Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА СИСТЕМЫ ОБРАБОТКИ ИНФОРМАЦИИ И УПРАВЛЕНИЯ

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

НА ТЕМУ:

Анализ методов поиска ближайшего соседа с использованием графов

Студент ИУ5-32М _____ **А.А. Павловская**
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель _____ **Ю.Е. Гапанюк**
(Подпись, дата) (И.О.Фамилия)

2023 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ
Заведующий кафедрой ___ИУ5____
(Индекс)
_____ ___В.И. Терехов__
(И.О.Фамилия)
«_04_» _____сентября_____ 2023 г.

З А Д А Н И Е
на выполнение научно-исследовательской работы

по теме ___Анализ методов поиска ближайшего соседа с использованием графов _____
_____

Студент группы _ИУ5-32М__

_____Павловская Анастасия Андреевна___
(Фамилия, имя, отчество)

Направленность НИР (учебная, исследовательская, практическая, производственная, др.)
_____ИССЛЕДОВАТЕЛЬСКАЯ_____
Источник тематики (кафедра, предприятие, НИР) _____КАФЕДРА_____

График выполнения НИР: 25% к __._ нед., 50% к _._ нед., 75% к _._ нед., 100% к __._ нед.

Техническое задание _проанализировать и сравнить методы поиска ближайшего соседа с_____
использованием графов HNSW и NSG, рассмотреть различные иерархические структуры, способы
_индексации, сценарии_____
_____
_____
_____

Оформление научно-исследовательской работы:

Расчетно-пояснительная записка на _39_ листах формата А4.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)
_____
_____
_____
_____

Дата выдачи задания «_04_» ____сентября_____ 2023 г.

Руководитель НИР _____ _____Ю.Е. Гапанюк__
(Подпись, дата) (И.О.Фамилия)

Студент _____ **_____А.А. Павловская_**
(Подпись, дата) (И.О.Фамилия)

Примечание: Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

Оглавление

Введение

В настоящее время объем информации, нуждающейся в обработке, постоянно растет. Это закономерно привело к необходимости использования эффективных алгоритмов поиска, снижающих затраты на просмотр всего объема данных.

Одним из известных методов является поиск ближайшего соседа (K-NNS), суть которого заключается в поиске в некотором метрическом множестве K элементов, расположенных ближе всего к заданному. Мера близости является функцией, задаваемой предметной областью: чем меньше значения близости, тем более схожими можно считать сравниваемые элементы и наоборот. Проблема поиска ближайшего соседа встречается в множестве областей, например, в задачах распознавания образов, классификации объектов, рекомендательных системах, задачах сжатия данных, размещении рекламы в сети Интернет, семантического поиска и др.

Однако, сложность классического подхода к алгоритму K-NNS растет линейно в зависимости от количества хранимых элементов и размерности этих элементов, что для крупномасштабных данных зачастую приводит к слишком долгой обработке или большим объемам требуемой памяти, к значительным потерям в точности. Поэтому стала актуальной проблема разработки наиболее эффективных и быстрых алгоритмов поиска.

Методы поиска ближайшего соседа с использованием графов значительно снижают сложность вычислений и ускоряют процесс получения результата. К таким методам относятся, например NSG и HNSW. В данной работе будет проведен анализ и сравнение этих двух методов с целью выявления особенностей, преимуществ, недостатков и возможных вариантов применения.

1. Методы ANNS, основанные на графах

Алгоритмы поиска ближайших соседей можно разделить на две основные группы:

- Точные методы (Nearest neighbor search, NNS)
- Приближенные методы, или методы аппроксимации (Approximate nearest neighbor search, ANNS)

К точным методам относится, например, классический алгоритм поиска ближайших соседей. Такие методы всегда возвращают истинных ближайших соседей, но работают очень медленно и затратно в случае высоких размерностей данных.

Отличие приближенных методов в первую очередь в том, что они допускают небольшую долю ошибок в результате. Методы аппроксимации включают в себя методы, основанные на деревьях, хешировании, квантовании, а также методы, основанные на графах.

Алгоритмы ANNS, основанные на графах, используют индекс в виде графа. Граф в таком случае состоит из вершин, каждая из которых соответствует элементу набора данных, и ребер, которые отображают отношение соседства между вершинами. Для поиска в таких структурах обычно используются вариации жадного алгоритма, который в своем самом простом представлении заключается в выборе самой ближайшей вершины к текущей для продолжения поиска уже из нее. В последние десятилетия было разработано множество графов для эффективного приближенного поиска ближайших соседей, которые сейчас называют графами близости (proximity graphs).

Некоторые графы близости, такие как графы Делоне (или триангуляция Делоне) и MSNET (Monotonic Search Networks), гарантируют, что от любой вершины p до любой вершины q существует монотонный путь. Также можно выделить RNG (Randomized Neighborhood Graph), который гарантирует полулогарифмическую временную сложность поиска. В графах NSWN (Navigable Small-World Networks) средняя длина пути поиска также растет

полулогарифмически с увеличением объема данных, однако временная сложность построения таких графов очень высокая (по крайней мере O(n2)).

На аппроксимации графа Делоне основаны, например, GNNS, IEH, Efanna. Метод NSW аппроксимирует граф NSWN, FANNG аппроксимирует RNG. Алгоритм HNSW использует структуру, которая близка и к NSWN, и к RNG и к графам Делоне. Метод NSG предлагает собственный граф MRNG (Monotonic Relative Neighborhood Graph), приближением которого и является NSG.

2. NSG – Navigating Spreading-out Graph

NSG [1] — это основанный на графе алгоритм приближенного поиска ближайших соседей (ANNS). Он основан на аппроксимации графовой структуры Monotonic Relative Neighborhood Graph (MRNG). NSG устанавливает центральное положение в качестве навигационной вершины, а затем использует определенную стратегию выбора краев для управления степенью отклонения каждой точки. Таким образом, это может уменьшить использование памяти и быстро определить местоположение цели поблизости во время поиска векторов.

2.1. Monotonic Relative Neighborhood Graph (MRNG)

Введем сначала необходимые обозначения и термины. Пусть $lune_{pq}$ определяется как следующая область:

$$lune_{pq} = B(p, \sigma(p,q)) \cap B(q, \sigma(p,q))$$

где σ(p,q) – расстояние между p и q;

$B(p,r) = \{x \mid \sigma(x,p) < r\}$ – множество вершин x, расстояние от которых до p меньше r.

То есть, $lune_{pq}$ это все точки, расстояния от которых до p и q меньше, чем расстояние от p до q.

Приведем определение MRNG. Пусть есть конечное множество n точек S в пространстве E^d, тогда NRNG это ориентированный граф с множеством ребер, которые удовлетворяют следующему свойству: для любого ребра $\overrightarrow{pq}$, $\overrightarrow{pq} \in$ MRNG тогда и только тогда, если $lune_{pq} \cap S = \emptyset$ или для каждой вершины $r \in (lune_{pq} \cap S)$, $\overrightarrow{pr} \notin$ MRNG.

Для RNG стратегия выбора ребер иная. Пусть есть конечное множество n точек S в пространстве E^d, тогда для каждых двух точек p, q ∈ S ребро pq ∈ RNG тогда и только тогда, если $lune_{pq} \cap S = \emptyset$.

На рис.1 представлено сравнение между стратегией выбора ребер в RNG (а) и MRNG(б). RNG это неориентированный граф, в то время как MRNG – ориентированный. На рисунке (а), p и r соединены потому, что нет ни одной

вершины в $lune_{pr}$. Так как $r \in lune_{ps}$, $s \in lune_{pt}$, и $u \in lune_{pq}$– нет ребер между p и s, t, u, q. На рисунке (б), p и r соединены потому, что в $lune_{pr}$ нет вершин. Вершины p и s не соединены потому, что есть $r \in lune_{ps}$, и pr, sr ∈ MRNG. Ориентированное ребро $\overrightarrow{pt}$ ∈ MRNG потому, что $\overrightarrow{ps} \notin$ MRNG. Однако, $\overrightarrow{tp} \notin$ MRNG, потому что $\overrightarrow{ts}$ ∈ MRNG. Видно, что MRNG определен рекурсивно, и что стратегия выбора ребер в RNG более строгая, чем в MRNG. В RNG (a) есть монотонный путь от q к p, но нет монотонного пути от p к q. В MRNG(б) существует по крайней мере один монотонный путь от одной вершины к другой.

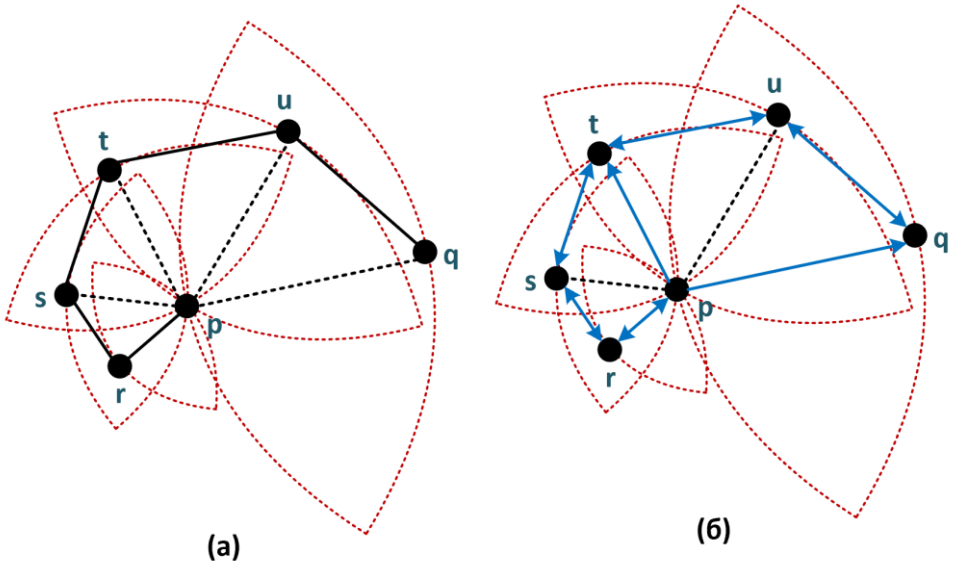

Рис.1. Сравнение между стратегией выбора ребер в RNG (a) и MRNG (b)

MRNG может быть построен, если просто применять стратегию выбора ребра к каждой вершине. Подробнее:

1) Для каждой вершины p, определить множество оставшихся вершин в S как R = S/{p}.

