

Федеральное государственное автономное образовательное
учреждение высшего образования

«Национальный исследовательский университет
«Высшая школа экономики»

Факультет компьютерных наук

Основная образовательная программа
Прикладная математика и информатика

КУРСОВАЯ РАБОТА

**Программный проект на тему
«Сервис поиска соседних объектов»**

Выполнил студент группы БПМИ 195, 3 курса,
Амеличев Константин Вадимович

Руководитель КР:

Руководитель группы разработки, Tinkoff.ru, Тощев Андрей Александрович

Москва 2022

Содержание

1 Аннотация	2
2 Введение	3
2.1 Цель курсовой работы	3
2.2 Задача поиска ближайших векторов	3
2.3 Требования к SimSearch	4
2.4 Актуальность и значимость	5
2.5 Задачи курсовой работы	7
3 Обзор алгоритмов поиска	8
3.1 K-d деревья	8
3.2 Алгоритмы на основе хеширования	9
3.3 Алгоритмы на основе квантизации	11
3.4 HNSW	13
3.5 Сравнение алгоритмов	13
3.6 Изученные библиотеки	14
3.7 Сравнение библиотек	15
3.8 Выводы	16
4 Реализация SimSearch	17
4.1 API SimSearch	17
4.2 Архитектура SimSearch	17
4.3 Масштабирование	19
4.4 Объект поиска	19
4.5 Storage Manager	20
4.6 Storage	20
4.7 Vectorizer	21
4.8 Search	22
4.9 Тестирование SimSearch	22
4.9.1 Тестирование корректности	22
4.9.2 Тестирование производительности	23
5 Поиск похожих картинок на основе SimSearch	24
6 Заключение	27
6.1 Результаты	27
6.2 Дальнешее развитие	27

1 Аннотация

В рамках курсовой работы описывается исследование задачи поиска ближайших соседей и разработка поискового сервиса SimSearch. SimSearch позволяет находить объекты, максимально похожие по свойствам на запрошенный пользователем объект. Описывается возможность реализовать поисковое приложение для любого типа объектов, у которых определена функция перевода их характеристик в вектор: для изображений, документов, пользователей рекомендательной системы и прочего. Данный подход демонстрируется разработанным поиском похожих изображений на основе SimSearch.

Ключевые слова: Задача поиска ближайшего соседа, поисковые системы.

The course work describes the study of the problem of finding nearest neighbors and the development of the search service SimSearch. SimSearch allows you to find objects that are most similar in properties to the object requested by the user. The possibility is described to implement a search application for any type of objects that have a function for translating their characteristics into a vector: for images, documents, users of a recommender system, and others. This approach is demonstrated by the developed search for similar images based on SimSearch.

Keywords: Nearest neighbor search, search engine.

2 Введение

При работе с данными часто возникает необходимость в поиске некоторого элемента в имеющемся хранилище данных. В случае, если объекты в базе покрывают все возможные поисковые запросы, нужно искать точное совпадение в множестве. В случаях, когда база фиксирована, а поисковое пространство неограничено, принято искать не точное, а «похожее» совпадение: основные характеристики объектов должны либо совпадать, либо отличаться незначительно. Например, на «похожих» изображениях находятся объекты одного и того же класса: изображения двух разных чашек кофе считаются «похожими», но изображение чашки кофе не похоже на изображение гоночного автомобиля.

В рамках работы произойдет переход от интуитивного определения «похожести» к более формальной поисковой задаче, где объектом выступит вектор, а «похожесть» окажется связано с расстоянием между векторами.

2.1 Цель курсовой работы

Целью курсовой работы является создание сервиса SimSearch (от англ. *similarity search*), который будет находить объекты, «похожие» на объект, который запросил пользователь.

Важно заметить, что *объект* — абстрактное понятие, и единственное требование, на-кладываемое на него в рамках данной работы — векторизуемость, переносящее метрику «похожести» в некоторое расстояние между двумя векторами. Чем меньше расстояние между векторами, тем более похожи объекты. Такая векторизация называется *эмбеддингом* [1]. Соответственно, разработанный сервис должен быть гибким и доступным для использования с различными поисковыми пространствами: от пользователя требуется только определить функцию перевода объекта в вектор и предоставить необходимые данные для поиска.

2.2 Задача поиска ближайших векторов

Опишем задачу поиска соседних векторов более подробно. Пусть объект описывается

вектором $v = \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ v_d \end{pmatrix} \in \mathbb{R}^d$, а также у нас есть множество векторов $S \subseteq \mathbb{R}^d$. Таким об-

разом, S — это *датасет* из n векторов в d -мерном пространстве. Определим некоторое расстояние между векторами как $dist(v_1, v_2)$. Поиск «достаточно близких» векторов принято называть задачей *c-ANN* (Approximate Nearest Neighbors), где c — параметр задачи ($c > 1$). Положим q — искомый вектор, а $v^* = \arg \min_{v \in S} dist(v, q)$. Таким образом, v^* — самый близкий вектор к запрашиваемому. Ответом на задачу *c-ANN* считается вектор v , удовлетворяющий неравенству:

$$dist(v, q) \leq c \cdot dist(v^*, q)$$

Обычно в качестве расстояний используется одна из следующих метрик:

- Евклидово расстояние: $dist(v_1, v_2) = \sqrt{\langle v_1 - v_2, v_1 - v_2 \rangle}$
- Косинусное расстояние: $dist(v_1, v_2) = \frac{\langle v_1, v_2 \rangle}{\sqrt{\langle v_1, v_1 \rangle \cdot \langle v_2, v_2 \rangle}}$
- Расстояние Минковского: $dist(v_1, v_2) = \sum_{i=1}^d |v_{1,i} - v_{2,i}|$
- Расстояние Джакарда: $dist(v_1, v_2) = 1 - \frac{\sum_{i=1}^d [v_{1,i} = v_{2,i}]}{2 \cdot d - \sum_{i=1}^d [v_{1,i} = v_{2,i}]}$

Такие расстояния, как расстояние Джакарда или расстояние Хэмминга (частный случай расстояния Минковского) принято использовать на булевых векторах, каждая координата которого равна либо 0, либо 1. Это связано с тем, что в некоторых случаях сложно трактовать «численную близость» векторов, и гораздо проще думать о задаче в бинарном виде, с подходом из теории множеств: единица соответствует наличию «признака», а 0 его отсутствию. Подобный подход встречается, например, в машинном обучении, под названием One Hot Encoding [2].

Также есть понятие задачи c - k -ANN, где ответом является множество из k различных векторов, каждый из которых является ответом на задачу c -ANN.

2.3 Требования к SimSearch

Кроме главного функционального требования, которое мы накладываем на SimSearch (решение задачи поиска соседних объектов), имеет смысл сформулировать дополнительные требования: необходимый функционал сервиса, возможные характеристики нагрузки.

Функционал, помимо решения задачи поиска соседних объектов, включает в себя следующие пункты:

- Одна инсталляция SimSearch поддерживает различные типы объектов независимым образом — например, поддерживается работа одновременно с изображениями и текстами.
- SimSearch является масштабируемым [40] — он может быть запущен на нескольких серверах [41], которые будут пользоваться одной общей базой данных.
- SimSearch поддерживает запросы добавления, изменения и удаления объектов из множества.
- SimSearch имеет настраиваемые стратегии поиска объектов, выбор стратегий ложится на пользователя сервиса.
- SimSearch хранит не только информацию об объекте, необходимую для поиска, но и мета-информацию, которая может быть полезна пользователю сервиса.

Нагрузка может быть трех видов:

1. Статическая: множество объектов для поиска фиксировано и не меняется. Может использоваться для данных, собранных разово и не меняющихся. Например, при построении систем для рекомендации классической литературы — новые объекты в базе не появляются.
2. Динамическая: множество объектов постоянно меняется, происходит добавление или удаление объектов. Общее ограничение на размер множества известно заранее. Например, это полезно для нахождения пользователей со схожими интересами и использования этих данных в рекомендательных системах.
3. Нерегулярные изменения: множество объектов изменяется редко, но изменение или добавление затрагивает множество объектов. Такая нагрузка может обрабатываться так же, как и статическая, при массовых обновлениях достаточно просто заменить старое множество на новое. Такие обновления происходят редко, и можно перестроить базу с нуля, так что такая нагрузка будет обрабатываться так же, как и статическая.

2.4 Актуальность и значимость

Идея поиска похожих объектов достаточно хорошо исследована для специфичных объектов (например, для изображений или текста). Сервисы, которые реализуют подобные поиски, обычно завязываются на специфику задачи, за счет чего перестают быть применимыми в других задачах.

