RAG

流程

读取文件提取文本，文本划分为 chunk，文本嵌入，构建索引，提问嵌入，相似匹配，构造 prompt，LLM 生成

优势：可解释性好，知识库可以实时更新，资源消耗低（与 SFT 相比）

框架：langchain、graphrag 等

model API

以 google gemini 为例 `AIzaSyCh_mIfQy51Kt8mYZV5A5_I7bYsH9RDycY1`

chat

```python
# pip install -U -q "google-genai"

from google import genai

# The client gets the API key from the environment variable `GEMINI_API_KEY`.
client = genai.Client(api_key="")

response = client.models.generate_content(
    model="gemini-2.5-flash", contents="Explain how AI works in a few words"
)
print(response.text)
```

embedding

```python
from google import genai

client = genai.Client(api_key="")

result = client.models.embed_content(
    model="gemini-embedding-001",
    contents="What is the meaning of life?")

print(result.embeddings)
```

其他模型或本地部署

略

分块策略

递归分块：依据分隔符逐步分割文本，直至大小符合 chunk_size

Python

```python
# 稍微复杂一点的示例文本，包含段落和换行
text = """RAG（Retrieval-Augmented Generation）是一种结合了检索和生成技术的自然语言处理模型。
它的核心思想是，在生成答案之前，先从一个大规模的知识库中检索出与问题相关的文档片段。

然后将这些片段作为上下文信息，引导生成模型（Generator）产生更准确、更丰富的回答。
这个框架显著提升了大型语言模型在处理知识密集型任务时的表现，是当前构建高级问答系统的热门技术。
"""

# 导入递归字符分块器
from langchain.text_splitter import RecursiveCharacterTextSplitter

# 初始化分块器
# 这次我们把 chunk_size 设置为80，overlap为10
# 注意看，分隔符列表是默认的
text_splitter = RecursiveCharacterTextSplitter(
    separators=["\n\n", "\n", " ", ""], # 这是默认的分隔符
    chunk_size = 80,
    chunk_overlap  = 10,
    length_function = len,
)

# 进行分块
chunks = text_splitter.split_text(text)

# 我们来看看分块结果
for i, chunk in enumerate(chunks):
    print(f"--- Chunk {i+1} ---")
    print(chunk)
    print(f"(长度: {len(chunk)})\n")
```

```
--- Chunk 1 ---
RAG（Retrieval-Augmented Generation）是一种结合了检索和生成技术的自然语言处理模型。
它的核心思想是，在生成答案之前，先从一个大规模的知识库中检索出与问题相关的文档片段。
（长度：79）

--- Chunk 2 ---
然后将这些片段作为上下文信息，引导生成模型（Generator）产生更准确、更丰富的回答。
（长度：46）

--- Chunk 3 ---
这个框架显著提升了大型语言模型在处理知识密集型任务时的表现，是当前构建高级问答系统的热门技术。
（长度：57）
```

代码与 markdown

```python
python_splitter = RecursiveCharacterTextSplitter.from_language(
    language=Language.PYTHON, chunk_size=150, chunk_overlap=0
)

markdown_splitter = RecursiveCharacterTextSplitter.from_language(
    language=Language.MARKDOWN, chunk_size=100, chunk_overlap=0
)
```

基于文本检索

TF-IDF

1. 词频：一个词语在一个文档中出现的频率越高，那么文档的相关性也越高

$$\mathrm{TF}(t,d) = \frac{\text{词t在文档d中的出现次数}}{\text{文档d的总词数}}$$

2. 逆向文档概率：一个词在整个文档集合中的稀有度，如果一个词在多个文档中出现，则 IDF 较低

$$\mathrm{IDF}(t, D) = \log\left(\frac{\text{文档集合D的总文档数}}{\text{包含词t的文档数} + 1}\right)$$

