетчи и для них вычисляется искомая величина. Как правило, размер скетча зависит от количества случайных перестановок, на практике хэш функций. Таким образом решается проблема больших размеров сравниваемых множеств, каждая операция сравнения затрачивает гораздо меньше ресурсов. Однако всё ещё не решается проблема количества этих сравнений. Пусть чтобы выянисть похожесть множеств требуются миллисекунды, но объём базы данных этих множеств может насчитывать сотни миллионов объектов, и искомый объект необходимо сравнить с каждым из них, таким образом количество сравнений никак не меняется. Решить эту проблему помогают индексы, которые аггрегируют в себе все имеющиеся объекты и упрощают поиск похожих множеств. Целью этой работы является сравнение индексов и определение их производительных характеристик.

Многие задачи, которые трудно решить точно, достаточно просто решаются с помощью вероятностных алгоритмов. Согласованное хеширование (consistent hashing) часто применяется для балансировки нагрузки и в распределенных базах данных. Алгоритм Mincut может быть использован для решения задач в области сетевой маршрутизации. Фильтры Блума (Bloom filter) присутствуют во многих СУБД и сетевых приложениях. Алгоритм быстрой сортировки (quicksort) на сегодняшний день является стандартом. Даже обычные хеш-таблицы, на самом деле, являются вероятностными. В своей повседневной работе программист постоянно имеет дело с вероятностными алгоритмами и структурами данных. Обычно они незаметны, потому что просто работают.

Действительно, вероятностные алгоритмы повсюду вокруг нас. Они позволяют решать сложные задачи без дополнительного оборудования. Кроме того, некоторые задачи невозможно решить без их помощи. Вероятностные алгоритмы должны быть обязательным инструментом в арсенале разработчика крупномасштабных систем.

2 Литературный обзор

2.1 MinHash

Minhash - вероятностная структура данных для подстчёта индекса Жаккара двух множеств большой мощности. Каждому множеству сопоставляется набор так называемых 'shingles' - объектов определённой структуры, характеризующих соответствующее им множество. Например, если множества представляют собой некие статьи, то в качестве одного shingle можно взять строку трёх идущих подряд слов. Для удобства можно захешировать эту строку. Тогда множество всевозможных троек идущих подряд слов, обработанных хеш функцией, и будет множеством shingles.

В ходе алгоритма используется сигнатура - объект фиксированного по памяти размера, сопоставляемая каждому из сравниваемых множеств. В целях

эффективности рекомендуется использовать сигнатуры небольшого размера, около 10-ти компонент. Каждая компонента есть минимальное значение фиксированной хеш-функции, применённой на множестве shingles. Тогда вторая компонента - минимальное значение второй фиксированной хеш-функции и так далее.

Чтобы посчитать индекс Жаккара для двух множеств, необходимо сравнить соответствующие им сигнатуры, тогда количество совпадающих компонент, делённое на их общее число (размер сигнатуры), есть приближение индекса Жаккара. Это справедливо потому, что вероятность, что одна компонента множества будет соответствовать другой компоненте множетсва и есть индекс Жаккара. Хеш-функция здесь выполняет роль случайной перестановки, то есть сопоставляет строке (в рассмотренном выше примере это строка из трёх идущих подряд слов) число, то есть реализует случайную перестановку. Такой подход работает значительно быстрее прямого подсчёта отношения мощности пересечения множеств к их объединению.

2.2 R-дерево

R-дерево — структура данных, представленная Антонином Гуттманом в 1984 году в [1]. Она похожа на бинарное дерево, но применяется для работы с пространственными данными, такими, например, как географические точки. Оно разделяет многомерное пространство на несколько вложенных и, вероятно, пересекающихся, гиперплоскостей.

Алгоритмы вставки и удаления используют эти ограничивающие прямоугольники для обеспечения того, чтобы «близкорасположенные» объекты были помещены в одну листовую вершину. В частности, новый объект попадёт в ту листовую вершину, для которой потребуется наименьшее расширение её ограничивающего прямоугольника. Каждый элемент листовой вершины хранит два поля данных: способ идентификации данных, описывающих объект, (либо сами эти данные) и ограничивающий прямоугольник этого объекта.

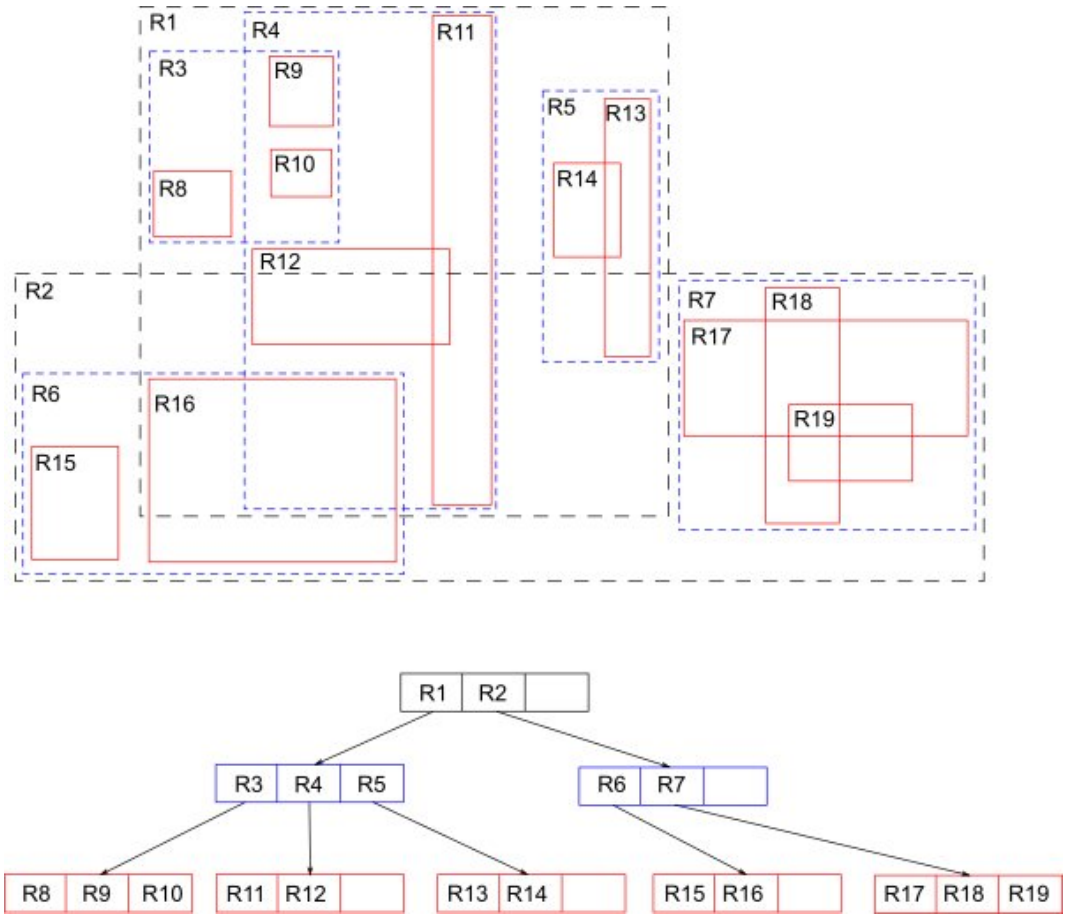


Рис. 1: Пример R-дерева.

Процесс запроса использует эти элементарные объекты, чтобы понять, нужно ли искать в дочерней вершине. Применяя такой подход, большинство вершин не используются в ходе поиска. Аналогично бинарным деревьям, такая характеристика R деревьев делает их эффективными для баз данных.

2.3 BallTree

K-NN - алгоритм классификации, концептуально один из самых простых для понимания. Его также называют «ленивым учеником» в сравнении с «нетерпеливым учеником». Большинство алгоритмов классификации - нетерпеливые ученики; существует набор учебных данных с примерами классификаций. Данные обучения используются для построения модели классификации. Модель используется для оценки тестовых данных, где известны классификации дан-

ных. Если оцениваемые результаты являются удовлетворительными, конечная модель затем используется для прогнозирования классов по данным с неизвестными классификациями. Поэтому заинтересованные ученики уже сделали большую часть своей работы по разработке модели заранее. С другой стороны, ленивый ученик не разрабатывает какую-либо модель заранее; он ждет данные, а затем прокладывает путь через алгоритм для прогнозирования классификации. Поэтому ленивые ученики занимают много времени - каждый раз, когда нужно сделать прогноз, все усилия по созданию модели должны быть выполнены снова.

$$D^B(t) = \begin{cases} \max(|t - B.pivot| - B.radius, D^{B.parent}), & \text{if } B \neq Root \\ \max(|t - B.pivot| - B.radius, 0), & \text{if } B = Root \end{cases}$$

Где $D^B(t)$

Минимально возможное расстояние от произвольной точки гиперсферы B до некоторой точки t

В алгоритме k-ближайшего соседа данные примера сначала отображаются в n-мерном пространстве, где «n» - количество атрибутов данных. Каждая точка в n-мерном пространстве помечена своим значением класса. Чтобы обнаружить классификацию неклассифицированных данных, на этом n-мерном пространстве нанесена точка, и отмечены метки классов ближайших k точек данных. Обычно k является нечетным числом. Этот класс, который встречается для максимального количества раз среди k ближайших точек данных, берется как класс новой точки данных. То есть решение - путем голосования по соседним точкам. Одним из больших преимуществ этого универсального алгоритма K-Nearest Neighbor для обнаружения классификации является то, что он поддается параллельным операциям.