Например, Shazam^[3] решает задачу поиска похожих аудиозаписей — сервис находит фрагмент песни в поисковом индексе, максимально похожий на аудиофрагмент, записанный пользователем. Но подобный сервис нельзя использовать с другими типами данных, даже в похожих сценариях использования. Например, пользователь Shazam не может сфотографировать картину в музее и узнать, что это за картина, хотя поисковые задачи явно похожи друг на друга.

Есть и более нестандартные варианты. Например, существует веб-сервис для поиска других веб-сервисов [33], где объекты являются описанием веб-сервисов. Таким образом, можно найти онлайн-калькулятор или сервис для компьютерного перевода с английского на русский языки. Разумеется, подобный сервис не применим для любых других поисковых задач.

Сервисов же, которые решают универсальную поисковую задачу, не существует. В отличие от заточенных на специфичные объекты сервисов, SimSearch реализует некоторую общую логику и предлагает модель построения универсального поискового сервиса. Сервис, описываемый в данной работе, не завязывается на специфику объектов. Таким образом, поисковая задача разбивается на две подзадачи — задачу эмбеддинга [1] и задачу поиска соседних векторов в многомерном пространстве.

SimSearch готовит для пользователя такую модель решения поисковой задачи, что при возникновении любой специфической задачи поиска, пользователю SimSearch нужно ду-

мать только о переводе задачи поиска соседних объектов в задачу поиска соседних векторов с помощью эмбеддинга. Непосредственно разработкой поискового движка заниматься не потребуется, так как за это отвечает SimSearch. Более того, для общей логики работы с часто встречающимися объектами (например, изображениями) эмбеддинги можно переиспользовать, что заметно ускоряет разработку прототипов нужных приложений.

Некоторые возможности для применения сервиса:

1. Поиск референсов [39] в базе доступных картинок. Дизайнер, имеющий некоторое изображение, может найти похожее изображение, на которое у компании приобретены авторские права. Такое веб-приложение было разработано в рамках данной работы.
2. Рекомендательные системы. Зная, что пользователь А похож на пользователя В, можно предлагать пользователю В то, что понравилось пользователю А и не предлагать то, на что А отреагировал негативно. Таким образом, по критериям поиска можно определять аудиторию клиентов со схожими интересами. Всем представителям одной аудитории можно рекомендовать одинаковые или похожие интересные предложения.
3. Поиск песен, которые могут понравиться человеку на основе другой песни, которая ему уже нравится.

Подробное изучение задачи эмбеддинга останется за рамками данной работы. Отметим, что для изображений можно использовать сверточные нейронные сети [4], для аудиосигналов [25], рекомендательных систем [21] и для текстов [34] также существуют свои эмбеддинги.

Можно заметить, что SimSearch не предлагает качество поиска выше, чем у специальных сервисов. Зато есть достаточно применений, в которых от сервиса нужно не столько хорошее качество сравнения, сколько отсев объектов, которые гарантированно не подходят под поисковый запрос. Например, такое актуально, когда поиск делается на небольших объемах данных. В терминах задачи c - k -ANN можно привести такой пример: пусть специализированный сервис решает задачу с параметром $c = 1.01$, а SimSearch решает эту же задачу с параметром $c = 1.02$. Тогда, с одной стороны, специализированный сервис давал погрешность вдвое меньшую, чем SimSearch. С другой стороны, на базах размером несколько тысяч объектов это не будет сильно отражаться на качестве поиска, в область со свойством $1.01 \cdot dist(v^*, q) < dist(v, q) \leq 1.02 \cdot dist(v^*, q)$ попадет мало объектов v , а значит, они не сильно ухудшат поисковую выдачу.

Главные преимущества использования SimSearch вместо специализированных сервисов:

Гибкость. SimSearch дает пользователю возможность невероятно гибкого использования. При решении конкретной поисковой задачи SimSearch предоставляет возможность гибко задать правила, влияющие на поисковую выдачу, ведь он разрешает настройку произвольного эмбеддинга и выбор нужной поисковой стратегии.

Более того, мощность решаемых SimSearch поисковых задач упирается только в мощность эмбеддинга. Например, есть возможность создать две модели эмбеддинга — первая переводит слова в пространство \mathbb{R}^d , вторая переводит изображения в пространство \mathbb{R}^d . Если модели на стадии обучения «знали» друг о друге и похожим объектам (картинке, похожей на свое текстовое описание) соответствуют похожие точки, то тогда поисковая задача для SimSearch окажется общей для двух типов. Например, это может быть полезно при получении изображения по текстовому описанию.

Универсальность. При регулярном решении различных поисковых задач поможет то, что у SimSearch одинаковый интерфейс для всех объектов, на которых ведется поиск. Таким образом, при возникновении задачи поиска в стороннем сервисе разработчику не потребуется искать новый поисковый сервис и делать с ним интеграцию. Это заметно уменьшает ресурсы на разработку, если разработчик решает задачу поиска сразу в нескольких вариациях.

Переиспользование кода. Если разработчик SimSearch добавил нестандартный алгоритм поиска (например, для какой-то необычной функции расстояния), он становится доступен всем остальным пользователям SimSearch, вне зависимости от объектов, для которых решалась поисковая задача. Таким образом, если в SimSearch существуют n различных эмбеддингов и m различных поисковых алгоритмов, то конфигурации сервиса можно настраивать по принципу «каждый с каждым» — всего $n \cdot m$ различных конфигураций.

Сервис планируется использовать в качестве внутреннего поисковика для различных объектов финансового мобильного приложения — изображений, иллюстрирующих «истории», пользователей, ресторанов и прочего.

Сервис, похожий на SimSearch, был разработан в Google Cloud для демонстрации возможностей платформы Google Cloud Platform [5]. Авторы реализуют сервис для семантического поиска в тексте, в некотором смысле делая связку между эмбеддингом текста в векторы и библиотекой для решения задачи c - k -ANN. Тем не менее, несмотря на похожую идею, авторами была решена конкретная поисковая задача для текстов, их решение не применимо для объектов другого типа.

2.5 Задачи курсовой работы

Разработанный сервис должен решать задачу c - k -ANN, иметь веб-интерфейс для взаимодействия с клиентами, а также поддерживать работу с несколькими типами хранилищ одновременно.

Таким образом, в данной курсовой работе решаются следующие задачи:

1. Анализ задачи c - k -ANN и методов ее решения.
2. Проектирование сервиса поиска соседних объектов.
3. Разработка, отладка и тестирование SimSearch.
4. Разработка веб-приложения для поиска похожих изображений на основе SimSearch.
5. Описание полученных результатов в тексте курсовой работы.

3 Обзор алгоритмов поиска

Поскольку для решения задачи поиска соседних объектов нужен предварительный эмбеддинг, универсального способа решить задачу поиска не существует. В свою очередь, задача поиска ближайших векторов достаточно хорошо изучена. Одна из задач работы — сделать обзор существующих подходов, чтобы найти наиболее подходящий под задачу.

3.1 K-d деревья

Алгоритмы на основе деревьев [38] разбивают поисковое пространство полуплоскостями в гиперпространстве. На рис. 3.1 можно посмотреть пример для пространства \mathbb{R}^2 . K-d дерево — двоичное дерево, которое строится над множеством объектов для поиска — каждый узел дерева содержит один вектор из множества. Дерево строится «сверху вниз». Каждая новая вершина выбирается так, чтобы решить задачу разбиения: половина текущего множества вершин окажется в ее левом поддереве, другая половина — в правом поддереве. Для разбиения в самой тривиальной реализации можно использовать сравнение фиксированной координаты с соответствующей координатой вектора в вершине дерева. Таким образом, чтобы множество разбилось на две приблизительно равные части, сначала можно выбрать функцию, по которой вести сравнение (самое простое — выбрать одну из координат), а затем положить в вершину вектор из множества, который дает медианное значение функции. Полученная вершина разобьет множество на две части, и для них запустится рекурсивный процесс построения (рис. 3.2). Итоговая глубина дерева получится равной $O(\log n)$.

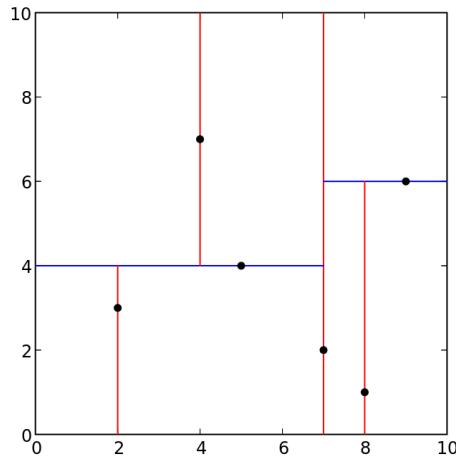


Рис. 3.1: Разбиение плоскости на области с помощью k-d-дерева. Первое разбиение делается по x-координате относительно точки $(7;2)$, чему соответствует вертикальная красная прямая при $x = 7$.