3.TF–IDF 值：计算出文档的每个词的TF–IDF值，然后按降序排列，取排在最前面的几个词

$$\text{TF-IDF}(t, d, D) = \mathrm{TF}(t, d) \times \mathrm{IDF}(t, D)$$

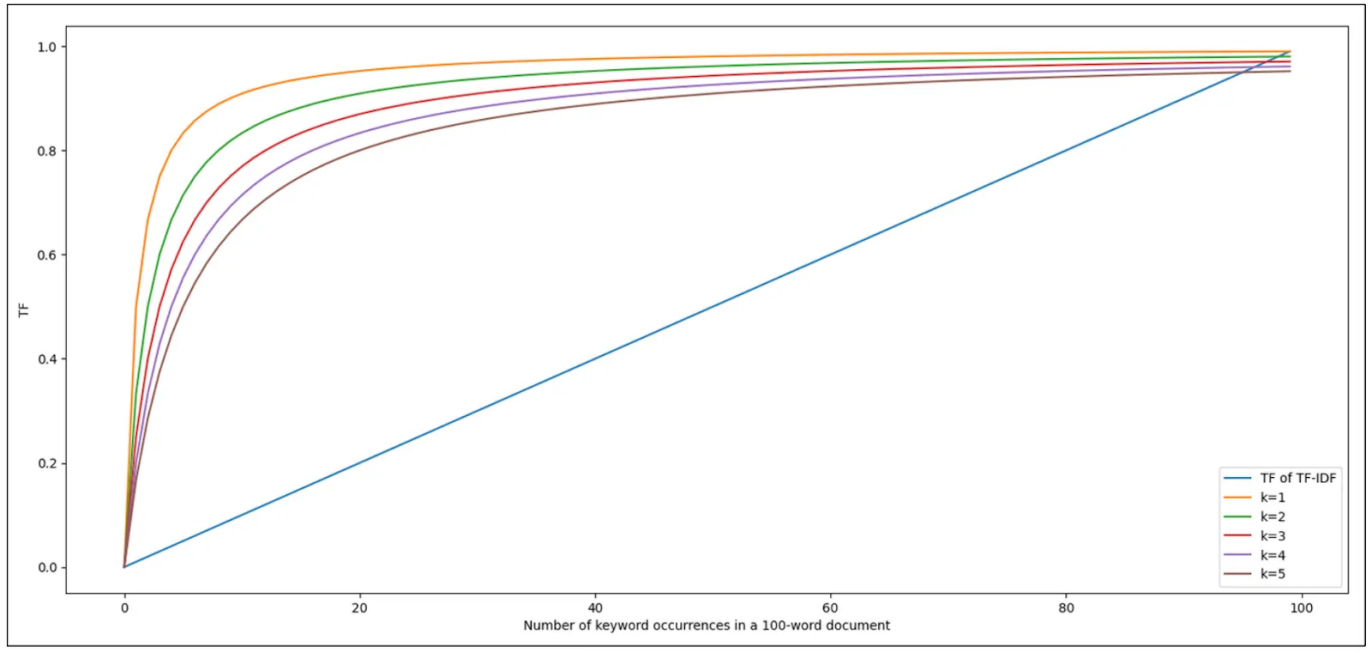
bm25

bm25 是对TF–IDF 的改进，主要考虑`关键词出现次数`和`文档长度`

1. 关键词出现次数：假设文档长度不变为 100，一个关键词从出现 2 次增长到出现 4 次，肯定比从出现 50 次增长到 52 次更关键。

因此为 TF 引入一个参数 k，如下图所示，加入 k 后，TF 曲线逐渐变缓，说明随着关键词次数越多，TF 增长越小。另外可以发现，k 越大，TF 缓和的越慢，对于长文档来说，这是合理的。

$$TF(x, y) = \frac{TF(x, y)}{(TF(x, y) + k)}$$



2.文档长度：一篇包含一个关键词的 10 字文档比一篇包含 10 个关键词的 1000 字文档更有优势。因此需要惩罚过长的文档。

|D|表示文档长度，avg(D)表示语料库中文档的平均长度。如果文档长度高于平均长度，则相当于间接变大了 k，对于长文档来说是合理的。

$$TF(x,y) = \frac{TF(x,y)}{(TF(x,y) + k * \frac{|D(y)|}{avg(D)})}$$

但是，不排除某些场景下，较长的文档也是重要的。因此，在TF方程中引入了一个额外的参数b，以控制文档长度在总得分中的重要性。

如果将b的值设为 0，则|D|/avg(D) 的比率不会被考虑，这意味着不重视文档的长度。如果 b 是一个大于 1 的值，且|D|/avg(D)

>1，即当前文档大于平均长度，同理，相当于间接选择了更大的k，是合理的。

$$TF(x,y) = \frac{TF(x,y)}{(TF(x,y) + k * (1 - b + b * \frac{|D(y)|}{avg(D)}))}$$

3.IDF： q_i 对应原公式的 t， N 对应文档数，修改主要为了使数值更平滑。

$$IDF(q_i) = ln(1 + \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5})$$

最终：

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

在实际应用中，k = 1.5 和 b = 0.75 的值在大多数语料库中效果良好。

```python
# pip install rank_bm25
# https://github.com/dorianbrown/rank_bm25

from rank_bm25 import BM25Okapi

# 知识库
corpus = [
    "Hello there good man!",
    "It is quite windy in London",
    "How is the weather today?"
]

tokenized_corpus = [doc.split(" ") for doc in corpus]
bm25 = BM25Okapi(tokenized_corpus)

# 查询
query = "windy London"
tokenized_query = query.split(" ")

# 1.计算相似分数
doc_scores = bm25.get_scores(tokenized_query)
# array([0.        , 0.93729472, 0.        ])

# 2.直接检索
bm25.get_top_n(tokenized_query, corpus, n=1)
# ['It is quite windy in London']
```

基于语义检索

bm25 根据关键词来匹配文档，embedding model 通过向量相似度来匹配。

常用 embedding model：M3E,BGE

使用

```python
# pip install sentence-transformers

from sentence_transformers import SentenceTransformer

# 加载一个预训练好的中文Embedding模型
model = SentenceTransformer('moka-ai/m3e-base')

# 准备几个待转换的句子
sentences = [
    "我喜欢吃苹果",
    "我喜欢吃香蕉",
    "今天天气真好",
    "我讨厌上班"
]

# 使用模型将句子编码为向量
embeddings = model.encode(sentences)

# 我们来看看结果
for sentence, embedding in zip(sentences, embeddings):
    print("句子:", sentence)
    # 打印向量的前5个维度和向量的总维度
    print(f"向量 (前5维): {embedding[:5]}")
    print(f"向量维度: {len(embedding)}")
    print("-" * 20)
```

```
句子: 我喜欢吃苹果
向量 (前5维): [ 0.01391807 -0.01953284  0.01596547 -0.01229419 -0.00160986]
向量维度: 768
--------------------
句子: 我喜欢吃香蕉
向量 (前5维): [ 0.01850123 -0.01908993  0.00392336 -0.01168233 -0.00832363]
向量维度: 768
--------------------
句子: 今天天气真好
向量 (前5维): [ 0.00445524 -0.03813957  0.01150338 -0.0321528  -0.03158003]
向量维度: 768
--------------------
句子: 我讨厌上班
向量 (前5维): [-0.00890695 -0.03367128  0.03842103  0.0210134  -0.01174621]
向量维度: 768
--------------------
```

计算相似度

```python
from sentence_transformers import util

# 计算"我喜欢吃苹果"和其它所有句子之间的余弦相似度
query_embedding = embeddings[0]
other_embeddings = embeddings[1:]

# util.cos_sim会返回一个张量(tensor)，包含查询向量和其它所有向量的相似度
cosine_scores = util.cos_sim(query_embedding, other_embeddings)

print(f"查询句子: '{sentences[0]}'")
for i in range(len(other_embeddings)):
    print(f"与 '{sentences[i+1]}' 的相似度: {cosine_scores[0][i]:.4f}")
```

```
查询句子: '我喜欢吃苹果'
与 '我喜欢吃香蕉' 的相似度: 0.9038
与 '今天天气真好' 的相似度: 0.5847
与 '我讨厌上班' 的相似度: 0.6120
```

embedding model 训练

infoNCE（Information Noise Contrastive Estimation）

对于 1 个查询样本 q，为它准备 1 个正样本 k_+，K 个负样本 k_i（噪声样本）。

分子是正样本的相似度，分母是负样本的相似度。embedding 效果越好，分子越大，分母越小，log 越大，负 log 越小。

$$L_q = -log\frac{exp(q \cdot k_+/\tau)}{\sum_{i=0}^{k} exp(q \cdot k_i/\tau))}$$

CoSENT（Contrastive Sentence Embedding with Normalization）

$$loss_{CoSENT} = log(1 + \sum_{(i,j)\in\Omega_{pos},(k,l)\in\Omega_{neg}} e^{\lambda(cos(u_k,u_l)-cos(u_i-u_j))})$$