В общей модели k-NN каждый раз, когда нужно предсказать предсказание для точки данных, сначала необходимо определить расстояние этой точки данных от всех других точек, и тогда для голосования могут быть обнаруже-

ны только ближайшие k-точки. Этот подход также известен как подход грубой силы. Когда объем данных огромен и его размер также очень велик, скажем, в сотнях или тысячах, эти повторные вычисления расстояния могут быть очень утомительными и трудоемкими. Чтобы закрепить этот процесс и чтобы избежать измерения расстояний от всех точек набора данных, делается предварительная подготовка учебных данных. Эта обработка помогает найти точки, которые, вероятно, будут находиться по соседству.

Один из способов - создать отсортированную иерархическую структуру данных, называемую деревом k-d или k-мерным деревом. K-мерное дерево является двоичным деревом.

Структура данных BallTree очень эффективна, особенно в ситуациях, когда количество измерений очень велико. BallTree также является двоичным деревом с иерархической (двоичной) структурой. Создаются два кластера (каждый из которых напоминает шар). Поскольку это многомерное пространство, каждый шар может быть соответственно назван гиперсферой. Любая точка в n-мерном пространстве будет принадлежать любому одному кластеру, но не к двум. Он будет принадлежать кластеру, чей центр (расстояние) меньше. Если расстояние от этой точки до центроидов обоих шаров одинаково, оно может быть включено в любой из кластеров. Возможно, что обе (виртуальные) гиперсферы могут пересекаться, но точки будут принадлежать только одному из двух. Затем каждый из шаров снова подразделяется на два подкластера, каждый из которых напоминает шар; что означает, что в этих подкластерах снова есть две центроиды, а нахождение в точке на шаре определяется исходя из его расстояния от центра тяжести субкластера. Мы снова разделим каждый из этих шаров на подшар и так далее до определенной глубины.

Неклассифицированная (целевая) точка должна попадать в любой из вложенных шаров. Ожидается, что точки в пределах этого гнездового шара будут ближе к целевой точке. Точки в других соседних шарах (или охватывающих шарах) также могут быть ближе к нему (например, эта точка может находиться на границе одного из шаров.) Тем не менее, не нужно рассчитывать расстоя-

ние этой точки от всех точек в n-мерном пространстве. Это ускоряет процесс классификации. Первоначально образование BallTree требует много времени и памяти, но как только вложенные гиперсферы создаются и помещаются в память, открытие ближайших точек становится проще.

2.4 KDTree

KD-Tree(K-мерное дерево), специальная структура данных, которая позволяет разбить K-мерное пространство на 'меньшие части', посредством сечения этого самого пространства гиперплоскостями($K > 3$), плоскостями ($K = 3$), прямыми ($K = 2$) ну, и в случае одномерного пространства-точкой (выполняя поиск в таком дереве, получаем что-то похожее на binary search).

Логично, что такое разбиение обычно используют для сужения диапазона поиска в K-мерном пространстве. Например, поиск ближнего объекта (вершины, сферы, треугольника и т.д.) к точке, проецирование точек на 3D сетку, трассировка лучей (активно используется в Ray Tracing) и т.п. При этом объекты пространства помещаются в специальные параллелепипеды-bounding box-ы(bounding box-ом назовем такой параллелепипед, который описывает исходное множество объектов или сам объект, если мы строим bounding box лишь для него. У точек в качестве bounding box-а берется bounding box с нулевой площадью поверхности и нулевым объемом), стороны которых параллельны осям координат.

Итак, прежде чем использовать KD-Дерево, нужно его построить. Все объекты помещаются в один большой bounding box, описывающий объекты исходного множества (при этом каждый объект ограничен своим bounding box-ом), который затем разбивается (делится) плоскостью, параллельной одной из его сторон, на два. В дерево добавляются два новых узла. Таким же образом каждый из полученных параллелепипедов разбивается на два и т.д. Процесс завершается либо по специальному критерию (см. SAH), либо при достижении определенной глубины дерева, либо же при достижении определенного числа

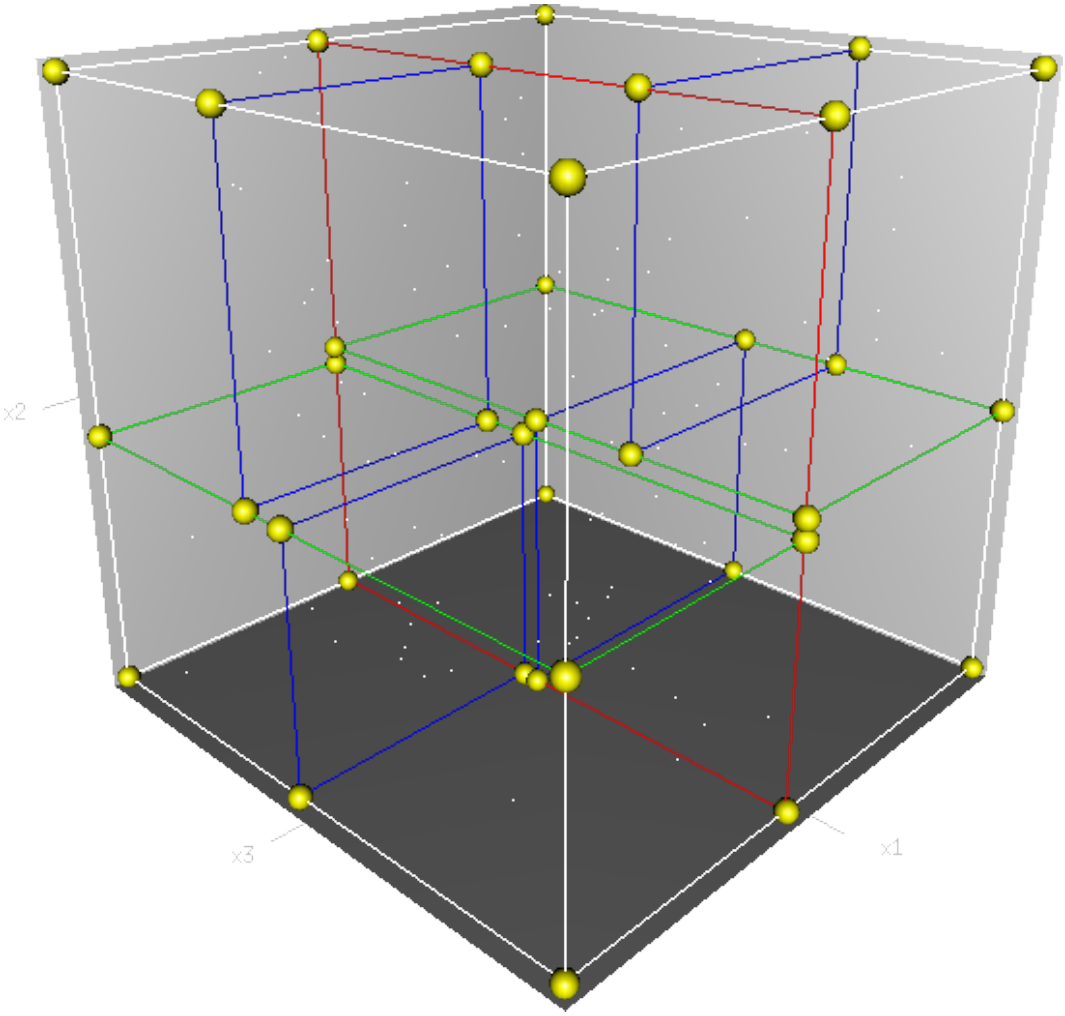


Рис. 2: Трёхмерное KD-дерево.

элементов внутри узла дерева. Некоторые элементы могут одновременно входить как в левый, так и в правый узлы (например, когда в качестве элементов дерева рассматриваются треугольники).