2) Вычислить расстояние между каждой вершиной в R и p, затем отсортировать R по расстоянию до p в возрастающем порядке.

3) L – множество выбранных вершин. Добавить ближайшую к p вершину из R в L.

4) Выбрать вершину q из R и вершину r из L, чтобы проверить, является ли pq самым длинным ребром в треугольнике pqr.

5) Если pq не самое длинное ребро в треугольнике pqr, тогда для любого r ∈ L добавить q в L.

6) Повторять процесс до тех пор, пока не будут проверены все вершины в R.

2.2. Алгоритм NSG

Хотя MRNG может гарантировать малое время поиска, его высокое время индексирования непрактично для проблем большого масштаба. Поэтому был предложен практический подход построения приближенного MRNG, названный Navigating Spreading-out Graph (NSG). Алгоритм построения NSG (Алгоритм 2.2) состоит из следующих шагов:

1. Построить kNN граф одним из популярных в данное время методов
2. Найти приближенный медоид датасета:
   2.1. Рассчитать центроид датасета
   2.3. Принять его за запрос, выполнить поиск по kNN-графу и принять возвращенного ближайшего соседа как приближенный медоид. Эта вершина называется навигационной вершиной, потому что все поиски будут начинаться с этой конкретной вершины
3. Для каждой вершины сгенерировать множество кандидатов в соседи и выбрать соседей из множеств кандидатов. Это можно сделать следующими шагами:
   3.1. Для каждой вершины p принять ее за запрос и выполнить поиск по графу, начиная с навигационной вершины на заранее построенном графе kNN.
   3.2. Во время поиска каждую посещенную вершину q (то есть такую, расстояние от которой до p рассчитывалось) добавить в множество кандидатов (расстояние также сохраняется)
   3.3. Выбрать m ближайших соседей для p из множества кандидатов с помощью стратегии выбора ребер MRNG

4. Выполнить поиск в глубину (Depth-First-Search) на графе, полученном на предыдущих шагах, при этом принять навигационную вершину как корень дерева. Когда DFS завершится проверить, есть ли вершины, которые не были затронуты поиском. Если такие вершины есть, соединить их с их приближенными ближайшими соседями, которые были затронуты поиском, и продолжить DFS.

Стратегия выбора ребер в MRNG принимает все остальные вершины в S как кандидатов в ближайшие соседи текущей вершины, что приводит к большой сложности вычислений по времени. Для ускорения этого процесса в NSG генерируется небольшое множество кандидатов для каждой вершины.

Так как процесс построения точного NNG занимает много времени, используется приближенный kNN граф. В нем допустимо, чтобы только несколько вершин не были соединены со своими ближайшими соседями.

Так как поиск в NSG всегда начинается с навигационной вершины n, для данной вершины p нужно рассматривать только те вершины, которые находятся на пути поиска от pn к p. Поэтому p принимается как запрос и выполняется поиск на заранее построенном kNN графе. Вершины, затронутые поиском, и ближайшие соседи вершины p сохраняются как кандидаты (рис.2). Вершины, формирующие монотонный путь от pn до p имеют большой шанс быть добавленными в кандидаты. Затем производится стратегия выбора ребер MRNG на этих кандидатах, и есть большая вероятность, что NSG наследует монотонный путь MRNG от pn до p.

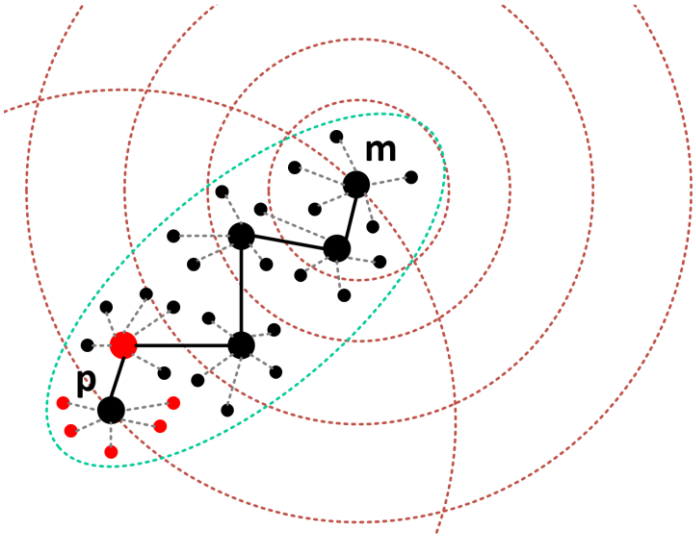

Рис.2. Кандидаты для выбора ребра в NSG

Параметр m вводится для того, чтобы решить проблему с вершинами, имеющими очень много исходящих из них ребер. Количество исходящих ребер для всех вершин ограничено параметром m << n с помощью удаления длиннейших ребер. Как следствие, после такого вмешательства связность графа больше не гарантируется.

Поиск в NSG проводится с использованием алгоритма 2.1, и он всегда начинается с навигационной вершины.

Алгоритм 2.1. Поиск по графу, Search-on-Graph (G,p,q,l,k)

Входные данные: граф G, стартовая вершина p, запрос q, размер пула кандидатов l.
Выходные данные: k ближайших соседей q.
$i \leftarrow 0$
$S \leftarrow \emptyset$ // пул кандидатов
$S \leftarrow S \cup p$
пока $i < l$ **выполнить**
    $i \leftarrow$ индекс первой непроверенной вершины в S
    пометить p_i как проверенную
    для каждой n в $neighborhood(p_i)$ в G **выполнить**
        $S \leftarrow S \cup n$
    конец для
    отсортировать S по возрастанию расстояния к q
    если $|S| > l$ **тогда**
        установить $|S| = l$ //изменение размера пула кандидатов
    конец если

конец пока
вернуть k первых вершин в S

Алгоритм 2.2. Построение NSG, NSGBuild (G,l,m)

Входные данные: k-NN граф G, размер пула кандидатов l для жадного поиска, максимальная полустепень исхода m.
Выходные данные: NSG с навигационной вершиной n.
$c \leftarrow$ вычислить центроид датасета
$r \leftarrow$ случайная вершина
$n \leftarrow$ Search-on-Graph(G,r,c,l,k=1) //навигационная вершина
для каждой вершины $v \in G$ **выполнить**
    Search-on-Graph(G,n,v,l)
    $E \leftarrow$ все вершины, проверенные во время поиска
    $E \leftarrow E \cup neighborhood(v)$
    отсортировать E в порядке возрастания расстояния до v
    $R \leftarrow \emptyset$ // множество результата
    $p_0 \leftarrow$ ближайшая к v вершина в E
    $R \leftarrow R \cup p_0$
    пока $E \neq \emptyset$ и $|R| < m$ **выполнить**
        $p \leftarrow$ первый элемент в E
    убрать первый элемент из E
        для каждой вершины r в R **выполнить**
            если ребро pv конфликтует с ребром pr **тогда**
                break
            конец если
        конец для
        если никаких конфликтов не возникает **тогда**
            $R \leftarrow R \cup p$
        конец если
    конец пока
конец для
пока True **выполнить**
    выполнить поиск в глубину (DFS) в NSG из корня n
    если не все вершины были посещены поиском **тогда**
        добавить ребро между не посещенными вершинами и их ближайшим соседом из списка посещенных (с помощью алгоритма 1.1)
    иначе
        break
    конец если
конец пока

2.3. Depth-first search (DFS)

Чтобы решить проблему связности, в NSG предлагается решение, основанное DFS [2]. После процесса поиска в глубину всем вершинам гарантирован хотя бы один путь из навигационной вершины. Алгоритм 2.3 описывает основные шаги данного поиска.

Алгоритм 2.3. Поиск в глубину

Входные данные: граф G, стартовая вершина ep.
Выходные данные: список посещенных вершин V
$V \leftarrow \emptyset$ // список посещенных вершин
$S \leftarrow ep$ // стек кандидатов для рассмотрения
пока $S \neq \emptyset$ **выполнить**
    $node \leftarrow$ последняя вершина из S // текущая вершина
    $S \leftarrow S/node$
если $node \notin V$ **тогда:**
        $V \leftarrow V \cup node$
        $S \leftarrow neighborhood(node)$
    конец если
конец пока

2.4. NN-Descent

В алгоритме NSG указано, что стадию инициализации можно сделать любым из популярных методов построения аппроксимации kNN графа. Одним из таких методов является NN-Descent [3], и он будет применяться в данной работе в начальной стадии NSG.

NN-desent это алгоритм для построения приближенного KNN-графа, основанный на следующем принципе: сосед соседа скорее всего тоже будет соседом. Другими словами, если у нас есть аппроксимация K ближайших соседей для каждой точки, мы можем улучшить это приближение с помощью исследования соседей каждого соседа точки, которые определены текущим приближением.

