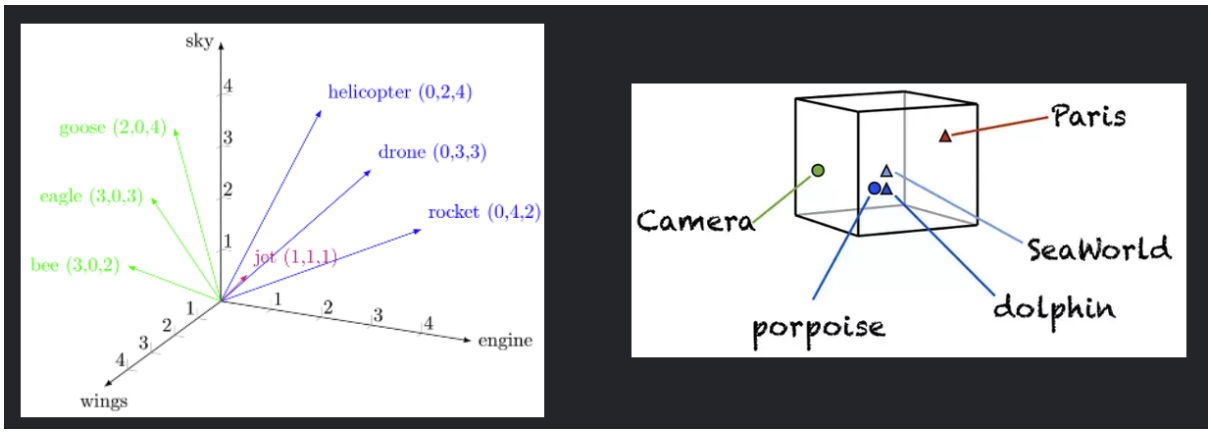
KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект > 6 урок > Приближенный поиск ближайших соседей

> Оглавление

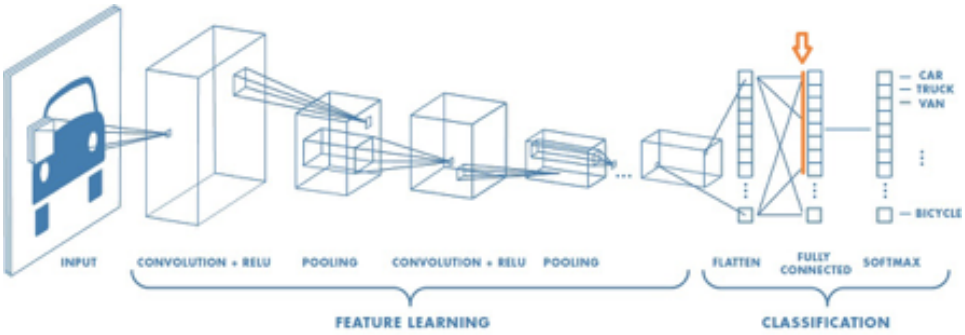
> Эмбеддинги



Вектор задаёт конкретную точку в пространстве

Точки в этом пространстве должны быть тем ближе друг к другу, чем более похожи объекты. В случае слов, как мы уже обсуждали, в качестве меры схожести выступает некоторый “смысл” слова. То есть слова "дельфин" и "морская свинья" ("porpoise") куда ближе друг к другу, чем к словам "Париж" и "камера".

Имеет смысл, с точки зрения применимости, кодировать сотни тысяч и даже миллионы слов, чтобы модель, получающая на вход эмбеддинги, выдавала приемлемые результаты, так как количество пропусков в данных будет минимальным и модель сможет в полной мере понять текст.



При работе с изображениями тоже используют векторы

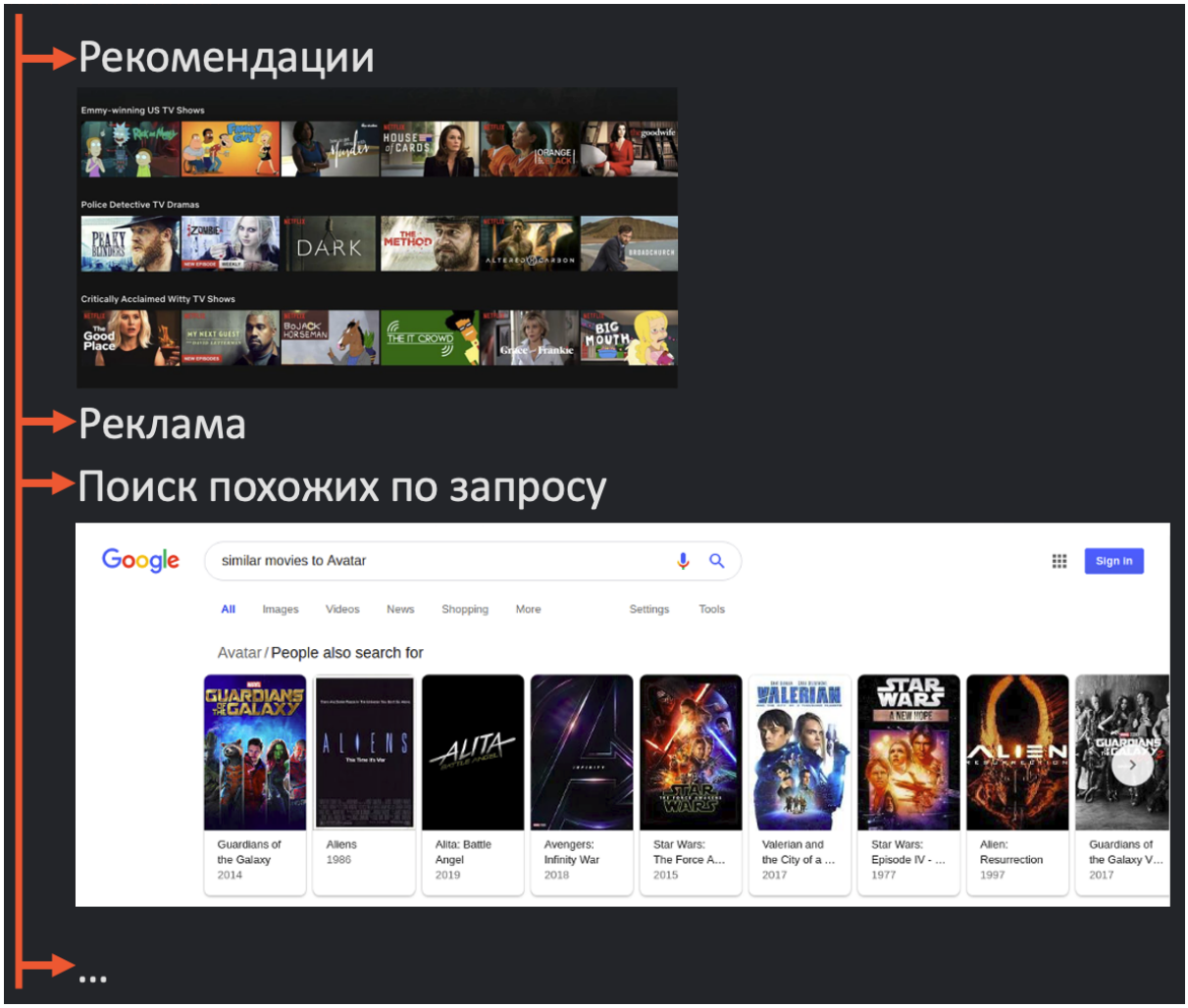
Нейросети в компьютерном зрении принимают на вход тензор с попиксельным описанием картинки, формируют некоторый вектор признаков, который впоследствии поступает в классификатор, чаще всего Fully Connected (полносвязные слои), для решения финальной задачи.

Промежуточный вектор, который формируется из картинки (на изображении выше на такой вектор указывает оранжевая стрелочка), можно назвать эмбеддингом изображения, который кодирует визуальную информацию.

Иногда такой вектор называют дескриптором, поскольку признаки, на основе которых строится классификация и выносится решение о том, что именно представлено на изображении, должны быть дискриминативны и давать вполне чёткое описание объекта на входном изображении.

Эти вектора, даже без специальной тренировки на разделение объектов, также обладают свойством, определённым для текстовых эмбеддингов — для схожих входных изображений такие вектора близки к друг другу, и наоборот.

> Ближайшие соседи



Примеры использования эмбеддингов пользователей

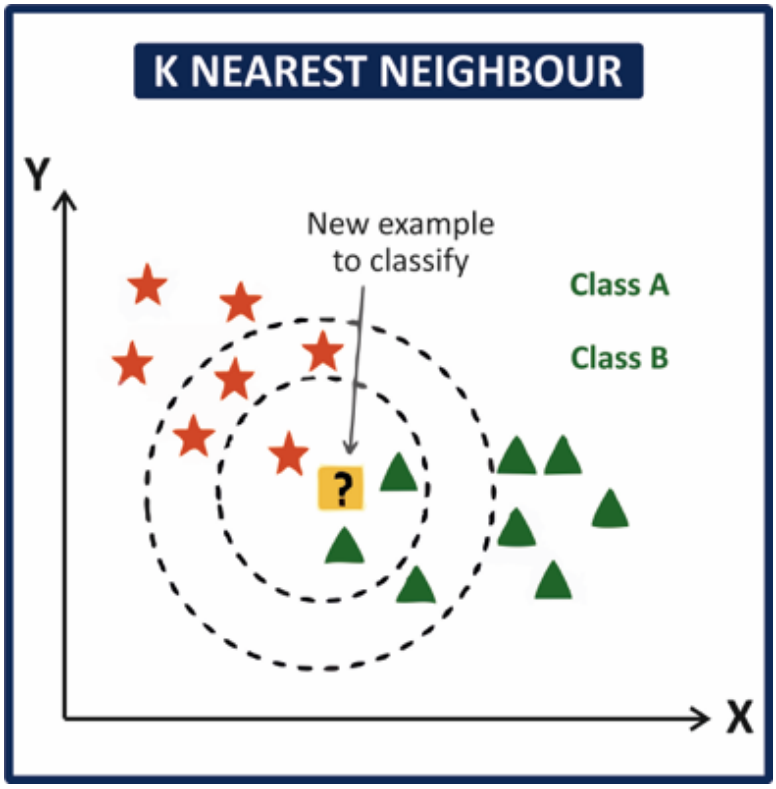
Если очень захотеть, то можно построить эмбеддинг чего угодно: и эмбеддинг пользователя в интернете по истории запросов, и эмбеддинг фильма по отзывам или через учёт людей, которые смотрели конкретный фильм — это пример коллаборативной фильтрации, которая используется в рекомендательных системах.

Её суть заключается в поиске людей, с которыми пользователь разделяет вкусы в кино или любой другой деятельности. Рекомендательная система предлагает человеку то, что было высоко оценено ими, но под руку ему ещё не попалось. Так работает множество онлайн кинотеатров, онлайн-магазинов. Они оперируют векторами, эмбеддингами пользователей, которые указывают на их предпочтения.

Для создания новой рекомендации нужно, при уже имеющихся векторах, найти пользователей, схожих по интересам, и затем применить какую-то логику отбора из числа потреблённого ими контента, музыки, фильмов, товаров и т.д.

Во всех разобранных кейсах ключевой компонент системы — это поиск схожих объектов. Эта концепция наверняка вам знакома по классическому методу машинного обучения KNN, пригодному для классификации и регрессии.

У этого метода нет этапа обучения — он просто запоминает всю выборку, копирует её, затем для поданного на вход объекта ищет k ближайших объектов и выдаёт усреднённый (возможно, взвешенный) таргет по известным значениям.



Жёлтым обозначен некоторый объект (запрос) для поиска.

В некоторой окрестности лежат три объекта: одна звездочка и два треугольника. Скорее всего, жёлтый объект относится к классу треугольников.

Однако если вердикт выносится не по трём, а по семи объектам, то оказывается, что вероятность принадлежности к тому или иному классу — 4:3 в пользу звездочек. Из этого примера становится понятно, что у алгоритма, по сути, два параметра:

- Количество соседей для рассмотрения и определения ответа;
- Мера, с помощью которой определяется дистанция между объектами.

> Формальная постановка

M D-dim query vectors $\quad Q = \{\boldsymbol{q}_1, \boldsymbol{q}_2, \dots, \boldsymbol{q}_M\}$

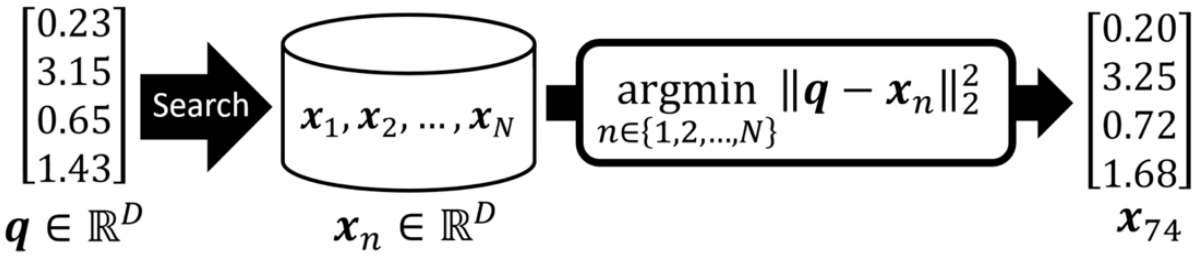
N D-dim database vectors $\quad \mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N\} \quad M \ll N$

Дан набор запросов размера M, где каждый запрос сформулирован в виде вектора размерности D; дан набор документов, или база для поиска размера N, где каждый объект также представлен в виде вектора размера D.