(i,j)是正样本对，(k,l)是负样本对，embedding 效果越好，cos(k,l)−cos(i,j)越小，loss 越小。

向量数据库

向量数据库存储待检索内容 embedding 后的向量，并建立索引加速查询。

why：相似度检索一般的解决方案是暴力检索，循环遍历所有向量计算相似度然后得出TopK，当向量数量级达到百万千万甚至上亿级别，很耗时。

faiss

基本使用

Python

```python
# pip install faiss-cpu

import faiss
import numpy as np

d = 64                                           # 向量维度
nb = 100000                                      # 知识库文本的数据量
nq = 10000                                       # query的数目
np.random.seed(1234)

xb = np.random.random((nb, d)).astype('float32')
xb[:, 0] += np.arange(nb) / 1000.                # 知识库文本向量
xq = np.random.random((nq, d)).astype('float32')
xq[:, 0] += np.arange(nq) / 1000.                # query向量

index = faiss.IndexFlatL2(d)    # 精确检索，使用L2距离计算相似度
print(index.is_trained)         # 输出为True，代表该类index不需要训练，只需要add向量进去即可
index.add(xb)                   # 将向量库中的向量加入到index中
print(index.ntotal)             # 知识库文本数量

k = 4                          # topK的K值
D, I = index.search(xq, k)  # xq为待检索向量，返回的I为每个待检索query最相似TopK的索引list，D为其对应的距离
print(I[:5])
print(D[-5:])

print(np.array(D).shape)
print(np.array(I).shape)

"""
True
100000
[[ 381  207  210  477]
 [ 526  911  142   72]
 [ 838  527 1290  425]
 [ 196  184  164  359]
 [ 526  377  120  425]]
[[6.5315704 6.9787292 7.003937  7.013794 ]
 [4.335266  5.2369385 5.3194275 5.7032776]
 [6.072693  6.576782  6.6139526 6.7323   ]
 [6.6374817 6.6487427 6.8578796 7.0096436]
 [6.2183685 6.4525146 6.548767  6.581299 ]]
(10000, 4)
(10000, 4)
```

```
44  """
```

常用索引

Faiss之所以能加速，是因为它用的检索方式并非精确检索（例如 L2），而是模糊检索。既然是模糊检索，那么必定有所损失。

上面的代码也能构建索引，构建索引的更规范方式：

```python
dim, measure = 64, faiss.METRIC_L2
param = 'Flat'
index = faiss.index_factory(dim, param, measure)

# dim为向量维数
# param代表需要构建什么类型的索引
# measure为向量相似度的度量方法（以下8种）
"""
METRIC_INNER_PRODUCT（内积）
METRIC_L1（曼哈顿距离）
METRIC_L2（欧氏距离）
METRIC_Linf（无穷范数）
METRIC_Lp（p范数）
METRIC_BrayCurtis（BC相异度）
METRIC_Canberra（兰氏距离/堪培拉距离）
METRIC_JensenShannon（JS散度）
"""
```

