RAG

Retrieval Augmented Generation

Contents

1. Vector 데이터 베이스

- Sentence Encoder – 문장 단위 유사성 결정을 위한 Sentence Embedding 기법(2019)
- HNSW(Hierarchical Navigable Small World) 탐색 – 고차원 벡터 데이터 검색 효율성을 높이기 위한 색인 기법(2016)

2. RAG(Retrieval Augmented Generation) 시스템

- Sentence Splitter - LangChain
- Vector DB + LLM

1

벡터 데이터베이스 개요 (Vector DB)

벡터 데이터베이스

Vector DB란? 고차원 벡터(vector) 데이터를 저장하고, '의미적 유사성'을 기준으로 가장 가까운 데이터를 빠르고 효율적으로 검색(ANN: Approximate Nearest Neighbors)하는 데 특화된 데이터베이스

① 고차원 벡터 문장 혹은 단락을 하나의 고차원 벡터로 표현

- LLM에서 토큰을 고차원 벡터로 표현하고 next token을 예측하도록 학습
- Vector DB에서는 문장을 하나의 고차원 벡터로 표현하고 이들 간의 유사성을 학습

② ANN(Approximate Nearest Neighbors) 색인-검색을 효율적으로 하기 위한 근사 탐색

- 근사 탐색을 위한 HNSW(Hierarchical Navigable Small World) 알고리즘
- 고차원 벡터 DB에서 정확한 k-nearest neighbors를 찾기 위해서 DB안에 있는 모든 벡터들과 유사성(거리) 비교하는 대신에 계층적 색인 구조를 만들어 효율적-효과적 k-nearest neighbors를 찾는 알고리즘

기본 실습 ChromaDB

① 클라이언트 생성

```
client = chromadb.Client()
```

② 컬렉션 생성 (테이블 개념)

```
db = client.create_collection(name="my_docs")
```

③ 임베딩 모델 로드

```
model = SentenceTransformer('all-MiniLM-L6-v2’)
```

```
# - 한국어 포함 다국어 임베딩 모델
# - "paraphrase-multilingual-MiniLM-L12-v2" (다국어)
# - "all-MiniLM-L6-v2" (영문 위주, 가볍고 빠름)
```

ChromaDB 모델 예

특징	paraphrase-multilingual-MiniLM-L12-v2	all-MiniLM-L6-v2
언어 지원	다국어 (50+ 언어) (한국어, 영어, 일본어 등)	영어 (English) 전용
모델 구조	12-Layer Transformer (L12)	6-Layer Transformer (L6)
임베딩 차원	384	384
학습 방식	다국어 의역(paraphrase) 및 병렬 코퍼스(번역 쌍)	방대한 영어 NLI 및 STS 데이터셋 (All-NLI)
주요 용도	다국어 의미 검색, 교차 언어(cross-lingual) 검색	영어 의미 검색, 클러스터링 (속도가 빠름)

기본 실습 ChromaDB

④ 문서 추가

```
texts = ["AI is transforming education.",
         "Chroma is a vector database.",
         "Large language models use transformers."]
embeddings = model.encode(texts)


db.add(
    documents=texts,
    embeddings=embeddings.tolist(),
    ids=["1", "2", "3"]
)
```

기본 실습 ChromaDB

⑤ 검색

```
query = "What is Chroma?"
query_embedding = model.encode([query])


results = db.query(
    query_embeddings=query_embedding.tolist(),
    n_results=2
)
```

⑥ 검색 결과 출력

```
for key, value in results.items():
    print(f"{key}: {value}")
```

기본 실습 ChromaDB

- ids: [['2', '3']]
- embeddings: None
- documents: [['Chroma is a vector database.', 'Large language models use transformers.']]
- uris: None
- included: ['metadatas', 'documents', 'distances']
- data: None
- metadatas: [[None, None]]
- distances: [[0.5973263382911682, 1.9461071491241455]]

2

Sentence Encoder

Sentence-BERT: Sentence Embeddings(2019)

- Sentence BERT(SBERT): 문장과 문장 간의 의미적(semantic) 유사성을 측정할 수 있는 기법 제안
- 기존의 LM 모델 중의 하나인 구글의 BERT를 기반으로 3가지 학습 방법 제안