Будем считать, что природа, порождающая вектора, одинакова для запросов и документов. M значительно меньше N и даже может равняться единице.

Задача поиска ближайшего соседа

Заключается в нахождении такого индекса из множества N для каждого объекта из M, который даёт минимальное значение некоторой функции дистанции между документом с таким индексом и запросом.



$argmin$ указывает на нахождение минимальной меры расстояния, в данном случае — Евклидова.

Для нахождения ближайшего соседа нужно пройтись по всем документам, то есть, во-первых, они все должны быть в памяти или иметь оперативный доступ, а во-вторых, линейная сложность такого расчёта от количества документов: чем больше N, тем дольше работает алгоритм.

Пояснение масштаба

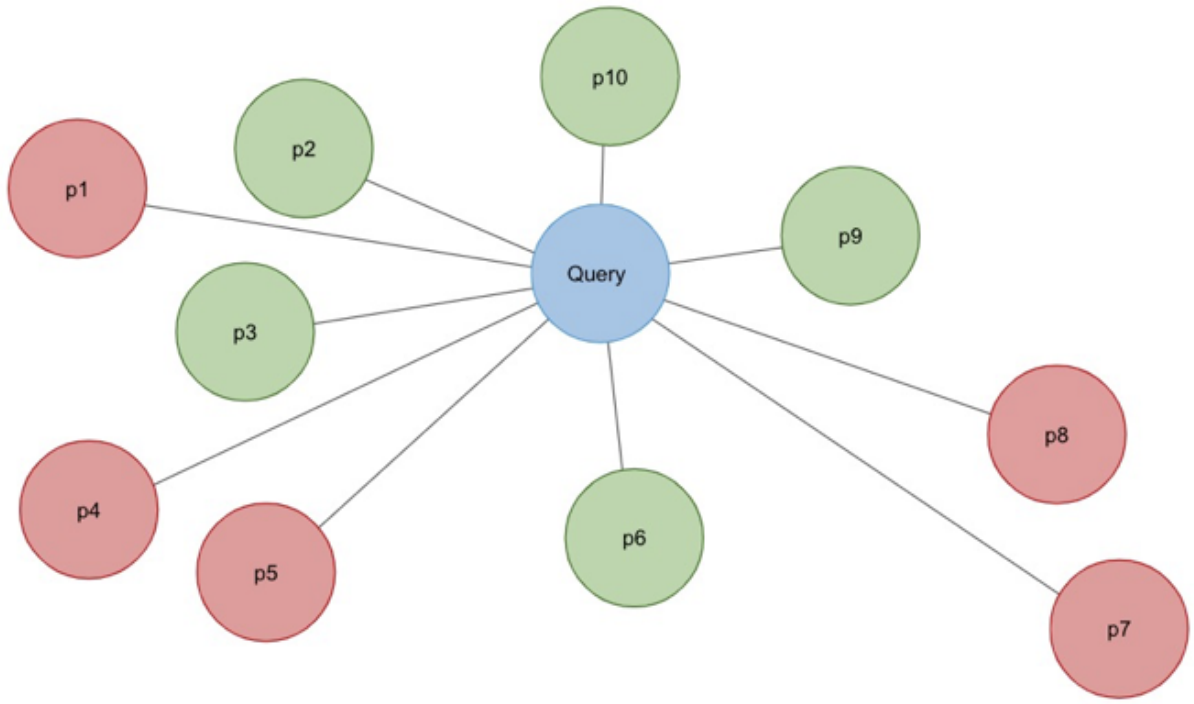
- Размер базы — миллионы объектов;
- Размерность каждого вектора — около сотни объектов.

Время на работу — десятки миллисекунд!

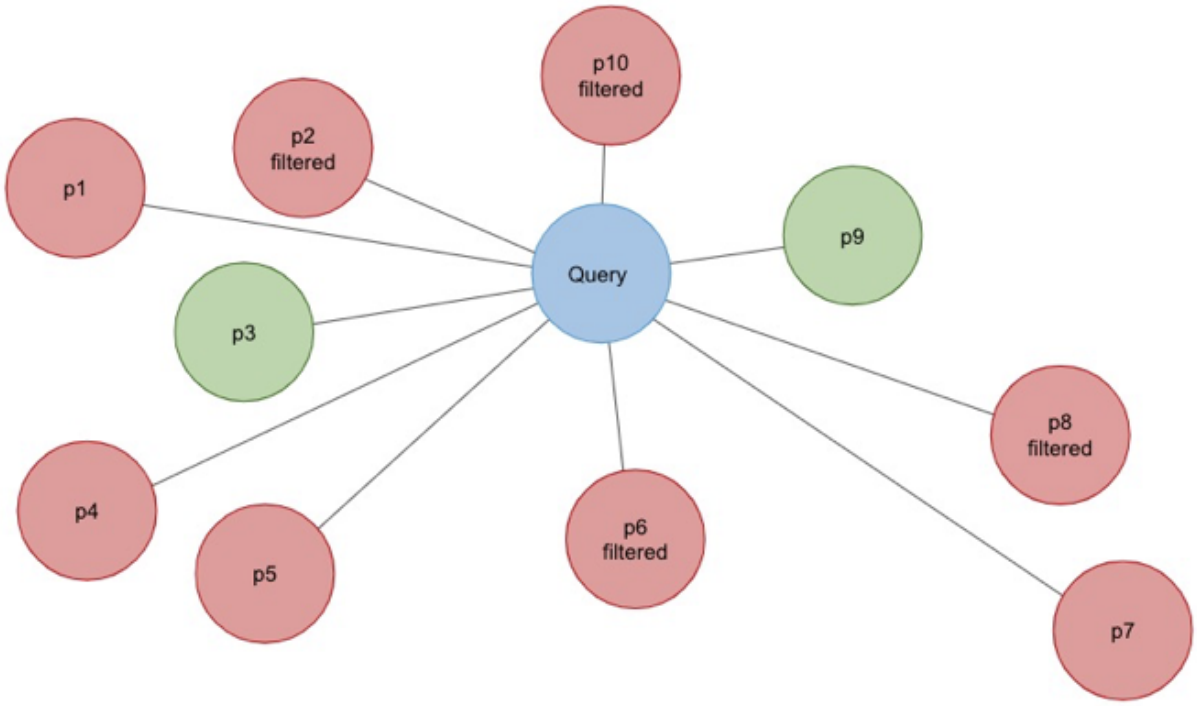
Если показатель RPS, *requests per second*, должен быть на уровне тысячи, такое решение в лоб, по всей видимости, не годится для применения в продакшене.

> Приближённый поиск ближайших соседей

Можно искать не N самых-самых близких соседей, а просто несколько достаточно близких объектов.



Пусть дан запрос query, обозначенный голубым цветом. Для него пятью ближайшими объектами являются объекты зелёного цвета.



Можно попробовать использовать объекты p3 и p9 и на их основе решить какую-то задачу.

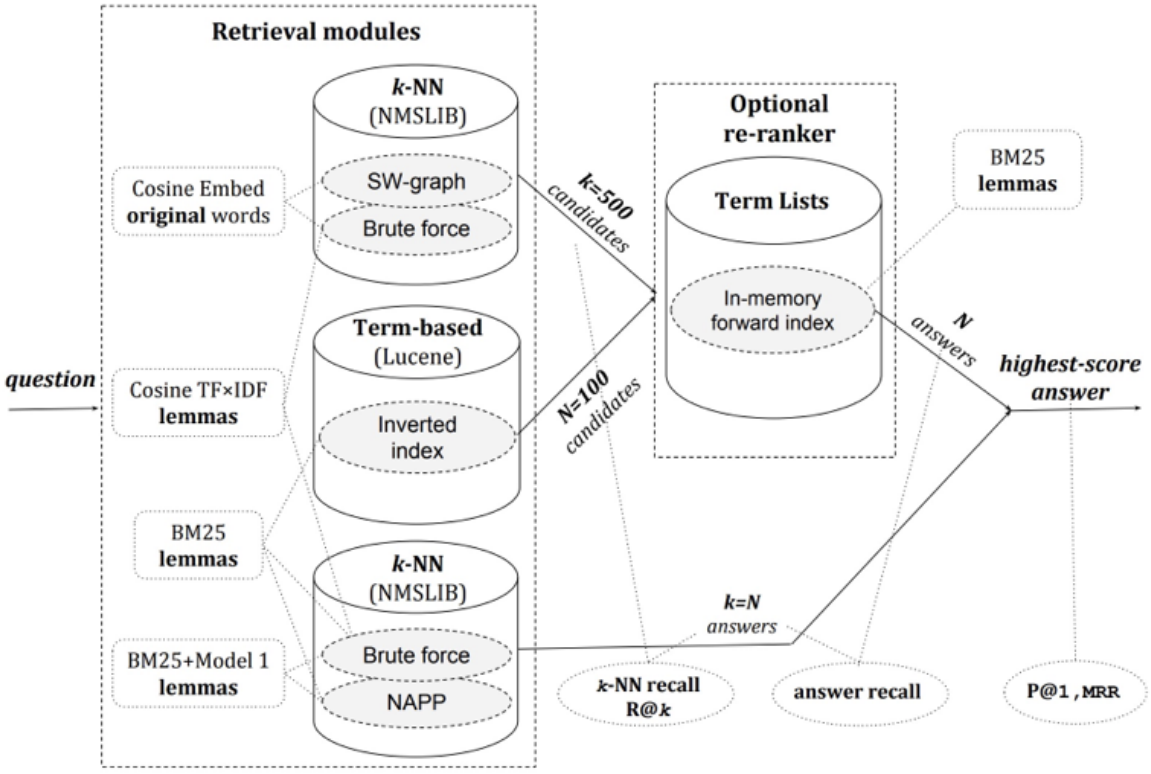
В качестве примера можно привести задачу рекомендации фильма. Каждая точка — это фильм, точка-запрос — это пользователь. Тогда можно порекомендовать к просмотру фильмы p3 и p9, и они будут вполне себе хорошей (по нашей оценке) рекомендацией. А фильмы p2, p10 и p6 в выдаче не встретятся, хотя и могли бы понравиться пользователю. Конечно, это плохо, что мы их пропустили, но ничего критичного не произошло — это и есть приближённый поиск.

Понятно, что условный Google не выдаёт всего две ссылки, Мы просто рассматриваем относительный пример — тут может быть 500 и 200 соседей вместо 5 и 2. Выходит, мы используем не k nearest neighbours, а ANN (*approximated nearest neighbours*), то есть приближённые вычисления, аппроксимацию.

Используя такой подход, мы, очевидно, теряем в точности, но взамен получаем быструю работу алгоритма. Более того, для KNN мы не используем время на подготовку данных и их обработку — у нас есть готовая база векторов, и поиск происходит в ней.

Давайте перед разбором конкретных подходов узнаем, какое место алгоритмы поиска ближайших соседей занимают в системах ранжирования и матчинга.

Пример



Вопросно-ответная система, ранжирующая документы согласно вероятности того, что в документе содержится ответ на заданный вопрос.

- В левой части изображения несколько баз, которые можно называть индексами.
- Сверху вниз идут базы с векторами, или эмбеддингами текста.
- Далее база для поиска по определённым словам, основанная на `TF-IDF`
- В самом низу некоторая модель в связке с индексом `BM25`

Каждая из компонент формирует некоторый список документов, которые могут отвечать на заданный вопрос, тем самым выполняя функцию кандидатной модели. На этом этапе из миллионов документов выбирается несколько сотен, которые будут отранжированы, или, лучше сказать, реранжированы для финальной выдачи.