Пусть V – датасет размером : $N = |V|$, и пусть $\sigma: V \times V \rightarrow R$ это функция расстояния. Для каждой точки v ∈ V, пусть $B_K(v)$ – это ближайшие соседи v, то есть K объектов в V (кроме v), которые ближе всего к v. Пусть $R_K(v) = \{u \in V | v \in B_K(u)\}$ – это обратные K ближайших соседей (то есть те точки, для

которых v – ближайший сосед. В алгоритме B[v] и R[v] используются для хранения аппроксимации $B_K(v)$ и $R_K(v)$ вместе со значениями близости. Пусть $\bar{B}(v) = B[v] \cup R[v]$ – это общие соседи v.

Базовый NN-descent алгоритм показан в Алгоритме 2.4. В начале алгоритма выбирается случайное приближение KNN для каждой вершины, это приближение итеративно улучшается с помощью сравнения каждой вершины с соседями ее соседей, включая прямых и обратных, и алгоритм заканчивается тогда, когда результат перестает улучшаться.

Алгоритм 2.4. NNDescent

Входные данные: датасет V, функция расстояния σ, K
Выходные данные: список K-NN B
B[v] ← Sample(V,K) x {∞}, ∀v∈V
    цикл
        R <- Reverse(B)
        $\bar{B}[v] \leftarrow B[v] \cup R[v],\ \forall v \in V$;
        c <- 0 //обновить счетчик
        для каждой точки $v \in V$ **выполнить**
            для каждой точки $u_1 \in \bar{B}[v], u_2 \in \bar{B}[u_1]$ **выполнить**
                $l \leftarrow \sigma(v, u_2)$
                $c \leftarrow c + UpdateNN(B[v], \langle u_2, l \rangle)$
            конец для
        конец для
        вернуть B если c = 0
    конец цикла
функция Sample (S, n):
    вернуть случайные n элементов из множества S
функция Reverse (B):
    $R[v] \leftarrow \{u | \langle v, \dots \rangle \in B[u]\ \forall v \in V$
    вернуть R
функция UpdateNN (H, $\langle u, l, \dots \rangle$):
    обновить K-NN heap H
    вернуть 1, если H изменился, 0, если нет

3. HNSW – Hierarchical navigable small world

Hierarchical NSW, HNSW [4] – подход к поиску K ближайших соседей, основанный на графах NSW (navigable small world) с управляемой иерархией. Данное решение полностью основано на графах, без необходимости каких-либо дополнительных структур для поиска. HNSW инкрементно строит многоуровневую структуру, состоящую из иерархического множества графов близости (уровней, слоев) для подмножеств хранимых элементов.

Идея алгоритма HNSW заключается в том, чтобы разделить связи, основываясь на масштабе их длины, в отдельные уровни, и затем проводить поиск в многоуровневом графе. В данном случае можно оценивать только необходимую часть связей для каждого элемента независимо от размера сети. В такой структуре поиск начинается с верхнего уровня, который имеет только самые длинные связи. Алгоритм жадно следует через элементы верхнего уровня, пока не будет достигнут локальный минимум (рис.3.). После этого, поиск перемещается на уровень ниже (который имеет более короткие связи), начинается заново с того элемента, который был локальным минимумом на прошлом уровне, и процесс повторяется. Максимальное количество связей на элемент на слое может регулироваться константой, что позволяет достичь логарифмической временной сложности поиска в сети NSW.

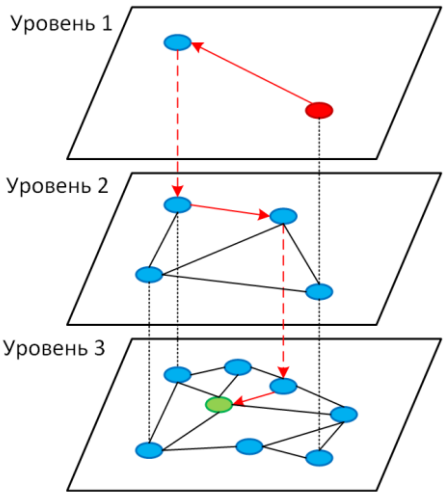

Рис.3. Поиск в HNSW

Для установлений связей во время добавления элемента в граф можно использовать эвристику, которая учитывает расстояния между кандидатами, чтобы создавать разнообразные связи вместо того, чтобы просто выбирать ближайших соседей. Эвристика исследует кандидатов начиная с самого ближнего к добавляемому элементу, и создает связь с кандидатом, только если он ближе к базовому элементу по сравнению с уже соединенными кандидатами (рис.4). Новый элемент добавляется на границу кластера 1. Все ближайшие соседи элемента принадлежат кластеру 1, между кластерами нет границ. Эвристика выбирает элемент e_2 из кластера 2, поддерживая общую связность в случае, если добавляемый элемент ближе к e_2, чем любой другой элемент в кластере 1.

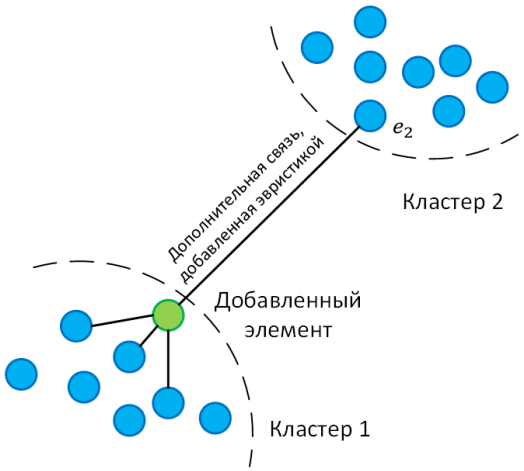

Рис.4. Эвристика, используемая для выбора соседей в графе для двух изолированных кластеров.

Алгоритм построения графа (алгоритм 3.1) основан на последовательном добавлении хранимых элементов в граф. Для каждого добавляемого элемента случайно выбирается максимальный уровень (слой) с экспоненциально затухающим распределением вероятности (нормализованным параметром m_L):

$$l \leftarrow [-\ln(unif(0..1)) \cdot m_L]$$

Первая фаза процесса добавления элементов начинается с верхнего уровня, путем жадного поиска по графу, чтобы найти ef ближайших соседей к добавляемому элементу q на слое. После этого, алгоритм продолжает поиск на

следующем слое, используя найденных на предыдущем слое ближайших соседей как точки входа, и процесс повторяется.

Ближайшие соседи на каждом слое находятся с помощью варианта алгоритма жадного поиска, описанного в алгоритме 3.2. Чтобы получить приближенных ef ближайших соседей на уровне l_c, во время поиска сохраняется динамический список W, содержащий ef ближайших найденных элементов (первоначально заполненный точками входа). Список W обновляется на каждом шаге с помощью оценивания соседей ближайшего неоцененного ранее элемента в списке, до тех пор, пока соседи каждого элемента в списке не оказываются оценены. Условие остановки в HNSW позволяет избавляться от кандидатов на оценивание, которые дальше от запроса, чем самый дальний элемент в списке W.

Во время первой фазы поиска параметру ef установлено значение 1 (простой жадный поиск), чтобы избежать добавления лишних параметров.

Когда поиск достигает уровня, который меньше либо равен l, начинается вторая фаза алгоритма построения графа. Вторая фаза отличается от первой следующим: 1) параметр ef увеличивается от 1 до efConstruction, чтобы контролировать метрику recall процедуры жадного поиска; 2) найденные ближайшие соседи на каждом слое также используются как кандидаты ля соединения с добавленным элементом.

Возможно использование двух методов выбора M соседей из кандидатов: простое соединение с ближайшими элементами (алгоритм 3.3) и эвристика, которая учитывает расстояния между элементами-кандидатами для создания связей в различных направлениях (алгоритм 3.4). Эвристика содержит два дополнительных параметра: 1) extendCandidates (по умолчанию false), который расширяет множество кандидатов и полезен для сильно кластеризованных данных; 2) keepPrunedConnections, который позволяет получить фиксированное количество соединений на элемент. Максимальное количество соединений, которое элемент может иметь на слой определяется параметром M_{max} для каждого уровня выше 0 (для нулевого уровня используется отдельный параметр M_{max0}). Если у вершины уже достаточно соседей на момент добавления новой

связи, то тогда ее расширенный список соединений уменьшается тем же алгоритмом, который используется для выбора соседей (алгоритм 3.3 или 3.4).

Процедура добавления элементов заканчивается, когда соединения добавленных элементов установлены на нулевом уровне.

Алгоритм поиска приближенных K ближайших соседей, используемый в HNSW, представлен в алгоритме 3.5. Он является грубым эквивалентом алгоритма добавления вершин на уровне l=0. Различие в том, что ближайшие соседи, найденные на нулевом уровне, которые используются как кандидаты для соединений, здесь возвращаются как результат поиска. Качество поиска контролируется параметром ef.

Алгоритм 3.1. Добавление элемента в граф, INSERT (hnsw, q, M, M_{max}, efConstruction, m_L)

Входные данные: многоуровневый граф hnsw, новый элемент q, количество устанавливаемых соединений M, максимальное количество соединений на элемент на уровне M_{max}, размер динамического списка кандидатов efConsctruction, нормализующий параметр для генерации максимального уровня присутствия элемента m_L .

