# Технический отчет команды «Три сигмы»

**Состав команды:** Елизарьев Ярослав Владимирович (ЭФ НГУ), Метлин Александр Дмитриевич (ФИТ НГУ), Кузнецов Никита Сергеевич (ЭФ НГУ)

## Введение

Перед нами стояла довольно распространенная в наше время задача в области искусственного интеллекта – поиск ближайших соседей. Проблема заключается в том, что точный поиск ближайших соседей оказывается очень затратным с точки зрения памяти и времени, и необходимо ускорить этот процесс для быстроты работы основного алгоритма (например LLM). Очевидно, что для увеличения скорости придется чем-то пожертвовать, в нашем случае это была точность поиска. Мы должны были разработать алгоритм приближенного поиска ближайших соседей (ANN).

## Существующие ANN-search алгоритмы

Очевидно, мы далеко не первые, кто решает подобную задачу. Существует множество ANN-search алгоритмов. Ниже представлена схема, опубликованная в статье **[1]**, в которой авторы сделали некую классификацию существующих методов ANN search.

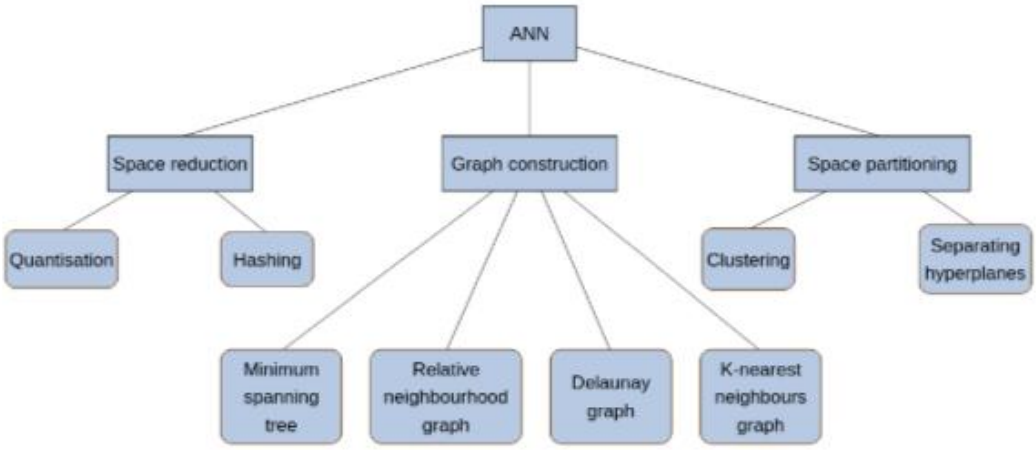


*Рисунок 1. Таксономия ANN методов. Источник: [1]*

В этой же статье авторы приводят примеры наиболее распространенных алгоритмов ANN search (IVFFlat, IVFFlatPQ, IVFADC, LSQ, ScaNN и др.). Также важно заметить, что авторы отмечают то, что использование ML алгоритмов не дает какого-либо значимого выигрыша и оптимизации в решении задачи поиска ANN, а иногда наоборот только увеличивают затраты времени и других ресурсов по причине того, что эти алгоритмы надо обучать.

## Какой алгоритм мы выбрали и почему

Наш выбор пал на алгоритм IVFFlat и его модификацию - IVFFlatPQ. Второй алгоритм был реализован в упрощенном виде. Эти алгоритмы были выбраны в силу их относительной популярности (статья [1] приводит данные, согласно которым IVFFlat – метод ANN, наиболее часто упоминаемый в научной литературе), простоты и относительной эффективности. Ниже представлен график, на котором сравнивается перфоманс различных ANN алгоритмов на примере датасета Profiset-100k.