Flat：暴力检索

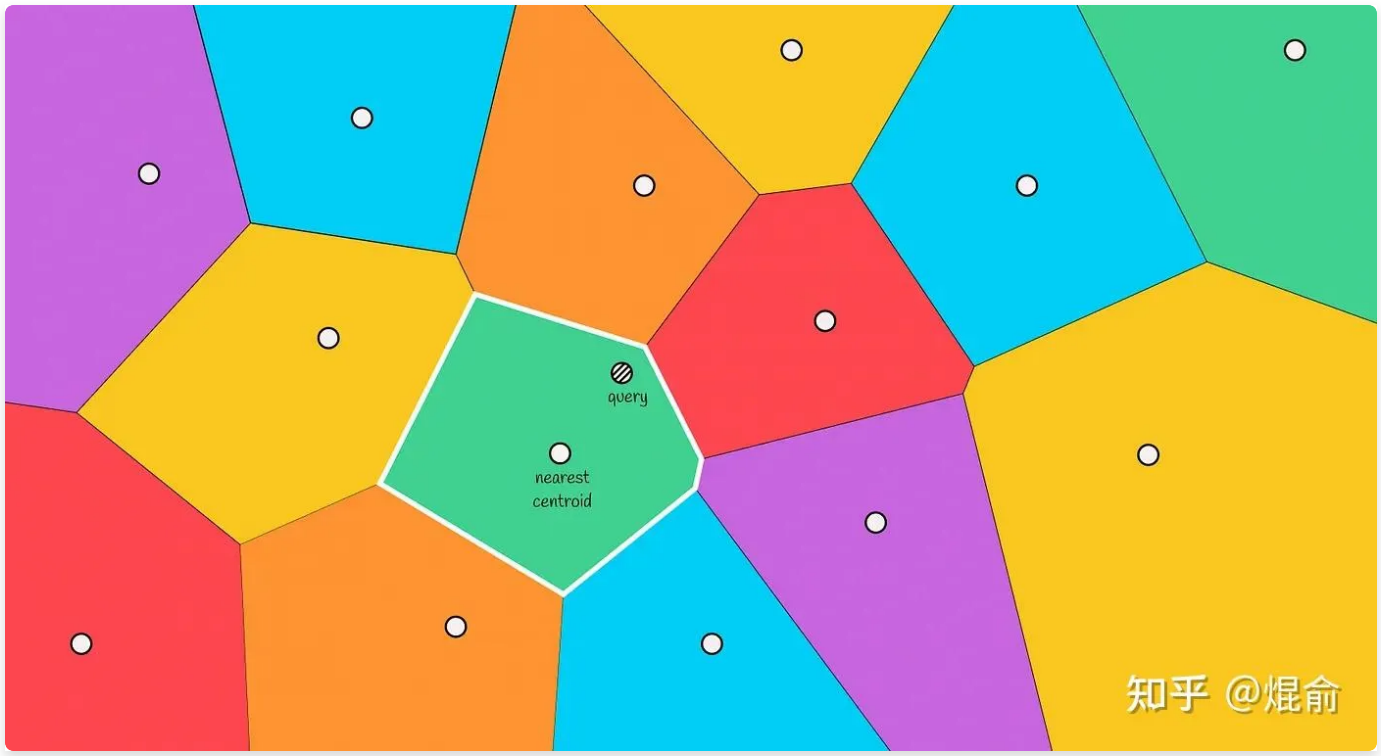
优点：该方法是Faiss所有index中最准确的，召回率最高的方法，没有之一；

缺点：速度慢，占内存大。

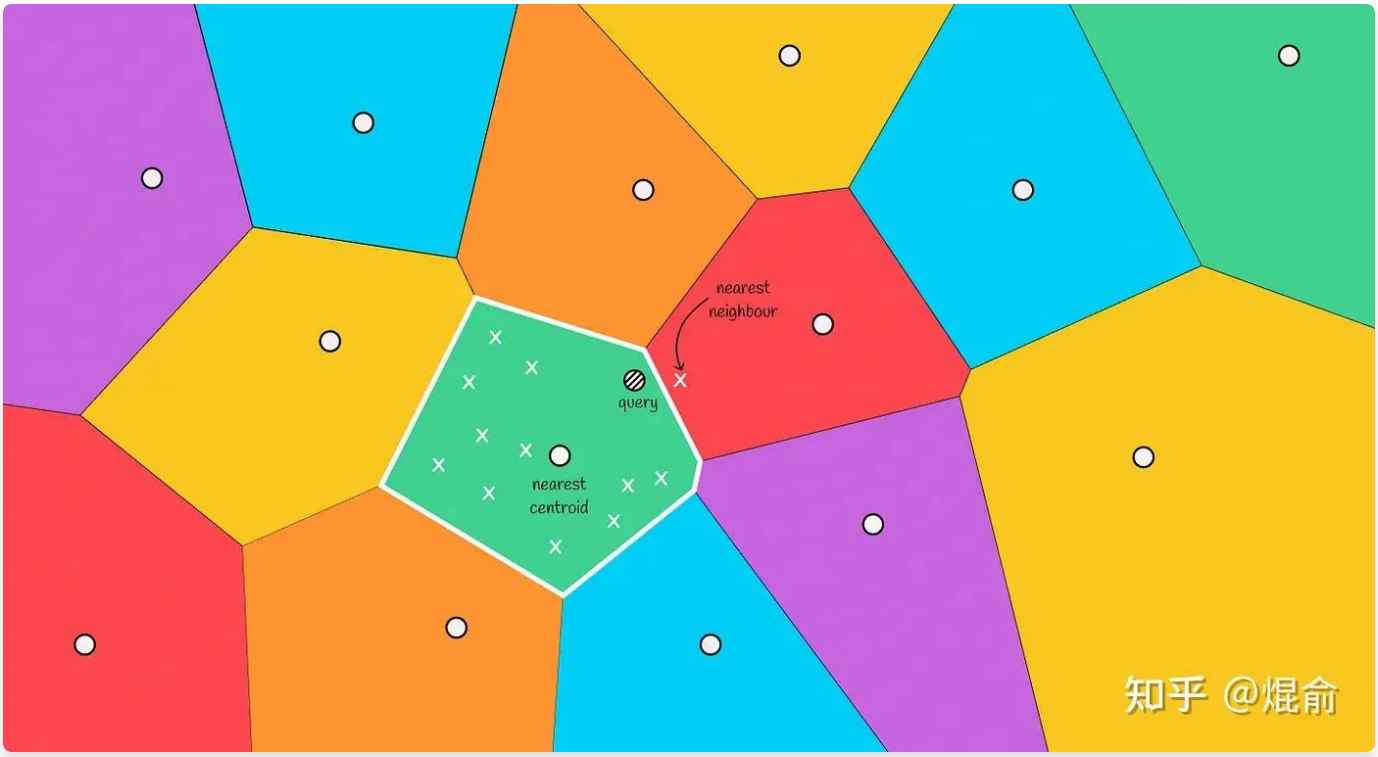
```python
dim, measure = 64, faiss.METRIC_L2
param = 'Flat'
index = faiss.index_factory(dim, param, measure)
index.is_trained                                    # 输出为True
index.add(xb)                                       # 向index中添加向量
```