Дерево строится от корня. В каждом узле дерева хранятся указатели на левое и правое поддеревья, если таковых у узла нет, то он считается листовым (листом). Каждый узел хранит bounding box, который описывает объекты данного узла. В листовых(и только в листовых!) узлах хранятся индексы тех объектов, которые входят в данный узел. Однако, в процессе построения память под узлы выделяется порциями (т.е. сразу для K узлов: во-первых, так эффективнее работать с менеджером памяти, во-вторых,-подряд идущие элементы идеальны для кэширования). Такой подход запрещает хранить узлы де-

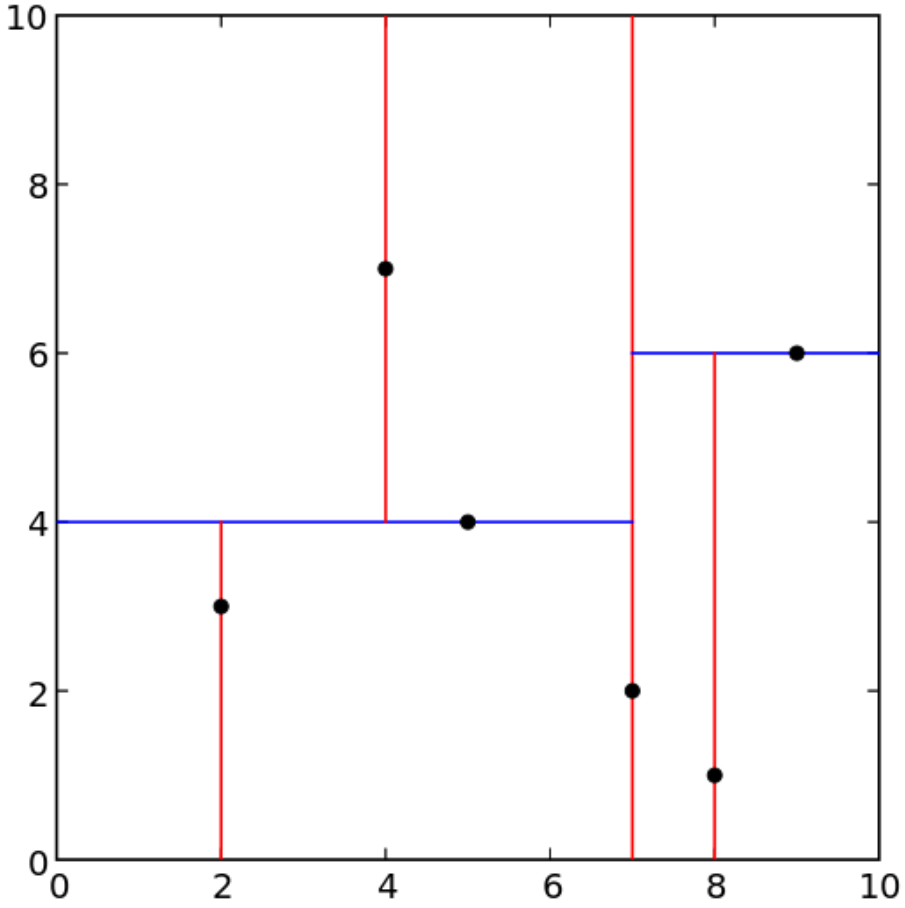


Рис. 3: Декомпозиция KD-дерева для множества точек (2,3), (5,4), (9,6), (4,7), (8,1), (7,2).

рева в векторе, ибо добавление новых элементов в вектор может приводить к реаллоцированию памяти под все существующие элементы в другое место.

Итак, дерево построено, перейдем к описанию реализации операции поиска в KD-Дереве.

Предположим, что поиск мы осуществляем в множестве треугольников: дана точка, требуется найти самый ближний к ней треугольник.

Решать поставленную задачу с применением bruteforce невыгодно: для множества из N точек и M треугольников ассимптотика составит O(N * M). Кроме

того, хотелось бы, чтобы алгоритм вычислял расстояние от точки до треугольника 'по-честному' и делал это быстро.

Выполняя поиск ближнего в узле, необходимо уметь быстро вычислять расстояние от точки до треугольника. Опишем простейший алгоритм:

Находим проекцию точки A (точка, к которой мы и ищем ближний) на плоскости треугольника. Найденную точку обозначим P. Если P лежит внутри треугольника, то расстояние от A до треугольника равно длине отрезка AP, иначе найдем расстояния от A до каждой из сторон (отрезков) треугольника, выберем минимум. Задача решена.

Описанный алгоритм является не самым эффективным. Более эффективный подход опирается на поиск и анализ (нахождение минимума градиента и т.п.) функции, значениями которой являются все возможные расстояния от данной точки до любой точки в треугольнике. Метод довольно сложен в восприятии, ознакомиться с ним можно по [1]. Данный метод по результатам тестов оказался в 10 раз быстрее того, что описывалось ранее.

Определить находится ли сфера с центром в точке O и радиусом R внутри конкретного узла, представленного bounding box, просто(3D).

Сперва мы сводим вычисления 4-ёх квадрантов к одному. Потом проверяем, лежит ли окружность в зеленой области. Если лежит, то пересечения нет. Затем проверим, входит ли окружность в оранжевую или серую области рисунка. Если входит, то пересечение обнаружено.

Дальше необходимо проверить, пересекает ли окружность угол прямоугольника (это делают следующие строки кода).

Осталось рассказать о скорости работы алгоритма поиска, а также о критичных ситуациях, замедляющих поиск в KD-Дереве.

Как было сказано выше, 'верные' ситуации порождают большое число элементов внутри узла, чем их больше-тем медленнее поиск. Более того, если все

элементы равноудалены от данной точки, то поиск будет осуществляться за O(N)(множество точек, которые лежат на поверхности сферы, а ближний ищем к центру сферы). Однако, если данные ситуации убрать, то поиск в среднем будет равносилен спуску по дереву с перебором элементов в нескольких узлах т.е. за O(log(N)). Понятно, что скорость работы поиска зависит от числа посещенных листовых узлов дерева. Пусть A - средняя длина поиска в одном пространстве, n - количество элементов, h - высота дерева, тогда

$$A_n = O(\frac{2^h(h-1)+1}{2^h-1}) = O(h\frac{2^h}{2^h-1}-1) = O(log(\frac{n}{N}+1)\frac{2^{log(\frac{n}{N}+1)}}{2^{log(\frac{n}{N}+1)}-1} - 1) = O(log(\frac{n}{N}+1)\frac{n+N}{n}-1) = = O(log(\frac{n}{N}+1)^{\frac{n+N}{n}}-1) = O(log(\frac{n}{N}+1)^{\frac{n+N}{n}}-1)$$

где N — количество элементов в узле.

Итак, KD-дерево разбивает пространство на области (каждая вершина производит разбиение по определенной координате), и каждый лист соответствует одному объекту из обучающей выборки. Обходя это дерево определенным образом, можно найти точку из обучения, ближайшую к заданной. Если размерность пространства небольшая (10-20), то данный подход позволяет находить ближайшего соседа за время порядка O(log ℓ). Экспериментально было установлено, что в пространствах большой размерности сложность поиска ближайшего соседа в kd-дереве сильно ухудшается и приобретает линейный порядок сложности.

2.5 FAISS

Индекс FAISS был представлен компанией Facebook в 2017 году в [4]. Однао идея, на которой он основывается, была представлена ещё в 2011 году в [5]. Эта идея состоит в том, чтобы разложить пространство на декартово произведение низкоразмерных подпространств и обрабатывать каждое подпространство отдельно. Вектор представлен коротким кодом, состоящим из индексов обработки подпространства. Евклидовое расстояние между двумя векторами может быть эффективно рассчитано по их кодам. Асимметричная версия увеличивает точ-

ность, поскольку она вычисляет приблизительное расстояние между вектором и кодом. Экспериментальные результаты показывают, что такой подход эффективно ищет ближайших соседей, в частности в сочетании с обратной файловой системой. Результаты для SIFT и дескрипторов изображений GIST показывают отличную точность поиска, опережая три современных подхода. Масштабируемость такогоподхода проверяется на наборе данных из двух миллиардов векторов.

2.6 LSH

Locality-sensitive hashing (LSH) — вероятностный метод понижения размерности многомерных данных. Основная идея состоит в таком подборе хеш-функций для некоторых измерений, чтобы похожие объекты с высокой степенью вероятности попадали в одну корзину[6]. Один из способов борьбы с «проклятием размерности» при поиске и анализе многомерных данных — при росте размерности исходных данных поиск по индексу ведет себя хуже чем последовательный просмотр. Метод позволяет строить структуру для быстрого приближенного (вероятностного) поиска n-мерных векторов, «похожих» на искомый шаблон.

LSH является, наверное, наиболее популярным на сегодняшний день среди предложенных приближенных алгоритмов поиска ближайших соседей (Approximate Nearest Neighbor, ANN). LSH в этом подходе отображает множество точек в высокоразмерном пространстве в множество бинов, т. е. в хеш-таблицу. В отличие от традиционных хэшей, LSH обладает свойством чуствительности к местоположению (locality-sensitive hash), благодаря чему способен помещать соседние точки в один и тот же бин.

Преимуществами LSH являются: 1) простота использования; 2) строгая теория, подтверждающая хорошую производительность алгоритма; 3) LSH совместим с любой нормой $L_p L_p$ при $0 < p \leq 20 < p \leq 2$. LSH можно использовать с евклидовой метрикой, с манхэттенским расстоянием. Существуют также варианты для расстояния Хэмминга и косинусного коэффициента.