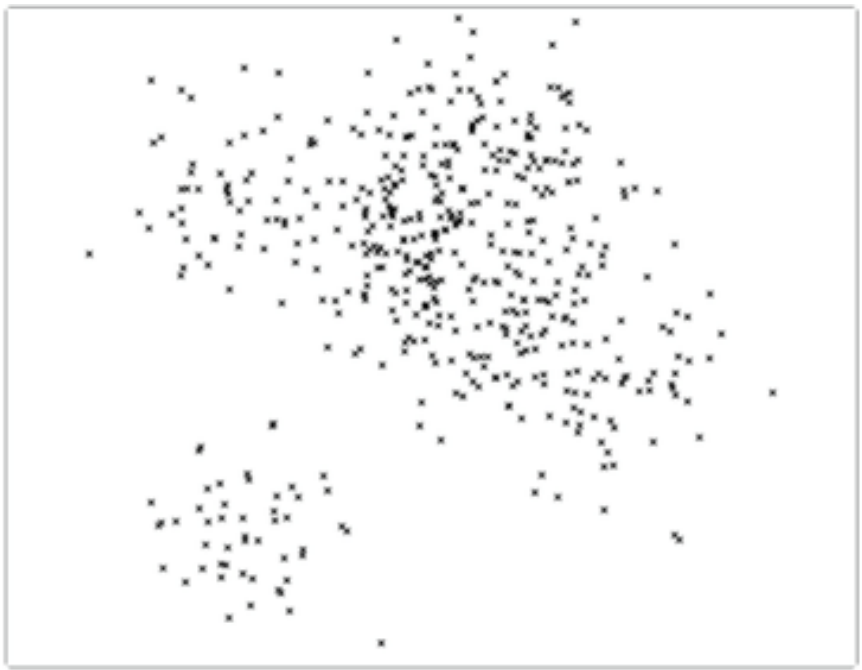
Наиболее подходящей в качестве примера в рамках текущего занятия является самая верхняя база, которая работает с эмбеддингами текстовых последовательностей.

1. Заранее строится represenation-focused модель и происходит векторизация документов.
2. При поступлении запроса происходит перевод в эмбеддинг вопроса на входе.
3. Выполняется выявление максимально схожих документов по некоторой мере расстояния между векторами запроса и документа.
4. 500 ближайших объектов уходят на реранжирование и более точную и качественную оценку тяжёлой моделью (более медленной и сложной).

Если на этапе работы кандидатной модели находится один мусор, то каким бы продвинутым ни был ранжирующий алгоритм на выходе, он всё равно не сможет взять документ из ниоткуда. Для разных компонент системы важно оценивать их отдельное качество, чтобы понимать воронку данных в пайплайне. На изображении этому можно найти подтверждение — для кандидатов оценивается `recall`, а для финального ранжирования метрика `MRR`, т.е. средний обратный ранг самого релевантного документа.

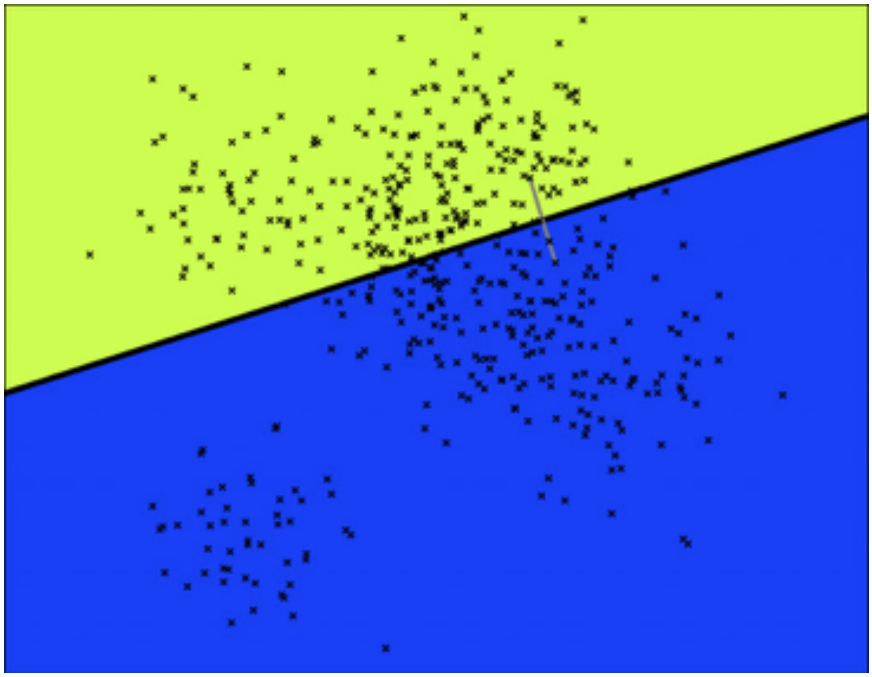
> Облегчение задачи поиска соседей

Допустим, имеется некоторое латентное пространство, и в нём задано множество объектов.



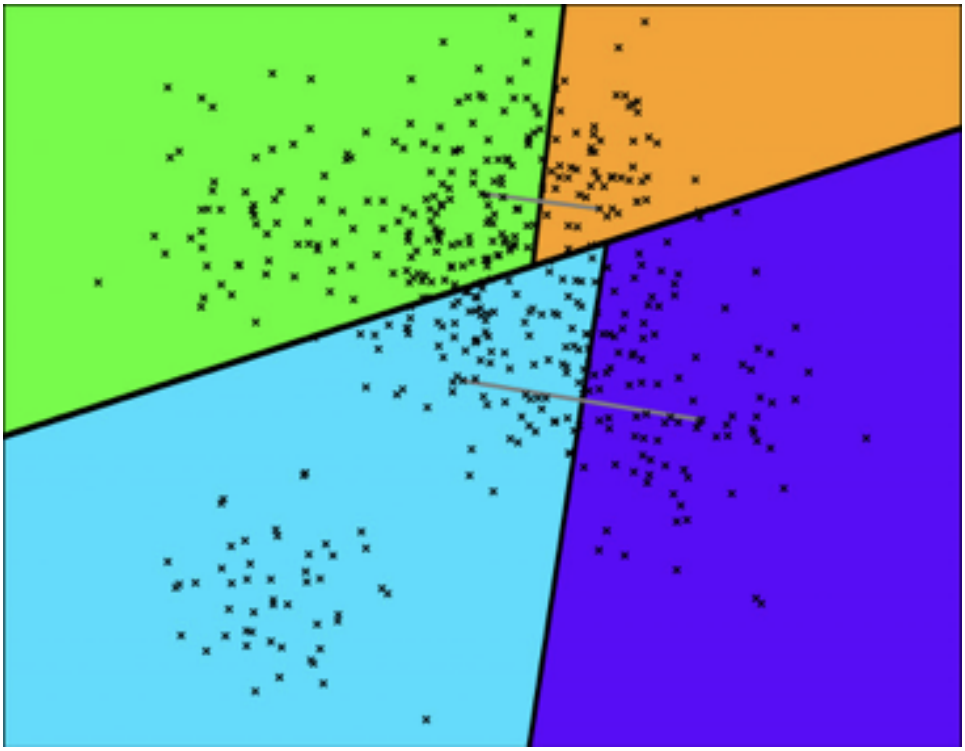
Объекты отмечены чёрными крестиками.

Наверняка вам в голову пришла гениальная идея — давайте просто разделим всё пространство на два подпространства, проведя некоторую гиперплоскость!



В случае с двумерным пространством такая гиперплоскость превращается в прямую, разделяющую прямоугольник на две фигуры.

Если две части примерно равны по количеству объектов и запрос генерируется из равномерного распределения в отношении латентного пространства, то можно просто определить, в какое из подпространств попадает точка запроса, в верхнее или нижнее, и выполнить поиск только среди объектов из этой части. Для этого нужно построить два алгоритма KNN, каждый из которых по отдельности работает в два раза быстрее единого, содержащего в себе все точки.



Можно продолжить рекурсивно разбивать пространства на подпространства и провести, например, ещё две разделяющие гиперплоскости.

Если подпространства теперь четыре, то строится четыре индекса KNN, каждый из которых примерно в 4 раза быстрее исходного. Идея кажется крайне привлекательной. Формализуем рассмотренный подход, обобщив идеи такой структуры данных, как KD-Tree.

> K-D Tree

Это улучшенная версия бинарного дерева, обобщённая на K пространственных компонент (*dimensions*).

Не является сбалансированным деревом в общем случае.

Концепция:

- Случайным образом выбираем измерение (из K).
- Находим медиану, распределяем данные в левое и правое поддерево.
- Повторяем.

В несбалансированном дереве:

- Может быть задан некоторый критерий важности или частоты встречаемости объектов, и его нужно учитывать при каждом разбиении.

- В каждую из ветвей может попадать абсолютно разное количество векторов.

> Пример

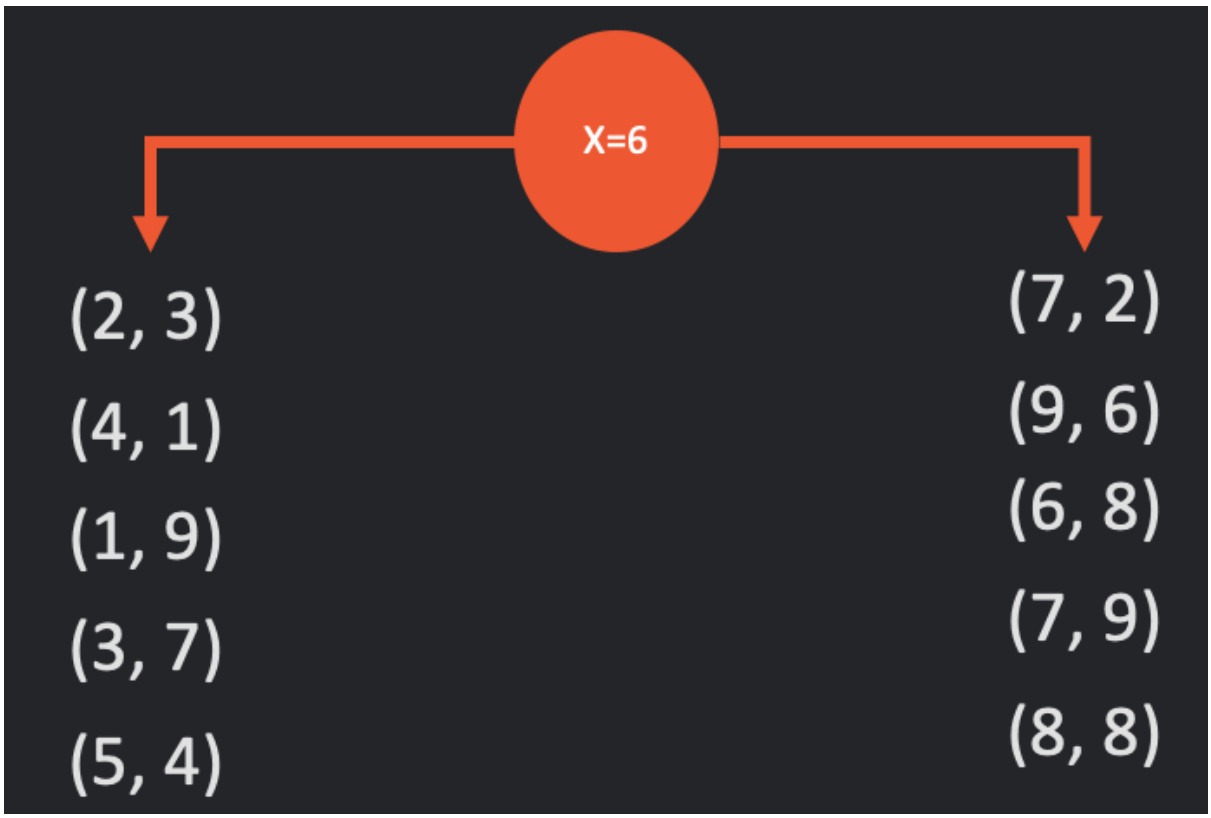
Пусть дано множество из 10 объектов:

$(1, 9), (2, 3), (4, 1), (3, 7), (5, 4), (6, 8), (7, 2), (8, 8), (7, 9), (9, 6)$

Каждый объект представляет собой точку на двумерной плоскости.