① Classification 모델: NLI(Natural Language Inference) fine-tuning 모델

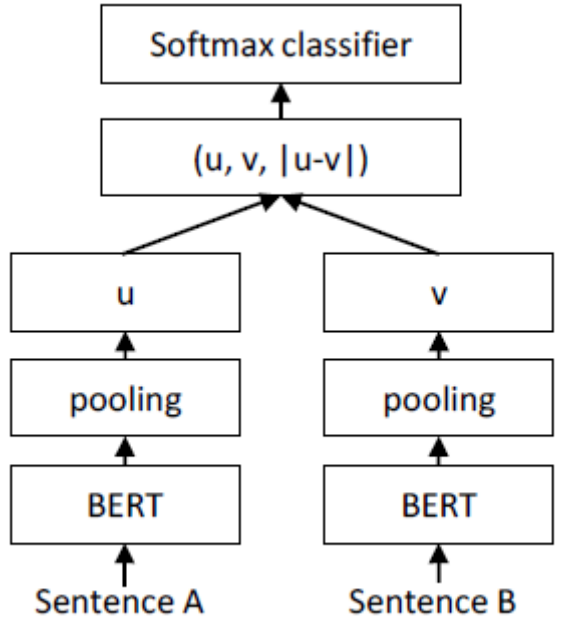
② Cosine-similarity 측정 모델: Semantic Textual Similarity (STS) 데이터 셋 학습 모델

③ Triplet 목적 함수 모델: Anchor 문장에 대하여 positive 문장과의 거리는 가깝고 negative 문장과의 거리는 멀도록 학습하는 모델

- SBERT는 다양한 목적에 사용될 수 있으며, Sentence Encoder(Embedding)을 위해서, ①번 학습을 한 후에, ②를 학습함
  - OpneAI Embedding은 정확하게 밝히고 있지는 않으나 ② 방식의 학습한 것으로 추정됨.
  - 혹은 CLIP 에서와 같이 Contrastive Learning을 할 수도 있음

① Classification – NLI Fine-Tuning

- Natural Language Inference 데이터 셋을 사용하여 학습



- 두 문장 (u, v)에 대하여 3가지 관계 중 하나의 관계를 선택하도록 학습

① entailment (포함 관계)

② contradiction (모순 관계)

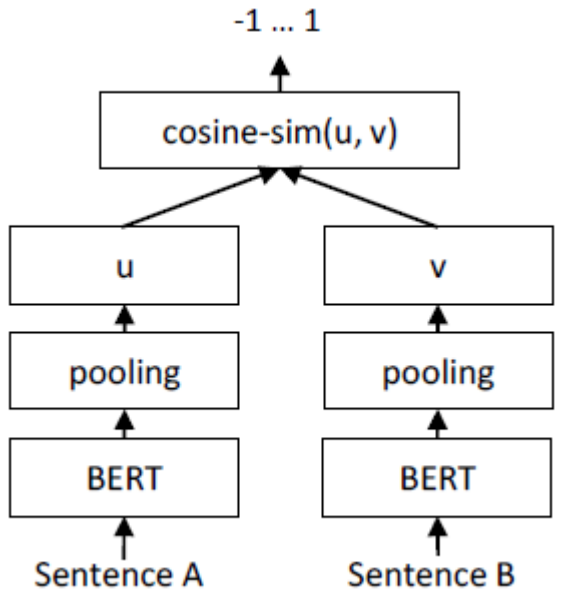
③ neutral (무관 관계)

- Sentence A와 B는 동일 구조를 갖는(Siamese) 네트워크에 입력됨
- Pooling: Average pooling을 사용하여 하나의 벡터 추출
- Softmax classifier: 3개의 입력 벡터, u, v, |u-v| 를 Linear Layer 통과 시킨 후에 Cross-Entropy Loss 함수 적용

② Cosine-Similarity 측정 모델

- Semantic Textual Similarity (STS) 데이터 세트 이용

- 각 문장 쌍에 사람이 매긴 유사도 점수(예: 0~5)를 regression 방식으로 학습
- $Loss = (similarity_{cosine}(u, v) - y)^2$, y: target similarity