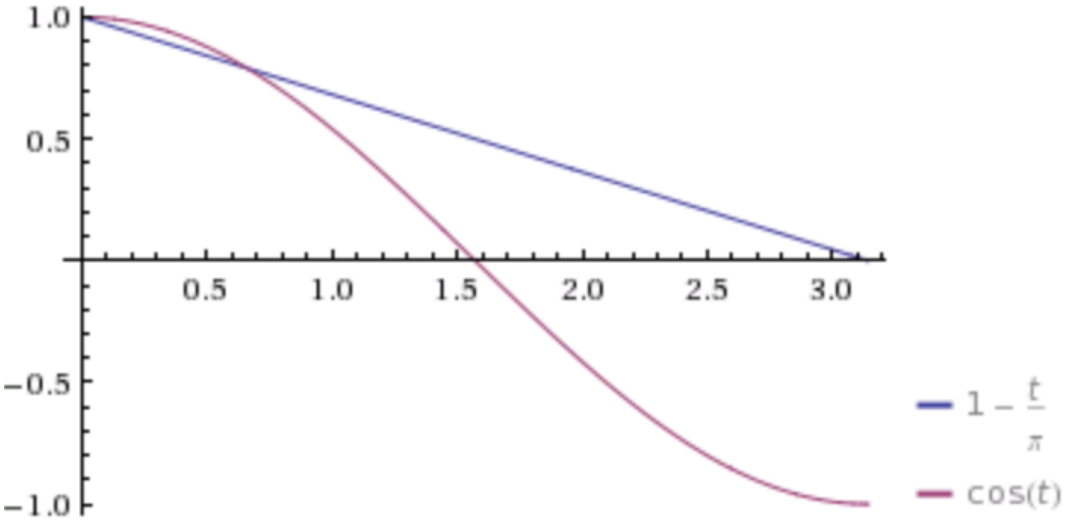


Рис. 4: Аппроксимация косинуса в подходе LSH

Идея LSH противоположна идее традиционного хеширования. При традиционном хешировании два входных значения, являющиеся похожими, но не одинаковыми, должны давать на выходе существенно различные результаты. В рамках концепции LSH похожие входные значения должны давать похожие выходные значения, имеющие значительно меньшую размерность. На практике с помощью LSH можно преобразовывать 1000000-мерные векторы в 512-битные хеши с хорошими результатами.

Очевидно, что если вектор большой размерности преобразуется в маленький хеш, часть данных теряется. Здесь на помощь приходит вероятность.

Существует ряд различных способов, позволяющих сгенерировать LSH-хеш. Рассмотренный ниже вариант является одним из простейших.

Допустим, мы имеем корпус книжных текстов. Пусть мы уже извлекли словарь, содержащий 100000 слов. Следовательно, вектор признаков для каждой книги будет иметь размерность 100000. Предположим, что мы решили создать 256-битную LSH-функцию. Для этого сгенерируем 256 абсолютно случайных 100000-мерных векторов. Эти случайные векторы являются основой нашей LSH-функции.

LSH-функция вычисляет скалярное произведение входного вектора и каж-

дого из 256 случайных векторов. Затем на основании знаков скалярных произведений формируется 256-битный хеш. Если скалярное произведение входного вектора и данного случайного вектора больше нуля – в соответствующий бит хеша записывается единица, если меньше нуля – ноль.

Выполняя описанные действия, мы аппроксимируем угол входного вектора в векторном пространстве. Следовательно, мы можем оценить косинусный коэффициент двух любых хешей, сгенерированных данной LSH-функцией. Косинусный коэффициент двух книг будет пропорционален расстоянию Хэмминга (Hamming distance) между их LSH-хешами.

2.7 Annoy

Мы собираемся построить дерево, которое позволяет нам делать запросы за O(log N), так работает Annoy. Фактически, это двоичное дерево, где каждый узел является случайным разбиением. Разделяем пространство один раз:

Annoy делает это, произвольно выбирая две точки, а затем расщепляя гиперплоскость на равном расстоянии от этих двух точек. Две точки обозначены серой линией, а гиперплоскость - толстой черной линией. Продолжим разбиение каждого подпространства рекурсивно.

Annoy делает это, произвольно выбирая две точки, а затем расщепляя гиперплоскость на равном расстоянии от этих двух точек. Две точки обозначены серой линией, а гиперплоскость - толстой черной линией. Продолжим разбиение каждого подпространства рекурсивно.

Пока ещё небольшое бинарное дерево начинает обретать очертания.

Мы продолжаем делать это, пока в каждом узле не останется элементов K. Сейчас пространство выглядит примерно так (для K = 10):

Ему соответсвует такое бинарное дерево:

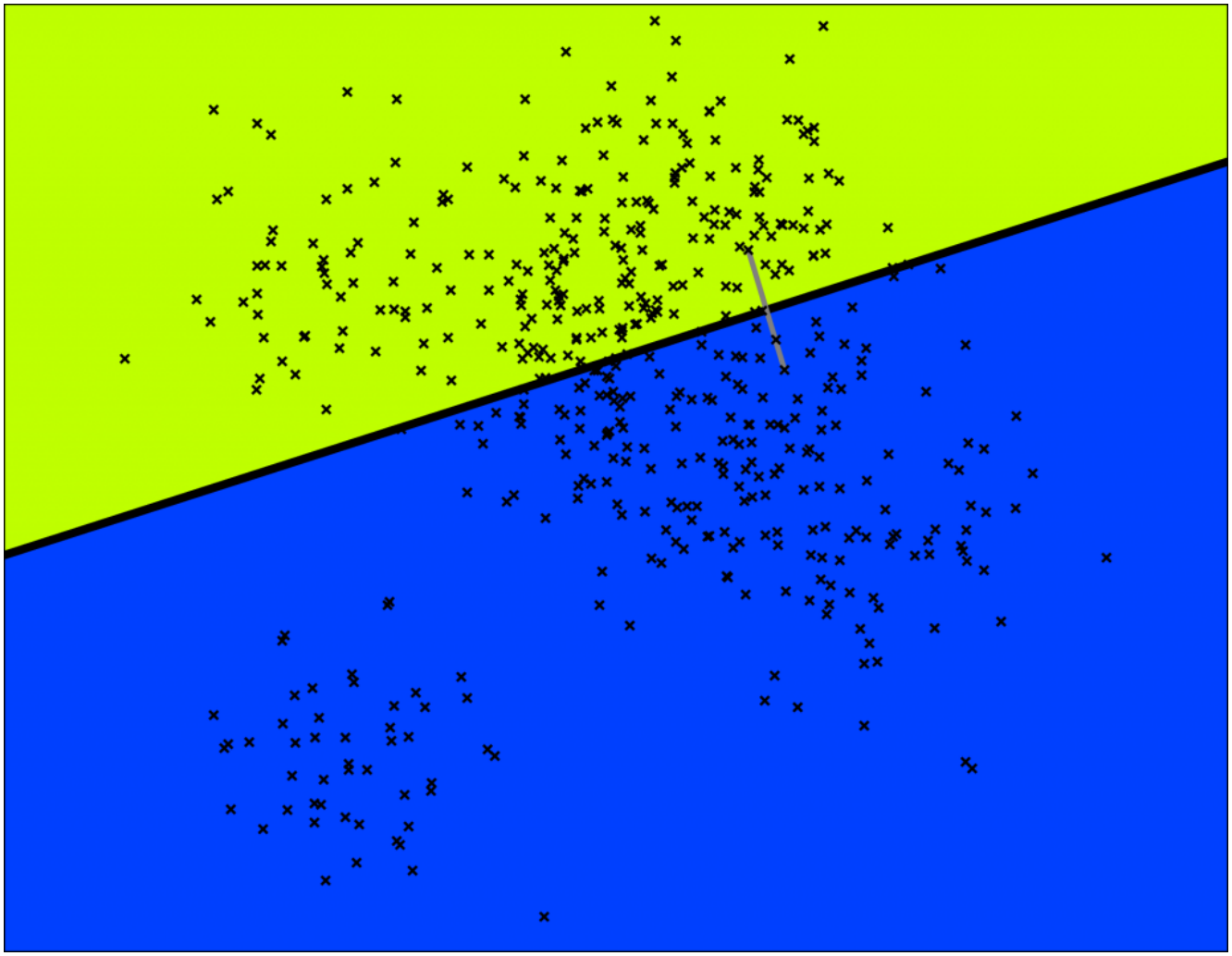


Рис. 5: Первое разделение пространства

В итоге мы получаем двоичное дерево, разбивающее пространство. Обратим внимание, что точки, которые находятся близко друг к другу в пространстве, скорее всего, будут близки друг к другу в дереве. Другими словами, если две точки находятся близко друг к другу в пространстве, маловероятно, что какая-либо гиперплоскость будет их разделять.

Для поиска любой точки в этом пространстве мы можем обойти двоичное дерево из корня. Каждый промежуточный узел (малые квадраты в дереве выше) определяет гиперплоскость, поэтому мы можем выяснить, какую сторону гиперплоскости нам нужно выбрать, что и определяет, будем ли мы двигаться к левому или правому дочернему узлу. Поиск точки может быть выполнен за логарифмическое время, так как это высота дерева.

