서강대학교
SOGANG UNIVERSITY

LLM_(6)

벡터 데이터베이스로 확장하기: RAG 구현하기

서강대학교 수학과 김건휘

목차

- 벡터 데이터 베이스
- 벡터 데이터 베이스 검색_KNN
- 벡터 데이터 베이스 검색_ANN_IVF
- 벡터 데이터 베이스 검색_ANN_HNSW
- 실습 (Code)

벡터 데이터베이스

- 벡터 데이터베이스란?
- 벡터 데이터베이스 : 벡터 임베딩을 키로 사용하는 데이터베이스
  (*벡터 임베딩 : 데이터의 의미를 담은 숫자 벡터)

- 데이터베이스 활용법
  - 데이터 수집 -> 임베딩 모델로 벡터 변환 ->
    벡터 데이터베이스 저장 -> **벡터 검색** 가능
  - **벡터 검색**
    - 임베딩 벡터의 거리 계산을 통해, 유사한 데이터를 검색할 수 있다.

결론 : 벡터 데이터베이스는 **벡터 사이의 거리 계산에 특화**된 데이터베이스

벡터 데이터베이스

- 벡터 데이터베이스의 개발 목적

- 임베딩 공간 특징 :
  데이터 특징을 추출하여, 이를 **숫자**로 표현한 **임베딩**을 공간 상에 배치
  ➔ 특징이 비슷한 데이터는 가깝게, 다른 데이터는 멀리 위치
  ➔ 임베딩 벡터 사이의 거리를 계산하여 비슷한 데이터를 찾을 수 있다.

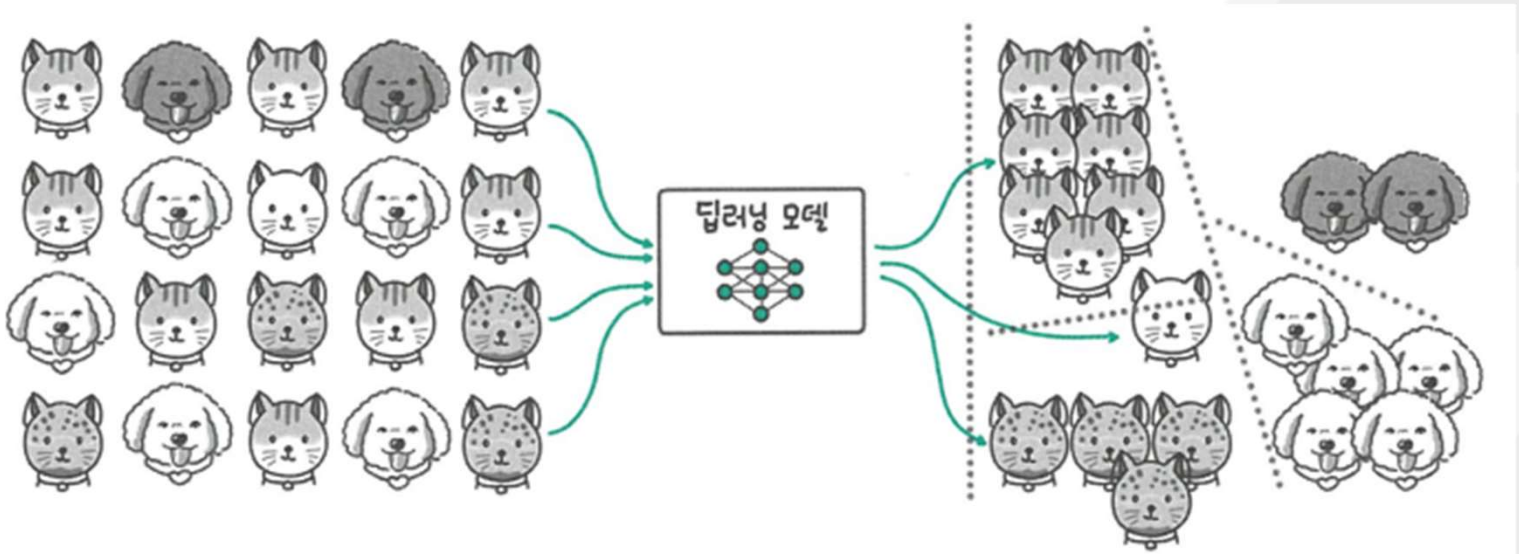


그림 12.4 표현 학습을 통한 임베딩 벡터 생성

벡터 데이터베이스는 위와 같은 **임베딩 벡터의 특징**을 이용하기 위한 목적으로 개발 되었다.

벡터 데이터베이스

- 벡터 데이터베이스 활용 단계
  1. **저장** :
     저장할 데이터를 임베딩 모델을 거쳐 벡터로 변환하고
     벡터 데이터베이스에 저장
  2. **검색** :
     검색할 데이터를 임베딩 모델을 거쳐 벡터로 변환하고
     벡터 데이터베이스에서 검색
  3. **결과 반환** :
     벡터 데이터베이스에서는 검색 쿼리의 임베딩과 거리가
     가까운 벡터를 찾아 반환한다.