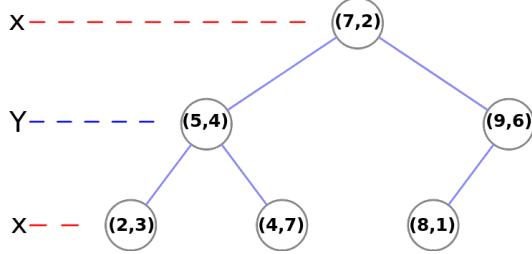


Рис. 3.2: Структура k-d-дерева для приведенного выше набора точек. Можно видеть, что глубина такого дерева равна трем, хотя подобное дерево создает 7 отличающихся областей.

Чтобы сделать поиск объекта, нужно «спуститься» вектором по дереву (рис. 3.3), определяя функцию на каждом уровне и делая сравнение с разделяющей вершиной. Это определит позицию вектора в дереве. Для поиска k соседей кроме позиции нужно найти некоторую окрестность, результаты в которой считаются позволительными. Обычно для этого предлагается следующее решение — при спуске по дереву алгоритм может сделать рекурсивный поиск у обоих сыновей вершины, если вектор запроса имел значение функции, близкое к значению разделяющего вектора. Это эквивалентно тому, чтобы сделать запрос поиска для каждой точки из некоторой окрестности вектора запроса.

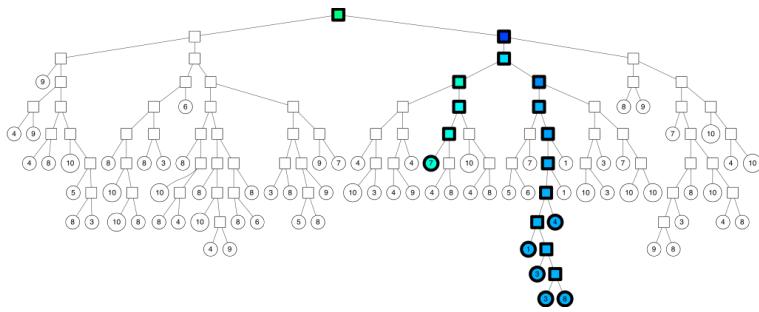


Рис. 3.3: Запрос на подмножестве поискового дерева рассматривает разных соседей одной вершины для продолжения поиска.

К достоинствам этого класса алгоритмов относится наглядность, возможность динамически менять поисковый индекс. Но, как правило, такие алгоритмы плохо работают для векторов больших размерностей, потому что неявно оставляют только $O(\log n)$ отличительных признаков у каждого объекта.

3.2 Алгоритмы на основе хеширования

LSH Идея алгоритма LSH [27] в том, чтобы отсесть все «непохожие векторы» с помощью k случайных хеш-функций. Более конкретно, для заданной задачи выбирается специальное семейство хеш-функций H , из которого выбирается k случайных хеш-функций. Часто хеш-функции являются бинарными, за счет чего из полученных k значений можно сформировать одно новое число. Объекты считаются одинаковыми, если все значения хеш-функций совпали (рис. 3.4). В некоторых вариациях алгоритма совпасть должны не все значения, а некоторая доля. Таким образом, при поиске достаточно рассматривать элементы, имеющие такое же значение хеш-функции, как и у искомого вектора.

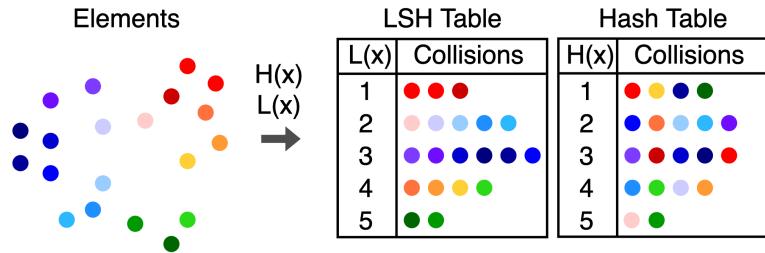


Рис. 3.4: Хеш-функция LSH для красных, похожих, объектов имеет значение 1, в то время, как для обычной хеш-функции красные объекты распределяются случайно.

Примером LSH-функции для бинарных векторов и расстояния Хэмминга может быть отдельно взятая координата: тогда при k взятых координатах векторы с одинаковым значением хеш-функций имеют хотя бы k общих бит — а значит, метрика Хэмминга между такими векторами с большей вероятностью оптимальна, чем с теми объектами, где значение хеш-функции не совпало.

Для косинусного расстояния существует другое семейство LSH-функций: $h_i(x) = \text{sgn}(x, w_i)$, где w_i — случайная полуплоскость, проходящая через точку 0. Тогда несколько полу плоскостей разбивают пространство на секторы, в каждом из которых лежат векторы с соседними углами (рис. 3.5).

Асимптотика для времени ответа на запрос и памяти для этих алгоритмов регулируемая и зависит от соотношения количества координат исходного вектора и количества хеш-функций.

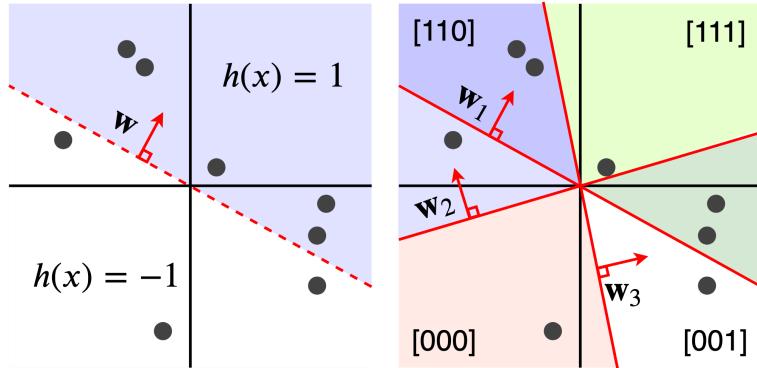


Рис. 3.5: Объекты с близкими косинусными расстояниями окажутся в одинаковых областях. Например, область [001] содержит все объекты, у которых $h_0(x) = -1, h_1(x) = -1, h_2(x) = 2$.

Вариаций LSH очень много, данная работа [29] делает обзор имеющихся результатов в сфере локального хеширования. Дополнительно опишем DMLSH, как метод, позволяющий обновлять поисковый индекс динамически.

DMLSH. Это алгоритм, представленный в 2013, который умеет динамически менять поисковый индекс [42]. Авторы алгоритма предлагают строить индекс «на ходу», постепенно добавляя в индекс новые хеш-функции. Такой алгоритм помогает решать задачу $c\text{-}k\text{-ANN}$ динамически с помощью локального хеширования.

Структура DMLSH строит дерево (рис. 3.6), аналогичное k-d-дереву, но строит его не над пространством \mathbb{R}^d , а над пространством значений хеш-функций. Если вершина дерева соответствует большому числу объектов, вершина создает новые листья, добавляя еще одну lsh-функцию.

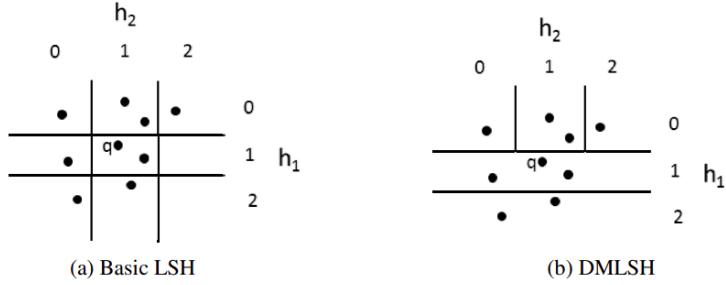


Рис. 3.6: DMLSH не создает хеш-функции без необходимости, за счет чего поисковый индекс можно менять динамически. На картинке добавлять вторую хеш-функцию h_2 нужно только для большой группы векторов с $h_1(v) = 0$.

К сожалению, проверенной реализации DMLSH в открытом доступе найти не получилось. Тем не менее, у LSH есть многообещающая возможность гибко настраивать количество хеш-функций, за счет чего поиск может работать очень быстро.

3.3 Алгоритмы на основе квантизации

Этот класс алгоритмов «сжимает» пространство, в котором ведется поиск. А именно, пространство \mathbb{R}^d меняется на $C \subset \mathbb{R}^d, |C| = k \in \mathbb{N}$. Точки в новом пространстве называются центроидами. Исходному вектору x в соответствие ставится ближайший центроид $q(x) \stackrel{\text{def}}{=} \arg \min_{c \in C} \text{dist}(c, x)$ (рис. 3.7). Таким образом, для каждого объекта достаточно хранить номер центроида, который занимает $\log k$ бит. Если кроме номера центроида для каждой вершины сохранить расстояние до центроида, то можно хранить сжатое множество, с небольшой потерей точности — теперь любой поиск расстояния: $\text{dist}(A, B) \simeq \text{dist}(A, q(B)) + \text{dist}(q(B), B)$ (рис. 3.8).