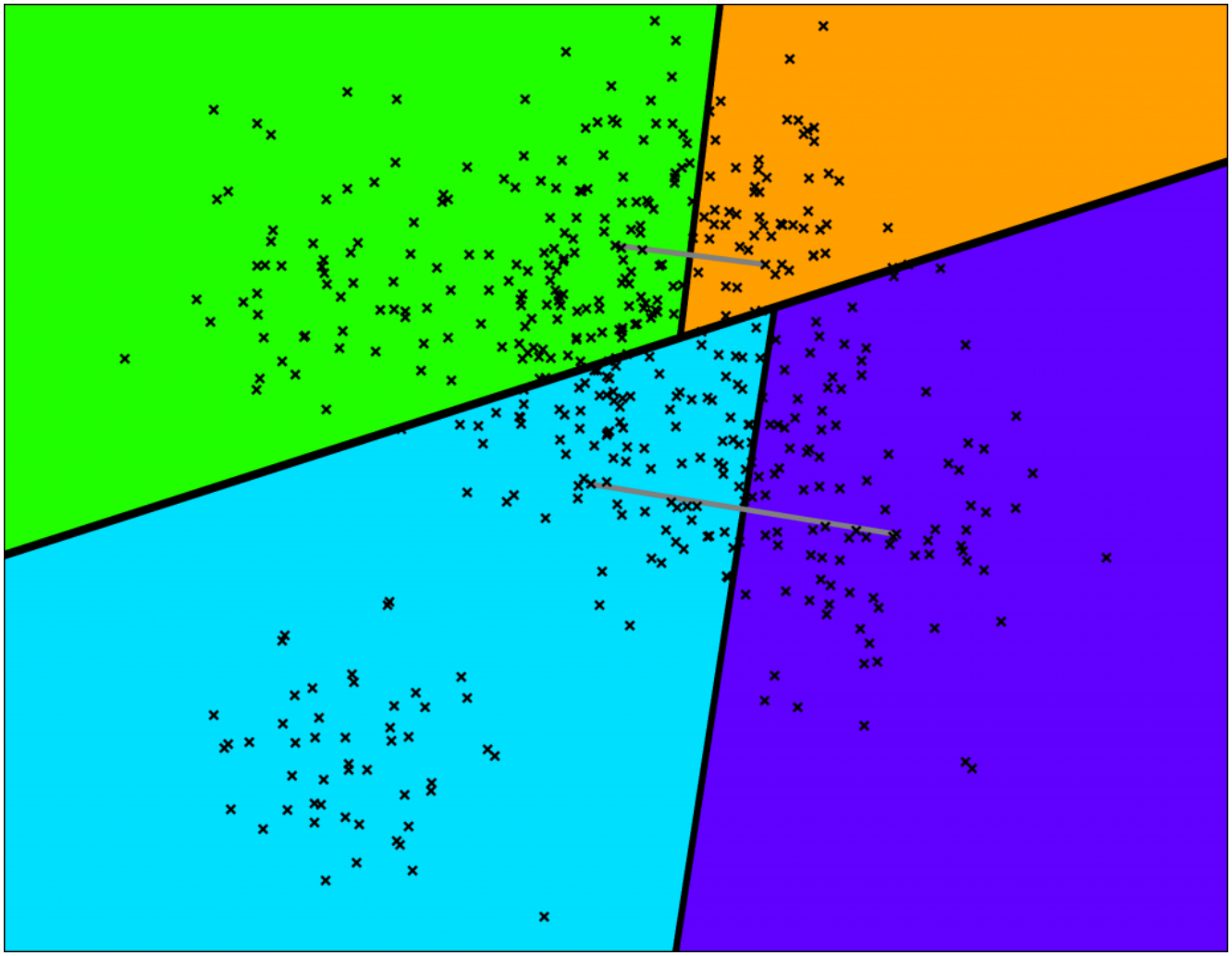


Рис. 6: Шаг 2 разделения пространства

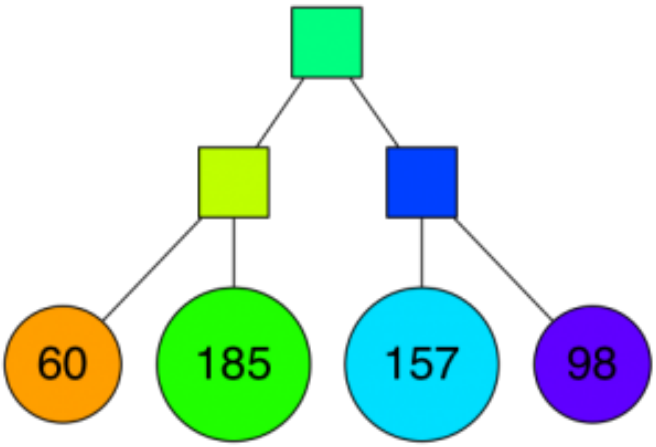


Рис. 7: Бинарное дерево на текущем этапе

Найдём точку, обозначенную красным X на приведенном ниже рисунке:

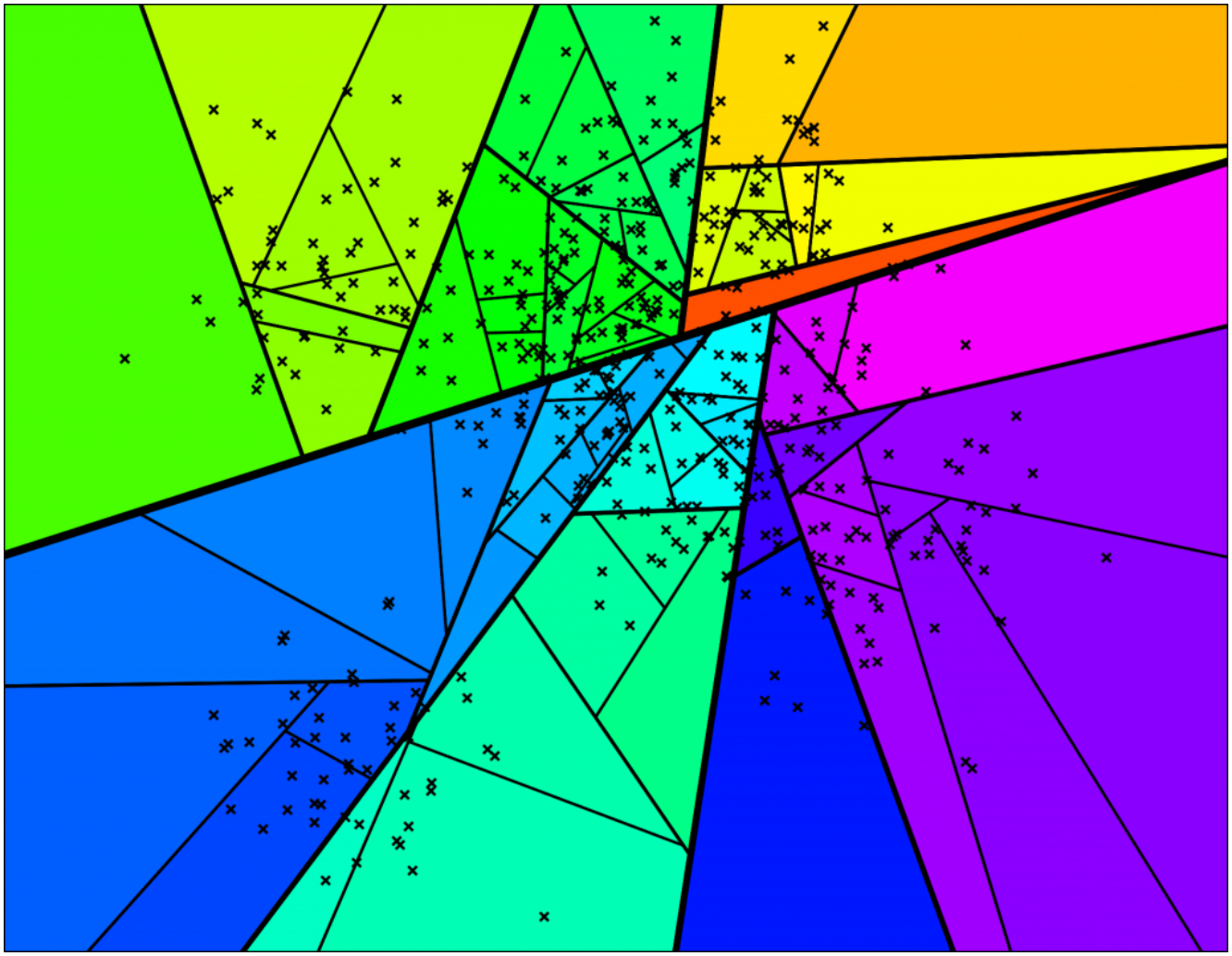


Рис. 8: Пространство

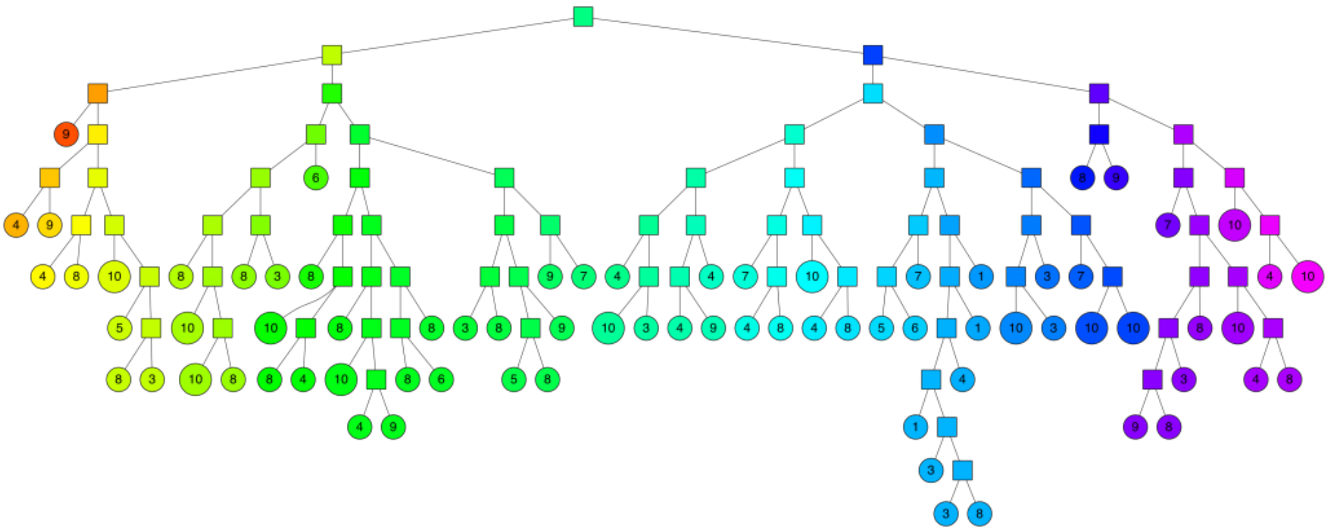


Рис. 9: Бинарное дерево

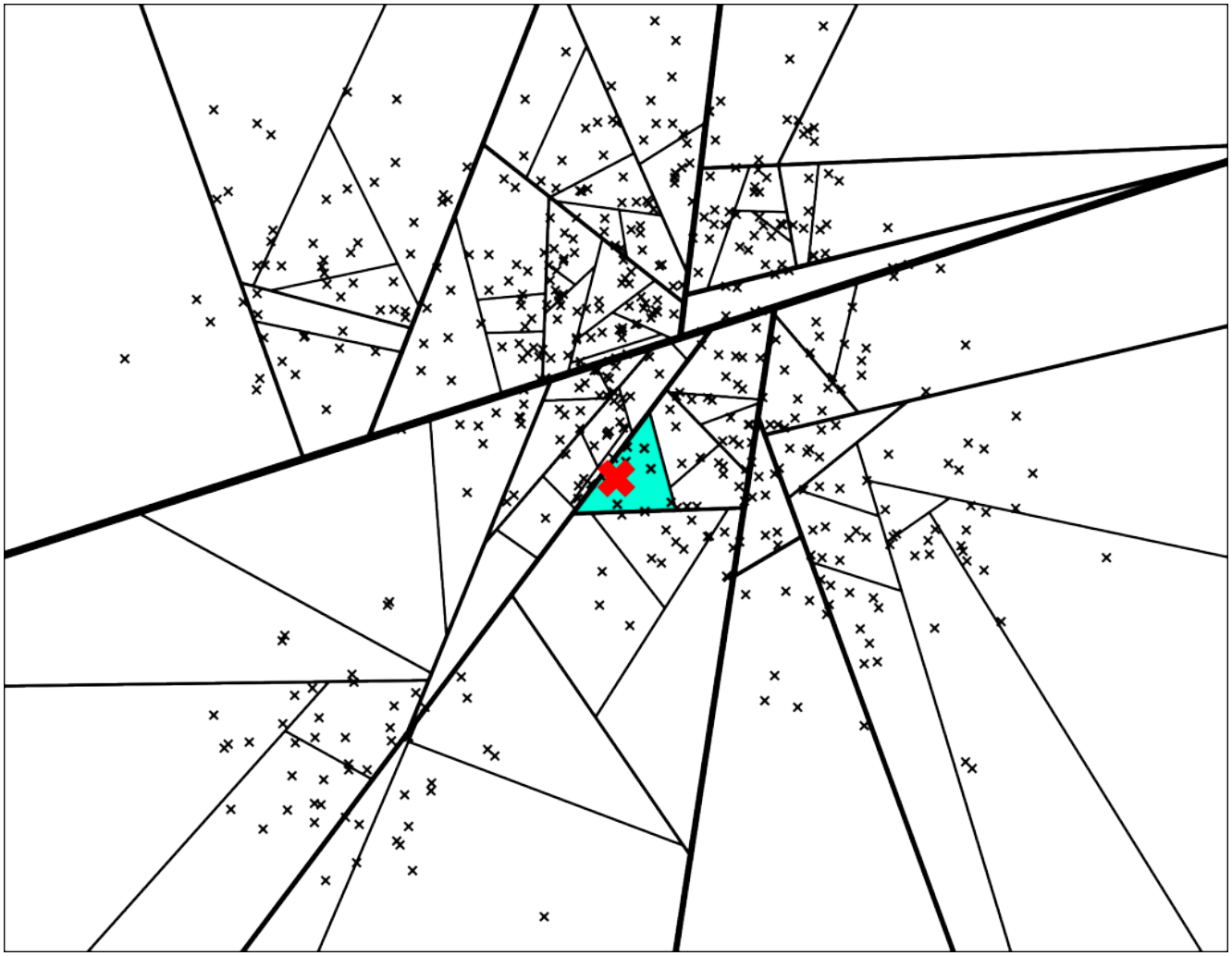


Рис. 10: Искомая точка

В итоге у нас 7 ближайших соседей. Однако текущий процесс имеет несколько недостатков. В данном случае количество соседей оказывается фиксированным. Во вторых, некоторые из ближайших соседей фактически находятся за пределами этого листового многоугольника.

Введём ряд поправок к текущему алгоритму. Во-первых, будем использовать очередь с приоритетом. Смысл состоит в том, чтобы спуститься по обе стороны от узла, если мы «достаточно близки». Вместо того, чтобы просто спуститься по одному пути двоичного дерева, рассмотрим еще несколько. Мы можем настроить порог того, насколько мы готовы пойти на «неправильную» сторону раскола. Если пороговое значение равно 0, мы всегда будем идти на «правильную» сторону разделения. Однако, если мы установим порог в 0,5,

получим путь поиска с расщеплением.

Суть в том, чтобы использовать очередь приоритетов для изучения узлов, отсортированных по максимальному расстоянию до «неправильной» стороны. Преимущество в том, что мы можем искать все более большие и большие пороги, начиная с 0.

Во-вторых, построим лес деревьев (forest of trees). Его суть состоит в том, чтобы построить много деревьев, образующих лес. Каждое дерево построено с использованием случайного набора расщеплений. Будем искать все эти деревья одновременно. Можем одновременно искать все деревья, используя одну очередь приоритетов. В этом заключается дополнительное преимущество - поиск будет сфокусирован на деревьях, которые являются «лучшими» для каждого запроса, - расщепления, которые наиболее удалены от точки запроса.

Каждое дерево содержит все точки, поэтому при поиске нескольких деревьев мы найдем несколько точек в нескольких деревьях. Если мы посмотрим на объединение листовых узлов, мы получим довольно хорошую область:

На этом шаге мы привязвли ее к небольшому набору точек. Обратим внимание, что до сих пор мы даже не вычисляли расстояния до одной точки. Следующим шагом является вычисление всех расстояний и ранжирование точек.

Затем мы сортируем все узлы по расстоянию и возвращаем верхние K ближайших соседей. Так работает алгоритм поиска в Annoy.

Отметим одну особенность алгоритма. Оказывается, что мы действительно пропустили пару соседей за пределами области.

Но A в Annoy означает приблизительно (approximate) и недобор нескольких соседей допустим. У Annoy есть параметр, который можно настроить ($search_k$), который позволяет обменивать производительность (время) на точность (качество).