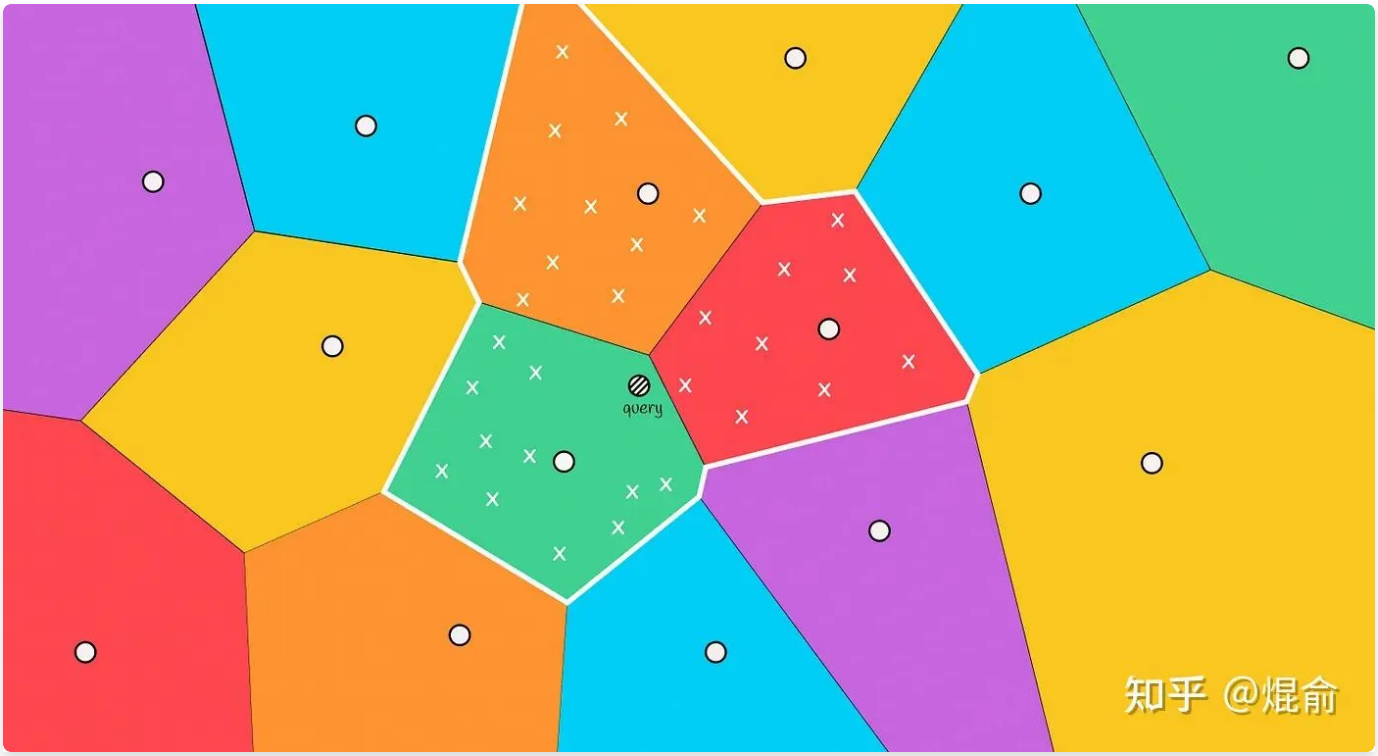
IVFx Flat：倒排文件索引

原理：将知识库构建为维诺图（聚类），query 与每个簇的质心计算相似度后，在最相似的簇中再逐一计算相似度。



可能会有边缘问题，如下图的 query 应与红色簇的 x 最相似，因此可以考虑将搜索范围扩展到前 k 个最相似的质心所在的簇。





使用

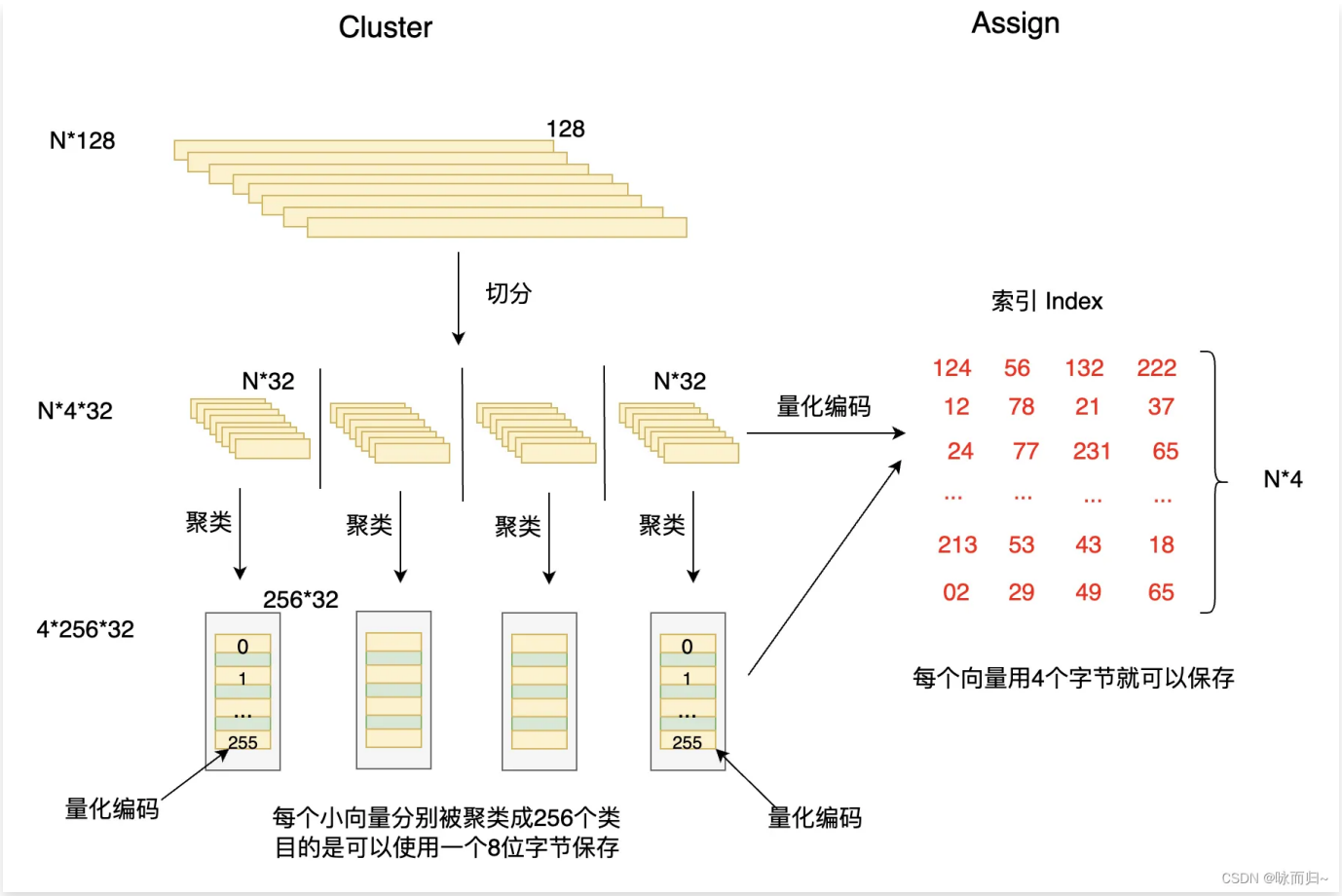
```python
dim, measure = 64, faiss.METRIC_L2
param = 'IVF100,Flat'                                    # 划分100个簇
index = faiss.index_factory(dim, param, measure)
print(index.is_trained)                                  # 此时输出为False，因为倒排索引需要训练k-means，
index.train(xb)                                          # 因此需要先训练index，再add向量
index.add(xb)
```

