智能问答系统实践

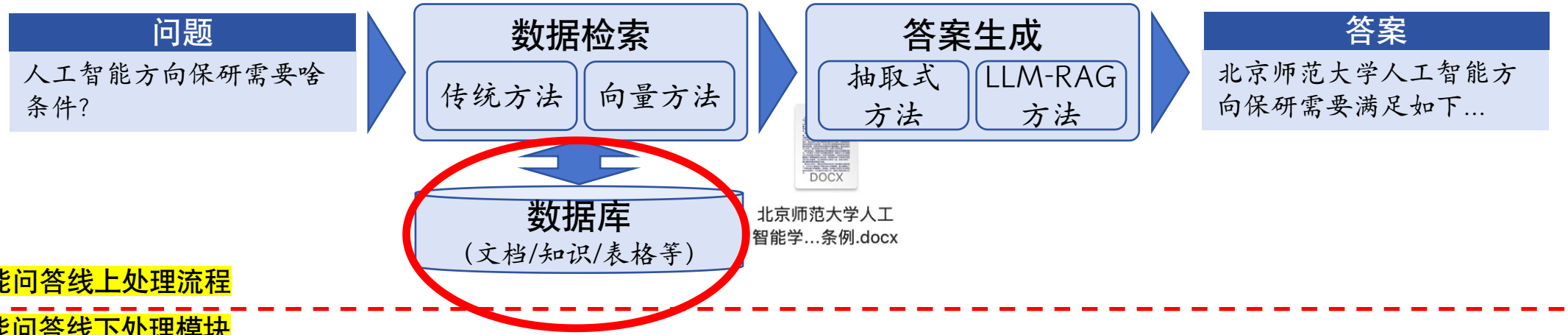
第三课：数据建库

姜文斌
北京师范大学人工智能学院
2025.03.13

我的位置

智能问答系统

针对用户提出的自然语言问题，从数据库中检索相关信息，并依据相关信息作出回答



实验设置

基于传统方法的数据建库

- 中文词法分析工具使用
  - 文档的词语切分及虚词判定
- 词语-文档的TF-IDF计算
- 基于倒排索引的文档建库
  - 文档的关键词集合提取
  - 记录关键词在文档中的位置

拓展：基于向量方法的数据建库

- 基于BERT的文本向量表示
  - 仅以文本模态展示向量化原理
- 基于K-Means算法的向量聚类
  - 给定聚类数目对文档向量进行聚类
- 基于HNSW向量索引的文档建库
  - 基于层次聚类建立层次化NSW图

目录

单词-文档矩阵

单词-文档矩阵是表达两者之间包含关系的概念模型

- 现有以下几个文档

  D1：乔布斯去了中国。

  D2：苹果今年仍能占据大多数触摸屏产能。

  D3：苹果公司首席执行官史蒂夫·乔布斯宣布，iPad2将于3月11日在美国上市。

  D4：乔布斯推动了世界，iPhone、iPad、iPad2，一款一款接连不断。

  D5：乔布斯吃了一个苹果。

单词-文档矩阵					
	D1	D2	D3	D4	D5
苹果		√	√		√
乔布斯	√		√	√	√
iPad2			√	√	

- 此时用户查询为“苹果 And (乔布斯 Or iPad2)”，表示包含单词“苹果”，同时还包含“乔布斯”或“iPad2”的其中一个

单词-文档矩阵

- 单词-文档矩阵可以从两个方向进行解读
  - 纵向：表示每个单独的文档包含了哪些单词，比如D1包含了“乔布斯”，D4包含了“乔布斯”和“iPad2”
  - 横向：表示哪些文档包含了该单词，比如D2、D3、D5包含了“苹果”

- 数据库索引是实现单词-文档矩阵的具体数据结构
  - 可以有不同的方式来实现上述概念模型，比如“倒排索引”、“签名文件”、“后缀树”等方式，但是“**倒排索引**”是实现单词到文档映射关系的最佳实现方式

正向索引

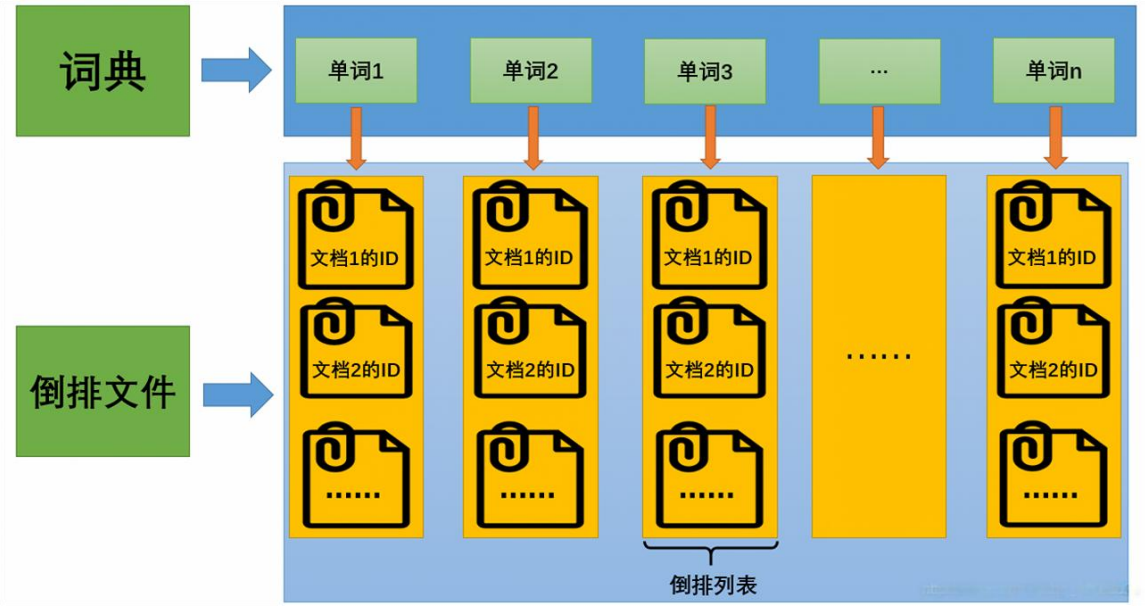
- 当用户发起查询时，搜索引擎会扫描索引库中的所有文档，找出所有包含关键词的文档，这样依次从文档中去查找是否含有关键词

反向索引

- 为了增加效率，把正向索引变为反向索引,即把“文档→单词”的形式变为“单词→文档”的形式

倒排索引定义

- 一种用于快速全文搜索的数据结构，通过记录关键词在文档中的位置来实现快速检索任务
  - 单词词典：文档的关键词集合提取并存储
  - 倒排列表：对于每个词，记录该词在哪些文档中出现，以及它在文档中的位置



基于倒排索引的文档建库

工作流程

- **前处理**：将文档内容分割成单词，并进行去停用词、词干提取等处理
- **构建索引**：为每个词创建倒排列表，记录其在文档中的出现位置
- **查询**：根据用户输入的关键词，在倒排索引中查找对应的文档

基于倒排索引的文档建库优点

- 高效的搜索：直接按词存储，快速查找某个词在哪些文档中出现
- 支持布尔查询：实现AND、OR、NOT等逻辑查询
- 支持词频和位置：记录词在文档中的具体位置，便于短语匹配和相似度计算

文档建库示例

- 假设有如下5个文档

  Doc1：乔布斯去了中国。

  Doc2：苹果今年仍能占据大多数触摸屏产能。

  Doc3：苹果公司首席执行官史蒂夫·乔布斯宣布，iPad2将于3月11日在美国上市。

  Doc4：乔布斯推动了世界，iPhone、iPad、iPad2，一款一款接连不断。

  Doc5：乔布斯吃了一个苹果。

- 这五个文档中的数字代表文档的ID，比如"Doc1"中的“1”