Выходные данные: обновленный hnsw с добавленным элементом q

W ← ∅

ep ← входная вершина hnsw

L ← уровень, на котором расположена ep // верхний уровень hnsw

$l \leftarrow [-\ln(unif(0..1)) \cdot m_L]$ //максимальный уровень для нового элемента

для l_c от L до l+1 **выполнить**

$W \leftarrow$ SEARCH-LAYER$(q, ep, ef = 1, l_c)$

$ep \leftarrow$ ближайший элемент из W к q

конец для

для l_c от min(L, l) до 0 **выполнить**

$W \leftarrow$ SEARCH-LAYER$(q, ep, ef = efConstruction, l_c)$

$neighbors \leftarrow$ SELECT-NEIGHBORS(q, W, M, l_c) // алгоритм 3 или алгоритм 4

добавить двунаправленные связи от $neighbors$ к q на уровне l_c

для каждой $e \in neighbors$ **выполнить** // убрать лишние связи если нужно

$eConn \leftarrow neighbourhood(e)$ на уровне l_c

если $|eConn| > M_{max}$ **тогда** // на уровне $l_c = 0$ $M_{max} = M_{max0}$

$eNewConn \leftarrow$ SELECT-NEIGHBORS$(e, eConn, M_{max}, l_c)$

назначить $eNewConn$ как $neighbourhood(e)$ на уровне l_c

конец если

конец для

$ep \leftarrow W$

конец для

если l>L **тогда**

назначить q входной вершиной hnsw

конец если

Алгоритм 3.2. Поиск по слою hnsw, SEARCH-LAYER (q, ep, ef, l_c)

Входные данные: элемент запроса q, входные вершины ep, ef – количество ближайших к q элементов для возврата, номер уровня l_c

Выходные данные: ef ближайших соседей q

$v \leftarrow ep$ // множество посещенных вершин

C $\leftarrow ep$ // множество кандидатов

$W \leftarrow ep$ // динамический список найденных ближайших соседей

пока $|C| > 0$ **выполнить**

c $\leftarrow$ ближайший элемент к q из C

$C \leftarrow C/c$

$f \leftarrow$ самый дальний к q элемент из W

если $distance(c, q) > distance\ (f, q)$ **тогда**

break // все элементы в W оценены

конец если

для каждой $e \in neighbourhood(c)$ на уровне l_c **выполнить** // обновление C и W

если $e \notin v$ **тогда**

$v \leftarrow v \cup e$

$f \leftarrow$ самый дальний к q элемент из W

если $distance(e, q) < distance\ (f, q)$ или $|W| < ef$ **тогда**

$C \leftarrow C \cup e$

$W \leftarrow W \cup e$

если $|W| > ef$ **тогда**

убрать самый дальний к q элемент из W

конец если

конец если

конец если

конец для

конец пока

вернуть W

Алгоритм 3.3. Простой подход выбора соседей, SELECT-NEIGHBORS-SIMPLE (q, C, M)

Входные данные: базовый элемент q, кандидаты C, количество соседей для возврата M
Выходные данные: M ближайших элементов к q
Вернуть M ближайших к q элементов из C

Алгоритм 3.4. Эвристика выбора соседей, SELECT-NEIGHBORS-HEURISTIC (q, C, M, l_c, extendCandidates, keepPrunedConnections)

Входные данные: базовый элемент q, множество кандидатов C, количество соседей для возврата M, номер уровня l_c, флаг extendCandidates (определяет, нужно ли расширять список кандидатов), флаг keepPrunedConnections (определяет, нужно ли добавлять удаленные связи).
Выходные данные: M элементов, выбранных эвристикой
$R \leftarrow \emptyset$
W $\leftarrow C$ // очередь для кандидатов
если extendCandidates = True **тогда** //расширить кандидатов их соседями
    для каждой $e \in$ C **выполнить**
        для каждой $e_{adj} \in neighbourhood(e)$ на уровне l_c **выполнить**
            если $e_{adj} \notin W$ **тогда**
                W $\leftarrow W \cup e_{adj}$
            конец если
        конец для
    конец для
конец если
$W_d \leftarrow \emptyset$ // очередь для отвергнутых кандидатов
пока $|W| > 0$ и $|R| < M$ **выполнить**
    $e \leftarrow$ ближайший к q элемент из W
    $W \leftarrow W/e$
    если e ближе к q чем любой элемент из R **тогда**
        $R \leftarrow R \cup e$
    иначе
        $W_d \leftarrow W_d \cup e$
    конец если
конец пока
если keepPrunedConnections = True **тогда** //добавить несколько удаленных связей из W_d
    пока $|W_d| > 0$ и $|R| < M$ **выполнить**
        $w \leftarrow$ближайший к q элемент из W_d
        $W_d \leftarrow W_d/w$

$R \leftarrow R \cup w$
конец пока
конец если
вернуть R

Алгоритм 3.5. Поиск K ближайших соседей, K-NN-SEARCH (hnsw, q, K, ef)

Входные данные: многоуровневый граф hnsw, элемент запроса q, количество ближайших соседей для возврата K, размер динамического списка кандидатов ef
Выходные данные: K ближайших элементов к q
$W \leftarrow \emptyset$ //множество текущих ближайших элементов
$ep \leftarrow$ входная вершина hnsw
$L \leftarrow$ уровень, на котором расположена ep // верхний уровень hnsw
для l_c от L до 1
$W \leftarrow$ SEARCH-LAYER$(q, ep, ef = 1, l_c)$
$ep \leftarrow$ ближайший к q элемент из W
конец для
$W \leftarrow$ SEARCH-LAYER$(q, ep, ef, l_c = 0)$
вернуть K ближайших элементов к q из W

3.1. HyRec

В алгоритме HNSW подразумевается, что на стадии добавления элементов датасета в граф у каждого элемента уже есть определенное множество соседей. Такое множество можно выбрать случайным образом для экономии времени, но есть интуитивный подход, позволяющий немного улучшить качество такого распределения – HyRec [5].

HyRec использует подход случайного выбора K вершин как K кандидатов в ближайшие соседи для каждой вершины. Сначала всем вершинам назначаются случайные соседи. Пусть Nu содержит аппроксимацию текущих K ближайших соседей вершины u. Множество кандидатов получается за счет объединения трех множеств:

- Nu – текущая аппроксимация ближайших соседей вершины u
- текущие ближайшие соседи вершин в Nu
- k случайных вершин

Установив конкретный размер множества кандидатов, можно ограничить стоимость расчетов. Использование случайных вершин предотвращает поиск от попадания в локальный минимум. Для HNSW можно оставить только одну итерацию для каждой вершины, так как дальнейшее построение по алгоритму HNSW исправляет неточности и занимает больше времени.

Пусть knn(u) – множество ближайших соседей вершины u, Sample (U, K) - функция, выбирающая K случайных вершин из U.

Алгоритм 3.6. HyRec

Входные данные: множество вершин U, K – количество соседей для вершины, k – множество случайных вершин, δ – константа для ранней остановки алгоритма, iter – количество итераций.
Выходные данные: приближенный kNN граф G
для каждого u ∈ U **выполнить:**
    $knn(u) \leftarrow Sample\ (U, K)$
конец для
для t от 1 до iter **выполнить:**
    c ← 0
    для каждого u ∈ U **выполнить:**
        W ← ∅ //множество кандидатов
        для каждого v ∈ knn(u) **выполнить:**
            W ← $W \cup knn(v)$
        W ← $Sample\ (U, k)$
        для каждого w ∈ W **выполнить:**
            d ← $distance\ (u, w)$
            c ← $c + knn(u).add(w, d)$ //значение функции add = 0, если множество kkn(u) не изменилось или 1, если изменилось
            конец для
        конец для
    конец для
    если $c < \delta \cdot K \cdot |U|$ **тогда**
        вернуть knn
    конец если
конец для

4. Реализация NSG и HNSW

Методы NSG и HNSW были реализованы [6] на языке программирования Python 3.8.7. Язык Python был выбран, исходя из его преимуществ: простота структур данных, которая делает его легко понимаемым и читаемым, поддержка большого количества библиотек, поддержка всех кодировок.

Для каждого метода был создан собственный класс, объект которого можно сохранять с помощью библиотеки pickle и производить поиск по сохраненной структуре графа, не тратя время на стадию построения.

Объект класса **NSG** имеет следующие входные параметры:

K_conns – количество устанавливаемых соединений на вершину на стадии инициации;

l_constr – размер пула кандидатов для жадного поиска;

m – максимальная полустепень исхода вершины (максимальное количество исходящих соединений);

num_processes – количество процессов;

is_pool – флаг, определяющий, использовать ли несколько процессов на стадии построения графа;

init – параметр, определяющий используемый алгоритм инициации, по умолчанию равен ‘nndescent’ – для вызова NNDescent; ‘hyrec’ – для вызова HyRec.

Функция поиска в NSG **search_on_graph** имеет следующие основные входные параметры:

l – размер пула кандидатов для жадного поиска;

K – количество ближайших соседей для возврата;

Объект класса **HNSW** имеет следующие входные параметры:

M – количество устанавливаемых соединений на вершину;

mL – нормализующий параметр для генерации максимального уровня присутствия элемента;

Mmax – максимальное количество соединений на вершину на уровне (так как при добавлении входящих связей количество соседей может превысить M);

efConstruction – размер динамического списка кандидатов;

num_processes – количество процессов;

is_pool – флаг, определяющий, использовать ли несколько процессов (по умолчанию False);

add_hyrec – флаг, определяющий, стоит ли включать после инициации стадию HyRec (по умолчанию True);

select – параметр, определяющий используемый алгоритм выбора соседей, по умолчанию равен ‘simple’- для вызова select_neighbors_simple; ‘heuristic’ – для вызова select_neighbors_heuristic.