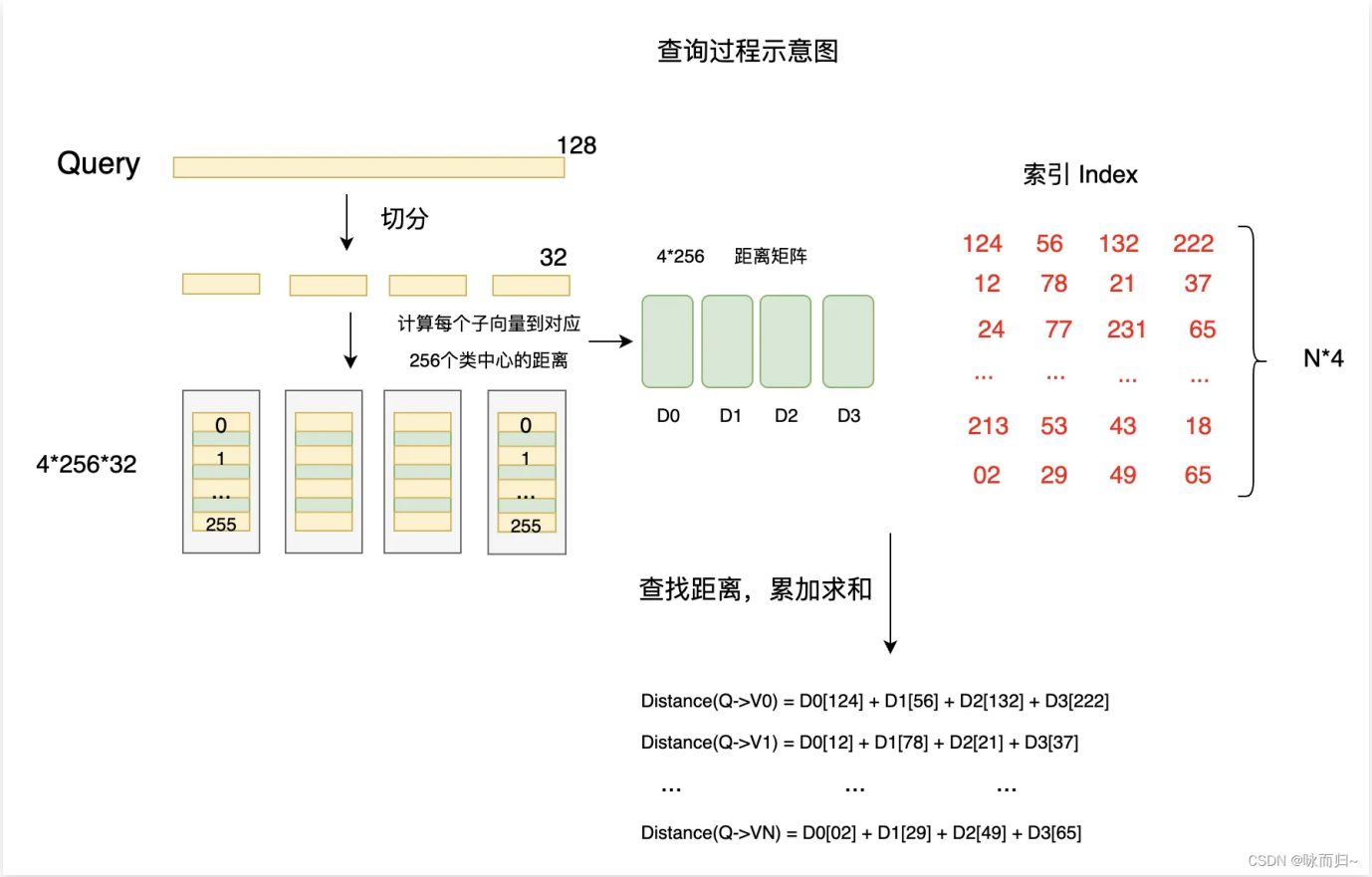
PQx：乘积量化

原因：高维向量需要分成更多的簇才能维持分类的质量。例如一个 128 维的向量，需要维护 2^64 个聚类中心才能维持不错的量化结果，但这样的码本存储大小已经超过维护原始向量的内存大小了。

方法：以 128=4*32 为例，一个 128 维向量分成 4 小组，每组 32 维。每一小组的 N 个 32 维子向量独自进行聚类，分为 256 类，$256=2^8$，也就是说每一个类别可以用 8bit=1 字节保存，那么每个32 维子向量就可以用 1 字节表示类别，一个 128 维向量就可以用 4 字节表示。

查询：将查询也编码为 4 字节即可。


Cluster
Assign
N*128
128
切分
索引 Index
N*32
N*32
N*4*32
量化编码
124 56 132 222
12 78 21 37
24 77 231 65
... ...
213 53 43 18
02 29 49 65
N*4
聚类
聚类
聚类
聚类
256*32
4*256*32
0
1
...
255
每个向量用4个字节就可以保存
量化编码
每个小向量分别被聚类成256个类
目的是可以使用一个8位字节保存
量化编码
CSDN @咏而归~


查询过程示意图
Query
128
切分
32
4*256 距离矩阵
索引 Index
计算每个子向量到对应
256个类中心的距离
124 56 132 222
12 78 21 37
24 77 231 65
... ...
213 53 43 18
02 29 49 65
N*4
4*256*32
0
1
...
255
D0 D1 D2 D3
查找距离，累加求和
Distance(Q->V0) = D0[124] + D1[56] + D2[132] + D3[222]
Distance(Q->V1) = D0[12] + D1[78] + D2[21] + D3[37]
... ...
Distance(Q->VN) = D0[02] + D1[29] + D2[49] + D3[65]
CSDN @咏而归~

优点：占用内存小

缺点：召回率相比暴力法下降

```python
dim, measure = 64, faiss.METRIC_L2
param =  'PQ16'                                    # 一个向量分为16个子向量
index = faiss.index_factory(dim, param, measure)
print(index.is_trained)                            # 此时输出为False，因为倒排索引需要训练k-means，
index.train(xb)                                    # 因此需要先训练index，再add向量
index.add(xb)
```

IVFxPQy 倒排乘积量化

最推荐使用，前两个的结合

```python
dim, measure = 64, faiss.METRIC_L2
param =  'IVF100,PQ16'
index = faiss.index_factory(dim, param, measure)
print(index.is_trained)                            # 此时输出为False，因为倒排索引需要训练k-means，
index.train(xb)                                    # 因此需要先训练index，再add向量 index.add(xb)
index.add(xb)
```