单词ID(WordID)	单词(Word)	倒排列表(DocID)
1	乔布斯	1，3，4，5
2	苹果	2，3，5
3	iPad2	3，4
4	宣布	3
5	了	1，4，5
…	…	…

文档建库示例

- 前处理
  - 通过分词将文档切分成单词序列，并对每个不同的单词赋予唯一的编号(WordID)
- 构建索引
  - 通过倒排索引为每个单词构建含有该单词的文档列表即倒排列表，如第一个单词ID“1”对应的单词为“乔布斯”，单词“乔布斯”的倒排列表为{1,3,4,5},即文档1、文档3、文档4、文档5都包含有单词“乔布斯”
- 查询
  - 找到所有包含查询词的文档，根据TF-IDF计算每个文档的相关性得分，并按得分从高到低排序

文档建库示例

- 包含TF和Pos的倒排索引

单词ID(WordID)	单词(Word)	倒排列表(DocID;TF;<Pos>)
1	乔布斯	(1;1;<1>),(3;1;<6>),(4;1;<1>),(5;1;<1>)
2	苹果	(2;1;<1>),(3;1;<1>),(5;1;<5>)
3	iPad2	(3;1;<8>),(4;1;<7>)
4	宣布	(3;1;<7>)
5	了	(1;1;<3>),(4;1;<3>)(5;1;<3>)
…	…	…

TF(term frequency)：单词在文档中出现的次数

Pos：单词在文档中出现的位置

- **包含TF和Pos信息的倒排索引**：前两列不变，第三列倒排索引包含的信息为(文档ID，单词频次，<单词位置>)，比如单词“乔布斯”对应的倒排索引里的第一项(1;1;<1>)意思是，文档1包含了“乔布斯”，并且在这个文档中只出现了1次，位置在第一个

代码实现

导入必要的库

```
import math
from collections import defaultdict
import jieba
```

- math：用于数学计算，特别是在计算TF-IDF时使用。
- defaultdict：用于创建默认值为列表的字典，方便存储倒排索引。
- jieba：用于中文分词，将中文文本分割成单词

代码实现

文档预处理类

```python
class Preprocessor:
    """文档预处理类，用于分词"""
    def preprocess(self, text):
        """对文本进行预处理，返回处理后的单词列表"""
        # 使用结巴分词进行分词
        words = jieba.lcut(text, cut_all=True)
        return [word.strip() for word in words if word.strip()]  # 过滤空格和空字符
```

- Preprocessor类：负责对文档内容进行预处理，使用结巴分词对文本进行分词，返回处理后的单词列表

代码实现

倒排索引类

```
class InvertedIndex:
"""倒排索引类，用于构建和查询倒排索引"""
def __init__(self):
    self.index = defaultdict(list)  # 倒排索引
    self.doc_id_counter = 0         # 文档ID计数器
    self.doc_lengths = defaultdict(int)  # 文档长度，用于TF-IDF计算
    self.doc_count = 0              # 文档总数
```

- InvertedIndex类：负责构建和查询倒排索引，初始化倒排索引、文档ID计数器、文档长度字典和文档总数

代码实现

- 添加文档到倒排索引
  - **预处理文档内容**：使用Preprocessor类对文档内容进行预处理
  - **记录单词位置**：遍历处理后的单词列表，记录每个单词在文档中的位置
  - **更新倒排索引**：将单词及其在文档中的位置信息添加到倒排索引中
  - **计算文档长度**：记录文档的长度（单词数）

```
def add_document(self, content):
    """添加文档到倒排索引"""
    doc_id = self.doc_id_counter
    self.doc_id_counter += 1
    terms = Preprocessor().preprocess(content)  # 预处理文档内容
    term_positions = defaultdict(list)          # 记录单词在文档中的位置

    # 遍历单词，记录其在文档中的位置
    for position, term in enumerate(terms):
        term_positions[term].append(position)

    # 更新倒排索引
    for term, positions in term_positions.items():
        self.index[term].append({"doc_id": doc_id, "positions": positions})

    # 计算文档长度
    self.doc_lengths[doc_id] = len(terms)
    self.doc_count += 1
def build_index(self, documents):
    """构建倒排索引"""
    for doc in documents:
    self.add_document(doc)
```

代码实现

- 查询倒排索引
  - 返回包含所有查询词的文档ID列表
- 流程
  - 遍历查询词：对每个查询词，获取其在倒排索引中的文档ID集合
  - 取交集：将所有查询词的文档ID集合取交集，得到同时包含所有查询词的文档ID列表

```python
def query(self, query_terms):
    """查询倒排索引，返回包含所有查询词的文档ID列表"""
    results = None
    for term in query_terms:
        if term in self.index:
            current_docs = {entry["doc_id"] for entry in self.index[term]}
            if results is None:
                results = current_docs
            else:
                results &= current_docs  # 取交集
        else:
            return []  # 如果有查询词不在索引中，直接返回空列表
    return sorted(results) if results else []
```

代码实现

- 对查询结果进行TF-IDF排序

```
def rank(self, query):
    """对查询结果进行TF-IDF排序"""
    query_terms = Preprocessor().preprocess(query)
    relevant_docs = self.query(query_terms)
    scores = {}
    # 计算每个相关文档的TF-IDF得分
    for doc_id in relevant_docs:
        score = 0
        doc_length = self.doc_lengths.get(doc_id, 1)
        for term in query_terms:
            if term in self.index:
                # 查找该词在文档中的出现信息
                postings = [entry for entry in self.index[term] if entry["doc_id"] == doc_id]
                if postings:
                    tf = len(postings[0]["positions"])  # 词频
                    idf = math.log(self.doc_count / (1 + len(self.index[term])))  # 逆文档频率
                    score += tf * idf
        scores[doc_id] = score / doc_length  # 归一化得分
```

代码实现

流程梳理

- **预处理查询词**：使用Preprocessor类对查询词进行分词
- **获取相关文档**：调用query方法获取包含所有查询词的文档ID列表
- **计算TF-IDF得分**：对每个相关文档，计算其TF-IDF得分
- **归一化得分**：将得分除以文档长度，进行归一化
- **排序**：按得分从高到低排序，返回排序后的文档ID和得分

代码实现

整体流程调用示例

```
documents = [
    "乔布斯去了中国",
    "苹果今年仍能占据大多数触摸屏产能",
    "苹果公司首席执行官史蒂夫·乔布斯宣布，iPad2将于3月11日在美国上市",
    "乔布斯推动了世界，iPhone、iPad、iPad2，一款一款接连不断",
    "乔布斯吃了一个苹果"
]

# 构建倒排索引
index = InvertedIndex()
index.build_index(documents)

# 查询示例
query = "乔布斯 苹果"
results = index.rank(query)
print("查询结果（按相关性排序）：")
for doc_id, score in results:
    print(f"文档ID: {doc_id}, 相关性得分: {score:.4f}")
```