Вся идея приближенных алгоритмов заключается в том, чтобы жертвовать

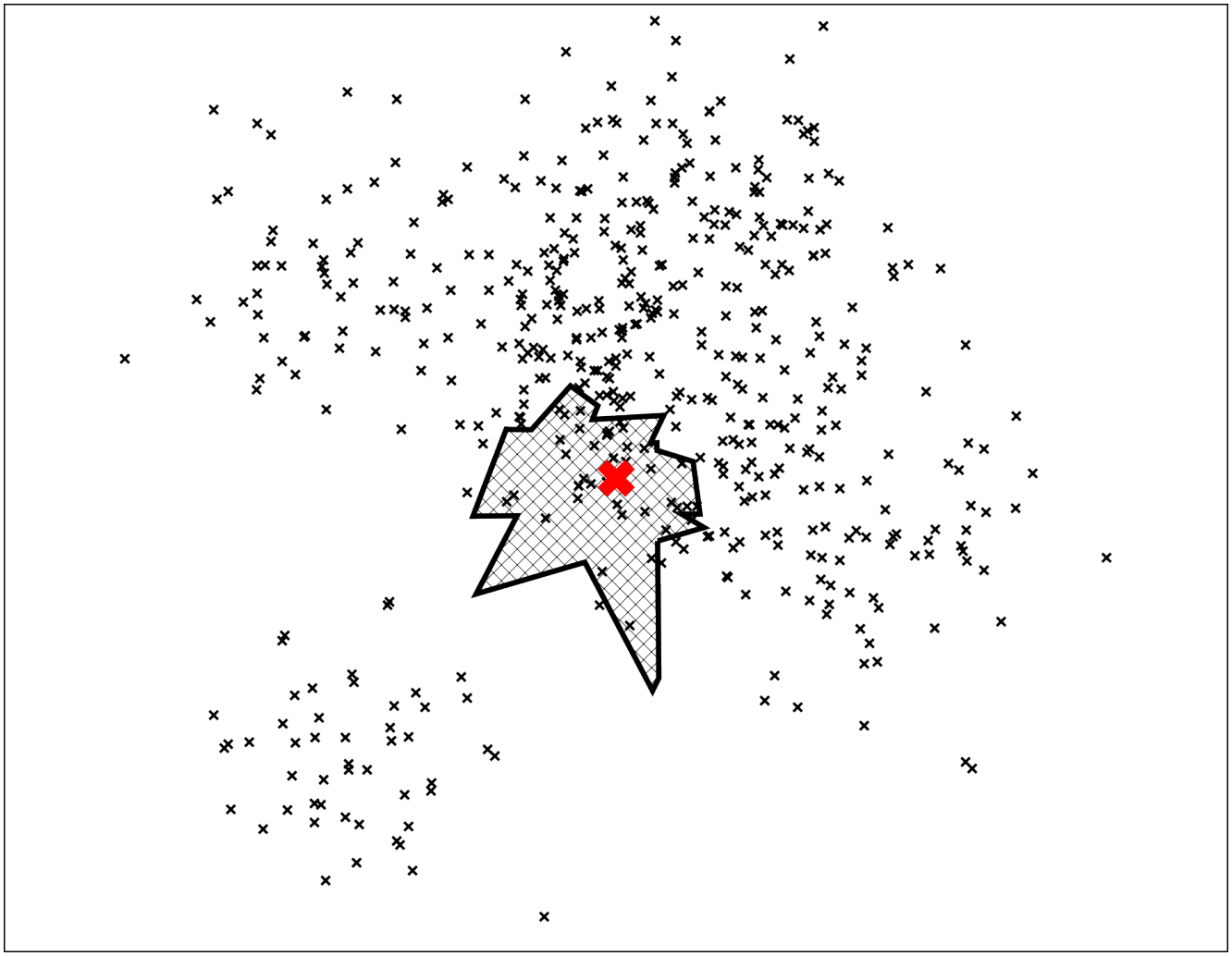


Рис. 11: Область ближайших соседей

небольшим количеством точности ради огромного прироста производительности (на порядок). Например, мы могли бы вернуть достойное решение, в котором мы действительно вычисляли расстояние только на 1

Больше деревьев всегда улучшает общую ситуацию с алгоритмом. Добавив больше деревьев, мы дадим Annoy больше шансов найти благоприятные расколы.

2.8 LSH Forest

LSH Forest - популярная структура данных для поиска ближайшего соседа. В [7] автор показывает, что его несложная модификация превосходит многие ал-

горитмы LSH. В этй же статье автор представляет экземпляр простого, практического алгоритма, который использует хеширование, зависящее от данных, для улучшения показателей LSH.

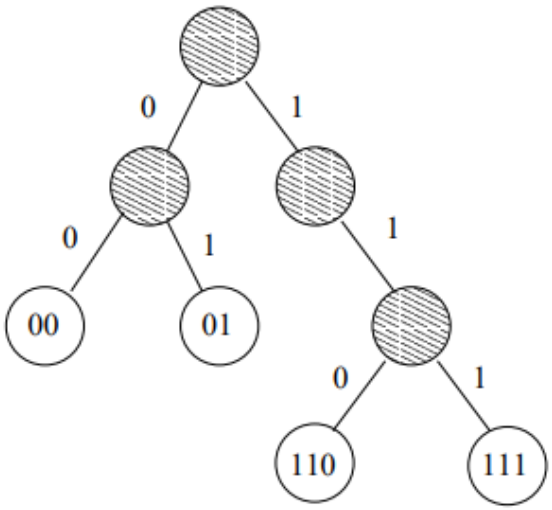
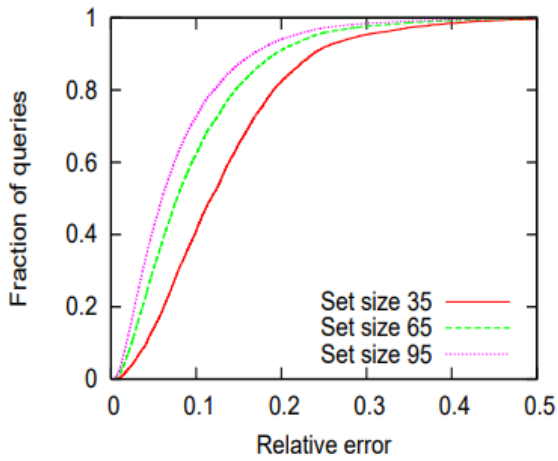


Рис. 12: Префиксное дерево LSH меток



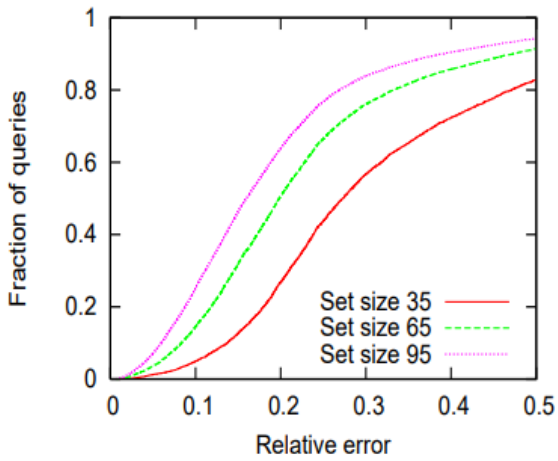


Рис. 13: Относительная ошибка метода

Вот полный алгоритм для d-мерного хэммингова пространства. LSH Forest для данного набора данных применяет случайную перестановку ко всем d-координатам, и создает метку trie на результирующих строках. В модификации

автора он дополняет эту метку: для каждого узла, сохраняет постоянное число точек, близких к среднему из соответствующего подмножества набора данных, которые сравниваются с точкой запроса, достигающей узел. Общая структура данных - это просто несколько таких меток trie, отобранных независимо.

3 Метод

Метод сравнения индексов, описанный в этой работе, прост и зависит только от самих выбранных структур данных. Программное обеспечение, использованное в ходе эксперимента:

- Операционная система Ubuntu 16.04 LTS
- Python 3.5
- Библиотека datasketch 1.2.3
- Библиотека nmslib 1.7.5,
- Библиотека scikit-learn 0.19.1
- Зависимости вышеуказанных библиотек (как правило, устанавливаются автоматически)

Основная идея метода состоит в сборе показателей эффективности каждого из индексов: объём памяти, использованной для индекса, время, необходимое, чтобы сделать запрос к измеряемому индесу, его качество recall.

На первом шаге процесса сравнения необходимо построить MinHash скетчи для каждого множества из рассматриваемого датасета. Используются 128 хэш-функций, то есть 128 случайных перестановок, поэтому скетч будет представлять собой 128 чисел. На текущем шаге новый датасет, состоящий из скетчей исходных множеств, сгенерирован и его можно направить на вход платформе,

чтобы разбить его на тренировочную и тестовую части (train/test) и подсчитать истинные расстояния между множествами в рассматриваемом наборе данных, которые будут использованы в дальнейшем для оценки качества recall каждого из индексов. Когда этот процесс будет закончен, можно интегрировать алгоритмы (индексы) в платформу и производить вычисления, то есть делать запросы (множества из test-датасета) к текущему индексу.

Основой для кодовой базы данного отчета является платформа сравнения ann-алгоритмов из этого источника [8]. Автор этой библиотеки задумал сравнение индексов и по метрике Жаккара тоже, однако, видимо, не тестировал ее, поэтому пришлось адаптировать эту платформу под функционирование в необходимой метрике, подключить свой датасет и наладить работу с базой данных (для индекса LSHForest).

Важно уточнить, что для репрезентативности результатов в качестве recall понимается следующее: выполняется запрос к индексу для поиска 10-ти ближайших соседей, затем вычисляется, какое количество этих самых соседей попадает в первую сотню истинных ближайших соседей. Число 100 разумно в рамках этих экспериментов, потому что общее количество множеств - 18000, таким образом подсчитывается, насколько исследуемый индекс соответствует точности около 1%. Модифицированную платформу можно найти здесь https://github.com/PhilSk/ann-jaccards

4 Результаты

Ниже приведены таблицы с результатами эксперимента.