③ Triplet 목적 함수 모델

- 문장 a에(anchor) 대하여, positive p, negative n 문장을 선택한다.
- 문장 a와 p의 거리가 문장 a와 n의 거리보다 가까워 지도록 학습한다.
- 수학적으로 다음 함수를 최소화 하도록 네트워크를 학습시킨다.
- $max(\left|\left|s_a - s_p\right|\right| - \left|\left|s_a - s_n\right|\right| + \epsilon, 0)$
- 여기서, s_a, s_p, s_n 는 sentence embedding을 의미하고 ||·||는 Euclidean거리를 의미하고, ϵ은 margin으로 실험에서 1로 셋팅 하였음.

3

HNSW - 색인·검색 (ANN)

Approximate Nearest Neighbors

HNSW: Hierarchical Navigable Small World

- Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs (2016년 arXiv, 2018년 TPAMI 온라인, 2020년 TPAMI 출판)

- Vector DB의 핵심 기술

① 효율적 효과적 Nearest Neighbors 검색-색인 – 고차원 벡터의 brute-force 방식 검색은 DB의 scale 제한.

② 트랜스포머 모델 기반 임베딩 – 의미론적 유사성에 대한 정확도

- Approximate Nearest Neighbors – 정확한 k-nearest neighbors 찾는 것 보다는 ground-truth에 근접한 k-nearest neighbors를 찾는 것이 응용에 실질적 적합
- HNSW – 기본 아이디어는 거의 모든 상용 벡터 DB에 적용되고 있음

① HNSW 개요

목표

- 고차원 벡터에서 근사 최근접 이웃(ANN) 을 매우 빠르고 정확하게 찾기 위한 그래프 인덱스

핵심 아이디어

- 데이터 전체 위에 여러 층(level)의 근접 그래프를 쌓음
- 맨 위층은 노드가 아주 적고, 아래로 내려갈수록 노드가 많아짐
- 질의 시 위층에선 탐욕적(greedy)으로 방향만 잡고, 바닥층(0층)에선 폭을 넓혀 탐색하여 정확도를 올림

구성요소와 파라미터

- level(층): 각 점은 무작위로 정해진 최대 층수까지 등장(지수분포; 스킵리스트 느낌)
- entry_point: 최상층 시작 노드(동적으로 갱신)
- M: 각 층에서 허용되는 최대 연결 수(차수). 보통 16
- M_max0: 0층에서의 최대 차수(보통 2*M)
- efConstruction: 삽입 시 후보를 수집하는 폭(기본 100~200). 클수록 품질↑, 구축시간/메모리↑
- efsearch(ef): 질의 시 0층 탐색 폭. 클수록 재현율↑, 시간 ↑