代码实现

整体流程运行结果

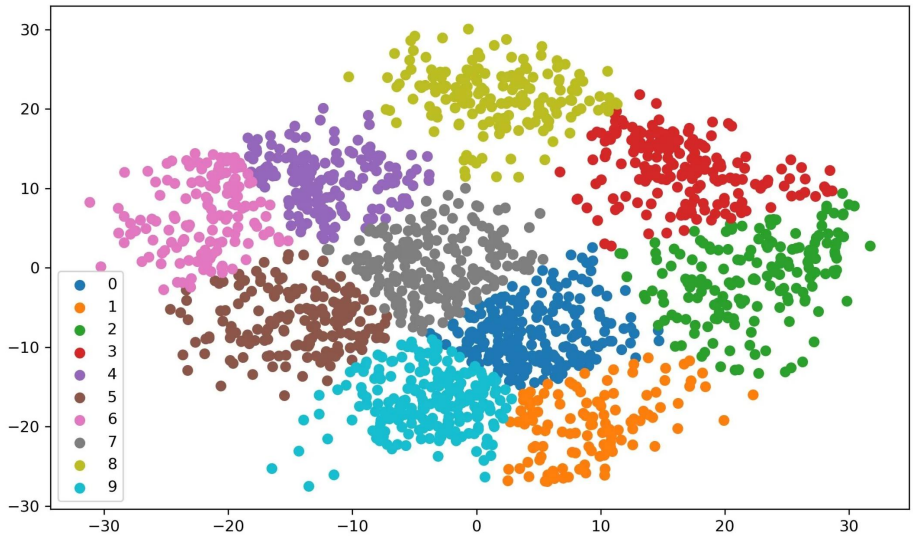
```
相关文档: [3, 5]
查询结果（按相关性排序）:
文档ID: 5, 相关性得分: 0.0319
文档ID: 3, 相关性得分: 0.0089
```

- 清理查询词：通过 query.strip() 去掉查询词首尾的空格，避免分词结果中出现不必要的空格字符
- 分词结果：查询词 "乔布斯 苹果" 被正确分词为 ['乔布斯', '苹果']
- 相关文档：找到所有包含 "乔布斯" 和 "苹果" 的文档
- 相关性得分：根据 TF-IDF 计算每个文档的相关性得分，并按得分从高到低排序

目录

K-Means算法

- K-Means是一种常用的无监督学习算法
  - 主要用于数据聚类，它的基本思想是通过迭代优化，将数据划分为 K 个簇，使得同一簇内的数据点彼此相似，而不同簇之间的数据点相异



  - 在向量化数据（如文本向量、图像特征、用户行为数据等）上，K-Means 通过计算样本之间的距离来进行聚类，常用于文本分类、推荐系统、搜索引擎优化等任务

K-Means算法

K-Means 算法的技术原理

- 目标是最小化簇内误差平方和（Sum of Squared Errors, SSE），即：

$$E = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|_2^2$$

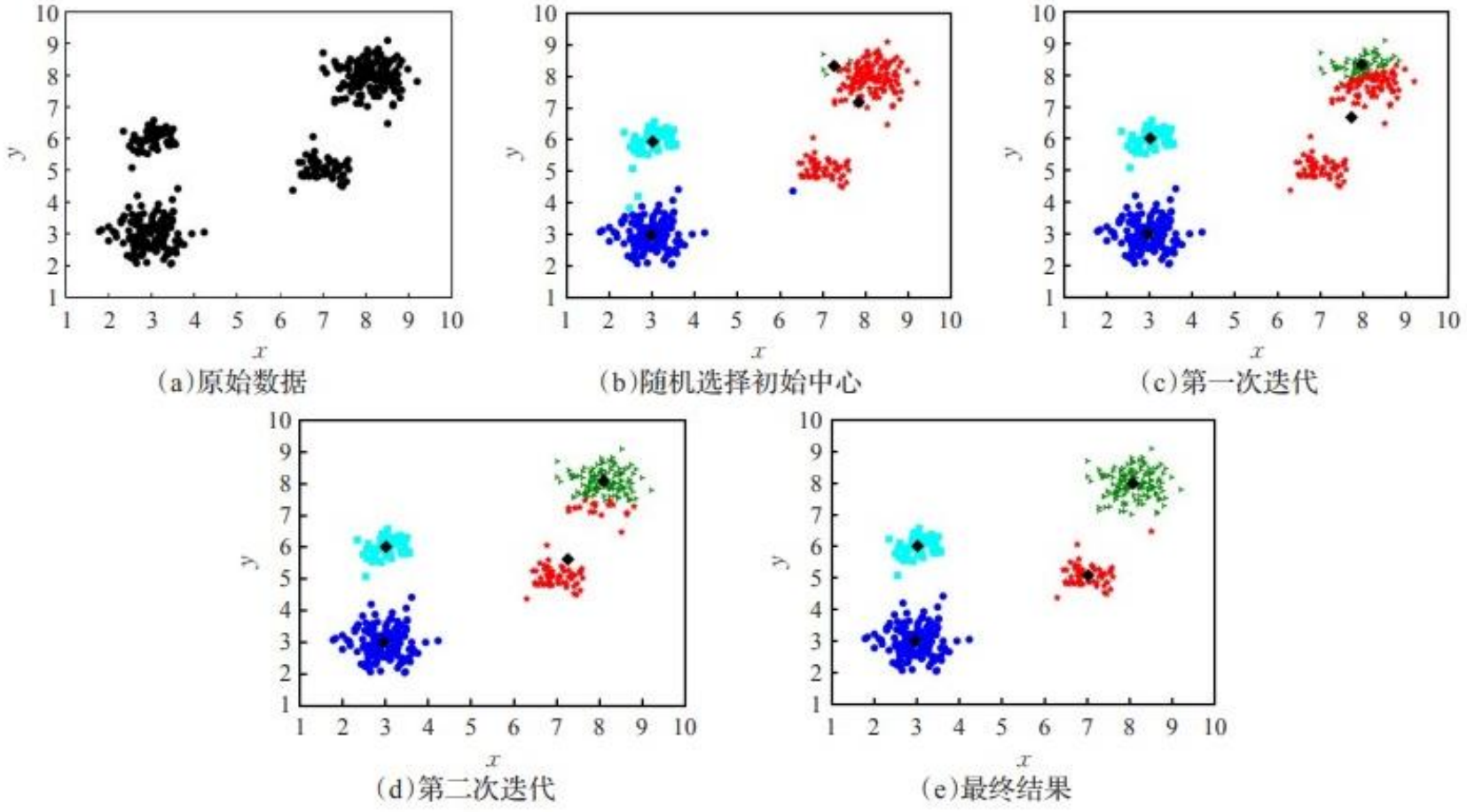
- 其中：
  - k是簇的个数
  - C_i代表第i个簇中的所有数据点
  - μ_i是第i个簇的中心点（均值向量）
  - $||x - \mu_i||^2$表示数据点到其簇中心的欧式距离

K-Means算法

- K-Means具体流程
  - **初始化**
    - 随机选择 K 个数据点作为初始聚类中心（可以是随机选取，也可以采用 K-Means++ 进行优化）
  - Repeat
    - **簇划分**
      - 计算每个数据点到 K 个簇中心的距离，并将其分配到最近的簇
    - **计算新的簇中心**
      - 计算每个簇内所有点的均值，将均值作为新的聚类中心
  - Until
    - 簇中心不再发生明显变化（收敛）或达到最大迭代次数

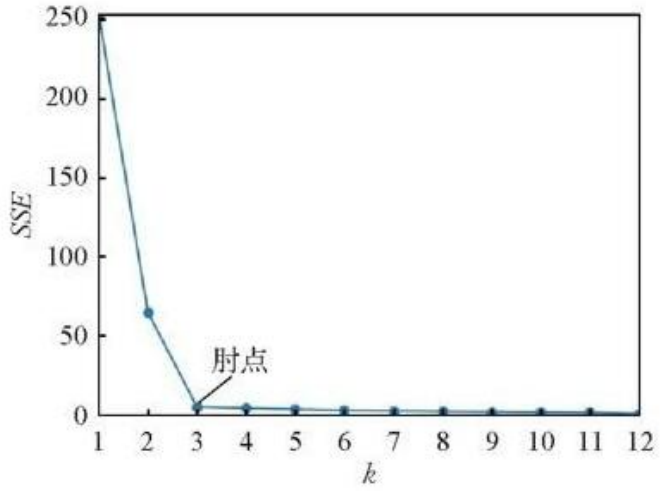
K-Means算法

- 一个简单的聚类迭代过程



K-Means算法的优化方法

- K-Means的缺点
  - K 值需要人为设定，不同 K 值得到的结果不一样
  - 对初始的簇中心敏感，不同选取方式会得到不同结果
- 合理选择 K 值
  - 肘部法（Elbow Method）：计算不同 K 值下的 SSE，选择拐点处作为最优 K