Первая итерация

Случайно выберем одно из двух измерений, пусть будет первое, то есть координата X точки. Для указанного множества медиана равняется 6

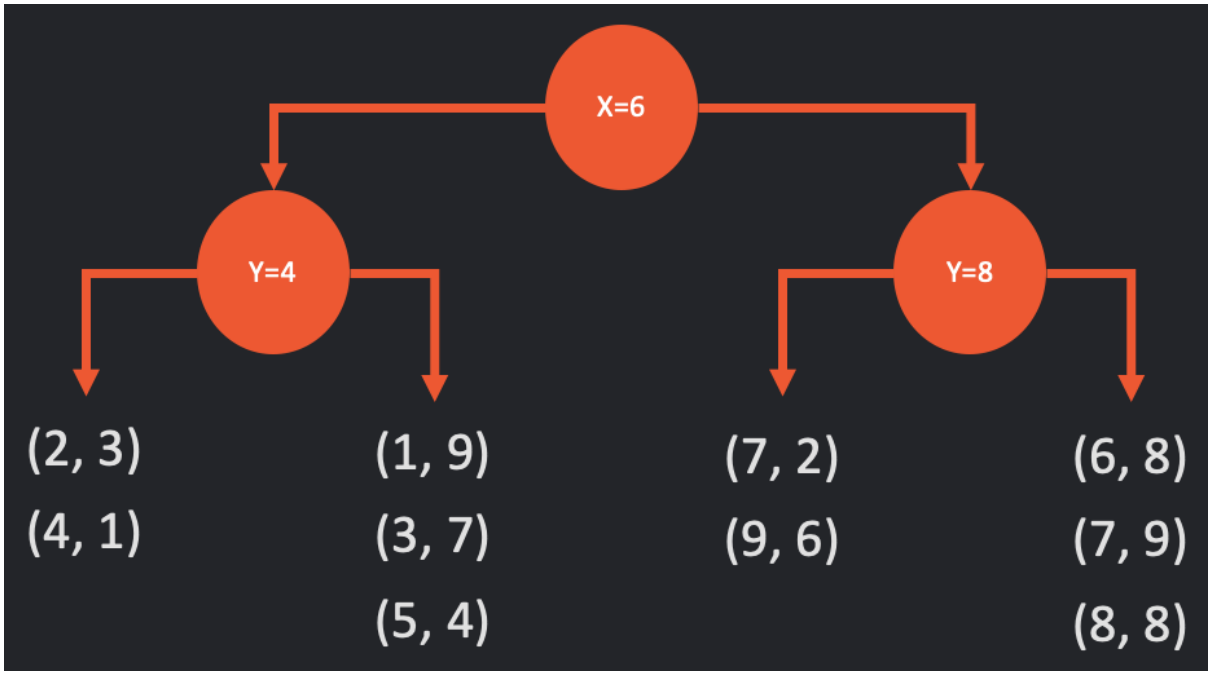


Так как мы взяли медиану, ровно пять объектов имеют значение координаты X меньше медианы, ровно пять — больше.

Следующие шаги

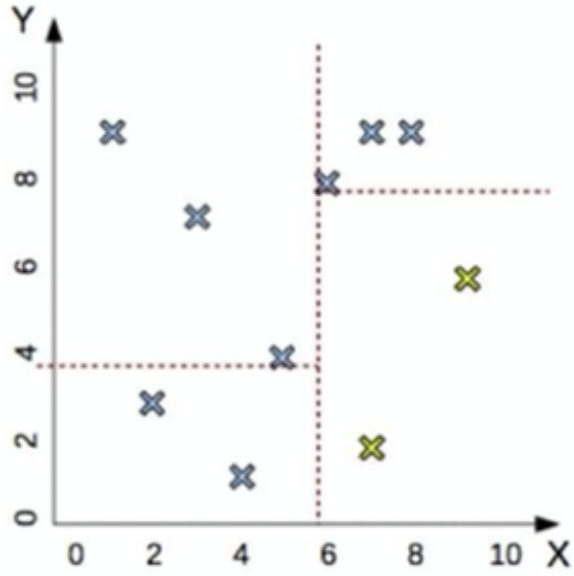
Рекурсивно для каждой полученной ноды повторим операцию.

Пусть для второго уровня дерева выбрана координата Y



Медианы в каждом отдельном множестве равняются 4 и 8.

Распределим объекты по нодам и назовём их терминальными нодами дерева.



Подпространства, которые задаёт данное дерево, можно отразить на плоскости, в которой расположены исходные точки.

Четыре листа задают 4 подпространства, в каждом 2 или 3 объекта. Сначала была проведена вертикальная черта в $X = 6$, и плоскость превратилась в две — правую и левую. Затем каждая из них была разбита вертикальной чертой, в $Y = 4$ и $Y = 8$ для левой и правой соответственно.

Ищем соседей точки

Допустим, эмбеддинг от запроса задаёт точку с координатами $(7, 4)$.

Нужно спуститься по дереву и определить подпространство для поиска, как мы обсуждали ранее. Координата $X = 7$. Это больше шести, поэтому отправляемся в правое поддерево.

$Y = 4$ и $Y < 8$, теперь выбираем левое поддерево, содержащее подсвеченные оранжевым точки.

Их мы и назовём приближённо вычисленными ближайшими соседями.

Красным крестом на картинке справа обозначена точка запроса, двумя жёлтыми — два соответствующих эмбеддинга из терминальной ноды дерева.

Однако кажется, что в процессе обнаружения соседей мы что-то упустили. Обнаружили точку $(7, 2)$, расстояние до которой равно 2, и $(9, 6)$ с расстоянием чуть больше 2. При полном переборе всех точек выясняется, что есть точка с координатами $(5, 4)$ в другом поддереве, и расстояние до этой точки также равно 2. Эта точка в дереве выделена красным — по сути она пропущена при поиске соседей, так как лежит в другом подпространстве.

Такая проблема случилась потому, что есть некоторое количество точек, крайне близко расположенных к границам, и если запрос также близок к границе, то появляются ошибки. Но помните про обмен скорости на качество. Сейчас для нахождения двух ближайших соседей было произведено 2 операции сравнения для определения терминальной ноды, что очень быстро, и 2 операции вычисления дистанции — и это всё вместо вычисления десяти дистанций до всех исходных точек множества.

Здесь есть две проблемы:

- Нет контроля над качеством соседей, выдаваемых алгоритмом.

Допустим, нам хочется получить в 90% случаев действительно самого близкого соседа, пусть даже придётся ждать в 2 раза дольше. Единственное, что можно сделать, это укоротить дерево, но тогда процесс расчёта слишком замедлится, и само качество будет привязано к некоторым константным дискретным значениям.

- Отсутствует возможность контролировать количество ближайших соседей.

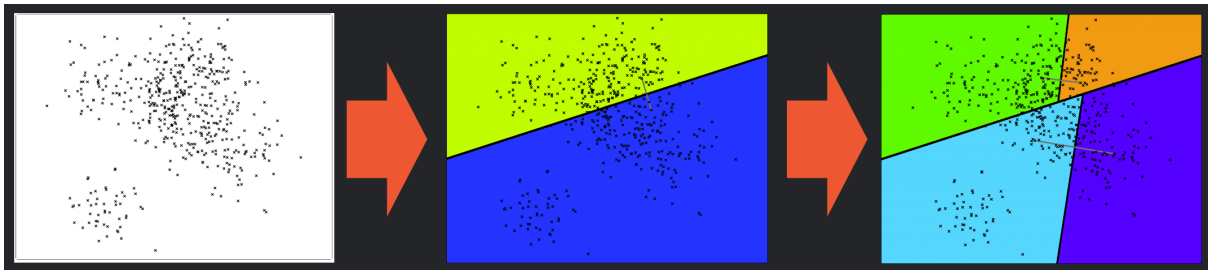
Предположим, формируется выдача топ-4 результатов для ответа на запрос пользователя. Можно попасть в ноду, которая содержит всего 2 объекта, и не ясно, что делать дальше. Конечно, опять же есть решение с обрезанием

дерева и эвристиками при построении, но тогда можно упереться в порог качества, если каждый лист дерева содержит ровно K объектов.

Постараемся разрешить эти проблемы, однако перед переходом к этой теме заметим, что мы обсудили не полный алгоритм `KD-Tree`. В классическом варианте есть сложный итеративный процесс обхода соседних подпространств. На выходе получается всегда именно ближайший сосед, однако в рамках задачи нас устраивает и просто хорошая аппроксимация. Поэтому эту часть мы не затронули. При желании и наличии интереса можно найти описание алгоритма, например, на Википедии, но на практике при наличии более продвинутых подходов полный вариант `KD-Tree` используется редко.

> ANNOY by Spotify

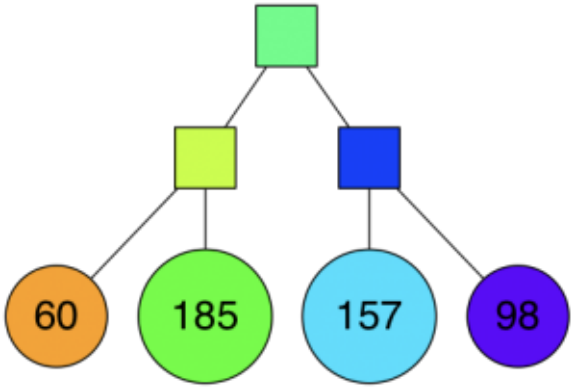
Первым таким алгоритмом будет `ANNOY` от Spotify, сервиса для прослушивания музыки с рекомендациями плейлистов.



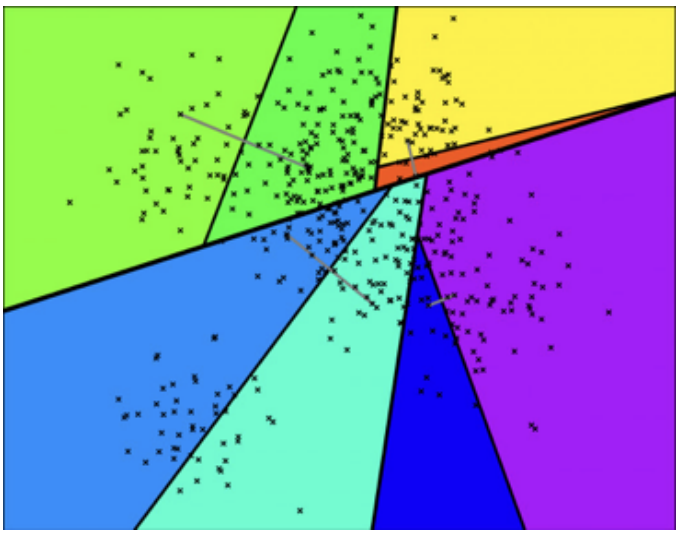
В целом метод похож на `KD-Tree`, кроме выбора разбивающих плоскостей. Для каждого разбиения случайным образом выбираются две точки, они соединяются отрезком, а затем через его середину строится перпендикулярная гиперплоскость.

> Пример

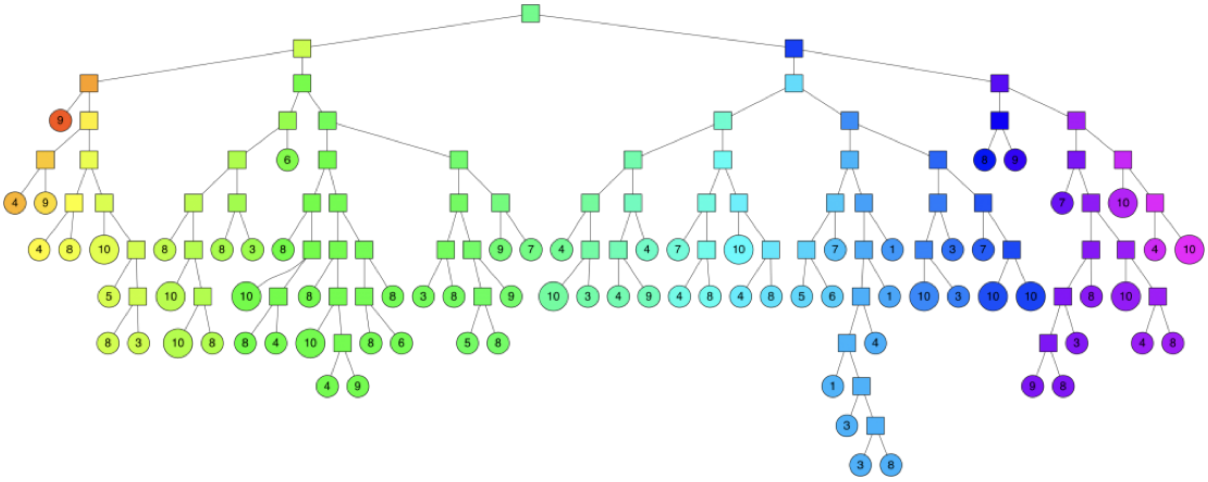
Вариант после третьей итерации представлен справа: строится аналогичное дерево, в узлах которого определяется отношение точки к уравнению плоскости, т.е. она лежит по одну сторону плоскости или по другую.



Кластеры могут сильно отличаться по размеру.



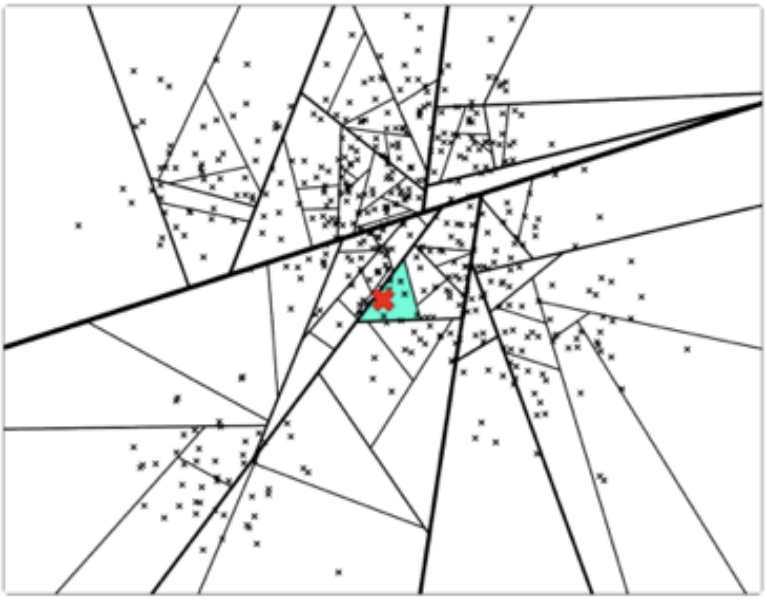
Алгоритм повторяется до тех пор, пока хотя бы в одной ноде точек не будет больше, чем некоторая константа K.