Функция поиска в HNSW **kNN_search** имеет следующие основные входные параметры:

K – количество ближайших соседей для возврата;

ef – размер динамического списка кандидатов;

select – параметр, определяющий используемый алгоритм выбора соседей, по умолчанию равен ‘simple’- для вызова select_neighbors_simple; ‘heuristic’ – для вызова select_neighbors_heuristic.

На рис.5. представлен пример структуры HNWS, которую алгоритм может построить при N=30 (количество вершин), M=5, Mmax =8, mL=1.

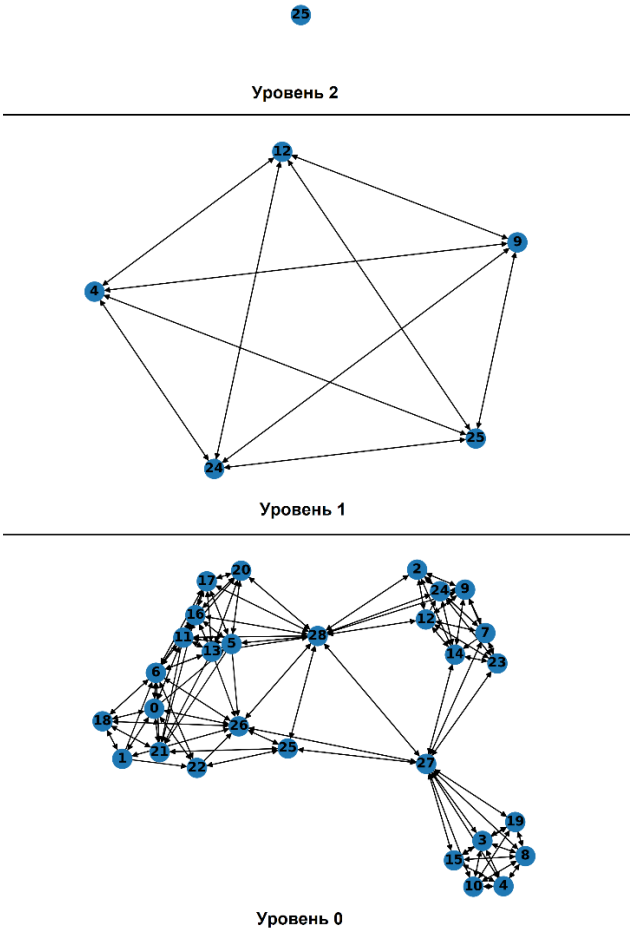

Рис.5. Структура HNSW для N=30, M=5, Mmax =8, mL=1

На рис.6. представлен пример структуры NSG, которую алгоритм может построить при N=20, K_conns =3, m =6. Навигационная вершина – 6.

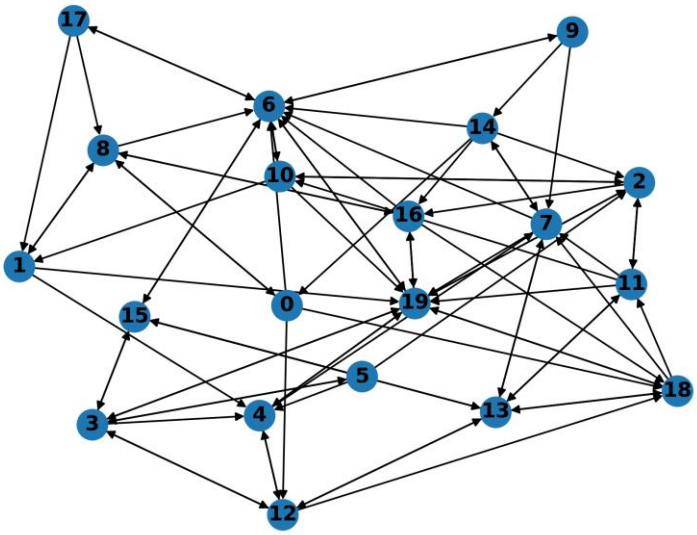

Рис.6. Структура NSG при N=20, K_conns =3, m =6

В качестве функции расстояния (близости) применялось евклидово расстояние, или L2 норма. Евклидово расстояние между двумя векторами x и y размерностью n определяется как:

$$\sigma_2(x, y) = \|x - y\|_2 = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Для оценки полученных результатов поиска была выбрана метрика Precision (Точность). Она показывает процент правильно отнесенных к некоторому классу элементов. Precision можно определить как:

$$precision = \frac{TP}{TP + FP}$$

где TP (True Positive, истинно положительные) – элементы, правильно определенные как соответствующие классу;

FP (False Positive, ложно положительные) – элементы, неверно определенные как соответствующие классу.

В терминах поиска K ближайших соседей получится:

$$precision = \frac{Q}{K}$$

где Q – количество найденных истинных ближайших соседей, то есть тех, которые принадлежат тому же классу, что и запрос;

K – общее количество найденных соседей.

4.1. Дополнения алгоритмов

В ходе реализации алгоритмов NSG и HVS были добавлены опциональные дополнения:

1) В алгоритме HNSW добавлена возможность инициализации графа алгоритмом HyRec.

Как уже было отмечено ранее, в самом алгоритме HNSW не указано, каким именно способом проходит инициализация, но она определенно подразумевается, так как алгоритм построения графа insert требует наличия у

вершин назначенных им соседей, связей. HNSW теперь можно инициализировать либо случайным выбором соседей, либо с помощью HyRec.

2) В HNSW и NSG добавлена возможность распределения этапа построения графа между несколькими процессами. Это значительно ускоряет довольно затратный по времени процесс построения в случае HNSW.

3) В HNSW добавлен этап DFS по слоям

HNSW без использования эвристики выбора соседей плохо работает с сильно кластеризованными данными. Часто происходит так, что HNSW связывает элементы внутри кластеров, а между самими кластерами не образует связей, что сразу ограничивает множество поиска кластером, в котором находится входная вершина. Связность графа нарушается, а этого допускать нельзя.

В качестве альтернативы использования эвристики был предложен подход, схожий с тем, который применяется в NSG. После этапа построения классического HNSW, на каждом слое проводится DFS (поиск в глубину). Выявляются вершины, не посещенные алгоритмом, и соединяются связью со своим ближайшим соседом из посещенных вершин. Для этой функции также предусмотрен вариант, разделяющий DFS на каждом слое между параллельными процессами (по процессу на слой).

Результатом стало то, что теперь даже в случае сильно кластеризованных данных связи между кластерами есть, все вершины достижимы из входной вершины хотя бы одним путем.

5. Сравнение HNSW и NSG на синтетических данных

Сравнение методов будет проводиться на случайно сгенерированных данных объемом 10000 элементов, размерностью 25 и 4 классами.

Был создан объект HNSW со следующими параметрами: M = 5, K = 5, Mmax = 10, mL = 0.62, ef = 100, efConstruction = 250. Результаты процессов построения и поиска в HNSW представлены на рис.7-рис.8.

```
num layers:  3
Object creation time:  1743.8047630786896
obj_size:  568081400
Memory peak during HNSW bulid:  (564770128, 958281038)

final precision: 1.0
Search time:  0.22758323669433594
Hops:  1141.884
Memory peak during HNSW search:  (586858483, 604713258)
```

Рис.7. Результаты построения и поиска в HNSW

```
q:  [-2.26895023  3.401278   -9.17970406 -4.45554687 -6.73586297 -7.87603942
 -6.18043124 -5.05214552 -1.84326529  0.92862161 -1.9081567   4.04105627
 -5.84144812  8.22684978 -9.02504518  4.84547995 -0.86704309  2.12933487
 -7.65492915 -5.79042844  5.3580672   8.9794194  -3.20804731  4.76034312
  7.70197197]
knns:  [(3.53290219565249, 7569), (3.852985999729752, 7986), (4.112115019912332, 3957), (4.302157026362
036, 7297), (4.406773863127157, 7214)]
time:  0.07600283622741699
hops:  297
q_label:  0
knns_labels:  [0, 0, 0, 0, 0]
----------------------
q:  [ 7.45537099 -7.60786199 -9.22574192 -6.96410819  6.44279207 -7.60826511
 -3.00210787  9.38975043  1.56887599  3.8051699  -3.23315755  3.35104985
  7.44452517 -8.43837624  3.58859549  9.81563752  4.33844864 -4.28331897
  5.60162849 -6.96102465 -0.0643676   8.49173804 -3.56758186 -3.3260867
 -6.46703243]
knns:  [(3.5619552020308616, 3605), (3.82022158073516, 2027), (3.9132137747336464, 5869), (3.9326833721
431598, 6639), (3.935728839772194, 2358)]
time:  0.0610044002532959
hops:  219
q_label:  1
knns_labels:  [1, 1, 1, 1, 1]
```

Рис.8. Примеры найденных соседей в HNSW

Был создан объект NSG со следующими параметрами: K = 5, l = 50, K_conns = 3, l_constr = 100, m = 6. Результаты процессов построения и поиска в NSG представлены на рис.9-рис.10.

```
init---->
build---->
DFS---->
Object creation time:  216.87611770629883
obj_size:  571903752
Memory peak during NSG bulid:  (567105235, 949563184)

final: 1.0
Hops:  346.4975
Search time:  0.046830599308013914
Memory peak during NSG search:  (242953, 284753)
```