基于K-Means算法的向量聚类

数据预处理

- 对于文本数据，通常使用 TF-IDF、Word2Vec、BERT 等方法进行向量化

```
from sklearn.feature_extraction.text import TfidfVectorizer

documents = [
    "机器学习是人工智能的一个分支。",
    "深度学习是机器学习的一部分。",
    "支持向量机是一种监督学习方法。",
    "太阳系有八大行星。",
    "地球是太阳系的一部分。",
    "行星围绕恒星运行。"
]

# 使用 TF-IDF 进行向量化
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(documents)
```

基于K-Means算法的向量聚类

计算距离

- K-Means 算法默认使用欧式距离
- 对于两个n维空间中的点$A=(a_1,a_2,\ldots,a_n)$和$B=(b_1,b_2,\ldots,b_n)$，其欧氏距离公式为：

$$d(A,B)=\sqrt{\sum_{i=1}^{n}(a_i-b_i)^2}$$

```python
# 计算欧氏距离
def euclidean_distance(vec1, vec2):
    return math.sqrt(sum((a - b) ** 2 for a, b in zip(vec1, vec2)))

# 示例
vec1 = [1, 2]
vec2 = [4, 6]

print("欧氏距离:", euclidean_distance(vec1, vec2))  # 输出 5.0
```

基于K-Means算法的向量聚类

计算距离

- 但在高维数据（如文本向量）中，余弦相似度可能更合适

$$\cos(\theta) = \frac{A \cdot B}{||A|| \times ||B||}$$

```python
# 计算余弦相似度
def cosine_similarity(vec1, vec2):
    dot_product = sum(a * b for a, b in zip(vec1, vec2))
    norm_vec1 = math.sqrt(sum(a ** 2 for a in vec1))
    norm_vec2 = math.sqrt(sum(b ** 2 for b in vec2))

    if norm_vec1 == 0 or norm_vec2 == 0:  # 防止除零错误
        return 0.0

    return dot_product / (norm_vec1 * norm_vec2)
```

基于K-Means算法的向量聚类

K-Means 聚类实现

- 步骤 1：随机选择 K 个数据点作为初始簇中心

```
# 初始化 K 个随机中心
def initialize_centroids(vectors, k):
    return random.sample(vectors, k)
```

- 步骤 2：计算每个数据点到 K 个簇中心的距离，分配数据点到最近的簇

```
# 分配数据点到最近的簇
def assign_clusters(vectors, centroids):
    clusters = [[] for _ in range(len(centroids))]
    for vec in vectors:
        distances = [euclidean_distance(vec, centroid) for centroid in centroids]
        min_index = distances.index(min(distances))
        clusters[min_index].append(vec)
    return clusters
```

基于K-Means算法的向量聚类

- K-Means 聚类实现
  - 步骤 3：计算新的簇中心

```python
# 计算新的簇中心
def compute_new_centroids(clusters):
    new_centroids = []
    for cluster in clusters:
        if cluster:  # 避免空簇
            centroid = [sum(x) / len(cluster) for x in zip(*cluster)]
            new_centroids.append(centroid)
        else:
            new_centroids.append(random.choice(cluster))  # 重新随机选择
    return new_centroids
```

  - 计算每个簇的均值向量，作为新的簇中心
  - 处理空簇问题：如果某个簇为空，则随机选择一个已有数据点作为新的簇中心

基于K-Means算法的向量聚类

K-Means聚类实现

步骤 4： K-Means 算法

```
def kmeans(vectors, k, max_iters=100, tolerance=1e-4):
    centroids = initialize_centroids(vectors, k)

    for _ in range(max_iters):
        clusters = assign_clusters(vectors, centroids)
        new_centroids = compute_new_centroids(clusters)

        # 计算中心点变化量
        shift = sum(euclidean_distance(c1, c2) for c1, c2 in zip(centroids, new_centroids))

        if shift < tolerance:
            break  # 如果中心点变化量小于阈值，则收敛

        centroids = new_centroids

    return clusters, centroids
```

基于K-Means算法的向量聚类

- K-Means进行文档聚类

```
# 设置聚类数目
k = 2

# 运行 K-Means 聚类
clusters, final_centroids = kmeans(vectors, k)

# 输出聚类结果
for i, cluster in enumerate(clusters):
    print(f"簇 {i+1}:")
    for vec in cluster:
        doc_index = vectors.index(vec)
        print(" -", documents[doc_index])
    print()
```

- 输出每个簇内的文档信息
- documents[doc_index] 还原原始文本

基于K-Means算法的向量聚类

- 运行结果示例

```
簇 1:
 - 机器学习 是 人工智能 的 一个 方向
 - 深度学习 是 机器学习 的 分支
 - 支持向量机 是 一种 监督学习 方法

簇 2:
 - 太阳系 有 八 大 行星
 - 地球 是 太阳系 的 一部分
 - 行星 围绕 恒星 运行
```