LSH：局部敏感哈希、HNSWx：分层可导航小世界

略，不常用

chromadb

faiss 向量数据库只能存于内存，不能持久化保存。且无法将向量与原文本段一一对应，需要手动管理。最后，不方便更新、删除

等操作。

chromadb：

https://docs.trychroma.com/docs/overview/introduction

基本使用

Python

```python
# pip install chromadb sentence-transformers

import chromadb
# 1.创建
# 初始化一个持久化的客户端，数据将存储在'my_chroma_db'目录下（数据库）
client = chromadb.PersistentClient(path="my_chroma_db")

# 创建一个名为"rag_series_demo"的集合，如果该集合已存在，get_or_create_collection会直接获取它（表）
collection = client.get_or_create_collection(name="rag_series_demo")

# 2.添加数据
# 文本段
documents_to_add = [
    "RAG的核心思想是检索增强生成。",
    "FAISS是Facebook开源的向量检索库。",
    "ChromaDB是一个对开发者友好的向量数据库。",
    "今天天气真不错，适合出去玩。",
]

# 附加的描述
metadatas_to_add = [
    {"source": "doc1", "type": "tech"},
    {"source": "doc2", "type": "tech"},
    {"source": "doc3", "type": "tech"},
    {"source": "doc4", "type": "daily"},
]

# 每个文本段的id
ids_to_add = ["id1", "id2", "id3", "id4"]

# 只需一个add命令，ChromaDB会自动处理：
# 调用默认的Embedding模型将documents转换为向量（我们也可以指定自己的模型）
# 存储向量、文档原文、元数据和ID
collection.add(
    documents=documents_to_add,
    metadatas=metadatas_to_add,
    ids=ids_to_add
)

# 3.查询
# 定义查询
query_texts = ["什么是向量数据库？"]

# 执行查询
```

```python
results = collection.query(
    query_texts=query_texts,
    n_results=2  # 我们想找最相关的2个结果
)

# 打印结果
import json
print(json.dumps(results, indent=2, ensure_ascii=False))

# 4.根据附加描述筛选，例如只在tech搜索
results_filtered = collection.query(
    query_texts=["什么是向量数据库？"],
    n_results=2,
    where={"type": "tech"}
)
```

多路召回

假设在检索阶段，使用了 bm25 和 embedding，他们各自召回了 topK 个文本段，需要进行融合

算法：倒数排名融合（Reciprocal Rank Fusion, RRF）

公式：对于每个列表，计算 `1 / (k + rank)`，rank 是文档在该列表中的排名（从1开始），k 是一个常数（通常设为60），用于降低低排名结果的影响（可以理解为，排名越靠前，分母越小，RRF 越大）

举例：例如一个文本段，在 bm25 的检索结果中，排名第 2，在 embedding 的相似度排名第 3，那么

$$RRF = \frac{1}{(2+60)} + \frac{1}{(3+60)}$$

对所有文本段的 RRF 再次降序，筛选 topK 即可

reranker

在实际应用中，最相似 ≠ 最相关，因此有些文本段可能与查询相似，但却不足够相关，因此需要对检索结果再次排序

原理：前面用于Embedding的，叫Bi-Encoder（双塔编码器）。它将问题和文档分开编码成向量，再计算相似度。速度快，但无法捕捉两者之间深层的交互信息。而 rerank 使用 Cross-Encoder，则是将问题和文档拼接在一起（[CLS] 问题 [SEP] 文档 [SEP]）后，再输入给一个预训练模型。模型给出相关性判断。

常用 rerank 模型：bge-reranker-base

Python

```python
# pip install sentence-transformers

from sentence_transformers import CrossEncoder

# 加载一个预训练好的Cross-Encoder模型
reranker_model = CrossEncoder('bge-reranker-base')

# 模拟一个用户查询
query = "Mac电脑怎么安装Python? "

# 模拟从向量数据库召回的3个文档
# 注意它们的初始顺序
documents = [
    "在Windows上安装Python的步骤非常简单，首先访问Python官网...", # 最不相关
    "Python是一种强大的编程语言，适用于数据科学、Web开发和自动化。", # 有点相关，但不是教程
    "要在macOS上安装Python，推荐使用Homebrew。首先打开终端，输入命令 'brew install python' 即可。" # 最相关
]

# Re-ranker需要的是[查询, 文档]对的列表
sentence_pairs = [[query, doc] for doc in documents]

# 使用predict方法计算相关性分数
# （注意：它不是0-1之间的相似度，而是一个可以排序的任意分数值）
scores = reranker_model.predict(sentence_pairs)

print("原始文档顺序:", documents)
print("Re-ranker打分:", scores)
print("-" * 20)

# 将分数和文档打包并排序
scored_documents = sorted(zip(scores, documents), reverse=True)

print("精排后的文档顺序:")
for score, doc in scored_documents:
    print(f"分数: {score:.4f}\t文档: {doc}")
```

```
原始文档顺序: ['在Windows上安装Python的步骤非常简单，首先访问Python官网...', 'Python是一种强大的编程语言，适用于数据科学、Web开发和自动化。', "要在macOS上安装Python，推荐使用Homebrew。首先打开终端，输入命令 'brew install python' 即可。"]
Re-ranker打分: [-4.6853375  -1.370929   7.9545364]
--------------------
精排后的文档顺序:
分数: 7.9545      文档: 要在macOS上安装Python，推荐使用Homebrew。首先打开终端，输入命令 'brew install python' 即可。
分数: -1.3709     文档: Python是一种强大的编程语言，适用于数据科学、Web开发和自动化。
分数: -4.6853     文档: 在Windows上安装Python的步骤非常简单，首先访问Python官网...
```