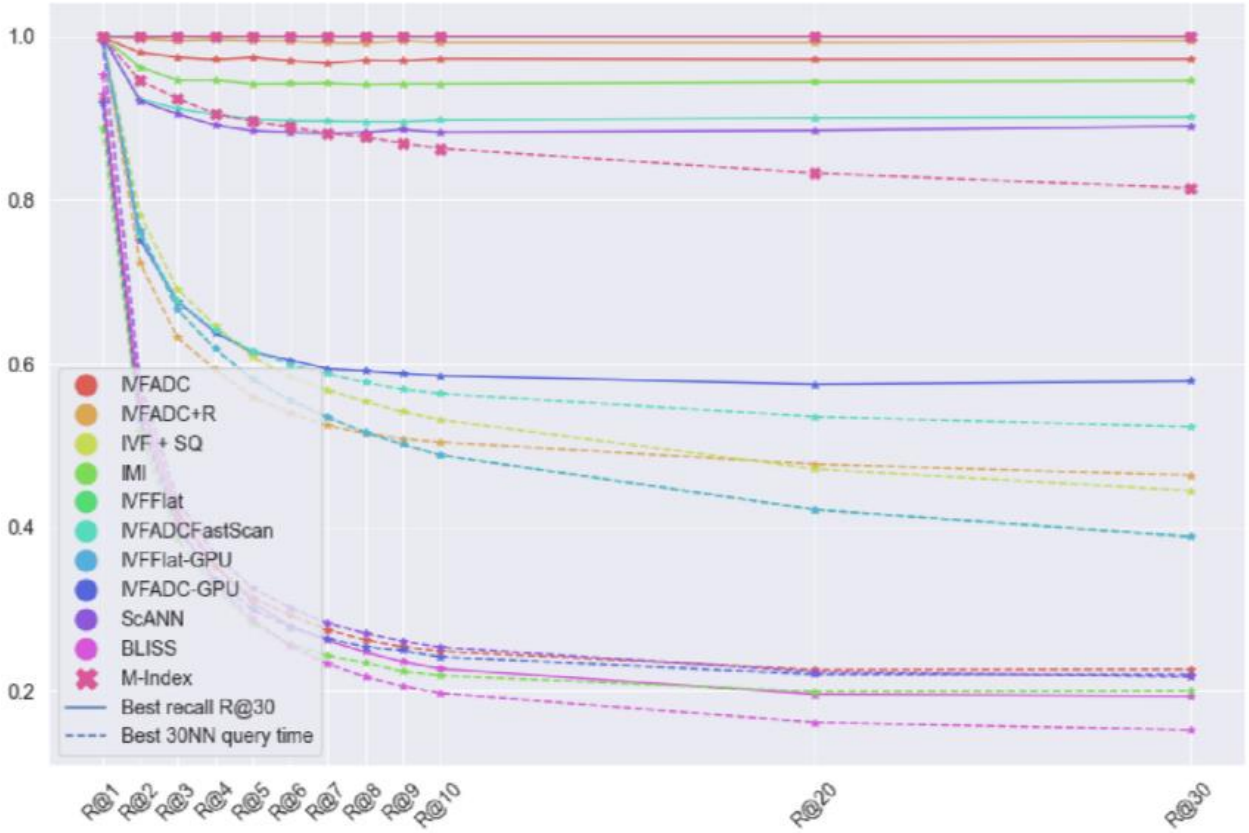


*Рисунок 2. Recall различных алгоритмов поиска ANN на примере датасета Profiset-100k. Источник: [1]*

Можем видеть, что IVFFlat имеет очень хорошие метрики на этом датасете по сравнению с другими алгоритмами.

## Как реализовали алгоритм

Выбрав алгоритм, мы приступили к его разбору и составили план его создания. План был таков:

1. Реализовать свой алгоритм KMeans;
2. Обучить модель KMeans на датасетах;
3. Реализовать сохранение размеченных датасетов на диск в подходящем формате;

4. Реализовать сохранение на диск обученной модели KMeans.
5. Реализовать парсинг вектора, для которого нужно найти соседей.
6. Реализовать поиск 10 предположительных ближайших соседей.