- K-Means 成功地将文本分为技术（机器学习）和天文学（太阳系）两个类别

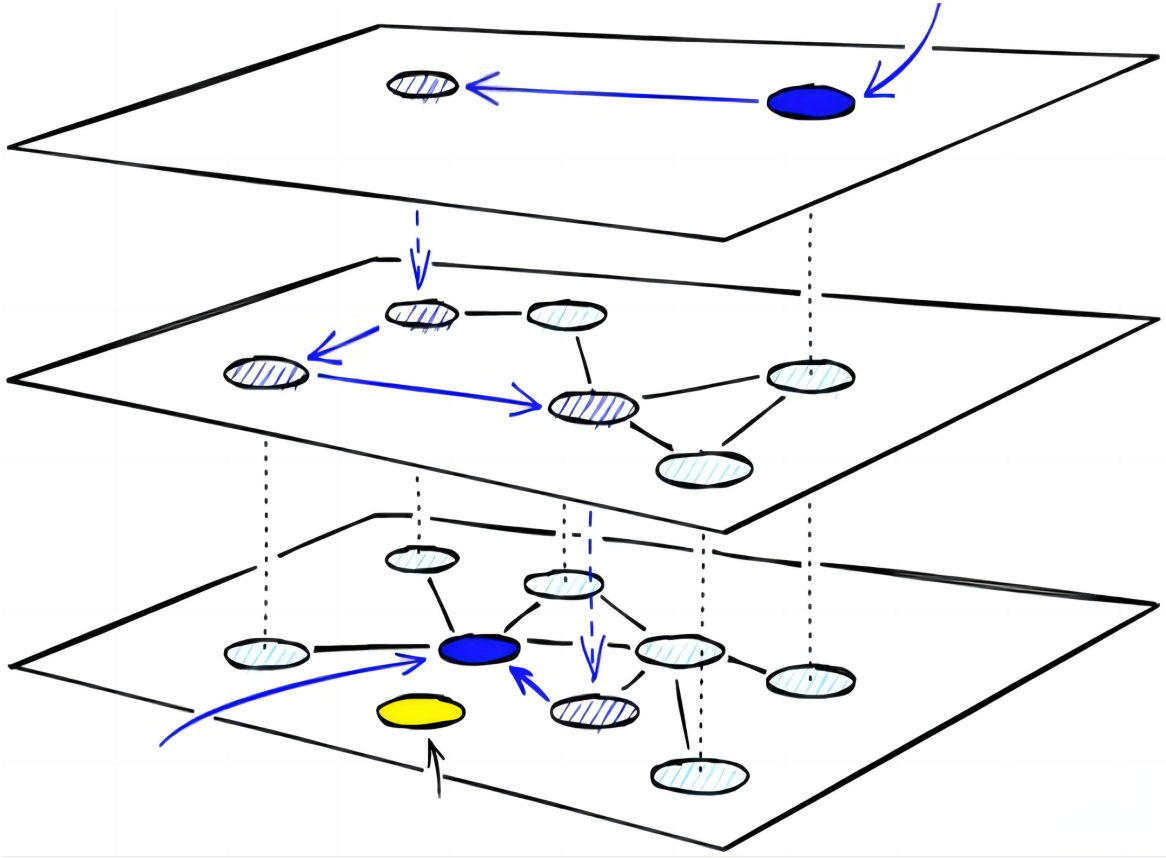
基于K-Means算法的向量聚类

- K-Means算法的改进：K-Means++
  - 在K-Means 算法中，簇中心的初始化是随机的，该方法存在缺点
    - 初始中心选择可能不理想：如果初始中心选得不合理（例如，选在了数据密集度较低的区域），K-Means 算法容易收敛到局部最优解，而不是全局最优解
    - 可能产生较差的聚类效果：由于初始化的不稳定性，最终的聚类结果可能相差较大，甚至可能得到错误的聚类
  - 改进方法
    - 初始化方法：首先随机选择一个数据点作为第一个聚类中心，然后根据已选择的中心点，选择下一个中心点，使得新的中心点距离现有中心点尽可能远
    - 这种方法倾向于在初始步骤中分散中心点，从而降低了陷入局部最优解的风险，并通常导致更好的聚类效果和更快的收敛速度

目录

向量索引的核心挑战

- **挑战**
  - 维度灾难（Curse of Dimensionality）、召回率与延迟的权衡
- **传统方法对比**
  - 暴力搜索（高精度但低效）、LSH（局部敏感哈希）、KD-Tree（低维有效）
- **HNSW**
  - 平衡效率与精度的现代解决方案
  - 工业界影响力最大的基于图的近似最近邻搜索算法



HNSW理论基础

- ANN算法
  - ANN算法旨在高维空间中快速找到与查询向量近似最近的邻接点，牺牲少量精度以换取计算效率
  - 我们可以将 ANN 算法分为三个不同的类别：树、哈希和图
- HNSW属于图类别中的一种
  - 更具体地说，它是一种接近图(proximity graph)，其中两个顶点根据它们的接近程度（越接近的顶点之间有链接）进行连接，近似距离通常以欧几里德距离来定义
- HNSW算法两个最重要的技术
  - 可导航小世界图（navigable small world graphs）
  - 概率跳表（the probability skip list）

可导航小世界图（NSW）

介绍

- NSW是一种高效的图结构，专为高维向量相似性搜索设计，结合了“小世界现象”和“贪婪路由”算法
- 通过短程（局部邻居）和长程（跨区域连接）链接交织，形成可快速导航的网络

小世界特性

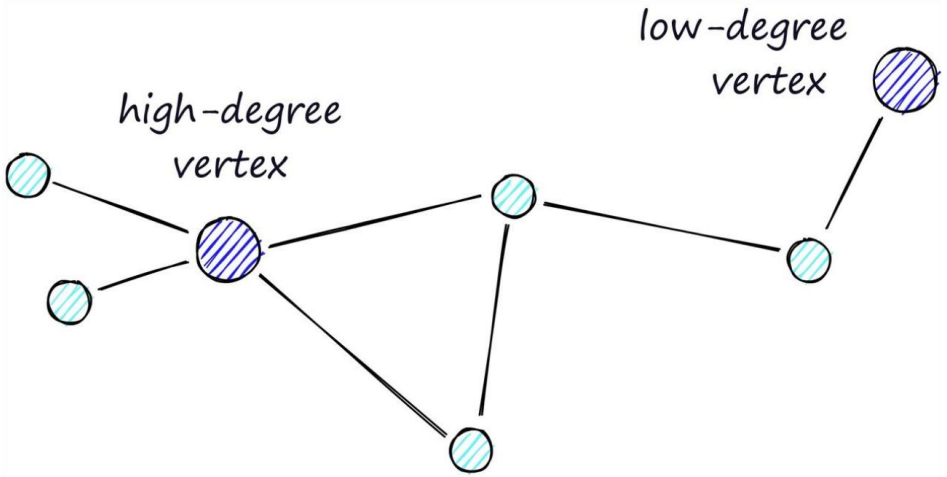
- 兼具高聚类系数（局部连接紧密）和低平均最短路径（全局跳转高效），在规则性与随机性之间取得平衡

结构特点

- 节点逐个插入，每个节点维护“朋友列表”（邻居节点），长程连接帮助跨越远距离，短程连接确保局部精度

NSW搜索机制

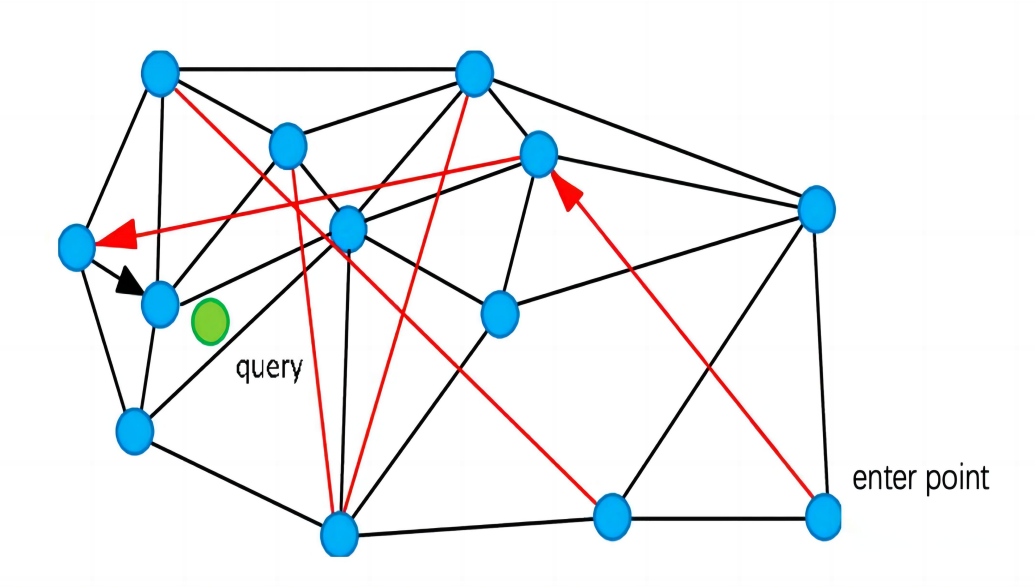
- 贪婪路由算法：从入口点出发，逐步逼近目标
  - 局部邻近性识别：在邻居中选择距离查询向量最近的节点
  - 迭代搜索：重复上述过程直至达到局部最小值（无法找到更近节点）
- 两阶段搜索
  - 放大阶段（Zoom-out）：通过长程连接快速缩小搜索范围
  - 缩小阶段（Zoom-in）：通过短程连接精细定位目标