Рис.9. Результаты построения и поиска в NSG

```
Search:
q:  [ -0.82862345   4.4126034  -10.30806941  -5.28819714  -6.97574512
  -8.91978396  -7.52284018  -3.13529954  -1.95501287   0.80098257
  -2.2425611    3.64563937  -5.55760252   7.5363048   -9.08627366
   4.69799999  -1.64191582   0.951737    -7.96400772  -6.879609
   4.58488303  10.27327861  -3.04639052   3.62075041   8.38314106]
knns:  [(3.2160033273273667, 4331), (3.2587020577663437, 5456), (3.290564192118131, 1123), (3.476302451
6543233, 6674), (3.491684294149529, 623)]
time:  0.05338025093078613
hops:  387
q_label:  0
knns_labels:  [0, 0, 0, 0, 0]
----------------------
q:  [ -7.5694813    7.76957928  -3.1340506    4.74889194   4.76529583
   7.58080485   1.08093658   6.26963854  -2.89614809  -5.3469364
   8.72083948  -2.26730894   8.347337     4.72627952   2.21000589
  -7.15370988   9.55924834  -0.97528322   0.48267535  -2.70597146
  -4.65172603   9.15999232   0.80557008 -10.1078916    1.84093515]
knns:  [(3.8187995170339653, 5052), (3.831593986947795, 2032), (3.886265104110747, 5270), (3.9113937183
86658, 5583), (4.007065345421941, 7773)]
time:  0.05046391487121582
hops:  361
q_label:  3
knns_labels:  [3, 3, 3, 3, 3]
```

Рис.10. Примеры найденных соседей в NSG

HNSW требует значительно больше времени на построение графа, чем NSG, но при этом время поиска отличается не настолько сильно. Точность обоих методов на синтетических данных достигла максимального значения precision = 1, все для всех 2000 элементов подобраны их 5 ближайших соседей.

6. Обработка реальных данных

В качестве реальных данных для тестирования моделей выбран Pistachio Dataset [7]. Он содержит данные характеристик двух классов фисташек Kirmizi Pistachio и Siirt Pistachio, всего 2148 записей размерностью 29 (1 из атрибутов – метка класса). Набор данных был разработан для классификации фисташек с изображений.

Датасет содержит 12 морфологических признаков:

1. Area
2. Perimeter
3. Major_Axis
4. Minor_Axis
5. Eccentricity
6. Eqdiasq
7. Solidity
8. Convex_Area
9. Extent
10. Aspect_Ratio
11. Roundness
12. Compactness

Также в данных есть 4 признака формы:

13. Shapefactor_1
14. Shapefactor_2
15. Shapefactor_3
16. Shapefactor_4

Кроме того, выделено еще 12 признаков, основанных на цвете:

17. Mean_RR
18. Mean_RG
19. Mean_RB
20. StdDev_RR

21. StdDev_RG

22. StdDev_RB

213. Skew_RR

24. Skew_RG

25. Skew_RB

26. Kurtosis_RR

27. Kurtosis_RG

28. Kurtosis_RB

29. Class

Обработка данных производилась в Jupiter Notebook. Удаление дубликатов показало, что таковых в данных не было (рис.11).

```
Ввод [234]: data.shape
Out[234]: (2148, 29)

Ввод [235]: data = data.drop_duplicates()

Ввод [236]: data.shape
Out[236]: (2148, 29)
```

Рис.11. Удаление дубликатов

Проверка на пустые значения также не выявила в пропусков (рис.12).

```
Ввод [183]: # Проверка наличия пустых значений
            # Цикл по колонкам датасета
            for col in data.columns:
                # Количество пустых значений - все значения заполнены
                temp_null_count = data[data[col].isnull()].shape[0]
                print('{} - {}'.format(col, temp_null_count))

Area - 0
Perimeter - 0
Major_Axis - 0
Minor_Axis - 0
Eccentricity - 0
Eqdiasq - 0
Solidity - 0
Convex_Area - 0
Extent - 0
Aspect_Ratio - 0
Roundness - 0
Compactness - 0
Shapefactor_1 - 0
Shapefactor_2 - 0
Shapefactor_3 - 0
Shapefactor_4 - 0
Mean_RR - 0
Mean_RG - 0
Mean_RB - 0
StdDev_RR - 0
StdDev_RG - 0
StdDev_RB - 0
Skew_RR - 0
Skew_RG - 0
Skew_RB - 0
Kurtosis_RR - 0
Kurtosis_RG - 0
Kurtosis_RB - 0
Class - 0
```

Рис.12. Проверка на пустые значения

Типы данных – все признаки имеют тип float64, кроме Area, Convex_Area (int64) и Class (object) (рис.13).

```
Ввод [184]: data.dtypes

Out[184]: Area              int64
          Perimeter       float64
          Major_Axis      float64
          Minor_Axis      float64
          Eccentricity    float64
          Eqdiasq         float64
          Solidity        float64
          Convex_Area       int64
          Extent          float64
          Aspect_Ratio    float64
          Roundness       float64
          Compactness     float64
          Shapefactor_1   float64
          Shapefactor_2   float64
          Shapefactor_3   float64
          Shapefactor_4   float64
          Mean_RR         float64
          Mean_RG         float64
          Mean_RB         float64
          StdDev_RR       float64
          StdDev_RG       float64
          StdDev_RB       float64
          Skew_RR         float64
          Skew_RG         float64
          Skew_RB         float64
          Kurtosis_RR     float64
          Kurtosis_RG     float64
          Kurtosis_RB     float64
          Class            object
          dtype: object
```

Рис.13. Типы данных

Были построены гистограммы по всем признакам, кроме Class (рис.14 - рис.16).