Экспериментируя с количеством кластеров в KMeans, мы обнаружили, что чем больше кластеров ставим, тем больше оперативной памяти расходуется при обучении модели. Мы на основе метода локтя решили для датасета SIFT поставить количество кластеров равным 4, а для GIST – 5.

Датасет SIFT оказался довольно легким, и мы легко выгрузили его в оперативную память, обучили модель KMeans для 100 тысяч векторов и разметили оставшиеся обученной моделью. Добавили для каждого вектора метку того, к какому кластеру он относится. Потом для нахождения 10 предположительных ближайших соседей наш алгоритм был спроектирован следующим образом:

1. Парсится вектор, для которого нужно искать соседей;
2. Для вектора предсказывается кластер, к которому он относится;
3. Выгружается размеченный датасет base;
4. Датасет просматривается построчно, и если вектор из датасета base входит в тот же кластер, что и поданный вектор, то вычисляется евклидово расстояние между ними и заносится в список расстояний вместе с индексом вектора из base;
5. Список расстояний сортируется и выводятся 10 индексов ближайший векторов среди найденных.

С датасетом GIST все оказалось гораздо сложнее, так как он очень тяжелый и сразу выгрузить его целиком не получилось. Мы решили читать его по батчам и применить технику плоской квантизации (Flat Quantization) для векторов этого датасета с целью уменьшения нагрузки на оперативную память. Flat Quantization используется в алгоритме IVFFlatPQ. Кодировались векторы в формат numpy.uint8. На первых 100 тысячах векторах мы также обучили KMeans, закодировали размеченные вектора, записали в файл. Потом оставшиеся 900 тысяч векторов считали по батчам, разметили, закодировали и записали в файл. Алгоритм для поиска 10 ANN был следующий в данном случае:

1. Парсится вектор, для которого нужно искать соседей;
2. Предсказывается кластер, к которому относится данный вектор
3. Вектор кодируется в формат uint8;
4. Датасет base выгружается по батчам и кажду батч читается построчно. Если вектора из одного кластера, то они оба декодировались обратно в float и между ними ищется евклидово расстояние, а в список расстояний записывается индекс вектора из base и расстояние.
5. Список расстояний сортируется и выводятся 10 индексов ближайший векторов среди найденных.

## Результаты и выводы работы

Нам удалось добиться поставленной перед нами цели – реализовать алгоритм поиска ANN. Метрики на одном из тестов оказались следующие:

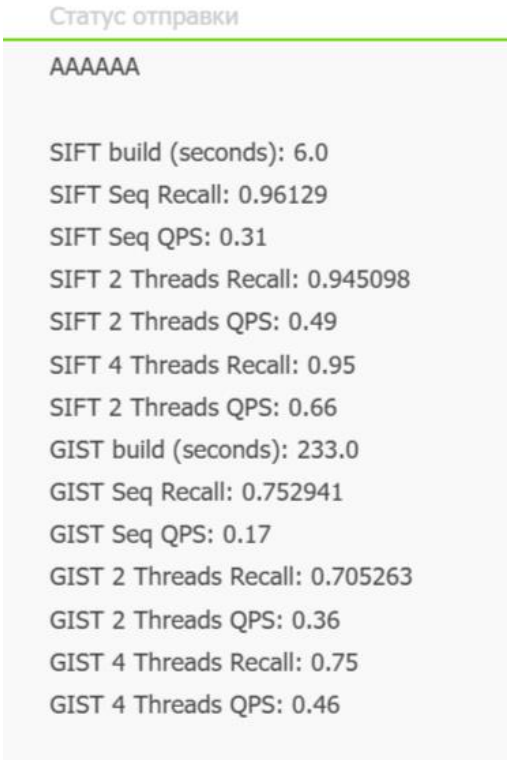


*Рисунок 3. Результаты работа алгоритма на одном из тестов. Источник: автор*

Для алгоритма приближенного поиска, на наш взгляд, алгоритм показывает хорошую метрику recall, однако скорость его работы невысока, и это есть недостаток нашего решения. Возможно, стоило как-то еще больше снизить нагрузку на оперативную память, попробовать другие способы уменьшения объема памяти, занимаемой векторами. Либо, как вариант, реализовать решение на языке C++.