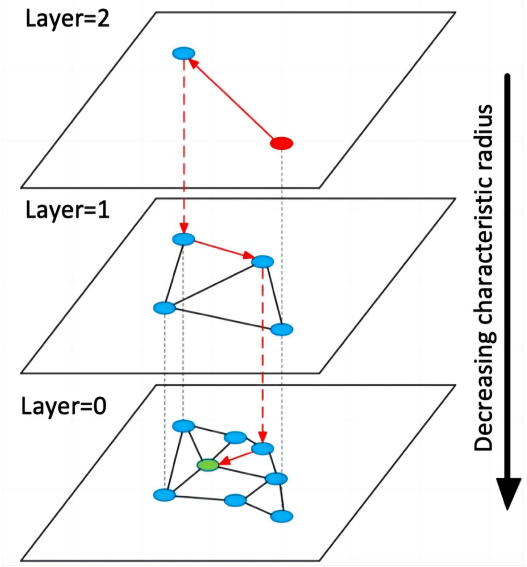


分层可导航小世界图（HNSW）

与NSW的关系

- NSW（左图）是HNSW的前身，HNSW（右图）通过引入分层结构优化NSW
- 高层：稀疏长程连接，加速全局导航
- 底层：密集短程连接，提升局部精度，显著减少搜索路径长度
- 优势：可以利用概率跳表的思想，自上而下逐层细化搜索





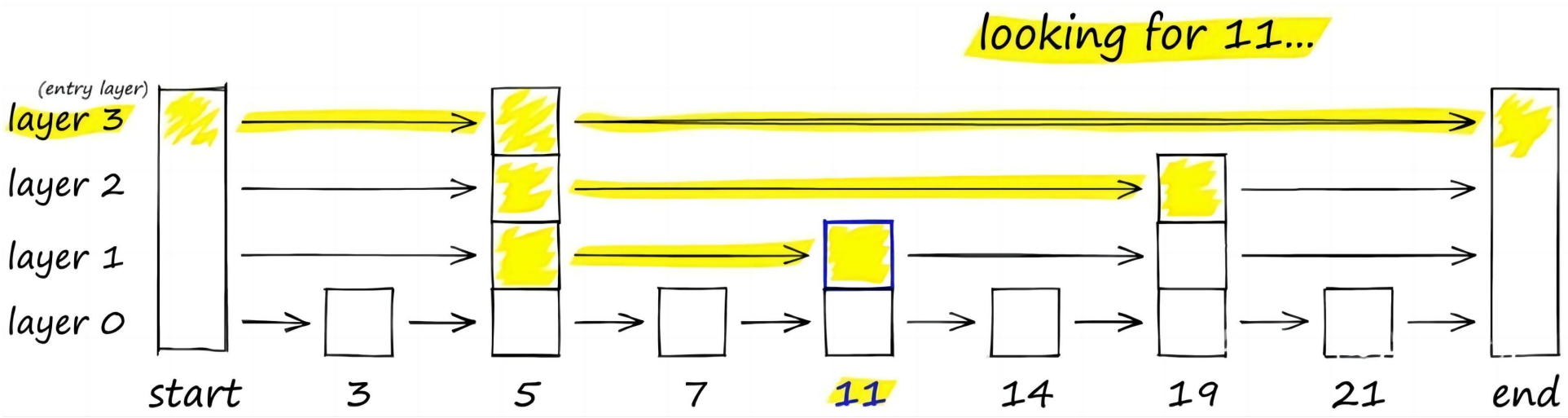
概率跳表

介绍

- 它可以像排序数组一样进行快速搜索，同时利用链表结构方便且快速地插入新元素
  这是使用排序数组无法实现的

实现

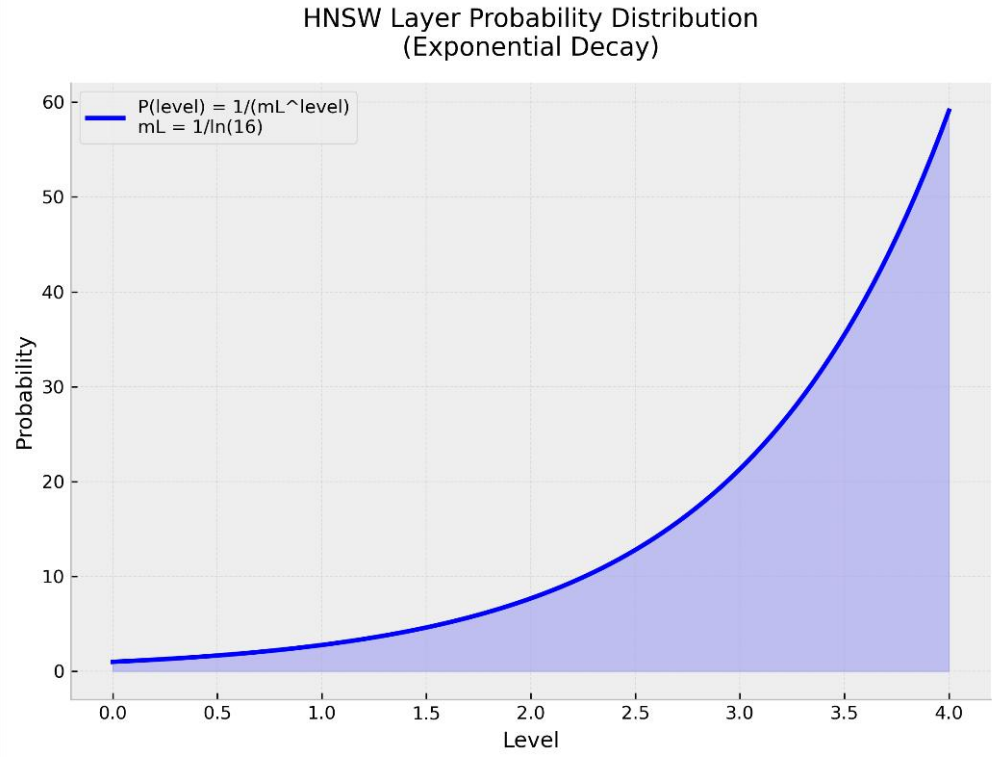
- 跳表通过构建多个链表层来工作。在第一层上，我们找到可以跳过多个中间节点的链接。随着我们向下移动到更低的层，每个链接跳过的节点数会减少