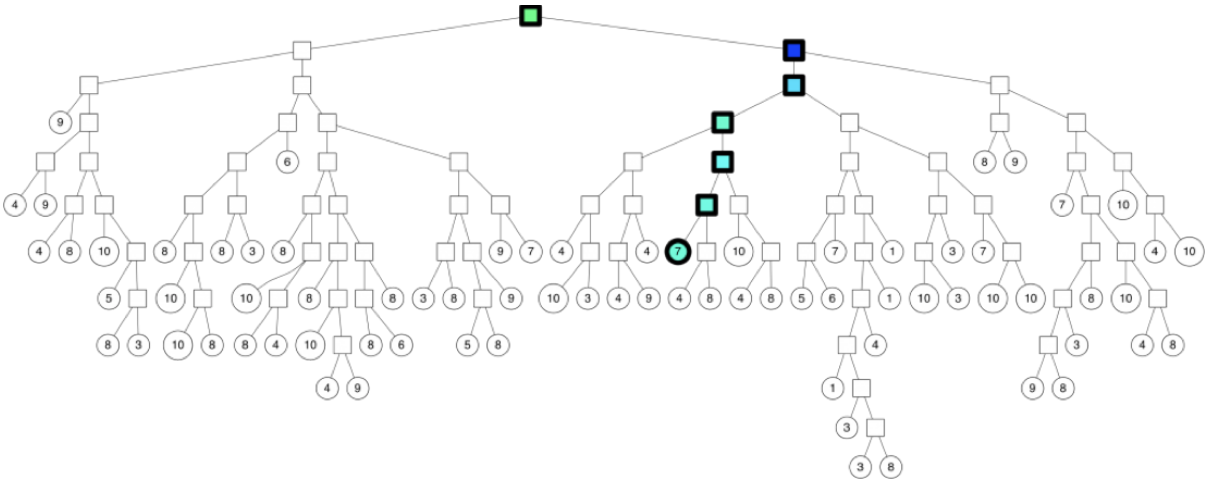
В конечном итоге, логику определения принадлежности точки к какой-либо части пространства можно описать деревом. Ни один лист этого дерева не превышает размера 10. Каждый узел дерева — это отдельная плоскость, с которой происходит сравнение поступающей в качестве запроса точки.

Получение результата

Для получения списка кандидатов для расчёта дистанций и приближённого определения ближайшего соседа необходимо спустить точку по дереву, достигнув определённого листа, и после сравнить с присвоенными этому региону векторами.



Найдём результат для красной точки.



Закрашены только те ноды, через которые прошла красная точка.

Кажется, что проблемы, озвученные для `KD-Tree` на этом этапе решены хотя бы частично. Появилась возможность контролировать количество финальных объектов в листах, однако не всегда такое нововведение способно улучшить алгоритм. И всё же мы построили дерево поиска, которое позволяет не перебирать все объекты для расчета дистанций.

Поскольку каждый раз выбирается случайное разбиение для создания ноды в дереве, пространство будет разнообразным в окрестности некоторой точки, характеризующей запрос, и она сама будет попадать в разные кластера с в целом одинаковыми, но всё же отличными друг от друга точками.

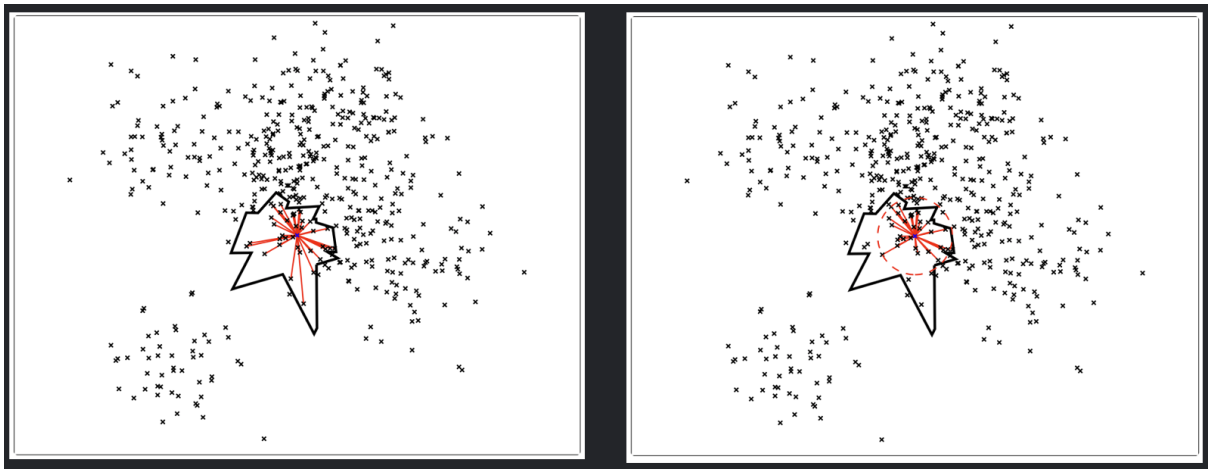
Также при спуске по дереву можно завести очередь с приоритетом (*priority queue*) — в неё заносятся посещённые ноды, и при этом добавляется возможность выбирать не исключительно левое или правое поддерево, а сразу оба.

Критерий захода — это расстояние до разделяющей плоскости. Если точка находится около границы и есть риск, что ближайшие соседи окажутся отрезаны, то логично включить этот кластер, записав его в очередь для вычислений, и позже вернуться для пересчёта дистанций и определения соседей. Приоритет задаётся расстоянием до плоскости: чем оно выше, тем глубже точка в кластере и тем выше шанс найти там более близких соседей и, возможно, самых близких.

> Достоинства

- Очередь с приоритетом (priority queue);
- Возможность при близости к критерию разбиения заходить в оба поддерева;
- Построение леса деревьев (с разными разбиениями);
- Очередь с приоритетом доступна всем деревьям (не работаем с «плохими» деревьями, в которых неудачное разбиение на подпространства).

При прогоне всех деревьев и объединении точек из их листовых нод образуется подпространство невыпуклой формы и расчёт дистанций производится только для точек, входящих в него.



Чем больше деревьев, тем:

- Требуется больше памяти;
- Проводится больше вычислений;
- Точнее расчёты.

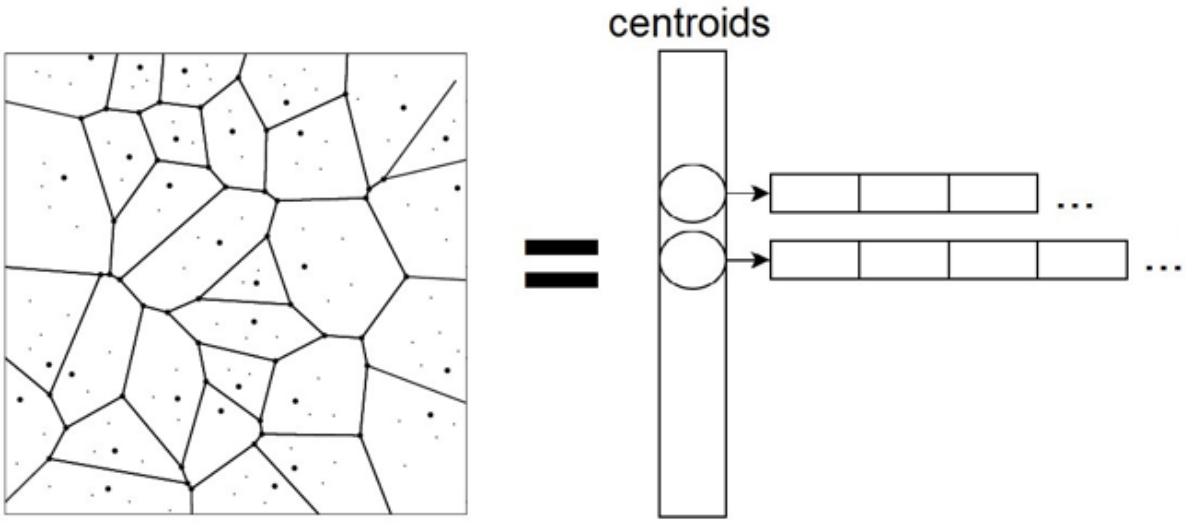
> IVF index: Inverted File Index

Inverted File Index — обратное отображение, маппинг полученных кластеров или подпространств в точки, принадлежащие этим кластерам. Если данных очень много и хранить их в памяти становится тяжело, то можно записать часть информации на диск.

> Пример

Простой пример применения такого метода — это кластеризация исходных данных методом `K-means`. Если все данные не влезают в оперативную память, то можно применить батчевое обучение для кластеризации, где на очередной итерации пересчета центров кластеров берётся лишь подвыборка из всех имеющихся данных.

Представьте, что у вас десятки миллионов документов, а на серверной ноде всего лишь 32 гб ОЗУ. Именно в такой ситуации подходит стохастический `K-means`.



Для каждого кластера определяются все входящие в него точки, затем они записываются в файл или отдельную часть большого файла с индексом. В памяти хранятся лишь центроиды кластеров и маппинг, подсказывающий, откуда нужно считать данные по определённому кластеру. Тогда, при применении метода поиска соседей, для запроса сначала будут рассчитаны расстояния до центров кластеров (т.е. до центроидов, которых значительно меньше, чем исходных точек), а затем для нескольких ближайших кластеров, возможно даже для одного, в память будут загружены данные из файла с обратным индексом. После этого будет произведён расчёт расстояний с точками из этих кластеров.

Если размер кластеров достаточно мал в пересчёте на количество точек, то такие вычисления будут достаточно быстры.

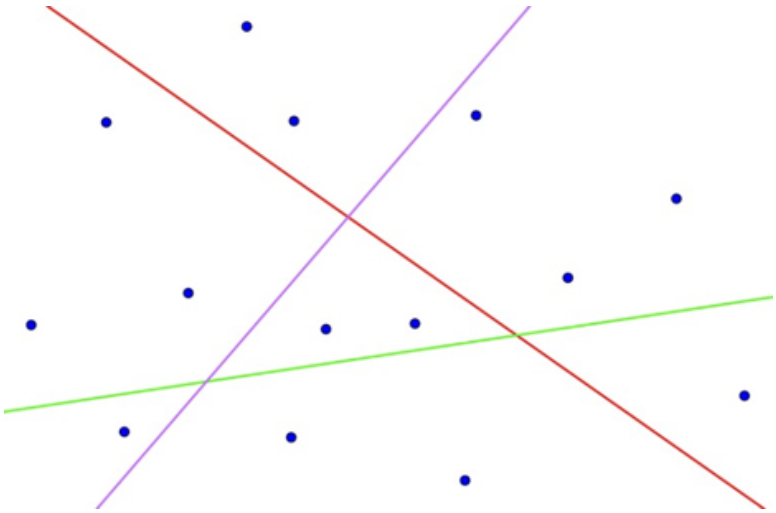
Данный алгоритм крайне похож на разобранные ранее подходы. Отсюда важно вынести, что IVF задаёт отображение кластера в множество принадлежащих ему точек.

> LSH: Local Sensitive Hashing

Если обычная хэш функция определяет один и тот же объект в один и тот же бин (запись в хэш-таблице) по строгому соответствию, то LSH определяет похожие в терминах локального расположения в пространстве объекты в один и тот же бин, исходя из предположения, что такие объекты можно назвать схожими.

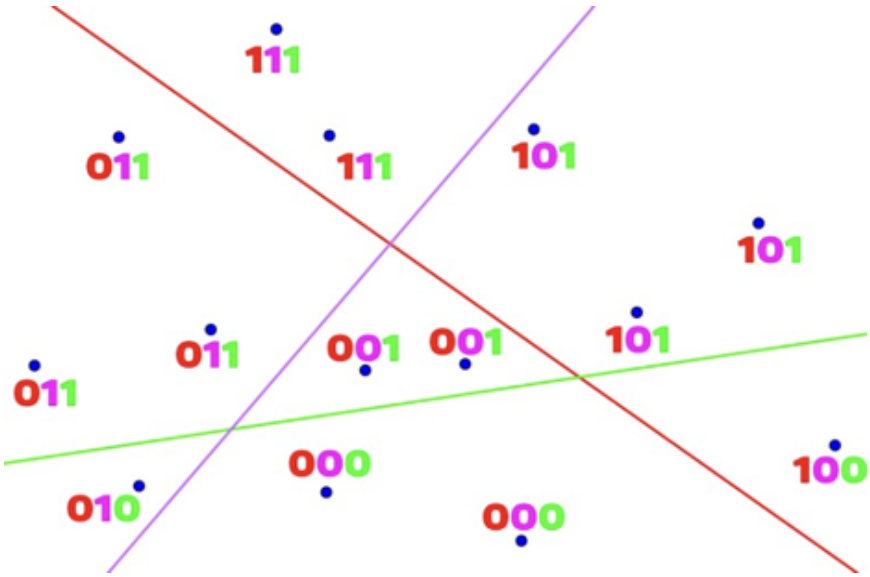
Есть много вариантов таких функций, рассмотрим самую простую и часто используемую. Она также базируется на уже рассмотренных концепциях, и

разобраться в ней не составит труда.