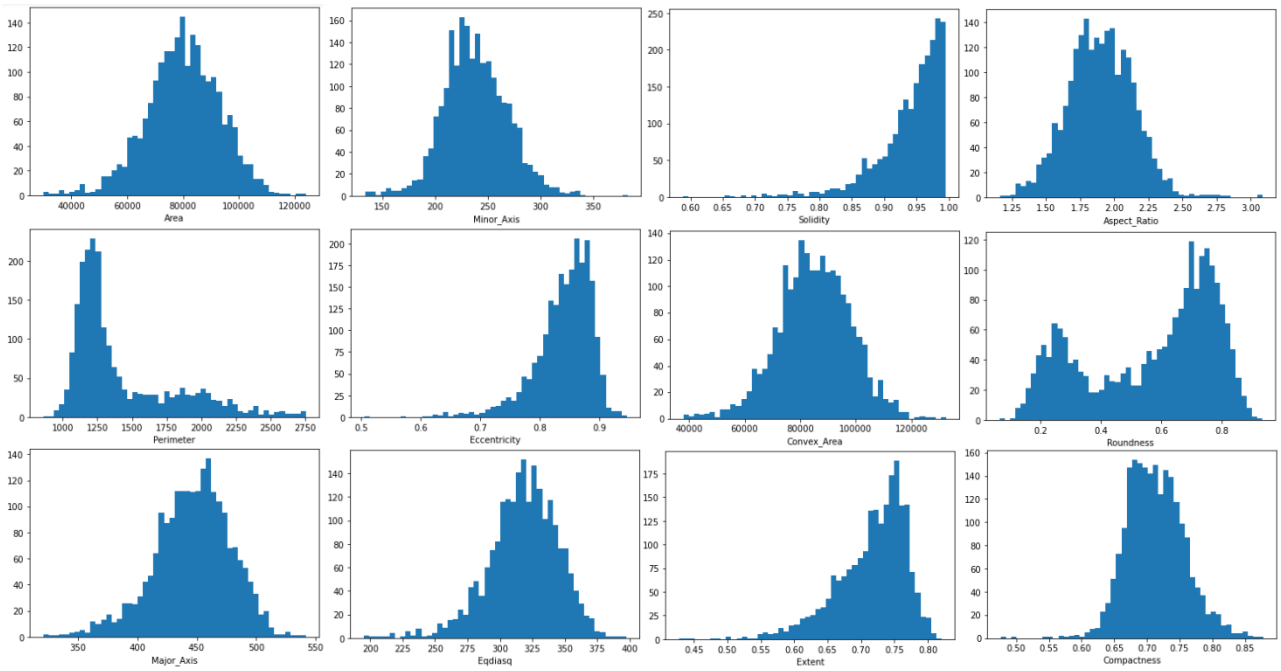

Рис.14. Гистограммы признаков

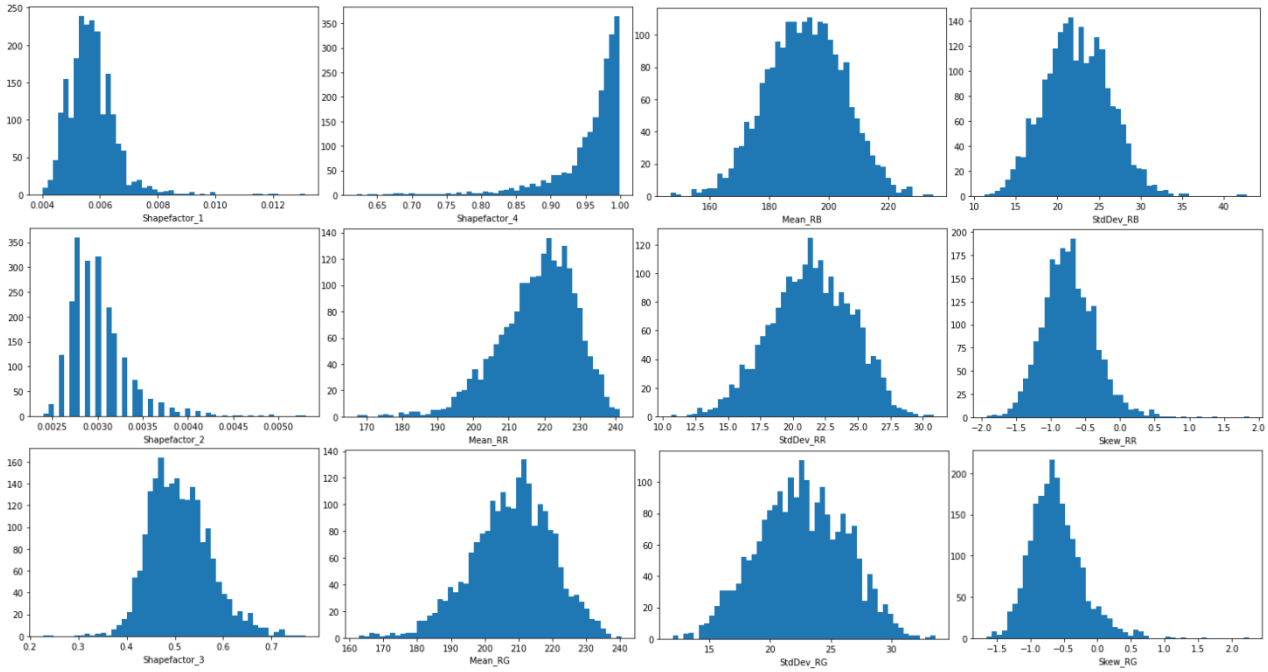

Рис.15. Гистограммы признаков

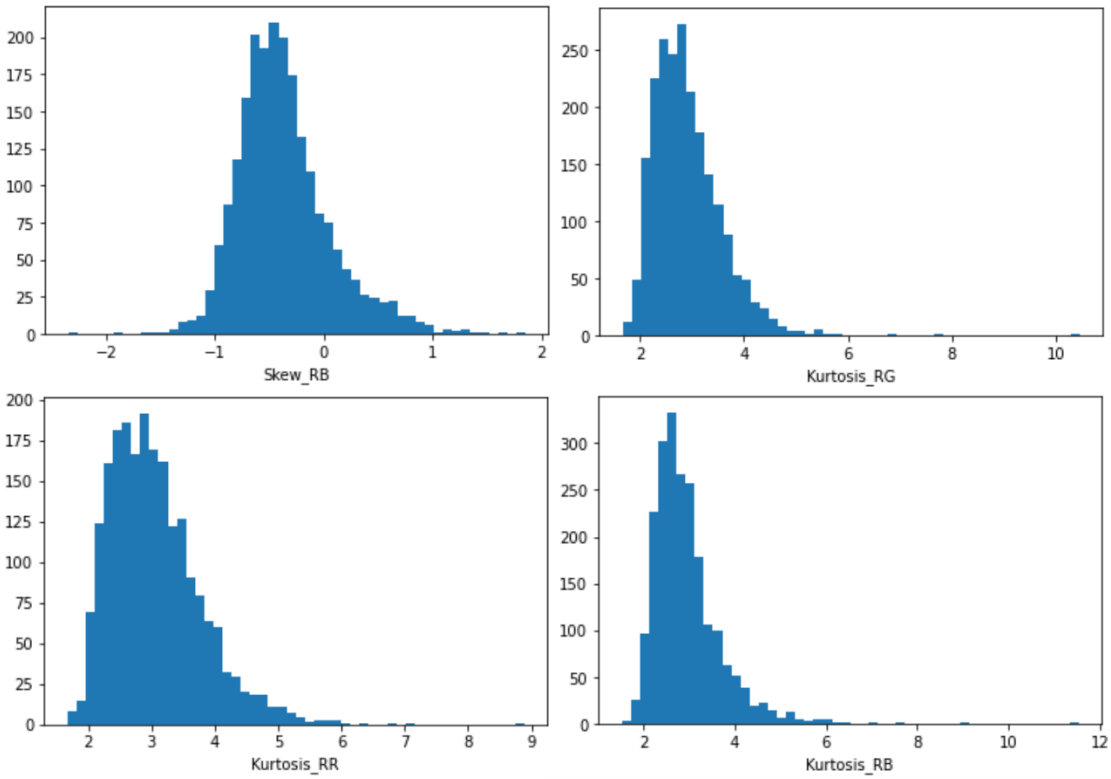

Рис.16. Гистограммы признаков

Для того, чтобы улучшить качество результатов поиска, было проведено масштабирование данных методом MinMax (рис.17 – рис.19).

Ввод [186]:
```
data.head()
```

Out[186]:

	Area	Perimeter	Major_Axis	Minor_Axis	Eccentricity	Eqdiasq	Solidity	Convex_Area	Extent	Aspect_Ratio	...	StdDev_RR	StdDev_RG	StdDev_RB
0	63391	1568.405	390.3396	236.7461	0.7951	284.0984	0.8665	73160	0.6394	1.6488	...	17.7206	19.6024	21.1342
1	68358	1942.187	410.8594	234.7525	0.8207	295.0188	0.8765	77991	0.6772	1.7502	...	26.7061	27.2112	25.1035
2	73589	1246.538	452.3630	220.5547	0.8731	306.0987	0.9172	80234	0.7127	2.0510	...	19.0129	20.0703	20.7006
3	71106	1445.261	429.5291	216.0765	0.8643	300.8903	0.9589	74153	0.7028	1.9879	...	18.1773	18.7152	29.7883
4	80087	1251.524	469.3783	220.9344	0.8823	319.3273	0.9657	82929	0.7459	2.1245	...	23.4298	24.0878	23.1157

5 rows × 29 columns

Рис.17. Данные до масштабирования

Ввод [241]:
```
columns = list(data.columns)
columns = columns[:(len(columns)-1)]
```

Ввод [242]:
```
# Числовые колонки для масштабирования
scale_cols = columns
```

Ввод [243]:
```
# Масштабирование данных
sc1 = MinMaxScaler()
sc1_data = sc1.fit_transform(data[scale_cols])
```

Ввод [244]:
```
# Добавим масштабированные данные в набор данных
for i in range(len(scale_cols)):
    col = scale_cols[i]
    data[col] = sc1_data[:,i]
```

Рис.18. Масштабирование методом MinMaxScaler библиотеки sklearn

Ввод [245]:
```
data.head()
```

Out[245]:

	Area	Perimeter	Major_Axis	Minor_Axis	Eccentricity	Eqdiasq	Solidity	Convex_Area	Extent	Aspect_Ratio	...	StdDev_RR	StdDev_RG	StdDe
0	0.356507	0.374359	0.315832	0.413713	0.657901	0.440817	0.684107	0.372582	0.539674	0.254397	...	0.351482	0.352163	0.3
1	0.409236	0.571430	0.408421	0.405724	0.715937	0.494734	0.708671	0.423680	0.635809	0.307010	...	0.795711	0.703947	0.4
2	0.464766	0.204660	0.595693	0.348827	0.834731	0.549439	0.808647	0.447405	0.726094	0.463083	...	0.415371	0.373796	0.3
3	0.438408	0.309433	0.492662	0.330881	0.814781	0.523723	0.911078	0.383085	0.700916	0.430343	...	0.374061	0.311144	0.5
4	0.533747	0.207288	0.672470	0.350349	0.855588	0.614752	0.927782	0.475910	0.810529	0.501219	...	0.633736	0.559540	0.3

5 rows × 29 columns

Рис.19. Данные после масштабирования

Далее данные были представлены в виде векторов, так как это нужно для работы алгоритмов поиска (рис.20).

```
Ввод [216]:  df_p.head()
```

Out[216]:

	label	vector
0	Kirmizi_Pistachio	[0.3565074309978769, 0.37435925744381204, 0.31...
1	Kirmizi_Pistachio	[0.40923566878980894, 0.5714303489649656, 0.40...
2	Kirmizi_Pistachio	[0.4647664543524416, 0.2046595902189614, 0.595...
3	Kirmizi_Pistachio	[0.43840764331210197, 0.30943338489168026, 0.4...
4	Kirmizi_Pistachio	[0.5337473460721869, 0.20728838577980824, 0.67...

Рис.20 Представление данных в виде векторов

Затем данные были перемешаны, так как датасет содержал строгое следование двух классов (рис.21).

```
Ввод [217]:  df_p = df_p.sample(frac=1)
```

```
Ввод [218]:  df_p.head()
```

Out[218]:

	label	vector
795	Kirmizi_Pistachio	[0.4686836518046709, 0.1700544966296742, 0.589...
1061	Kirmizi_Pistachio	[0.5929087048832271, 0.1739628924364448, 0.696...
2094	Siirt_Pistachio	[0.6763375796178344, 0.25879137301633626, 0.82...
1742	Siirt_Pistachio	[0.6146178343949045, 0.14109873004288898, 0.37...
81	Kirmizi_Pistachio	[0.29438428874734607, 0.5239833307156097, 0.42...

Рис.21. Данные после перемешивания

7. Сравнение HNSW и NSG на реальных данных

Теперь проверим модели на реальных данных (Pistachio Dataset) (Таблица 1 и Таблица 2, рис.22 и рис.23). Максимальное значение precision для обоих моделей составило 0,853. Скорее всего, это связано с тем, что кластеры находятся очень близко друг к другу, и, возможно, пересекаются, так как значения оцениваемых расстояний довольно малы (рис.24). Видно, что хоть HNSW тратит на построение графа почти в два раза больше времени, на реальных данных время поиска в HNSW куда меньше, чем в NSG, как и количество проверенных вершин и максимальная используемая память за поиск. NSG пришлось ввести большой параметр l_search = 80, чтобы достичь такой же точности, что и замедлило поиск.