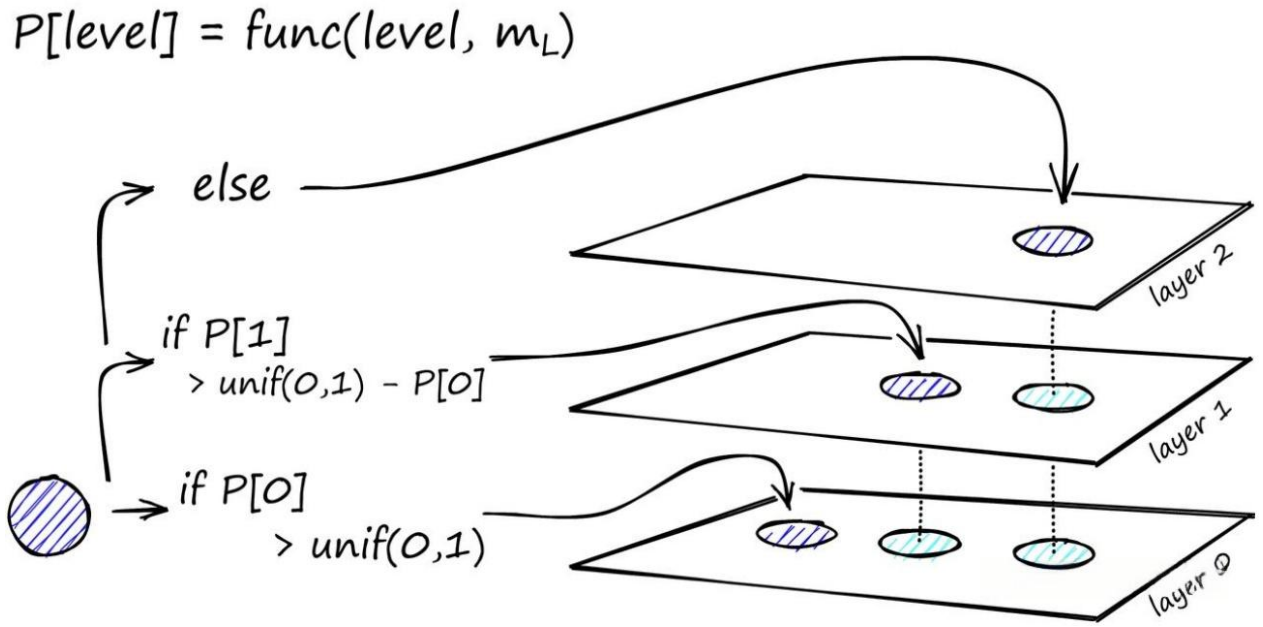


HNSW构建-插入层次分配规则

核心机制

- 向量插入的层级通过概率函数：P(level) = 1/(mL^level)决定，确保高层稀疏化
- 参数 mL 控制层级衰减速率，经验公式为 mL = 1/ln(M)