Такую функцию можно получить на основе разбиения пространства гиперплоскостями.

Для каждой плоскости можно определить, по какую сторону лежит точка, и закодировать информацию в один бит, то есть проставить бинарный флаг: 0 или 1. Соответственно, для K разбиений пространства код содержит K бит, K единиц или нулей.



Пример с кодами на плоскости.

Логично, что точки с одинаковым хэшем попадают в одно подпространство. Соответственно, они относительно близки друг к другу, а значит похожи.

Если подобное разбиение повторить несколько раз, и полученные коды соединить, или конкатенировать, то получится длинный бинарный и легковесный код, который работает аналогично разным деревьям в `ANNOY`. Для сравнения таких кодов и определения расстояния можно, к примеру, использовать расстояние Хэмминга.

Есть более продвинутые варианты LSH-функций, но сейчас они применяются достаточно редко, а сам подход называется "классическим" в значении "устаревший, некогда очень хороший метод, на смену которому пришли более новые и продвинутые". На смену ему пришёл `PQ` (*Product quantization*).

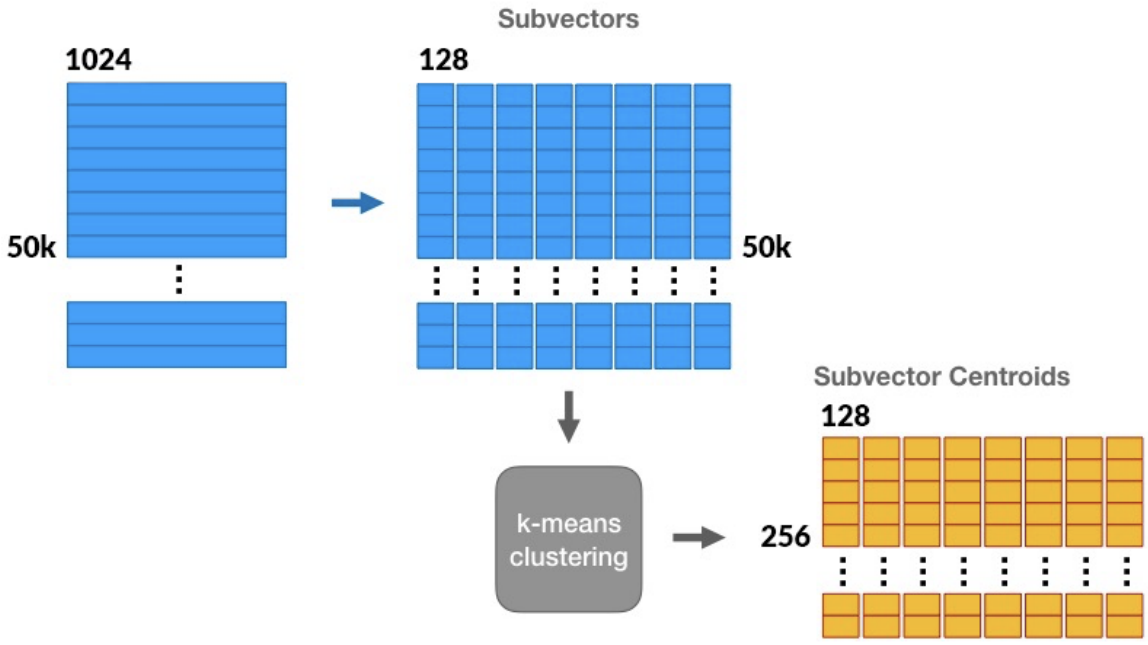
> PQ: Product Quantization

PQ применяется, когда данных настолько много, что хранить их даже на диске очень дорого, и считывание большого объёма данных из индекса в память замедляет процесс приближённого поиска ближайших соседей.

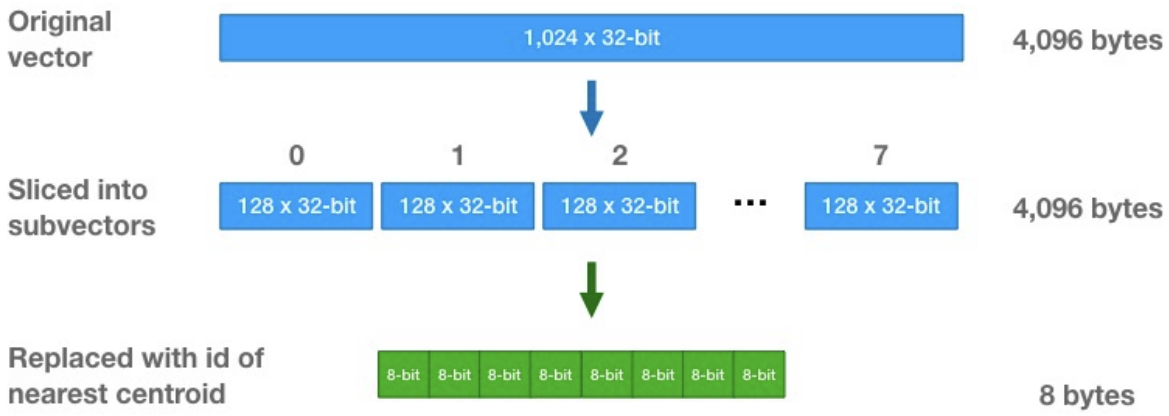
> Пример

Размерность вашего вектора 1024. Например, это может быть эмбеддинг картинки, так как многие нейросети из области компьютерного зрения выдают представления примерно такого размера. Если таких векторов будет 30 миллионов, то всё это будет весить чуть больше 120 гигабайт. Каждое значение вектора обычно представлено числом с плавающей точкой `fp32` и, как следствие, весит 32 бита или 4 байта.

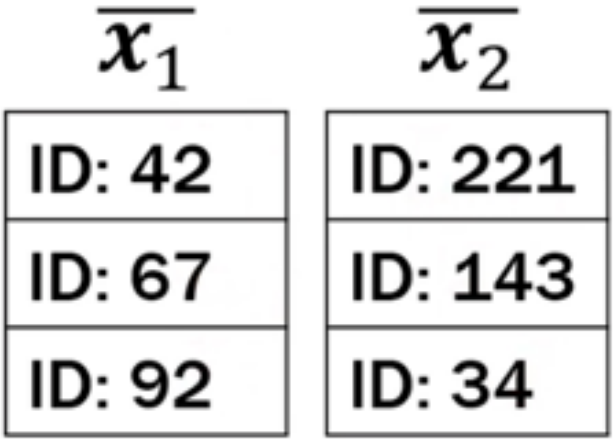
Выходит, можно оптимизировать как размер каждого значения вектора, так и саму размерность вектора.



Для этого достаточно проделать следующую операцию: исходный вектор разбивается на подвектора одинакового размера, скажем, по 128 элементов в каждом. При их конкатенации мы обратно получаем исходные вектора. Разделённые подвектора по всем объектам можно кластеризовать с помощью `k-means`, но использовать не произвольное количество кластеров, а константное — 256.



Связано это с тем, что в один байт можно записать целочисленное значение от 0 до 255, то есть 256 уникальных значений. И тогда такой подвектор превращается просто в id соответствующего кластера. Так как групп векторов имеется несколько, то схожая операция проделывается и для них.



Пример закодированных векторов X1 и X2 с помощью всего 3 байт.

Посмотрим на пример для x_1 и x_2 — это меньше, чем один элемент исходного вектора!

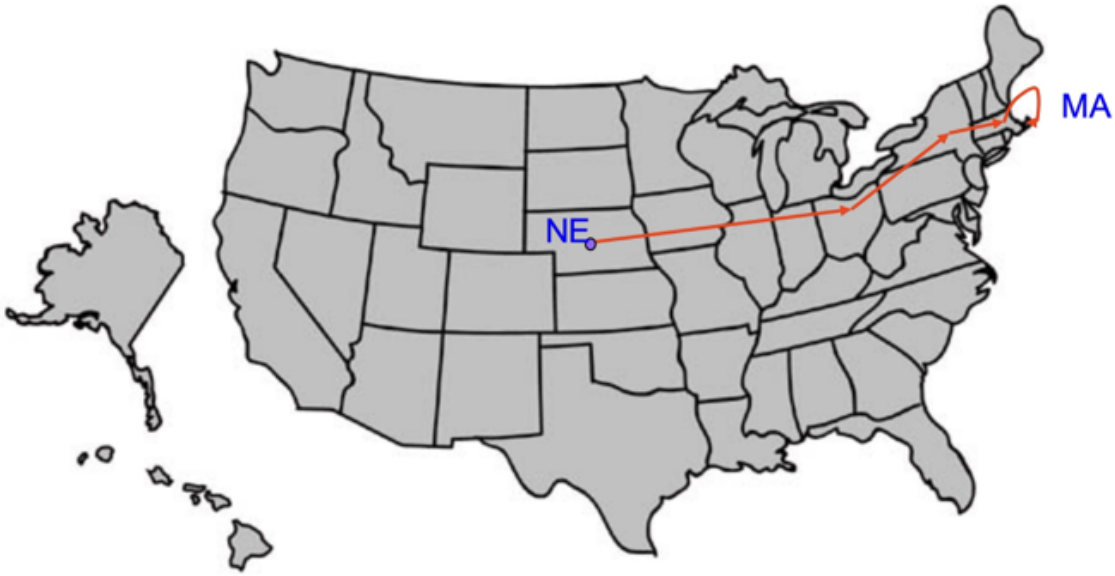
С помощью дистанции между квантизованными векторами можно эффективно аппроксимировать расстояние между исходными векторами.

Для PQ есть множество продвинутых улучшений, в которые мы не будем вдаваться. Например, кодирование не абсолютного расположения точки в пространстве, а направления, в котором лежит исходный подвектор относительно центроида, т.е. вектор покоординатных разностей центроида и подвектора. Тогда похожесть оценивается сонаправленностью исходных векторов, а не глобальными координатами.

А пока перейдём к графовым методам приближённого поиска ближайших соседей.

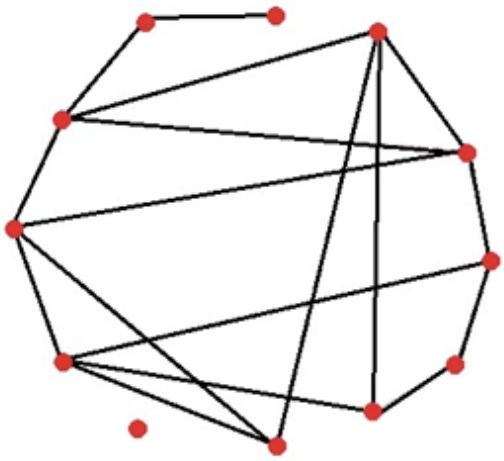
> Small World

Мир тесен — любые два человека на Земле разделены не более чем шестью уровнями связей (см. эксперимент доктора Милгрэма по доставке писем в другую часть Соединённых Штатов). Формальная математическая формулировка теории: диаметр графа знакомств не превышает 6.



В теории графов "мир тесен" определяется как сеть, в которой типичное расстояние L между двумя произвольно выбранными вершинами растёт пропорционально логарифму от числа вершин N в сети:

$$L \propto \log N$$



Граф "Мир тесен" имеет длинные и короткие "связи" (рёбра).

Рассмотрим пример графа: в нём есть всего одна изолированная недостижимая точка, а из любой другой можно перейти в произвольную точку этого графа.

Важной особенностью такого графа является наличие длинных и коротких рёбер. Под длиной здесь подразумевается удалённость точек в пространстве

или людей на планете.

> NSW и HNSW

Современные подходы к поиску ближайших соседей основаны именно на графах "мир тесен".