查询优化

构造对话 prompt

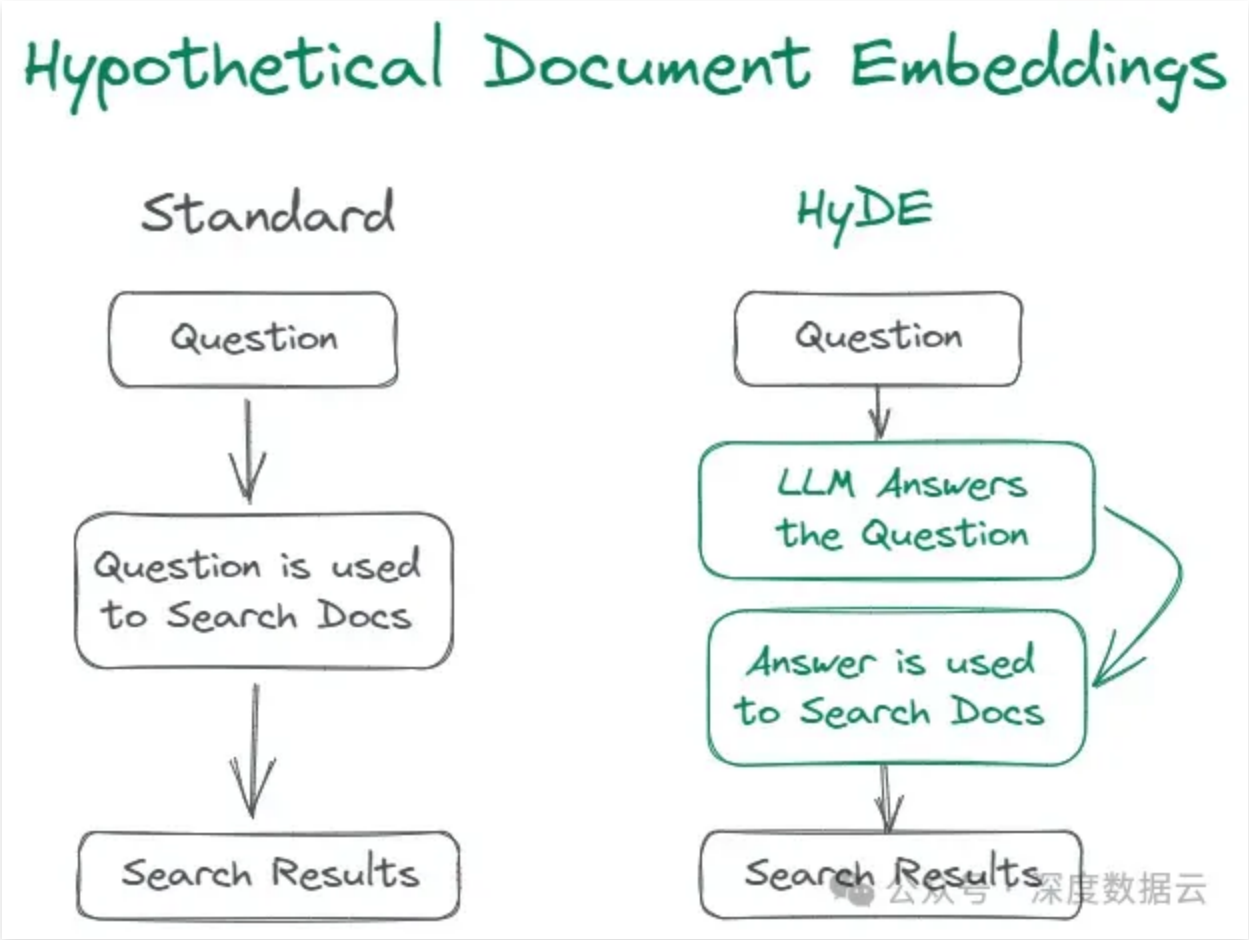
```
你是一个专业、严谨的问答助手。

请严格根据下面提供的“上下文”来回答用户的“问题”。
不要依赖你自己的任何先验知识。
如果“上下文”中没有足够的信息来回答“问题”，请直接回复“根据提供的资料，我无法回答您的问题。”。
绝不允许编造、杜撰答案。

---
上下文:
{context}
---
问题:
{question}
---

请根据以上规则，生成你的回答:
```

Hypothetical Document Embeddings

生成假设文档：给定一个查询，HyDE 使用一个指令遵循的语言模型（如 InstructGPT）生成一个假设文档。这个文档可能包含虚假信息，但它能够捕捉到与查询相关的模式。

编码与检索：生成的假设文档编码为嵌入向量，然后通过向量相似性在文档库中检索出最相关的真实文档。

（模拟一个答案替代原始查询）



评估

embedding

使用 recall 评估：topK 中真实相关的数量/所有相关的数量

reranker model

使用 MRR评估：Mean Reciprocal Rank, 平均倒数排名

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i}$$

其中：

- $|Q|$ 表示查询的总数。
- $rank_i$ 表示第 i 个查询中第一个相关项目的位置。

举例：例如有三次查询，最相关的在每次查询结果中分别排名1,2,3，则

$$MRR = \frac{1}{3} * (\frac{1}{1} + \frac{1}{2} + \frac{1}{3})$$

end to end

accuracy（LLM 基于 query、ground truth 以及生成的 answer 评估）

框架：Ragas

Python

```python
# pip install ragas openai "langchain-openai"

import os
from datasets import Dataset
from ragas import evaluate
from ragas.metrics import (
    faithfulness,
    answer_relevancy,
    context_recall,
    context_precision,
)
from langchain_openai import ChatOpenAI

# os.environ["OPENAI_API_KEY"] = "sk-..."

# --- 1. 准备评估数据集 ---
# ground_truth: 人类专家给出的标准答案
# answer: RAG系统生成的答案
# contexts: RAG系统召回的上下文
dataset_dict = {
    "question": ["macOS上怎么安装Python? "],
    "answer": ["要在macOS上安装Python，推荐使用Homebrew。首先打开终端，输入 'brew install python'。"],
    "contexts": [
        "要在macOS上安装Python，推荐使用Homebrew。首先打开终端，输入命令 'brew install python' 即可。",
        "Homebrew是macOS的包管理器。"
    ],
    "ground_truth": ["在macOS上安装Python，可以使用Homebrew包管理器，在终端执行命令 'brew install python'。"]
}
dataset = Dataset.from_dict(dataset_dict)

# --- 2. 运行评估 ---
llm = ChatOpenAI(model="gpt-4o-mini")

# 选择我们想评估的指标
metrics = [
    faithfulness,
    answer_relevancy,
    context_precision,
    context_recall,
]

# 执行评估
```

```python
result = evaluate(
    dataset=dataset,
    metrics=metrics,
    llm=llm
)

# --- 3. 查看结果 ---
print(result)
```

Python

```
{
  'faithfulness': 1.0,
  'answer_relevancy': 0.98,
  'context_precision': 1.0,
  'context_recall': 1.0
}
```

扩展–graphrag

目标：普通 RAG 很难回答 `该数据集的主题是什么` 这种high level的总结性问题

原理：

1. 由 LLM 针对每段文本建立知识图谱，并生成实体描述、关系描述。
2. 合并所有同名的实体，建立一个最终的知识图谱，由 LLM 为每个实体根据多个关系生成总结性的描述。
3. 使用莱顿社区检测，将知识图谱进行部分合并与抽象，形成高层知识图谱（例如，多个实体和关系可能被合并为一个更抽象的实体）。

4. 每一层都嵌入，并双向维护所有实体、关系与其来源的映射。
5. global search：从高层开始查询
6. local search：从底层开始查询