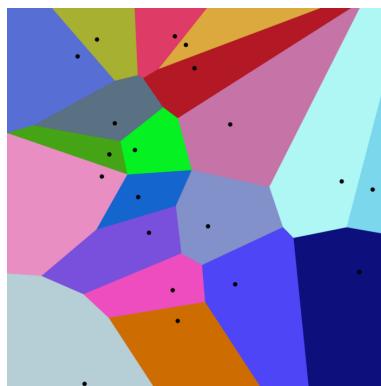


Рис. 3.7: Точками обозначены центроиды, а цветными областями разделены точки, соответствующие разным центроидам. Такая конструкция называется диаграммой Вороного

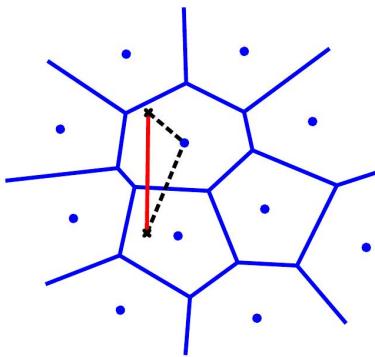


Рис. 3.8: Поиск расстояния теряет некоторую точность, потому что одна сторона треугольника меньше, чем сумма двух других.

Разные алгоритмы для квантизации предлагают разные функции $q(x)$. Самая простая такая функция может находиться с помощью алгоритма K-means методом выделения кластеров в данных (рис. 3.9). Центром кластера становится новый квант. Такой метод плохо применим, когда кластеров много, поэтому нужно рассмотреть другие техники.

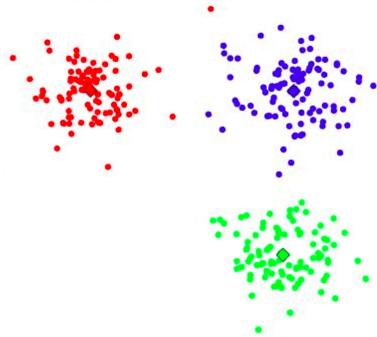


Рис. 3.9: Квантизация с помощью K-means

Метод Product Quantization ищет центроиды не для всего вектора размера d , а для его подвекторов меньших размеров. Затем полученные центроиды связывают вместе, создавая сетчатую структуру (рис. 3.10). Таким образом, задачу кластеризации требуется решать на пространстве меньшей размерности, а при объединении количество доступных центроидов заметно увеличивается, не требуя дополнительной памяти.

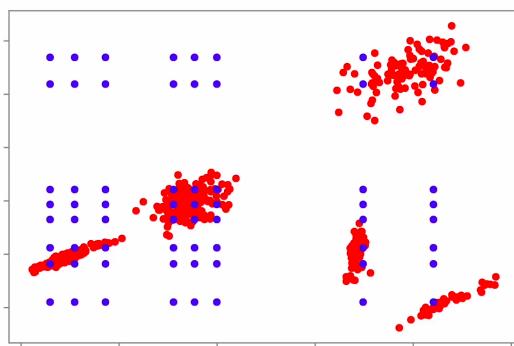


Рис. 3.10: Определение центроидов для Product Quantization. Синие центроиды были получены как декартово произведение центроидов по x-координате и центроидов по y-координате

К сожалению, у данного метода не получилось найти вариации, применимые в динамической постановке задачи. Это можно объяснить тем, что алгоритме на основе квантизации опираются на найденные центроиды, а изменение базы может достаточно сильно повлиять на множество центроидов.

3.4 HNSW

Алгоритм HNSW [32] строит граф связей между соседними объектами в несколько уровней — на верхних уровнях находится меньшее количество объектов, на нижних находится большее количество. Вектору соответствует вершина в графе. Каждая вершина соединена с k соседями. При запросе на поиск ближайшего поиска используется алгоритм локальной оптимизации [6]. А именно, если у заданной вершины есть сосед с меньшим расстоянием до искомого вектора, то алгоритм переходит к заданной вершине. Если такого соседа нет, то алгоритм переходит на более низкий уровень и продолжает поиск (рис. 3.11). В полученной структуре данных можно производить динамическое обновление индекса.

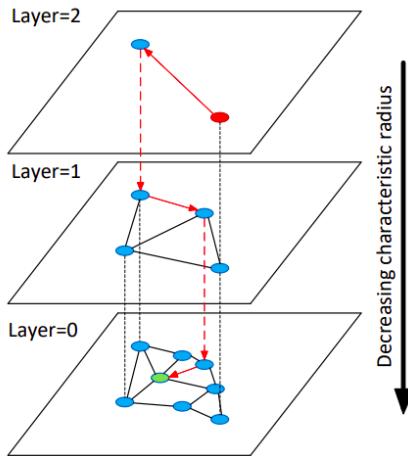


Рис. 3.11: Структура слоистого графа в алгоритме HNSW. При запросе поиска алгоритм делает поиск по слоям — сначала на первом слое находится самая близкая вершина, а затем делается переход на следующий слой.

3.5 Сравнение алгоритмов

При сравнении алгоритмов по основным критериям: времени обработки запросов, требуемой памяти, возможности динамической работы, получилась таблица для четырех рассмотренных подходов (табл. 3.1).

Таблица 3.1: Сравнение ANN-алгоритмов [7]. Размер базы n , размер одного вектора d , количество центроидов q . За T_h обозначим время работы lsh-функции $h : \mathbb{R}^d \rightarrow \mathbb{R}^k$

	Время	Память	Динамический?	Комментарий
K-D 3.1	$O(d \log n)$	$O(nd)$	да	лучше маленькие d
Квантизация 3.3	$O(qd)$	$O(qd + n \log q)$	нет	потеря точности
LSH 3.2	$O(T_h \log n)$	$O(nk)$	да	
HNSW 3.4	$O(d \log n)$	$O(nd)$	да	

Считая, что в SimSearch векторы редко бывают небольшого размера, HNSW оказывается предпочтительнее K-d деревьев. Квантизация не дает динамически обновлять базу данных и теряет в точности. Таким образом, есть смысл в первую очередь добавить стратегии на основе LSH и HNSW.

3.6 Изученные библиотеки

Были изучены крупные библиотеки с открытым исходным кодом, покрывающие различные подходы к решению задачи $c\text{-}k\text{-ANN}$. Среди критериев для выбора (которым подошли не все библиотеки), были:

- Наличие нескольких вариаций поискового алгоритма в одной библиотеке.
- Возможность подключения в проект на C++.
- Приемлемое время работы.

Annoy. Библиотека от Spotify [22], использующая в своем ядре KD-дерево (рис. 3.12). Авторы отмечают, что библиотека заточена под векторы небольшой размерности, но применять ее можно и при больших размерностях. Библиотека поддерживает C++/Python.

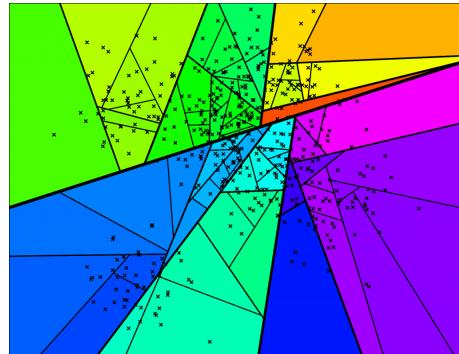


Рис. 3.12: Разбиение плоскости до листьев дерева в annoy

LSHKIT. LSHKit [8] — библиотека от автора статьи про LSH [26]. Библиотека поддерживает несколько видов расстояний и реализует LSH «как из статьи», с возможностью кастомизировать алгоритм под требуемую функцию расстояния. LSHKit не поддерживает динамическое обновление индекса. Динамический индекс для LSH не получилось найти в open-source, только в статьях. Библиотека поддерживает C++.

ScANN. ScANN [36] — библиотека от Google, выпущенная в 2020 году [28]. Эта библиотека используют специальную функцию потерь для определения центроида в группе, показывает state-of-the-art результаты. Библиотека поддерживает Python, но исходный код написан на C++, так что теоретическая возможность добавления в проект есть.

FaiSS. FaiSS [35]— библиотека от Facebook, выпущенная в 2017 году [31]. Библиотека использует большое количество инженерных решений, в частности умеет работать с GPU. Большим преимуществом этой библиотеки является большое количество реализованных алгоритмов: HNSW, LSH, Quantizer. Библиотека поддерживает C++/Python.