> NSW (*Navigable Small World*)

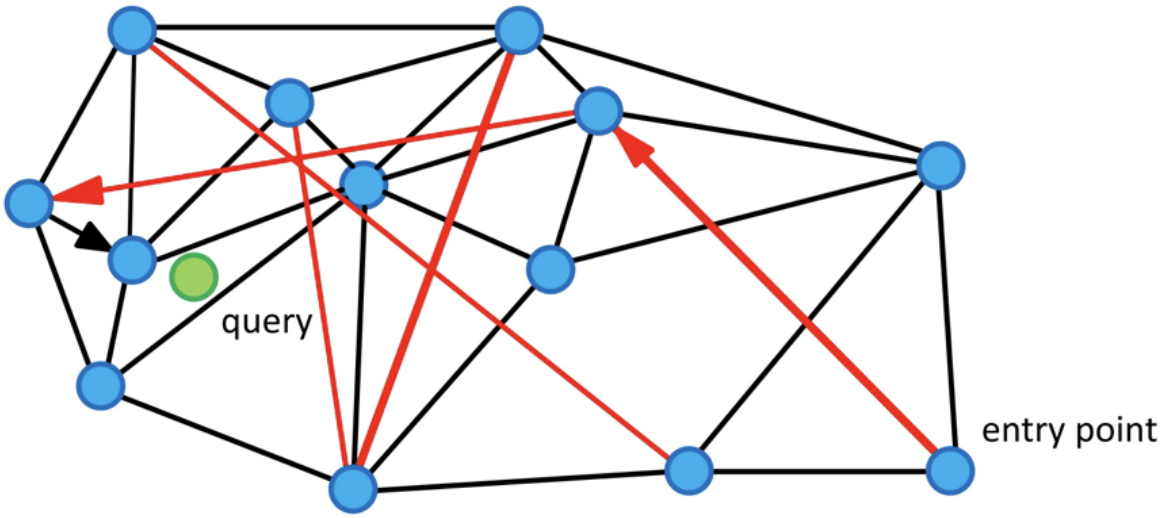
Сначала стоит задача построения такого графа по имеющимся векторам, или эмбеддингам для набора документов.

- Каждая точка такого графа представляет собой документ.
- Каждое ребро — запись в некоторой вспомогательной таблице, указывающая на существование возможности перехода из точки A в точку B.
- Граф ненаправленный, т.е. переходы взаимно обратны.
- Рёбра графа должны представлять и длинные, и короткие связи.

На этапе построения графа (так как это предварительная работа, которая выполняется разово, а не во время каждого запроса) можно высчитать много расстояний между точками. Скажем, для каждой точки случайно выбираются 5% точек из всей выборки, считается расстояние до них, и из 10 ближайших точек выбираются 5 точек, до которых прокладываются рёбра. Здесь 5 и 10 — это гиперпараметры алгоритма. Таким образом формируются короткие связи.

Из этих же 5% точек можно выбрать аналогичным образом ещё 5 точек, но уже максимально удалённых от исходной, и записать связи между ними в виде рёбер.

Повторим процедуру для всех точек и на выходе получим граф, в котором каждая точка связана как с удалёнными точками, так и с близлежащими. Это как раз то, что нам нужно.



Рассмотрим график: здесь красным цветом обозначены длинные ребра, чёрным — короткие. На вход поступает запрос, обозначенный зелёной точкой. Достаточно выбрать случайную точку из графа, рассчитать расстояния от связанных с ней точек до зелёной точки, характеризующей запрос, и передвинуться в точку с наименьшим значением расстояния.

Если взято крайне неудачное начальное приближение (та самая случайная точка), то прыжок будет совершён по длинной связи, что позволит сильно продвинуться и приблизиться к искомой точке. Здесь такой прыжок обозначен стрелкой на красном ребре в правой части изображения.

Итеративно повторив вышеописанную последовательность действий, в конечном итоге мы найдём точку, которая очень близка к искомой. Её можно назвать приближённым соседом. Поскольку на каждом этапе выбирается точка с минимальным расстоянием до целевой и может быть достигнут локальный минимум вместо глобального, то по сути это жадный (greedy) алгоритм, то есть стратегия формируется исходя не из глобальной задачи, а из выбора лучшего варианта каждый раз, как такой выбор предоставляется.

Чтобы улучшить качество и получить возможность находить действительно более близкие точки к заданной запросом, можно применить уже знакомые техники:

- Несколько инициализаций;
- Очередь с приоритетом.

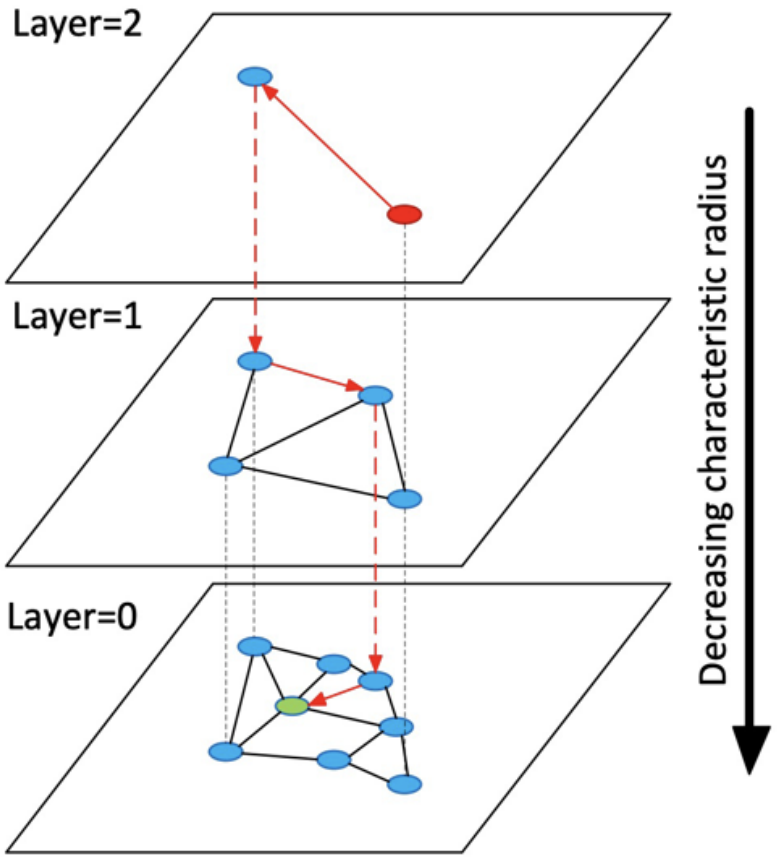
> HNSW (*Hierarchical Navigable Small World*)

HNSW используется в наше время очень часто. К примеру, именно этот алгоритм лежит под капотом у такого популярного решения для поиска схожих документов, как `Elasticsearch` в облаке Амазона. Также на нём основаны другие реализации, которые вы можете развернуть самостоятельно.

В названии метода указано *hierarchical*, или иерархический. В этом и заключается вся суть. Авторы подхода заметили, что выгоднее сначала перебирать только длинные связи для уточнения места в пространстве, где лежит искомая точка, а затем углубляться и перебирать всё более и более локальные связи.

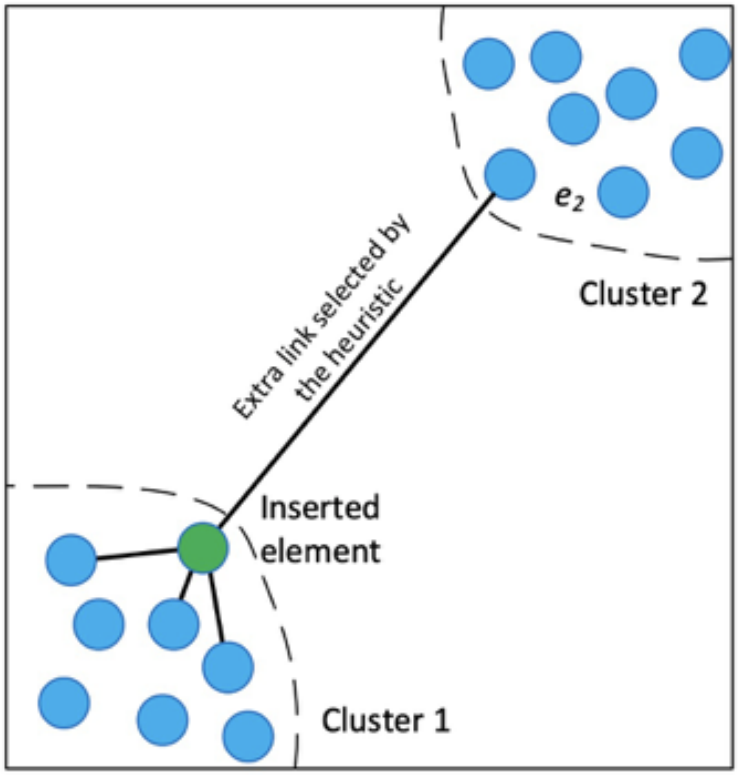
Если описывать всё более формально, то процесс поиска разбит на уровни. Слой с номером L содержит подмножество точек слоя $L-1$. Слой $L = 0$ содержит все точки (аналогично `NSW`).

Таким образом, останавливая поиск на определённом уровне на какой-то точке, можно опуститься на уровень ниже, который содержит больше точек, и продолжить итерации с того же места, но перебирая более локальные связи. Также при переходе на нижние слои можно увеличивать количество точек, которые будут выступать в качестве начальных для поиска.



Пример: на уровне с номером два используется всего одна точка.

Она сделала 4 прыжка. Тогда последнюю и предпоследнюю точку мы спроецируем на $L = 1$ уровень и начнём перебор с них. Затем выберем 4 точки в качестве локальных приближений для $L = 0$ уровня и будем продолжать перебор до тех пор, пока не найдём K точек, у которых нет рёбер к точкам, которые находились бы ближе к точке запроса Q, т.е. найденные точки — это идеальные кандидаты.



Некоторая эвристика позволяет добавлять точки в граф так, чтобы эффективно формировать и короткие, и длинные рёбра (см. картинку выше). Общая эффективность поиска при таком подходе не деградирует. Также с помощью некоторой вероятностной величины можно определять, на какие из L уровней стоит добавлять точку, исходя из созданных связей.

> Комбинации методов

FAISS — библиотека от Facebook. Поддерживает работу с GPU (для тех алгоритмов, где она возможна) и множество оптимизаций (даже для простого полного поиска на CPU). Индекс частично может быть сохранён на диске. Есть возможность быстро создавать разные индексы для поиска, комбинируя механики.

В одну строку можно создать индекс для работы с большим количеством данных:

```
index = faiss.index_factory(128, "PCA64,IVF16384_HNSW32,Flat")
```

В этом примере подразумевается:

- Работа с векторами размерности 128.
- Эта величина снижается до 64 методом главных компонент — на это указывает `PCA64` в начале строки, задающей индекс.
- Затем происходит кластеризация на чуть более чем 16 тыс. кластеров методом `K-Means`, и объекты, попавшие в каждый кластер, записываются в `IVF` индекс, простой файл, как мы обсуждали ранее.
- Поиск ближайших кластеров для рассмотрения будет осуществляться с помощью иерархической навигации по графу, где точки представляют собой центроиды кластеров. То есть происходит поиск не ближайших точек, а максимально релевантных кластеров для поиска кандидатов — всего будет выбрано 32 центроида, задающих 32 кластера из 16384. При этом поиск в рамках этих кластеров будет максимально честным, с полным переборов всех кандидатов из файлов обратного индекса, на что указывает алиас `Flat`.

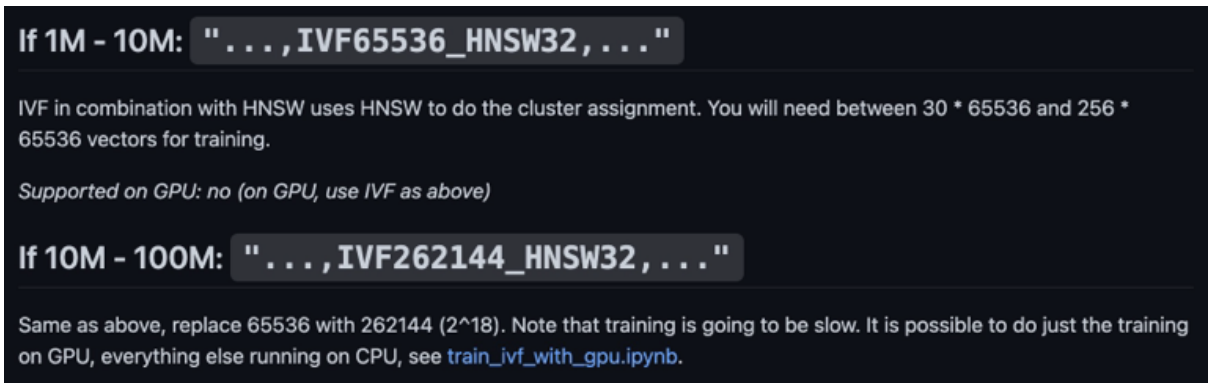
```
index = faiss.index_factory(dim, "IVF262144,PQ64", faiss.METRIC_L2)
```

Во втором примере более простой подход:

- Сначала создаётся целых 260 тыс. кластеров.
- Затем исходные вектора квантизуются в 64 бита. С одной стороны, это облегчает вычисления, ускоряет их, а также уменьшает расход памяти — как оперативной, так и на диске. С другой стороны, получается некоторое падение точности из-за сжатия векторов, однако по количеству кластеров (260 тыс.) можно предположить, что стартовое количество документов огромно, и по нему просто невозможно максимально точно за разумное

время производить поиск. Поэтому приходится идти на уступки с `Product Quantization`.