## Список использованной литературы

1. HANKO B. C. J. Indexing Data Using Machine Learning.

Приложение

| Name | KNN | Dist. f. | Citations | Year | Impl. |
|---|---|---|---|---|---|
| IVFADC [15] | ✓ | Euclidean | 2558 | 2010 | ✓ [32] |
| IVFADC+R [33] | ✓ | Euclidean | 272 | 2011 | ✓ [32] |
| CKM [34] | | Euclidean | 340 | 2013 | ✓ [34] |
| OPQ [22] | | Euclidean | 285 | 2013 | ✓ [35] |
| AQ [36] | | Euclidean | 222 | 2014 | ✓ [32] |
| LSQ [37] | | Euclidean | 52 | 2016 | ✓ [35] |
| RVQ [38] | | Euclidean | 39 | 2010 | ✓ [32] |
| RVPQ [39] | | Euclidean | 0 | 2022 | ✓ [32] |
| MSQ [40] | ✓ | Euclidean | 65 | 2017 | |
| KSSQ [41] | | Euclidean | 20 | 2016 | |
| OTQ [42] | | Euclidean | 5 | 2015 | |
| ScaNN [43] | ✓ | Euclidean/Cosine | 59 | 2022 | ✓ [43] |
| IMI [23] | ✓ | Euclidean | 399 | 2014 | ✓ [23] |
| GNOIMI [44] | ✓ | Euclidean | 67 | 2016 | ✓ [44] |
| NeuralLSH [45] | ✓ | General | 63 | 2019 | ✓ [45] |
| Neural Catalyser [46] | | General | 62 | 2018 | ✓ [46] |
| Learned k-d tree [47] | ✓ | General | 30 | 2007 | |
| Boosted Search Forest [48] | | Euclidean | 25 | 2011 | |
| Reinforcement learning graph [49] | ✓ | General | 9 | 2019 | ✓ [49] |
| AAQ [50] | ✓ | Cosine | 0 | 2022 | |
| IVFADC + G + P [51] | ✓ | Euclidean | 51 | 2018 | ✓ [51] |
| OPH [52] | ✓ | General | 19 | 2013 | |

Continued on the next page

| Name | KNN | Dist. f. | Citations | Year | Impl. |
|---|---|---|---|---|---|
| DCPQ [53] | | Euclidean | 7 | 2020 | |
| IRLI [54] | ✓ | General | 0 | 2021 | ✓[54] |
| AUCH [55] | | General | 3 | 2021 | |
| HPQ [56] | ✓ | General | 4 | 2019 | |
| Unsupervised space partitioning [57] | ✓ | General | 0 | 2022 | ✓[57] |
| BLISS [58] | ✓ | General | 0 | 2022 | ✓[2] |
| CH [59] | ✓ | Hamming | 81 | 2011 | |
| LOPQ [60] | ✓ | Euclidean | 280 | 2014 | ✓[3] |
| PTQ [61] | ✓ | Euclidean | 9 | 2015 | |
| GNIDL [62] | ✓ | Euclidean | 9 | 2017 | |
| QPQ [63] | ✓ | Euclidean | 8 | 2019 | |
| CQ [64] | | Euclidean | 212 | 2014 | ✓[4] |
| SCQ [65] | | Euclidean | 0 | 2015 | |
| ADSH [66] | ✓ | Hamming | 89 | 2018 | ✓[5] |
| CompQ [67] | ✓ | Euclidean | 24 | 2016 | ✓[35] |

*Рисунок 4. Данные по некоторым алгоритмам ANN search. Источник: [1]*