DiskANN. Библиотека от Microsoft, которая реализует графовые поисковые алгоритмы, являющиеся некоторым продолжением HNSW [30]. Библиотека поддерживает C++.

NMSLib. Библиотека NMSLib [23] используется в Elasticsearch [24] [9]. Эта библиотека, с одной стороны, реализует сразу несколько высокопроизводительных алгоритмов поиска; с другой стороны, их алгоритмы подходят для не-метрических поисковых пространств. Библиотека поддерживает C++/Python.

3.7 Сравнение библиотек

Все приведенные работы так или иначе решают одну и ту же задачу, при этом не дают точного ответа на то, что и когда предпочтительнее использовать. Существует большое исследование производительности подобных библиотек на разных датасетах [10] (рис. 3.13), но сделать вывод о тотальном доминировании одного из алгоритмов нельзя, потому что многое зависит от характера нагрузки. Заметим также, что все сравнение основано не на абстрактной скорости, а на queries per second относительно полноты поиска.

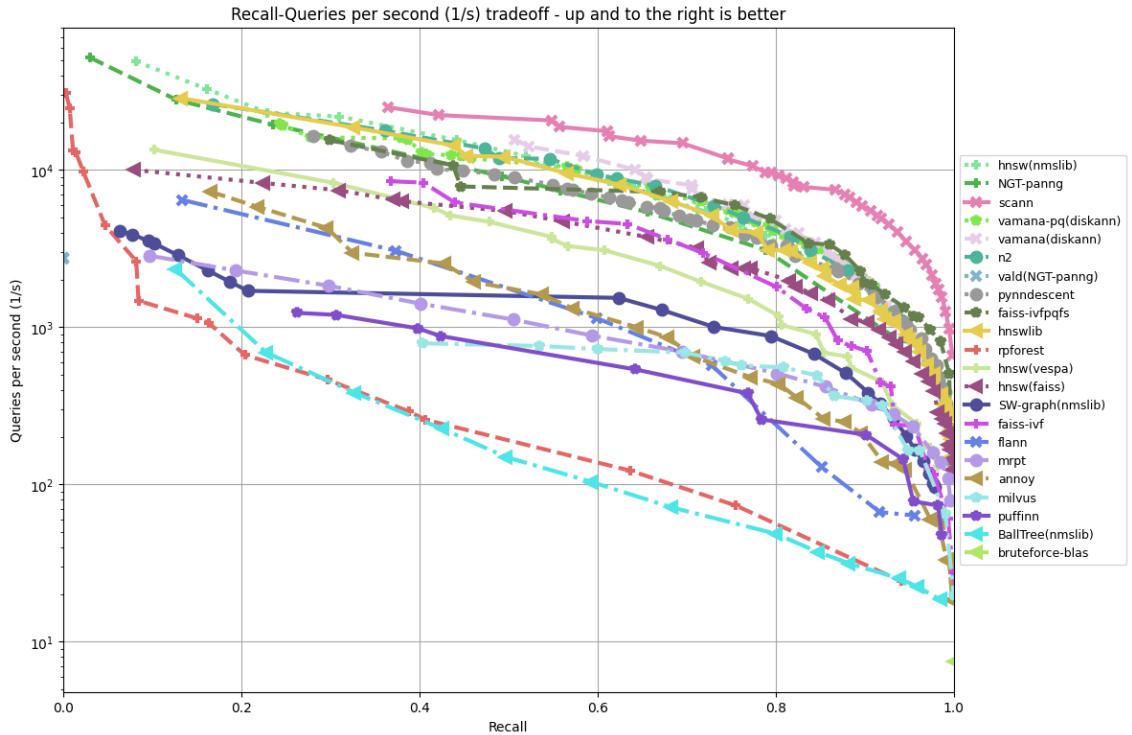


Рис. 3.13: Результат сравнения библиотек для решения задачи ANN [10] (тест glove-100-angular, делался поиск 10 ближайших векторов). Тестирование проводилось на L2-метрике (то есть, оптимизировалось Евклидово расстояние). Разные конфигурации DiskANN, Scann, Nmslib, Faiss в списке лучших.

3.8 Выводы

Было принято решение добавить в SimSearch поисковую стратегию на основе линейного поиска для небольших объемов данных и тестирований поисковой выдачи, библиотеку Lshkit для LSH-алгоритма — LSHKit проверенная временем и небольшая, в будущем ее можно дополнить алгоритмом DMLSH. Помимо этого, она поддерживает различные расстояния, в частности расстояние Хемминга. Также была добавлена библиотека Faiss (HNSW) по двум соображениям — во-первых, Faiss предлагает множество стратегий (не только HNSW), во-вторых, это одна из лучших реализаций HNSW с открытым исходным кодом на C++ в исследовании ann-benchmark [10], при условии поддержки динамического обновления индекса. В дальнейшем есть смысл добавить стратегии на основе квантизации и k-d-деревьев, а также реализовать DMLSH.

4 Реализация SimSearch

SimSearch реализован как сервис на C++, предоставляющий API для решения поисковой задачи.

4.1 API SimSearch

SimSearch реализует HTTP REST API [11].

1. GET `/status` — запрос статуса сервиса, возвращающий подтверждение, что все хранилища настроены и сервис работает.
2. GET `/search?id=ID&storage=STORAGE&n=N` — запрос поиска N объектов, похожих на объект ID, в хранилище STORAGE.
3. POST `/search?storage=STORAGE&n=N` — запрос поиска N объектов, похожего на объект, переданный в теле запроса, в хранилище STORAGE. Используется, если нужно передать метаинформацию для векторизатора.
4. POST `/add?storage=STORAGE` — добавление объекта, переданного в теле запроса, в хранилище STORAGE.
5. POST `/remove?storage=STORAGE` — удаление объекта, переданного в теле запроса, из хранилища STORAGE.

4.2 Архитектура SimSearch

Сервис сделан на основе модели хранилищ и менеджера (рис. 4.1). Менеджер перенаправляет приходящие запросы к нужному хранилищу. Хранилище — структура, содержащая информацию о данных в базе, а также *векторизатор* и *поисковую стратегию*. Векторизатор определяет правило перевода объекта в вектор (4.7), а поисковая стратегия отвечает за решение задачи поиска соседних векторов (4.8). Векторизатор и стратегия поиска определяются независимо от конкретного хранилища. Таким образом, сервис получает возможность гибко определять ключевые части хранилища, не зависеть от логики других хранилищ, и переиспользовать код там, где это возможно.

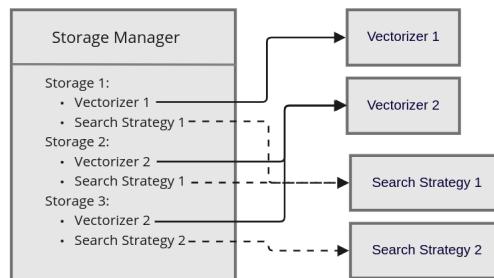


Рис. 4.1: Схема логики менеджер-хранилище. Можно видеть, что для трех хранилищ, которые суммарно используют 6 модулей, реализовать нужно всего 4 модуля, так как некоторые совпадают. Например, Vectorizer 2 используется и в Storage 2, и в Storage 3.

Предполагается, что сервис поддерживает несколько методов векторизации по умолчанию, а необходимые пользователю сторонние методы выполняются либо на стороне пользователя, либо в интеграции с сервисом. Иначе говоря, в сервисе можно задать векторизаторы «по умолчанию», а более сложные запросы придется реализовать на стороне клиента в удобном для пользователя формате. В таком случае хранилище потребует от пользователя обычные векторы.

Подобная структура не только сделает пользование сервисом гибким, но также позволит экспериментировать с алгоритмами поиска, так как заранее сложно предсказать, какой из предложенных алгоритмов покажет себя лучше всего. Также это даст возможность оптимизировать разные расстояния [2.2](#) в зависимости от задачи [2.4](#).

Опишем последовательность действий, которая используется SimSearch для обработки запросов (рис. [4.2](#)):

1. Получить запрос от пользователя на API. Получить нужную информацию: идентификатор Storage, идентификатор объекта, метаинформацию объекта.
2. С помощью StorageManager получить объект Storage, который обработает запрос.
3. Storage получает из объекта вектор с помощью Vectorizer.
4. При работе с вектором есть два варианта:
 - 4.1. При запросе поиска полученный вектор отдается в поисковую стратегию для получения поисковой выдачи.
 - 4.2. При запросе модификации делается запрос изменения в Redis [[12](#)], затем происходит обновление поискового индекса.
5. Возвращается ответ на запрос.