Гайдлайн FAISS по подбору параметров для задачи поиска предлагает использовать степени двойки в качестве количества кластеров.

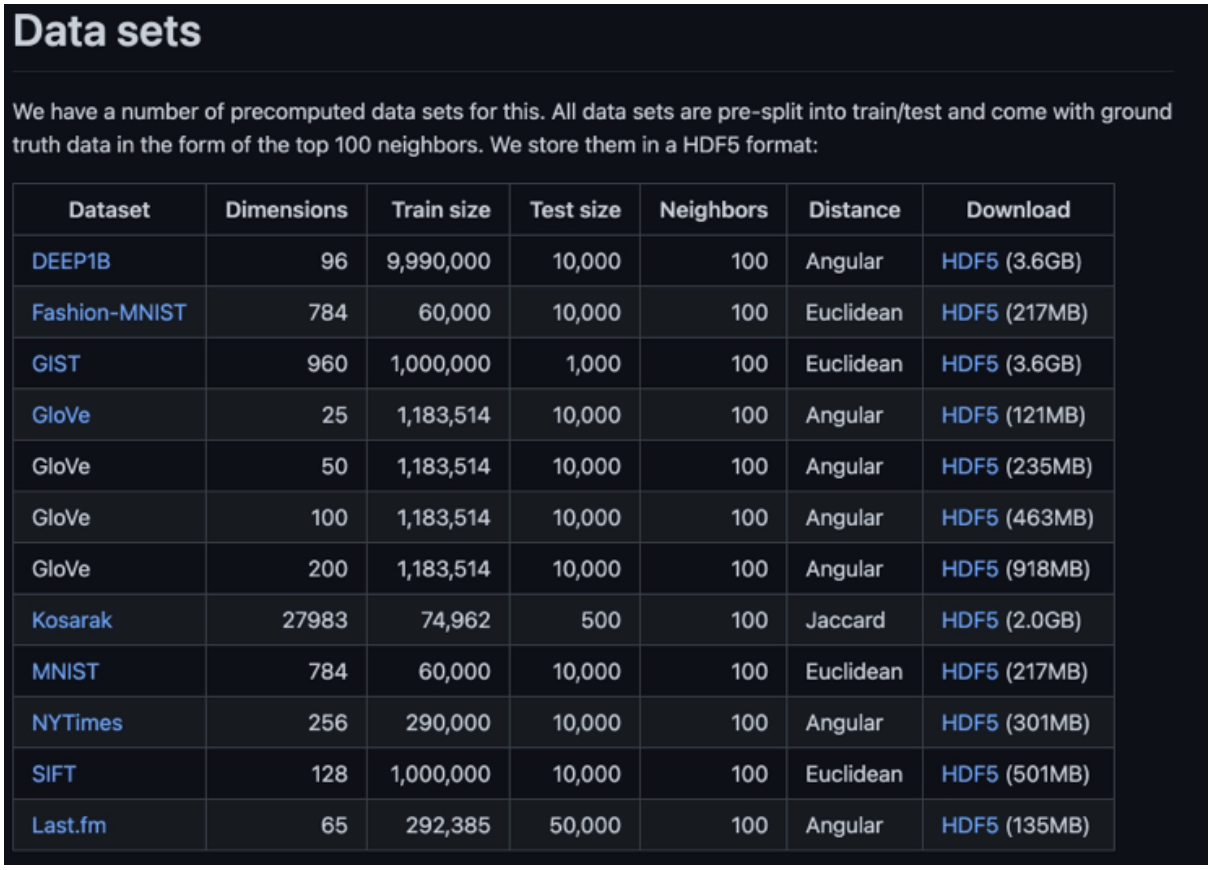


Примеры рекомендуемых комбинаций поиска для датасетов разного размера.

Приступая к решению прикладной задачи, полезно сначала оценить размеры базы документов, её особенности и пройтись по списку для определения того, какими методами стоит воспользоваться. Также следует попробовать несколько разных дистанций для расчёта близости векторов. Это может быть косинусное расстояние, `L1`, `L2` или что-то более сложное.

> Сравнение и оценка методов ANN

Бенчмарк большинства методов ANN: github.



Data sets

We have a number of precomputed data sets for this. All data sets are pre-split into train/test and come with ground truth data in the form of the top 100 neighbors. We store them in a HDF5 format:

Dataset	Dimensions	Train size	Test size	Neighbors	Distance	Download
DEEP1B	96	9,990,000	10,000	100	Angular	HDF5 (3.6GB)
Fashion-MNIST	784	60,000	10,000	100	Euclidean	HDF5 (217MB)
GIST	960	1,000,000	1,000	100	Euclidean	HDF5 (3.6GB)
GloVe	25	1,183,514	10,000	100	Angular	HDF5 (121MB)
GloVe	50	1,183,514	10,000	100	Angular	HDF5 (235MB)
GloVe	100	1,183,514	10,000	100	Angular	HDF5 (463MB)
GloVe	200	1,183,514	10,000	100	Angular	HDF5 (918MB)
Kosarak	27983	74,962	500	100	Jaccard	HDF5 (2.0GB)
MNIST	784	60,000	10,000	100	Euclidean	HDF5 (217MB)
NYTimes	256	290,000	10,000	100	Angular	HDF5 (301MB)
SIFT	128	1,000,000	10,000	100	Euclidean	HDF5 (501MB)
Last.fm	65	292,385	50,000	100	Angular	HDF5 (135MB)

Для оценки методов используется несколько датасетов. При выборе библиотеки и подхода хорошая практика — свериться с тем набором данных, который похож на ваш.

Варьируются:

- Размерность каждого вектора, что определённо влияет и на время поиска, и на затраты памяти;
- Размеры датасетов.

Здесь представлены как текстовые эмбеддинги, например `GLOVE`, схожий по логике с `Word2Vec`, так и векторы изображений — `MNIST`.

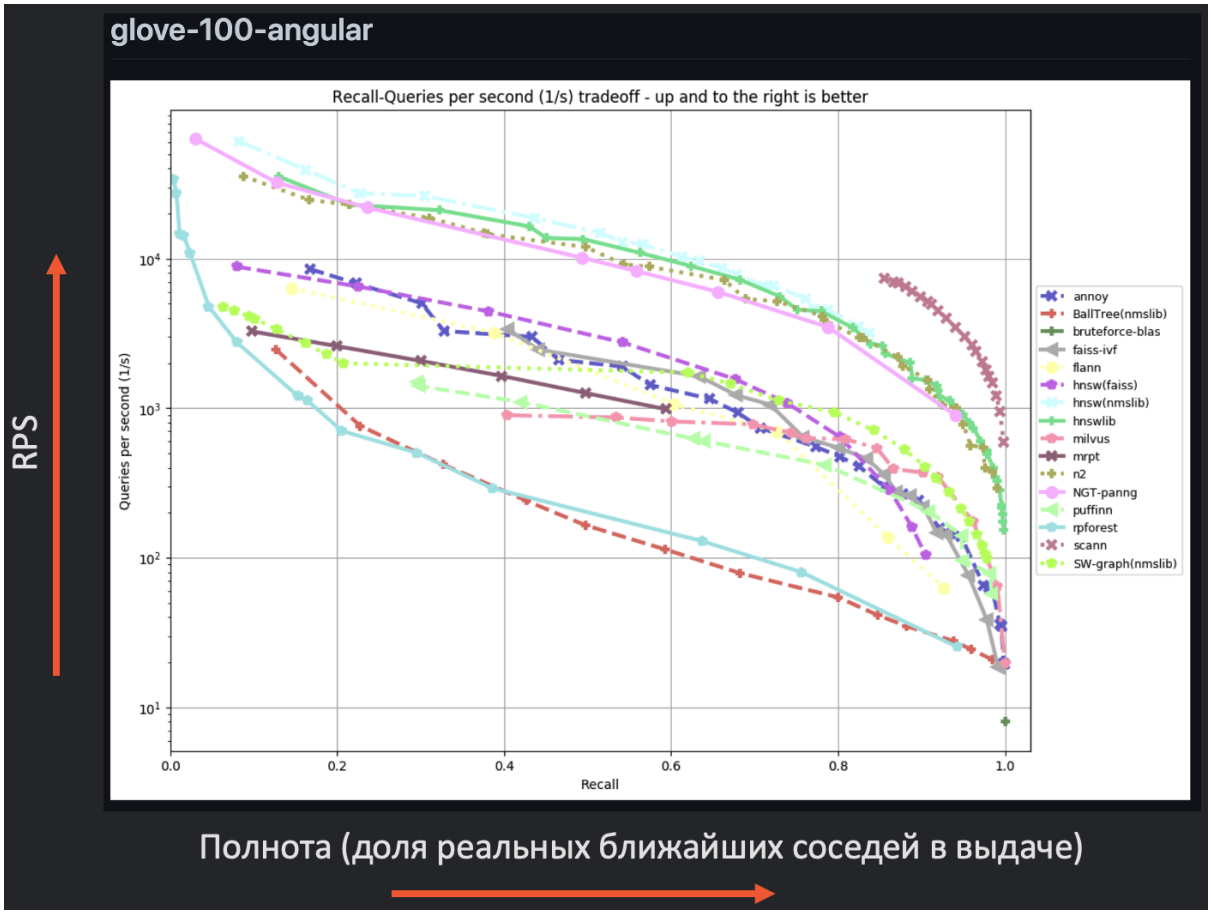
> Сравнение методов между собой

Рассмотрим на примере поиска похожих эмбеддингов `GLOVE` размерности 100.

Во время работы алгоритма меняются параметры, отвечающие за качество работы, и замеряется пропускная способность, т.е. сколько запросов в секунду (*RPS*) способна обработать система.

Качество работы оценивается следующим образом:

1. Для запросов заранее полным перебором были рассчитаны 100 ближайших соседей в корпусе документов. Ближе ним точно нет никаких других объектов.
2. Применяется поиск 100 ближайших соседей.
3. Рассчитывается доля истинных ближайших соседей в выдаче, т.е. это дробь, в числителе которой количество общих объектов между множествами настоящих ближайших соседей, рассчитанных заранее, и предложенных алгоритмом `ANN`



Такая мера называется полнотой, или `recall`, и чем ближе она к единице, тем точнее осуществляется поиск соседей. Но **чем выше полнота, тем ниже скорость работы** модели.

Соответственно, чем выше и правее на графике находится метод, тем лучше.

В самом нижнем правом углу зелёным крестиком отмечено значение пропускной способности при точном нахождении соседей и полном вычислении всех расстояний с учётом `BLAS`, т.е. процессорных оптимизаций вычислений.

Видно, что в таком случае количество обрабатываемых запросов не достигает и десятка.

Можно обратить внимание на салатовую линию в верхней части графика — это `HNSW`, с которым мы уже познакомились в реализации `hnswlib`. Если зафиксировать значение полноты на уровне 0.8, то пропускная способность будет около $7-8$ тысяч, что звучит очень неплохо — менее 1 миллисекунды на поиск в среднем для набора данных размера 1.1 млн.

Библиотека `ANNOY` лежит где-то посередине и обозначена тёмно-фиолетовой пунктирной линией с крестиками. По скорости работы она находится на уровне серой кривой со стрелочками, которая характеризует поиск по `IFV` индексу в реализации `FAISS` при разбиении пространства с помощью кластеризации на отдельные зоны, как было рассмотрено на занятии.

В верхнем правом углу выделяется метод с названием `ScaNN`, который невероятно хорошо работает при высоких порогах полноты: при той же скорости работы, что и `ANNOY`, он выдаёт значение метрики `recall` около 0.85 против 0.2 — это почти в 4 раза быстрее! `ScaNN` это один из самых новых и передовых методов, разработанных за последний год. Он опирается на продвинутую выучиваемую квантизацию векторов. Ключевое слово здесь "выучиваемая", т.е. нет жёстко заданной концепции, в отличие от `Product Quantization`. При желании вы можете ознакомиться с этим подходом: ссылка. Метод в большей степени опирается на математику, так что разобраться с наскока будет нелегко.

> Резюме

- Приближённый поиск ближайших соседей — важная часть систем ранжирования и матчинга, выполняющая роль кандидатной модели (выделение подмножества объектов для дальнейшей обработки).
- Основные идеи связаны с упрощением задачи методом разделения пространства на изолированные кластера. Для повышения точности можно обходить несколько кластеров либо брать несколько начальных инициализаций.
- Если данных много, то нужно применять квантизацию векторов — это слегка снизит качество, но улучшит время работы и вес индекса.
- $HNSW + PQ + IVF +$ метрика $=$ результат

- Библиотеки, с которых стоит начать: `nmslib`, `FAISS`, `ANNOY` (см. похожий датасет в бенчмарках, учитывая свой сценарий использования).