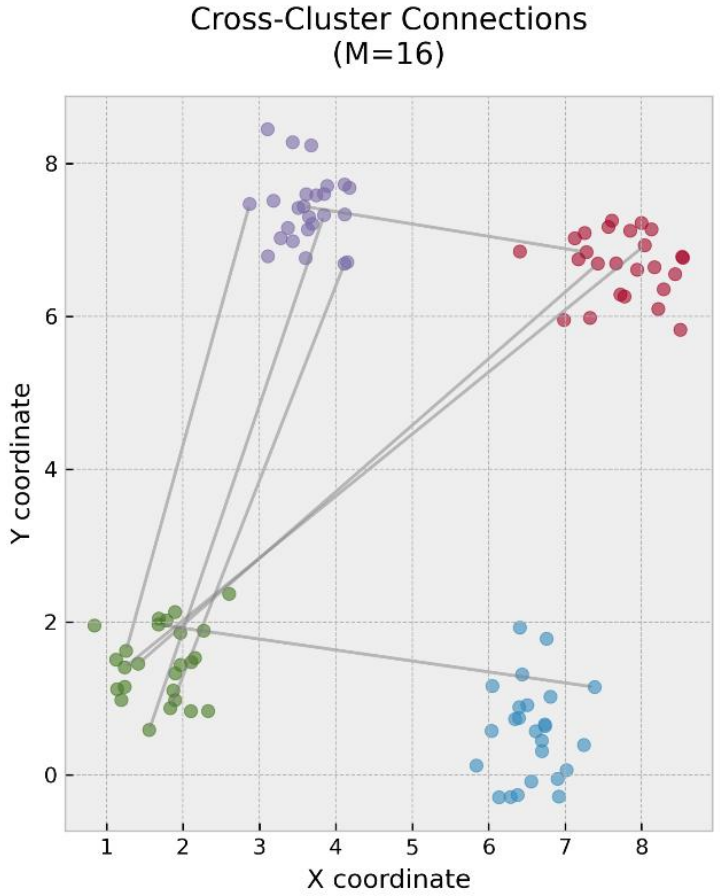
HNSW构建-构建可导航图

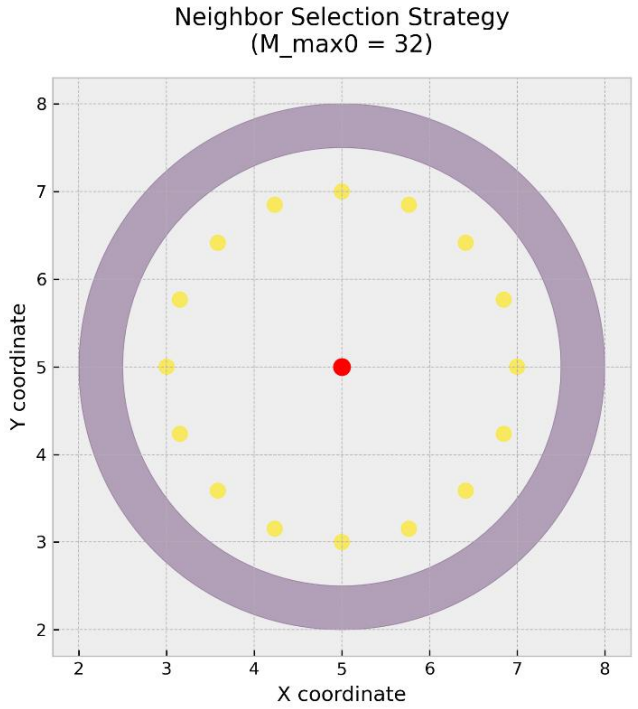
确定层连接数

- M：每层最大连接数，决定图密度
- M_max0：基础层（Layer 0）最大连接数，增强底层精度

启发式规则

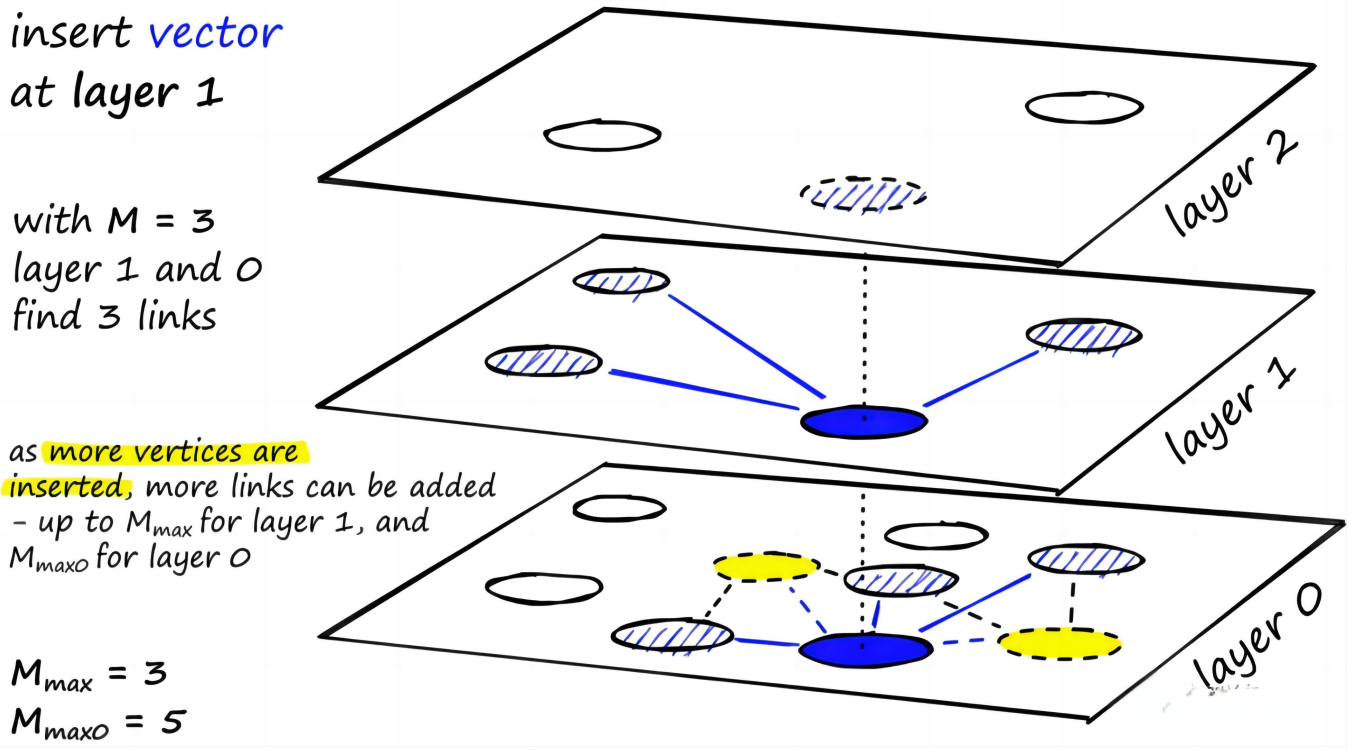
- 优先连接跨簇节点，创建“高速公路”式长边，提升全局搜索效率





HNSW构建-插入操作

- 阶段一（放大阶段）
  - 从顶层开始，ef=1快速定位入口点
  - 逐层下探至目标插入层，记录路径节点
- 阶段二（缩小阶段）
  - 在目标层扩展候选池ef，选择最近的M个邻居建立连接
  - 若节点连接数超过M_max，按距离剪枝保留最近邻



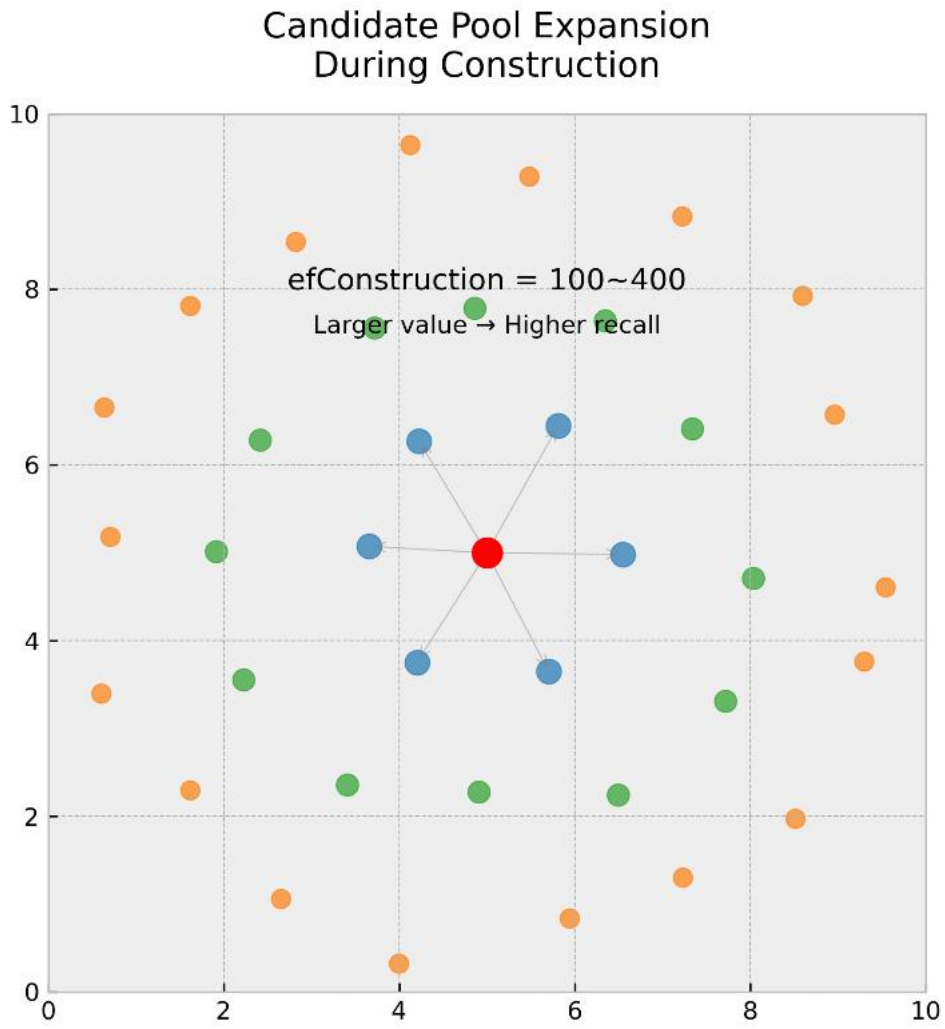
HNSW构建-邻居候选池

候选池扩展

- efConstruction：控制动态候选列表大小（默认100~400），值越大召回率越高，但构建时间增加
- 插入时从候选池选择最近M个邻居，避免局部最优

剪枝策略

- 移除冗余边（例如A→C边，如果A→B→C比它更优则删除），减少无效连接



HNSW构建流程

- **初始为空图**
  - 刚开始没有任何节点，第一个插入的节点会成为顶层的唯一成员
- **层次分配**
  - 每个新节点通过概率函数随机分配一个最高层。例如，一个节点可能被分配到第3层，那么它会出现在第0到第3层中，但更高层可能只有少数节点
- **节点插入与跨层连接**
  - 搜索路径：从顶层的入口点（初始随机选择）出发，用贪心算法找到当前层距离新节点最近的邻居，作为下一层的入口点，直到到达该节点的目标层
  - 跨层连接：在每一层中，新节点会与当前层的最近邻居建立连接。例如，在顶层可能只连接1个远距离邻居，而在底层可能连接多个近距离邻居
  - 动态候选集：在插入过程中，通过参数ef控制候选邻居数量，避免搜索范围过大

HNSW构建流程

邻居选择与剪枝

- 邻居选择：在每一层中，新节点会与当前层中距离最近的M个邻居建立连接（例如M=16）。这些邻居可能是“远亲”（长边，用于快速跳转）或“近邻”（短边，用于精细搜索）
- 剪枝冗余连接：如果某个邻居的连接数超过上限（例如顶层最多M=16），会剔除冗余连接，保留距离最近的邻居。类似“朋友圈不能无限扩大，只保留最亲密的朋友”

动态入口点更新

- 高层入口点优化：新节点如果位于高层（如第5层），可能成为该层的新入口点，后续搜索会优先从这里开始
- 保持小世界特性：通过长边（跨越多个节点的连接）和短边的结合，确保无论从哪个入口点开始，都能快速接近目标区域

基于HNSW的文档建库

- **数据预处理**
  - 文档解析与清洗：提取原始文档（如文本、图片、音频）中的结构化或非结构化数据，去除冗余字符或噪声
  - 向量生成：使用嵌入模型（如BERT）将文档内容转换为高维向量
- **HNSW索引构建**
  - 前面已经详细介绍
- **数据存储与更新**
  - 数据分片与内存管理：大规模数据需分片存储，避免单节点内存溢出
  - 增量更新与维护：支持动态插入和删除等

HNSW原理回顾

- 想象你要在一个巨大的城市地图中快速找到某个地点
  - 建高楼：先搭一个稀疏的顶层地图（只有主要地标），再逐步填充细节到底层
  - 动态修路：每新增一个地点，从顶层开始找最近的“地标”作为参考，逐层向下修路（连接邻居）
  - 控制路口数量：每个路口（节点）只保留最重要的几条路（邻居）
  - 优化导航起点：新增的高层地标会成为导航的推荐起点，缩短搜索路径
  - 这种设计让HNSW既能快速插入新数据，又能在搜索时兼顾效率与精度

谢谢大家！