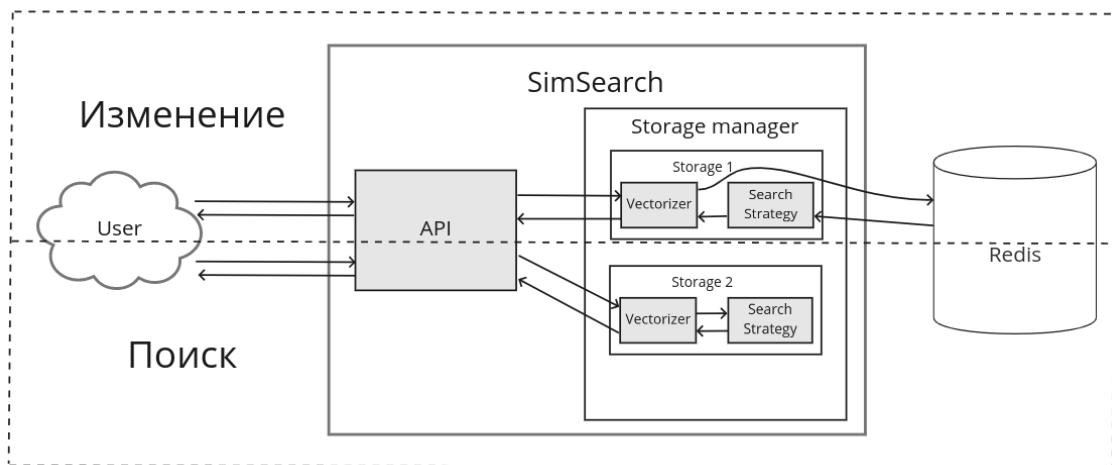


Рис. 4.2: Архитектура SimSearch, взаимодействие компонент при разных запросах.

4.3 Масштабирование

Для того, чтобы приложение могло быть распределенным, нужно, чтобы данные хранились не в какой-то инсталляции хранилища, а в общей базе, к которой будут иметь доступ все серверы [41], на которых запущен SimSearch. Поисковый индекс хранилища, для разграничения логики, создается на каждом сервере [41] независимо. Таким образом, нужно не только использовать базу данных, но и при обновлении объекта в базе обновлять поисковый индекс во всех инсталляциях. Для реализации такой логики был выбран Redis [12], так как он предлагает и NoSQL базу данных в kv-формате, и поддерживает каналы для PUB-SUB взаимодействия [13]. Таким образом, в SimSearch Redis одновременно хранит базу данных в удобном формате и рассыпает всем запущенным серверам [41] SimSearch уведомление об изменении данных.

Когда какой-то сервер SimSearch получает запрос на обновление данных, узел делает обновление в таблице Redis, после чего с помощью очереди сообщений оповещает остальные сервера о изменении в данных (рис. 4.3).

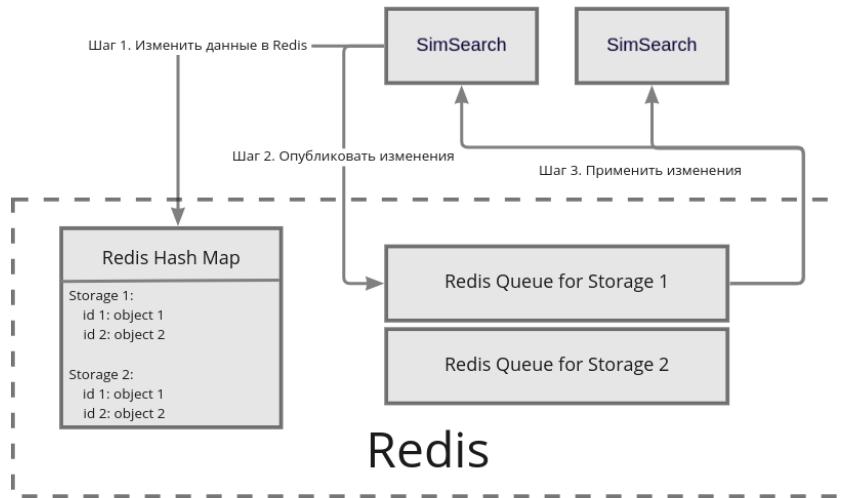


Рис. 4.3: Схема обновления данных между инсталляциями SimSearch с помощью Redis.

4.4 Объект поиска

Объект поиска для SimSearch — это структура, имеющая:

- Некоторый строковый идентификатор (например, в качестве идентификатора изображения может выступить его url).
- Векторное представление (предполагается, что вектор содержит либо bool, либо float).
- Поле, которое используется для хранения метаинформации объекта.

Метаинформация может как использоваться Vectorizer-ом для эмбединга, так и храниться для внешнего использования.

Внутри SimSearch объект поиска хранится в своем JSON-представлении (рис. 4.4).

```

1 - {
2   "id": "https://www.planetware.com/wpinages/2020/02/france-in-pictures-beautiful
-places-to-photograph-eiffel-tower.jpg",
3 -   "vector": [
4     | 0,
5     | 1,
6     | 1,
7     | 0,
8     | 1
9   ],
10  "data": "{\"city\": \"PARIS\", \"weather\": \"nice\"}"
11 }

```

Рис. 4.4: Представление изображения Эйфелевой башни [14] в виде json-объекта. Вектор сокращен для наглядности.

4.5 Storage Manager

Storage Manager решает две задачи:

1. Настроить хранилища на основе конфигурационного файла сервиса.
2. Передать пришедший запрос нужному хранилищу. Запрос может прийти как от API, так и от subscriber-a Redis-a, в случае обновления какого-то хранилища.

Таким образом, Storage Manager хранит хеш-таблицу, где хранилище хранится под своим именем. Для настройки хранилища менеджер достает из конфигурационного файла (рис. 4.5) стратегию поиска, стратегию векторизации, и задает их для хранилища.

```

1 - {
2   "storages": [
3     {
4       "name": "stories",
5       "searchStrategy": "faiss",
6       "vectorizerScript": "python3 scripts/downloader.py"
7     },
8     {
9       "name": "test",
10      "searchStrategy": "linear",
11      "vectorizerScript": "python3 scripts/downloader.py"
12    }
13  ]
14 }

```

Рис. 4.5: Конфигурация двух хранилищ: первое используется для поиска картинок, второе можно использовать для тестирования выдачи, поскольку линейный поиск находит оптимальные ответы.

4.6 Storage

Storage реализует нужный интерфейс:

- Build: загрузить базу из Redis, построить поисковый индекс. Используется для инициализации.
- Add: добавить объект в базу, обновить поисковый индекс. Если у поисковой стратегии нет реализации Add, обновить базу с нуля с помощью Build.
- Remove: удалить объект из базы (если присутствует там), обновить поисковый индекс. Если у поисковой стратегии нет реализации Remove, обновить базу с нуля с помощью Build.

- Search: Сделать поиск объекта в базе, вернуть n ближайших по мнению стратегии векторов.

Storage осуществляет взаимодействие с Redis [12], стратегией поиска и векторизатором. Логика Storage не зависит от стратегий поиска и векторизатора — объект сначала векторизуется с помощью нужного Vectiorizer-а, а затем происходит либо поиск с помощью поискового ядра, либо обновление базы и запись в publisher Redis-а. Если Storage получает от менеджера обновление данных в Redis, то просит хранилище перестроиться соответствующим образом (рис. 4.6).

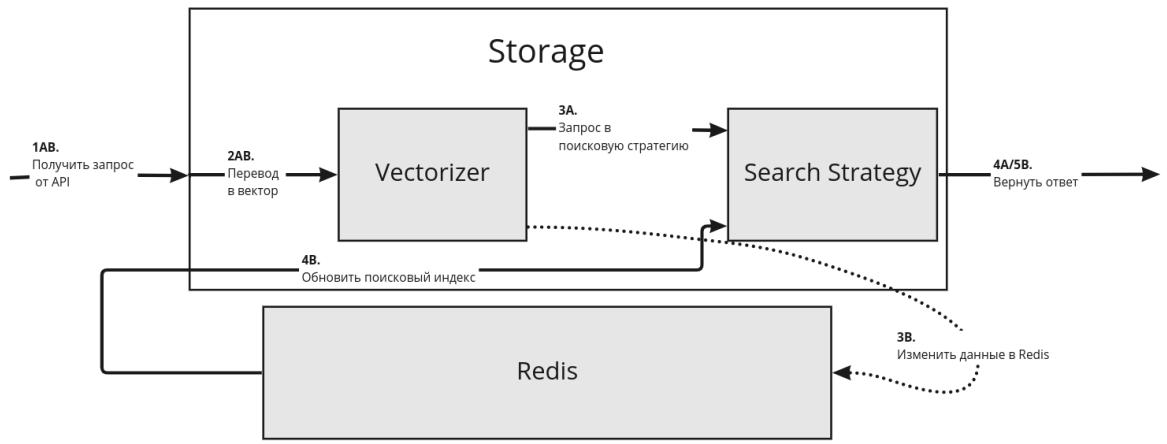


Рис. 4.6: Storage отвечает на запросы поиска (A) и обновления (B).