Индекс	Размер листа	recall	Запросов/сек
KDTree	10	0.654	351.915
	20	0.654	408.649
	40	0.654	426.426
	100	0.654	410.554
	400	0.654	380.898
	1000	0.654	347.738

Индекс	Размер листа	recall	Запросов/сек
BallTree	10	0.654	266.027
	20	0.654	197.337
	40	0.654	215.648
	100	0.654	200.882
	400	0.654	177
	1000	0.654	253.654

Индекс	Количество пар key-value	Количество проб	recall	Запросов/сек
FaissIVF	10	1	0.648	1850.373
	10	200	0.654	427.266
	20	1	0.643	5793.422
	20	200	0.654	419.329

Индекс	Бит/вектор	recall	Запросов/сек
FaissLSH	32	0.37	4775.658
	128	0.587	2764.965
	512	0.672	1838.447
	1024	0.687	1342.944

Индекс	Префикс деревьев	recall	Запросов/сек
LSHForest	5	0.671	87.822
	10	0.648	92.045
	20	0.547	94.601
	40	0.369	95.475

Индекс	Количество деревьев	Проверяемых вершин	recall	Запросов/сек
Annoy	100	20000	0.703	310.783
	100	40000	0.703	185.303
	100	400000	0.703	17.809
	200	100	0.679	1936.773
	200	1000	0.699	1094.977
	200	200000	0.703	39.287
	200	400000	0.703	19.961
	400	400	0.697	725.17
	400	4000	0.702	409.068
	400	10000	0.703	305.832
	400	100000	0.703	73.384

Индекс	recall	Запросов/сек
Bruteforce	1	0.437

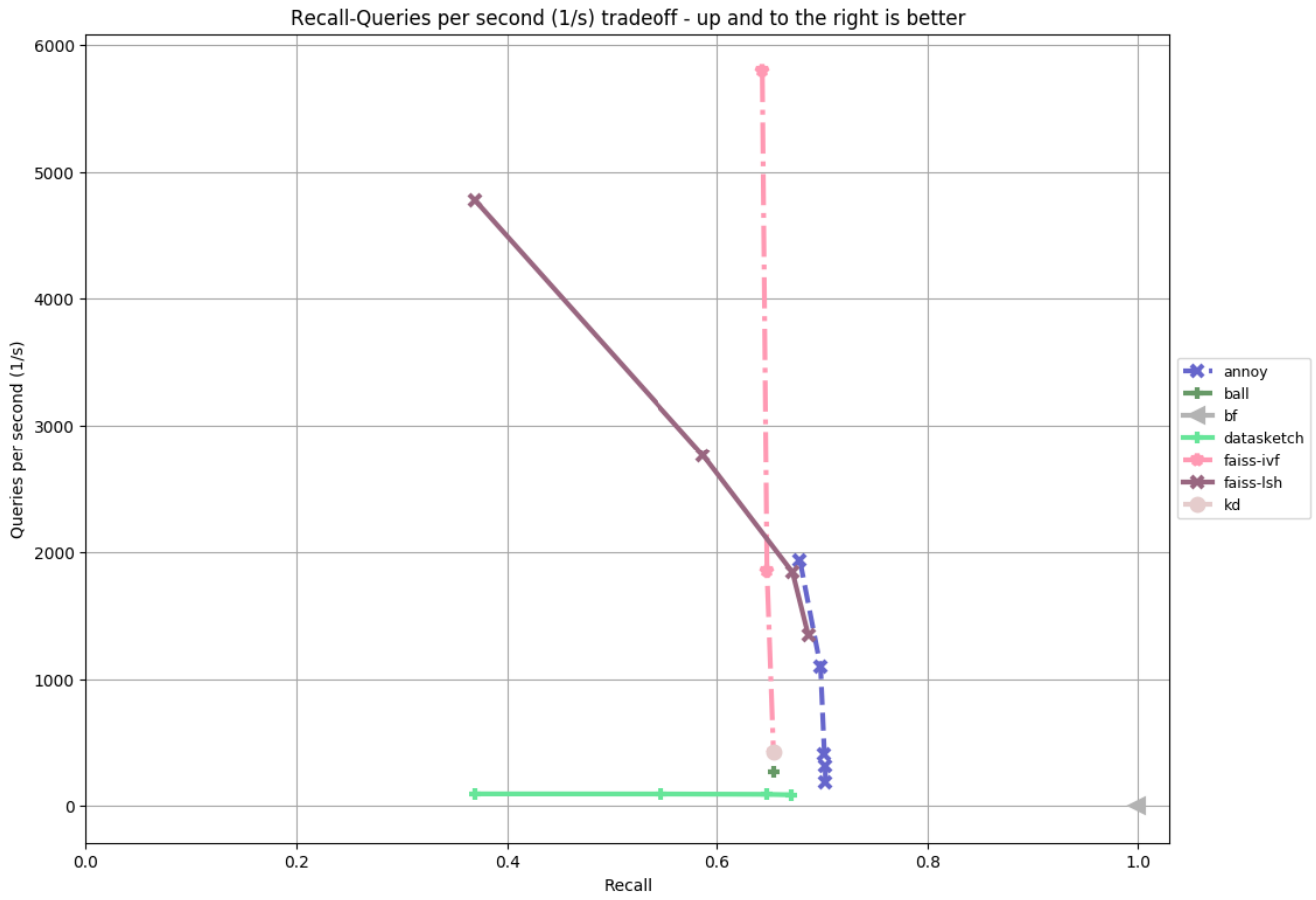


Рис. 14: График результатов изученных индексов

5 Анализ результатов

Как показывает ряд экспериментов, можно выделить несколько оптимальных решений в зависимости от необходимых характеристик. Наиболее быстрым и относительно точным является FaissIVF с параметрами (10, 1), чуть более точным, но почти в 3 раза медленным является Annoy с параметрами (200, 100). В зависимости от потребностей и предлагается выбрать один из этих индексов в качестве основного решения. Любое из выбранных решений значительно превосходит baseline-решение bruteforce - bf на графике.

Любопытно то, что индекс, изначально предназначенный для работы в метрике Жаккара (LSHForest) показал сравнимые по производительности результаты с индексами, функционирующими в других метриках. Также, как показывают эксперименты, чувствительность алгоритмов не в метрике Жаккара

сильно ниже, о чём свидетельствуют почти (логичная зависимость всё же присутствует) горизонтальные графики.

Следует отметить низкую эффективность алгоритмов BallTree и KDTree. на представленном графике частота запросов чуть превосходит стандартный подход - алгоритм LSH Forest. Такие результаты связаны с тем, что оба упомянутых выше индекса (BallTree и KDTree) основаны на рассмотренном в литературном обзоре R-Tree, который, в свою очередь, классифицируется как 'Ленивый ученик', то есть при каждом обращении к индексу производится инициализация модели, что при большом объёме данных ресурсоёмкая задача, требующая некоторого времени. Из результатов эффективности этих алгоритмов можно сделать вывод, что их применение в крупных стационарных вычислениях является нецелесообразном, потому что итоговая эффективность (частота запросов) будет сильно тормозить общий вычислительный процесс. С другой стороны, оба эти алгоритма можно применить в условиях динамичной системы, когда данные меняются достаточно часто. В этих условиях работа с другими индексами может стать затруднительной в силу того, что не для каждого индекса модификация существующего индекса (то есть добавление или удаление нового элемента) будет выполняться эффективно. Разумеется, эту проблему можно решить периодичностью выполнения модификации текущего индекса, однако грамотное выполнение этой задачи требует дополнительной проектировочной работы на общим алгоритмом проводимых вычислений.

Ещё одним результатом исследования является факт высокой точности алгоритмов, основанных на подходе деревьев, то есть таких индексов, которые составлены двумя и более индексами в зависимости от собствтвенных параметров. Изначальное предположение заключалось в том, что запрос к лесу индексов будет очень медленным процессом, потому что результирующий ответ будет складываться на основе данных, полученных не от одного-единственного индекса, а их некоторого количества. Как показывают результаты на графике (индекс Annoy), сам факт применения технологии комбинации нескольких индексов сам по себе не означает, что по итогу эффективность применяемого

алгоритма будет заведомо ниже, чем у алгоритмов, опрерирующих моделью, состоящей из одного индекса.

6 Заключение

В данной работе проведён сравнительный анализ алгоритмов, функционирующих в метрике Евклида и алгоритма LSH Forest в метрике Жаккара. Для этого был существенно модифицирован программный комплекс, а именно интегрирована метрика Жаккара, обработан и интегрирован датасет, на котором проводилось исследование, модифицированы критерии качества для более наглядной демонстрации полученных результатов.

7 Список литературы

[1] Гуттман А. A Dynamic Index Structure for Spatial Searching // ACM. — 1984.

[2] Бентли Дж. Л. Multidimensional binary search trees used for associative searching // ACM. — 1975.

[3] Мальков Ю. А. , Яшунин Д. А. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs [Электронный ресурс] URL: https://arxiv.org/abs/1603.09320

[4] - Джонсон Дж. , Дуз М. Billion-scale similarity search with GPUs [Электронный ресурс] URL: https://arxiv.org/pdf/1702.08734.pdf

[5] - Жегу Х., Дуз М. Product quantization for nearest neighbor search [Электронный ресурс] URL: https://lear.inrialpes.fr/pubs/2011/JDS11/jegou_searching_with_quantization.pdf

[6] - Раджараман, А., Ульман, Дж. Mining of Massive Datasets [Электронный ресурс] URL: http://infolab.stanford.edu/~ullman/mmds.html

[7] - Бава М., Конди Т., Ганесан П. LSH Forest: Self-Tuning Indexes for Similarity Search [Электронный ресурс] URL: http://infolab.stanford.edu/~bawa/Pub/similarity.pdf

[8] - Платформа для анализа алгоритмов поиска приблизительных ближайших соседей [Электронный ресурс] URL: https://github.com/erikbern/ann-benchmarks