Параметры HNSW: M=10, Mmax =20, mL = 0,8, efConsctruction = 40, K=5, ef= 5, num_processes = 5.

Таблица 1. HNSW на реальных данных

k_checked	t_constr, сек	t_search, сек	precision	obj_sz, Мб	peak_constr, Мб	peak_search, Кб	n_layers
99	53,44	0,022	0,853	15,19	31,3	41,4	4

Параметры NSG: K_conns =3, m = 6, l_constr = 40, K=5, l_search = 80, num_processes = 3.

Таблица 2. NSG на реальных данных

k_checked	t_constr, сек	t_search, сек	precision	obj_sz, Мб	peak_constr, Мб	peak search, Кб
277	24,35	0,0504	0,853	14,8	24,35	87,7

```
q:  [0.596135881104034, 0.18156773543076, 0.40123480743754225, 0.545248490701761, 0.5610972568578554, 0
.6715754549067721, 0.9724883321051336, 0.5228626127793702, 0.7436419125127163, 0.1866341514035179, 0.78
54846118511714, 0.8270714107987063, 0.08791208791208782, 0.24137931034482762, 0.7847426470588236, 0.981
7749603803487, 0.7240844501976813, 0.5773109837128536, 0.49138166553493745, 0.5139613985128936, 0.40492
02004697353, 0.23925917710990358, 0.327442717935212, 0.24878696562643646, 0.4418549289404147, 0.1069145
8454386985, 0.09229211232988938, 0.14168847513834232]
knns:  [(0.2963243174492279, 853), (0.2972343619814268, 3), (0.3214278422341409, 299), (0.3249112615337
9163, 1424), (0.3263048444412547, 127)]
time:  0.021373510360717773
hops:  103
q_label:  Siirt_Pistachio
knns_labels:  ['Siirt_Pistachio', 'Siirt_Pistachio', 'Siirt_Pistachio', 'Siirt_Pistachio', 'Siirt_Pista
chio']
----------------------
q:  [0.5701592356687899, 0.2231766764147215, 0.6416603796741833, 0.4056929547380843, 0.808886873724779,
 0.6481595129886959, 0.9245885531810367, 0.5147075933702124, 0.840793489318413, 0.4212629066569813, 0.6
615755627009647, 0.5697934809654142, 0.1648351648351648, 0.13793103448275867, 0.4970588235294119, 0.947
9661912308504, 0.6713415465438506, 0.6265691706202214, 0.536128987644692, 0.5716658756525864, 0.5389750
892312243, 0.3661750027725407, 0.23039241506452462, 0.1994739261453598, 0.40538481681624494, 0.24910766
165850426, 0.15275818123890583, 0.12667558982759658]
knns:  [(0.1513734438588622, 133), (0.20622807357231515, 1410), (0.2129454784214071, 471), (0.213111913
3164459, 949), (0.22228563663217502, 975)]
time:  0.0191957950592041
hops:  123
q_label:  Kirmizi_Pistachio
knns_labels:  ['Kirmizi_Pistachio', 'Kirmizi_Pistachio', 'Kirmizi_Pistachio', 'Kirmizi_Pistachio', 'Kir
mizi_Pistachio']
```

Рис.22. Пример выполнения HNSW на реальных данных

```
q:  [0.4693949044585987, 0.5281711612691208, 0.5767650806600078, 0.3505318861168606, 0.8277034686012243
, 0.5539091988555431, 0.8361581920903955, 0.4421162857112636, 0.8260427263479144, 0.4512011622477039, 0
.2365640790078089, 0.5200298581736753, 0.23076923076923073, 0.2068965517241379, 0.4461397058823531, 0.8
750660327522453, 0.5069050090505494, 0.4422618397991873, 0.378855294735136, 0.5988866476823288, 0.54303
9039816544, 0.3701928104057414, 0.4646563076112721, 0.40857551458195, 0.5837598495488847, 0.07519161063
611965, 0.0650857949114742, 0.10577941147092318]
knns:  [(0.22427495892878085, 1646), (0.28797507826165075, 66), (0.33627228318278185, 1155), (0.3411930
154072279, 1362), (0.3452342500234922, 1661)]
time:  0.04282999038696289
hops:  312
q_label:  Kirmizi_Pistachio
NNS labels:
knns_labels:  ['Kirmizi_Pistachio', 'Kirmizi_Pistachio', 'Kirmizi_Pistachio', 'Kirmizi_Pistachio', 'Kir
mizi_Pistachio']
----------------------
q:  [0.596135881104034, 0.18156773543076, 0.40123480743754225, 0.545248490701761, 0.5610972568578554, 0
.6715754549067721, 0.9724883321051336, 0.5228626127793702, 0.7436419125127163, 0.1866341514035179, 0.78
54846118511714, 0.8270714107987063, 0.08791208791208782, 0.24137931034482762, 0.7847426470588236, 0.981
7749603803487, 0.7240844501976813, 0.5773109837128536, 0.49138166553493745, 0.5139613985128936, 0.40492
02004697353, 0.23925917710990358, 0.327442717935212, 0.24878696562643646, 0.4418549289404147, 0.1069145
8454386985, 0.09229211232988938, 0.14168847513834232]
knns:  [(0.2963243174492279, 853), (0.2972343619814268, 3), (0.3214278422341409, 299), (0.3249112615337
9163, 1424), (0.3263048444412547, 127)]
time:  0.04314231872558594
hops:  278
q_label:  Siirt_Pistachio
NNS labels:
knns_labels:  ['Siirt_Pistachio', 'Siirt_Pistachio', 'Siirt_Pistachio', 'Siirt_Pistachio', 'Siirt_Pista
chio']
```

Рис.23. Пример выполнения NSG на реальных данных

```
q:  [0.6563269639065819, 0.1810030136246582, 0.5860926913261164, 0.517969515481703, 0.6962140104284744,
 0.7245838507170135, 0.9292557111274873, 0.6017367758586041, 0.8720752797558493, 0.28796762309967316, 0
.8435920992191089, 0.7034088081612342, 0.09890109890109888, 0.17241379310344818, 0.641544117647059, 0.9
661912308505016, 0.7191211585783144, 0.6387168013764151, 0.6277537009809202, 0.5726645309286506, 0.5185
12011539955, 0.3083033634880146, 0.3335791414274427, 0.2770570509219061, 0.4050991501416431, 0.10269500
013834706, 0.05438987756588229, 0.05190083305032267]
knns:  [(0.1979463943863062, 660), (0.2469875732960311, 500), (0.2664266130715988, 438), (0.28305067409
42302, 609), (0.29069355972017863, 1361)]
time:  0.031083345413208008
hops:  253
q_label:  Siirt_Pistachio
NNS labels:
knns_labels:  ['Siirt_Pistachio', 'Siirt_Pistachio', 'Siirt_Pistachio', 'Kirmizi_Pistachio', 'Siirt_Pis
tachio']
```

Рис.24. Расстояние до ошибочно определенного элемента

Заключение

Методы поиска ближайшего соседа, основанные на графах, имеют большое преимущество в сравнении с другими методами KNNS в отношении времени поиска ближайших соседей по графу. При правильно построенной структуре гарантируется высокая точность такого поиска, хоть и во многих подобных методах стадия построения графа является очень затратной по времени.

В ходе выполнения работы были реализованы два метода ANNS, основанных на графах – HNSW и NSG. HNSW использует иерархическую структуру, состоящую из уровней (слоев), которая обеспечивает более эффективный поиск благодаря разделению связей по масштабу. NSG основан на аппроксимации графа MRNG, использующей поиск из навигационной вершины для коррекции всех связей в графе с помощью стратегии выбора ребер MRNG.

Было проведено сравнение данных методов на синтетических и реальных данных, которые были заранее масштабированы и подготовлены. Результаты показали, что NSG намного быстрее строит структуру графа, чем HNSW, и при этом во времени поиска тоже выигрывает в большинстве случаев, хоть и не так значительно (что связано также с тем, что HNSW имеет многоуровневую структуру). Однако, оба алгоритма достигают высокой точности поиска, в большинстве случаев находя истинных ближайших соседей заданной точки.

Подытоживая результаты работы, можно сделать вывод, что NSG является более предпочтительным вариантом в случае больших объемов данных, но HNSW тоже может применяться при наличии достаточных вычислительных мощностей (можно потратить время на построение, чтобы впоследствии выполнять быстрый поиск в готовой структуре, в которую предусмотрена возможность добавления данных).

Список использованных источников

1) Fu C. et al. Fast approximate nearest neighbor search with the navigating spreading-out graph //arXiv preprint arXiv:1707.00143. – 2017.

2) Even, Shimon (2011), Graph Algorithms (2nd ed.), Cambridge University Press, pp. 46–48.

3) W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. Proceedings of the 20th international Conference on World Wide Web, pages 577–586, 2011.

4) Malkov Y. A., Yashunin D. A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs //IEEE transactions on pattern analysis and machine intelligence. – 2018. – Т. 42. – №. 4. – С. 824-836.

5) Antoine Boutet, Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, and Rhicheek Patra. Hyrec: leveraging browsers for scalable recommenders. In Middleware, 2014.

6) Репозиторий GitHub с реализованными в данной работе HNSW и NSG [Электронный ресурс] // github.com URL: https://github.com/PavlAA79/NIR_OBR_3sem.git, (дата обращения: 16.12.23).

7) Pistachio Dataset [Электронный ресурс] // kaggle.com URL: https://www.kaggle.com/datasets/muratkokludataset/pistachio-dataset, (дата обращения: 02.11.23).