4.7 Vectorizer

Vectorizer переводит объекты из описания в вектор. Реализуется одна функцию Transform, которая принимает объект и подсчитывает его вектор.

На текущий момент реализовано два Vectorizer'а:

1. IdentityVectorizer: принимает объект, уже хранящий свой вектор в метаданных, возвращает вектор из объекта. Может использоваться, если пользователь уже сделал эмбеддинг без привязки к SimSearch.
2. ScriptVectorizer, который запускает отдельным процессом скрипт, с которым взаимодействие идет через перенаправленные I/O-потоки. В скрипт передается строковое представление метаинформации объекта, в ответ скрипт возвращает текстовое представление векторного представления объекта. Примером реализованного ScriptVectorizer'а является скрипт, который переводит изображение в вектор, получая на вход url.

4.8 Search

Поисковые стратегии — одно из самых гибких мест в SimSearch. Разработчики сервиса могут добавить множество различных стратегий. Интерфейс стратегии покрывает следующие запросы:

- Запрос Search получает объект-запрос и параметр n и возвращает n объектов, которые по мнению стратегии являются самыми релевантными.
- Запрос Add получает объект, который добавляется в поисковый индекс.
- Запрос Remove удаляет объект, который был в поисковом индексе.

На текущий момент реализована стратегия линейного поиска, которая находит оптимальный ответ, стратегия поиска на основе lshkit с расстоянием Хемминга в качестве метрики, и стратегия на основе Faiss-HNSW. Стратегия на основе lshkit не поддерживает динамическое обновление хранилища, поэтому после каждого запроса изменения перестраивает его заново, что не подходит для динамической нагрузки, но подходит для нерегулярных изменений.

В качестве примера, линейный поиск в рамках SimSearch реализован следующим образом:

- В качестве поискового индекса хранится динамический массив объектов.
- При запросах обновления индекс добавляет/удаляет требуемый элемент.
- При запросе поиска в индексе перебираются все объекты, с помощью кучи [37] размера k выбирается k самых близких к искомому.

4.9 Тестирование SimSearch

Для SimSearch была настроена контейнеризация через Docker-образ [15], а также docker-compose [16] для сборки проекта в связке с Redis. Благодаря этому получилось создать изолированный экземпляр сервиса, который можно протестировать через публичное API.

4.9.1 Тестирование корректности

Было настроено автоматическое интеграционное тестирование проекта. Интеграционные тесты проверяют, что API сервиса начинает обрабатывать запросы, что запросы на добавление в пустую базу обрабатываются, и запросы на получения ближайших векторов получают непустой ответ. Интеграционные тесты реализованы на python с помощью pytest [17].

Для проверки функционала написаны следующие тесты:

1. Создание непустой базы, проверка, что для объекта x находится поисковая выдача.
2. Добавление объекта x в базу, проверка, что он вернется в поисковой выдаче, как самый близкий.

Помимо интеграционного тестирования с помощью pytest была реализована сборка проекта в GitLab CI/CD [18] и проверка тестов.

Таким образом, итоговая реализация SimSearch получилась готовой к использованию в реальных сервисах за счет сборки в GitLab CI/CD, сохранения docker-образа на удаленном GitLab сервере и выполнения на нем автоматического интеграционного тестирования.

4.9.2 Тестирование производительности

Для замеров производительности использовались тесты (табл. 4.1), замеряющие время ответа на запрос Add/Search. Замерялись все реализованные поисковые стратегии, в качестве векторизатора использовался IdentityVectorizer, который не требует никакой обработки векторов, за счет чего замерялось именно время работы поисковых стратегий.

В качестве расстояний использовалось расстояние Хэмминга для стратегий linear и lshkit, для faiss-hnsw использовалось Евклидово расстояние. Поскольку эмбеддинг создавал бинарные вектора, минимизации этих двух метрик эквивалентны и они были выбраны, как стратегии по умолчанию для lshkit и faiss-hnsw.

Таблица 4.1: Тестирование различных поисковых стратегий.

Стратегия	Размер хранилища	t(Add), мс	t(Search), мс
linear	1000	12	98
linear	5000	11	410
lshkit	1000	89	14
lshkit	5000	280	25
faiss-hnsw	1000	12	43.5
faiss-hnsw	5000	14	44

Видно, что сбалансированнее всего показала себя стратегия faiss-hnsw, поскольку стратегия одновременно динамическая и предлагает поиск за логарифмическое время. Можно заметить, что полученные результаты говорят о способности сервиса обрабатывать несколько десятков запросов в секунду, что в первой версии сервиса для внутреннего пользования является более, чем достаточной пропускной способностью.

5 Поиск похожих картинок на основе SimSearch

На основе SimSearch был разработан поиск похожих картинок. В качестве используемых картинок для поиска использовались фоновые изображения из «историй» (рис. 5.1) в финансовом мобильном приложении.

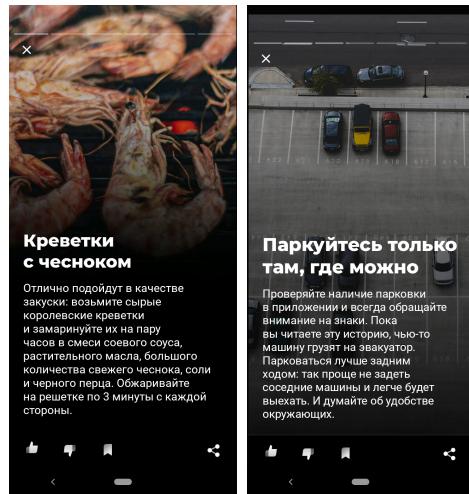


Рис. 5.1: Истории в мобильном финансовом приложении. Тематика произвольная, не привязана к финансам.

Для разработки подобного веб-приложения нужно определиться с несколькими вопросами:

1. Каким методом переводить объект в вектор?
2. Какое расстояние использовать для сравнения векторов?

Для перевода в вектор используется сверточная нейронная сеть, переводящая изображение в вектор. За основу взяты веса imagenet [19]. Таким образом, изображение переводится в вектор. За метрику взято расстояние Хэмминга. В качестве идентификатора изображений используется url.

Таким образом, веб-сервис для поиска изображений имеет очень простую логику: ему всего лишь нужно передавать запросы поиска в SimSearch, предварительно загрузив в него базу картинок. После этого нужно показать полученные результаты.

Для реализации веб-сервиса использовался flask [20]. Результаты получились впечатляющими: на относительно небольшой базе в 30000 изображений поиск находит достопримечательности, абстрактные объекты, кафе. Это хорошо решает задачу поиска картинок по референсам, потому что на базе таких размеров нужно искать не полное совпадение, достаточно что-то похожее по смыслу или внешнему виду.

SimSearch

Отправить

Query:



Results:

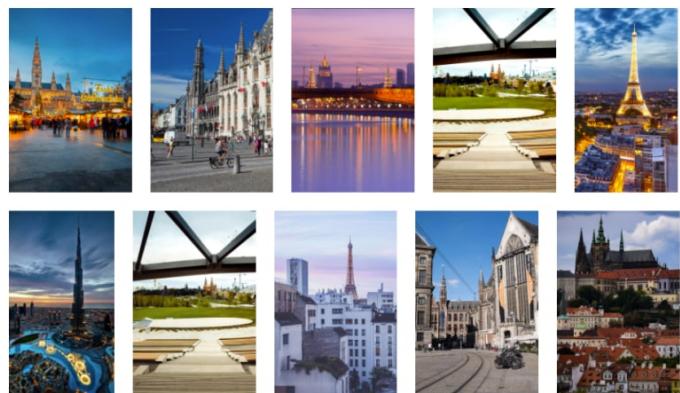


Рис. 5.2: Поиск картинок, похожих на изображение Эйфелевой башни. Приложение находит изображения городов и достопримечательностей.