① HNSW 개요

복잡도(경험치)검색 평균 시간

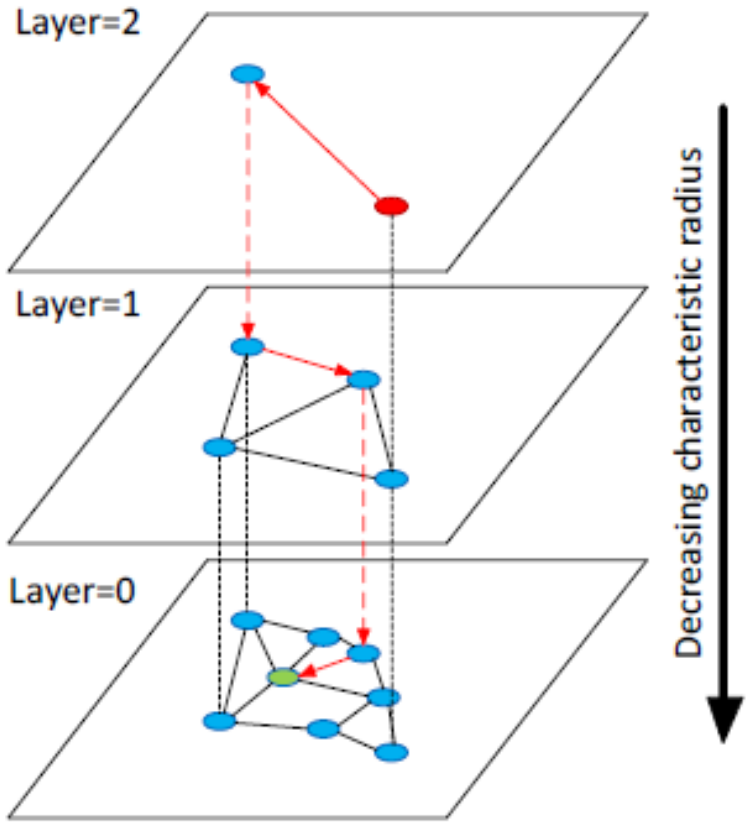
- 대략 로그 수준 + 작은 상수배

메모리

- 저장점당 평균 링크 수가 상수라 O(N)

정확도 vs 속도

- efsearch가 주된 요소



② Insert 알고리즘 – 전체 흐름

- 한 점 p는 여러 레벨에 들어 간다. → 어떤 레벨에 들어갈 지는 지수 분포 확률로 샘플링

1. 층수 샘플

- 새 점 p에 대해 최대 층 L_p를 지수분포로 샘플.
- 인덱스의 현재 최상층 L_max보다 크면 entry_point와 L_max 갱신.

2. L_max 층부터 L_p + 1 층까지는 탐욕 탐색(Greedy search)

- 각 층에서 efsearch=1 수준의 순수 탐욕(greedy)
- 현재 진입 노드에서 시작하여, 더 가까운 이웃이 없을 때까지 이웃으로만 이동
- 그 층에서의 최종 위치를 다음 아래층의 시작점으로 사용

3. L_p 층부터 0층까지 폭 넓은 탐색을 수행(실제 점 p가 삽입되는 층)

- 시작점: 직전 단계에서 얻은 "그 층의 진입점"
- efConstruction 만큼 후보를 모으는 best-first 탐색
- 후보 모음에서 select_Neighbors(다양성/품질 휴리스틱) 로 최대 M개(0층은 M_max0) 이웃을 선택
- p와 그 이웃들을 양방향으로 연결
- 이웃 노드의 차수가 초과하면 그 노드에서도 같은 휴리스틱으로 가지치기(pruning) 하여 차수 제한 유지.

4. entry_point 갱신

- L_p > L_max 였다면 entry_point = p, L_max = L_p

② Insert 알고리즘 – 핵심 자료구조 및 휴리스틱

- **핵심 자료구조/변수**
  - C(candidates): 아직 확장할 정점들의 우선순위 큐(질의점까지의 거리 오름차순)
  - W(topCandidates): 현재까지 발견한 좋은 정점들의 우선순위 큐(거리 내림차순, 최대 크기 = efConstruction)
  - visited: 재방문 방지 집합
- **select_Neighbors 휴리스틱 – 가까움 + 다양성을 동시에 확보하려는 가지치기(pruning)**
  - 가장 가까운 M개를 연결하면 편향성, 지역 클러스터를 형성하여 근사 탐색 성능 저하를 가져올 수 있다. 이를 개선하고자 가까움과 다양성을 동시에 고려하여 M개의 이웃들을 유지하는 알고리즘