SimSearch

Отправить

Query:



Results:

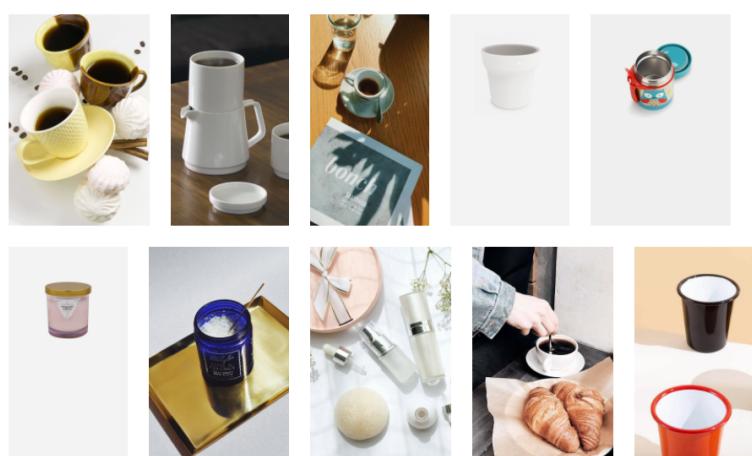


Рис. 5.3: Поиск картинок, связанных с кофе. Помимо кофе, можно найти разные кружки и даже круассаны!

SimSearch

Отправить

Query:



Results:

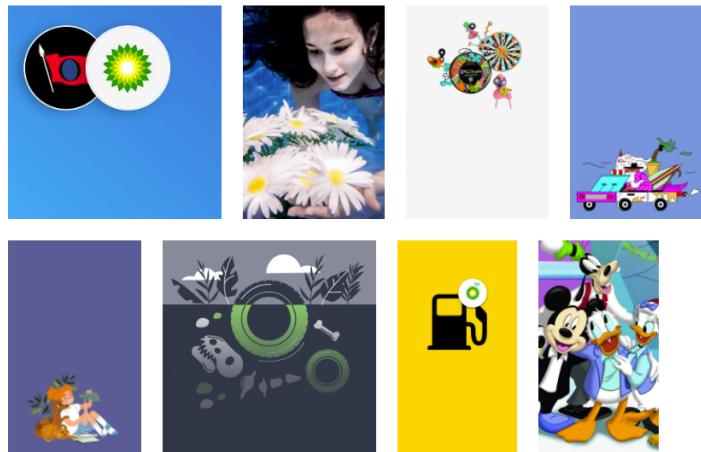


Рис. 5.4: Поскольку изображений с ромашками в базе почти нет, то находятся нерелевантные картинки и изображения с логотипом топливной компании, похожей на ромашку.

Приведем несколько примеров запросов. Для запросов достопримечательностей (рис. 5.2) и кофейной тематики (рис. 5.3) находятся схожие изображения. Есть и менее удачные запросы поиска, где в базе не было нужных изображений, за счет чего результаты достаточно случайны (хоть и получилось найти зависимости между ромашкой и логотипом топливной компании, как на рис. 5.4).

6 Заключение

Сформулируем, что было сделано в рамках курсовой работы, и какое дальнейшее развитие проекта возможно.

6.1 Результаты

В рамках курсовой работы был сделан сравнительный анализ алгоритмов для решения задачи $c\text{-}k\text{-ANN}$ (K-d деревья 3.1, LSH 3.2, квантизация 3.3, HNSW 3.4) и библиотеки (Annoy [22], Nmslib [23], Faiss [35], Scann [36], Lshkit [8]).

Был разработан сервис SimSearch, который можно использовать для поиска похожих объектов произвольного вида. SimSearch реализует HTTP REST API, выполняет векторизацию объекта и поиск ближайших векторов в базе, имеет возможность динамически менять базу. Для сервиса настроена контейнеризация и автоматическое интеграционное тестирование.

На основе сервиса SimSearch было создано приложение для поиска похожих изображений с UI интерфейсом, которое показывает релевантную поисковую выдачу.

6.2 Дальнешее развитие

Дальнешее развитие работы заключается в исследовании новых поисковых алгоритмов и векторизаторов, а также в улучшении функционала сервиса SimSearch.

Список литературы (или источников)

1. <https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526>. [Embedding].
2. <https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/>. [One Hot Encoding].
3. <https://www.shazam.com/ru/home>. [Shazam].
4. https://github.com/rom1504/image_embeddings. [ImageEmbeddings on github].
5. <https://github.com/GoogleCloudPlatform/realtime-embeddings-matching>. [Google Cloud Platform: Realtime Embedding].
6. <https://www.pinecone.io/learn/hnsw/>. [James Briggs, Vector Search with HNSW].
7. <https://lou.dev/talks/2018-08-09-lopq>. [Lou Kratz, Scaling Visual Search with Locally Optimized Product Quantization, Papers We Love, Philadelphia, 2018].
8. <http://lshkit.sourceforge.net/>. [LSHKit].
9. <https://www.elastic.co/elasticsearch/>. [Elasticsearch].

10. <http://ann-benchmarks.com/>. [ANN benchmarks, секция «glove-100-angular, k = 10»].
11. <https://otaviofff.github.io/restful-grounding>. [REST HTTP API].
12. <https://redis.io>. [Redis].
13. <https://redis.io/docs/manual/pubsub/>. [Redis channels documentation, Pub/Sub model].
14. <https://www.planetware.com/wpimages/2020/02/france-in-pictures-beautiful-places-to-visit-in-france-1000x667.jpg>. [Фотография Эйфелевой башни].
15. <https://www.docker.com/>. [Docker].
16. <https://docs.docker.com/compose/>. [Docker-compose].
17. <https://docs.pytest.org/en/7.1.x/>. [Pytest documentation].
18. <https://docs.gitlab.com/ee/ci/>. [Gitlab CI/CD documentation].
19. <https://www.image-net.org/>. [Imagenet].
20. <https://flask.palletsprojects.com/en/2.1.x/>. [Flask documentation].
21. Oren Barkan and Noam Koenigstein. Item2vec: neural item embedding for collaborative filtering. In *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6. IEEE, 2016.
22. Erik Bernhardsson. *Annoy: Approximate Nearest Neighbors in C++/Python*, 2018. <https://github.com/spotify/annoy>.
23. Yury Malkov David Novak Bilegsaikhan Naidan, Leonid Boytsov. *Non-Metric Space Library (NMSLIB)*, 2014. <https://github.com/nmslib/nmslib>.
24. Leonid Boytsov and Bilegsaikhan Naidan. Engineering efficient and effective non-metric space library. In Nieves R. Brisaboa, Oscar Pedreira, and Pavel Zezula, editors, *Similarity Search and Applications - 6th International Conference, SISAP 2013, A Coruña, Spain, October 2-4, 2013, Proceedings*, volume 8199 of *Lecture Notes in Computer Science*, pages 280–293. Springer, 2013.
25. Pedro Cano, Eloi Batlle, Emilia Gómez, Leandro de Campos Teixeira Gomes, and Madeleine Bonnet. *Audio Fingerprinting: Concepts And Applications*, volume 2, pages 233–245. 09 2005.
26. Wei Dong, Zhe Wang, William Josephson, Moses Charikar, and Kai Li. Modeling lsh for performance tuning. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 669–678, 2008.
27. Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529, 1999.

28. Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, pages 3887–3896. PMLR, 2020.
29. Omid Jafari, Preeti Maurya, Parth Nagarkar, Khandker Mushfiqul Islam, and Chidambaram Crushev. A survey on locality sensitive hashing algorithms and their applications, 2021.
30. Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.
31. Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2019.
32. Yury A. Malkov and Dmitry A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *CoRR*, abs/1603.09320, 2016.
33. Anne HH Ngu, Jiangang Ma, Quan Z Sheng, Lina Yao, and Scott Julian. Servicexplorer: A similarity-based web service search engine. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 1251–1252, 2014.
34. Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.
35. Facebook Research. *Faiss*, 2019. <https://github.com/facebookresearch/faiss>.
36. Google Research. *ScANN (Scalable Nearest Neighbors)*, 2020. <https://github.com/google-research/google-research/tree/master/scann>.
37. Wikipedia contributors. Heap (data structure) — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Heap_\(data_structure\)&oldid=1088468512](https://en.wikipedia.org/w/index.php?title=Heap_(data_structure)&oldid=1088468512), 2022. [Online; accessed 5-June-2022].
38. Wikipedia contributors. K-d tree — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=K-d_tree&oldid=1089227535, 2022. [Online; accessed 4-June-2022].
39. Wikipedia contributors. Reference — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=Reference&oldid=1083842428>, 2022. [Online; accessed 4-June-2022].
40. Wikipedia contributors. Scalability — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=Scalability&oldid=1086627524>, 2022. [Online; accessed 5-June-2022].

41. Wikipedia contributors. Server (computing) — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Server_\(computing\)&oldid=1086236982](https://en.wikipedia.org/w/index.php?title=Server_(computing)&oldid=1086236982), 2022. [Online; accessed 5-June-2022].
42. Shaoyi Yin, Mehdi Badr, and Dan Vodislav. Dynamic multi-probe lsh: An i/o efficient index structure for approximate nearest neighbor search. In *International Conference on Database and Expert Systems Applications*, pages 48–62. Springer, 2013.