<알고리즘(p, W, M)>

① W에 저장된 후보 리스트를 p와의 거리로 정렬

② 거리가 가까운 순으로 후보 점을 선택하되 이미 선택된 리스트의 점과의 거리가 p와의 거리보다 가까우면 제외시킴

③ 선택된 리스트의 크기가 M이 되면 결과를 반환

- 상층에서는 M(보통 16)으로 저층(0)에서는 M_max0 = 2 * M 정도로 더 많은 연결 허용

② Insert 알고리즘 – Pseudo 코드

```
INSERT(p):
  L_p ← sample_level()           # 지수분포 - 대부분 저층 0에 분포하도록 설계
  if L_p > L_max:
      L_max ← L_p
      entry_point ← p

  ep ← entry_point
  # 1) 윗층들: 탐욕 검색 (ef=1)
  for ℓ from L_max down to L_p+1:
      ep ← GREEDY_SEARCH_AT_LEVEL(query=p, entry=ep, level=ℓ)

  # 2) 포함될 각 층에서: 폭 넓혀 후보 수집 후 연결
  for ℓ from L_p down to 0:
      W ← SEARCH_LAYER(query=p, entry=ep, level=ℓ, ef=efConstruction)
      N ← SELECT_NEIGHBORS(W, M_at_level(ℓ))    # 휴리스틱
      for v in N:
          CONNECT_BIDIRECTIONAL(p, v, level=ℓ)  # 양방향 연결이기 때문에 v의 이웃수가 M을 초과할 수 있다.
          PRUNE_IF_DEGREE_EXCEEDS(v, level=ℓ)   # 동일 휴리스틱으로 가지치기
      ep ← ARGMIN_{x∈W} dist(p, x)              # 다음 아래층 entry로
```

③ 검색 알고리즘 – 전체 흐름

1. 위층: 탐욕 검색(ef=1)

- entry point에서 시작, 해당 층에서 이웃 중 질의점에 더 가까운 노드가 있을 때까지 이동
- 더 이상 개선이 없으면 한 층 내려감
- 최종적으로 0층에 도달하면 좋은 출발점을 확보하게 됨

2. 0층: 폭을 넓힌 best-first 탐색(ef)

- 0층에서 ef를 사용해 후보(C) 와 우수후보집합(W) 을 유지하며 확장
- 중단 조건: C가 비거나, min(C)의 거리가 W에서 가장 먼 거리보다 커질 때(더 좋은 후보가 안 나옴)
- 최종 반환: W에서 거리순 상위 k개를 정렬해 반환(대개 ef ≥ k로 설정해 품질 확보).

• 파라미터 ef

- 검색 속도와 정확도 관계 조절
- 추천 값: ef = max(32, 2~4 * k)

③ 검색 알고리즘 – Pseudo 코드

```
QUERY(q, k, ef):
  ep ← entry point
  # 1) 윗층: 탐욕
  for ℓ from L_max down to 1:
      ep ← GREEDY_SEARCH_AT_LEVEL(query=q, entry=ep, level=ℓ)

  # 2) 0층: 폭 넓힌 best-first
  W ← {ep}                          # max-heap by distance(q, ·), size ≤ ef
  C ← {ep}                          # min-heap by distance(q, ·), to expand
  visited ← {ep}
```

③ 검색 알고리즘 – Pseudo 코드

```
while C not empty:
      c ← pop_min(C)                    # q에 가장 가까운 후보

      if dist(q, c) > worst_distance(W) and size(W) ≥ ef:
          break

      for u in neighbors(c, level=0):
          if u in visited: continue
          visited.add(u)
          d ← dist(q, u)
          if size(W) < ef or d < worst_distance(W):
              push(C, u)
              push(W, u)
              if size(W) > ef: pop_worst(W)

 R ← top_k_from(W, k)                   # 거리 오름차순으로 정렬
 return R
```

Parameters in HNSW

파라미터	의미	영향
M	각 층에서 한 노드가 가질 수 있는 **최대 연결(링크) 수**	• 그래프의 연결 밀도를 결정 • M이 클수록 정확도↑, 메모리/구축시간↑ • 일반적으로 16 사용
M_max0	0층(가장 아래층)에서의 최대 연결 수	• 0층은 실제 탐색이 이루어지는 층이라 연결을 더 많이 둠 • 보통 M_max0 = 2 * M (예: M=16 → M_max0=32)
efConstruction	**삽입 시 탐색 폭**. 후보 이웃을 얼마나 넓게 수집할지	• 크면 더 좋은 그래프(정확도↑), 구축 시간↑ • 보통 **100~200** 사이
efsearch (또는 ef)	검색 시 탐색 폭	• 클수록 재현율(Recall)↑, 속도(QPS)↓ • 보통 **32~200**, k의 2~4배 정도로 설정
Distance metric	벡터 거리 측정 방식 (L2, Inner Product, Cosine 등)	• 데이터 특성에 따라 선택
Max_elements	전체 인덱스에 저장 가능한 최대 노드 수	• 미리 크기를 지정해 메모리 관리