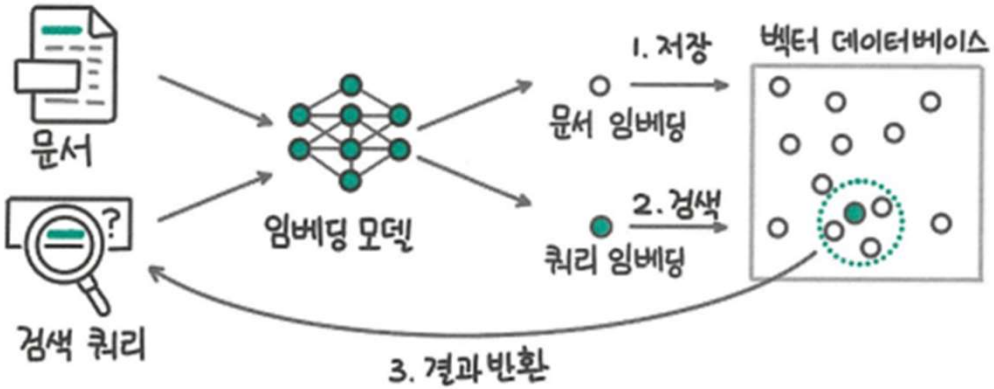


그림 12.5 벡터 데이터베이스 작동 과정

벡터 데이터베이스

- 벡터 데이터베이스 현황
  - 검색 방법 : 쿼리와 거리가 가장 가까운 벡터를 **검색**
  일반적으로 유클리드 거리, 코사인 유사도, 내적을 많이 활용

  - 딥러닝 기술의 폭넓은 활용에 따라,
    데이터의 특징을 추출한 임베딩을 활용하는 경우가 많아지고
    임베딩을 **저장하고 관리하는 기능**에 특화된 벡터 데이터베이스 등장
    EX) 이미지 **검색**을 제공하는 서비스
    사용자가 구매한 상품과 **유사한** 상품 **추천**하는 서비스 …

  - 2022년 말 Chat GPT출시 이후, 대부분의 데이터 베이스에 **검색 기능**이 추가 되고 있음
  이유 : 임베딩 벡터의 유사도를 기반으로 한 **문서 검색**을 하는 RAG가
  LLM의 환각 현상을 줄이고, 학습 없이도 최신 정보를 추가 가능

벡터 데이터베이스 검색

- 벡터 데이터베이스 작동 원리
  - 벡터 사이의 거리를 계산해서 유사한 벡터를 찾는다
    - **KNN** (K-Nearest Neighbor) :
      저장된 모든 임베딩 벡터를 조사해 가장 유사한 K개의 벡터를 반환하는 가장 기본적인 방법
    - **ANN** (Approximate Nearest Neighbor) :
      KNN의 한계를 극복하기 위해 사용되는 알고리즘으로, 정확성은 떨어지나 검색 속도가 매우 빠른 방법

벡터 데이터베이스 검색_KNN

- **KNN** (K-Nearest Neighbor)
  - 저장된 모든 임베딩 벡터를 조사해 가장 유사한 K개의 벡터를 반환하는 가장 기본적인 방법

  장점 : 모든 데이터를 조사하기 때문에 정확하다.
  단점 : 모든 벡터를 조사하기 때문에 연산량이 데이터 수에 비례하게 늘어나 속도가 느려진다
  ➔ 확장성이 떨어진다.

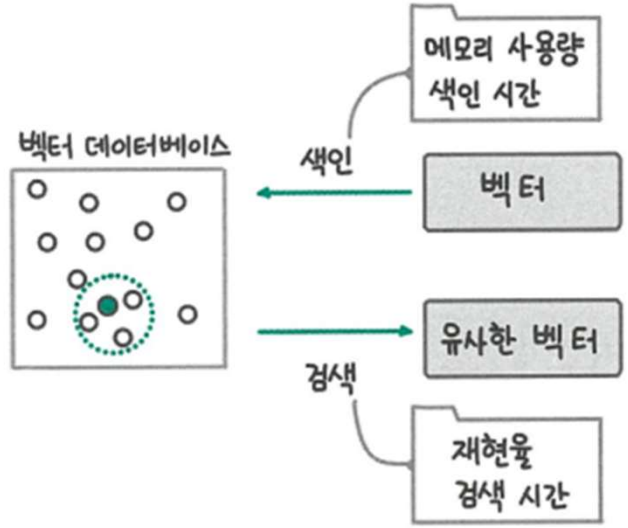


그림 12.8 색인과 검색 과정에서의 벡터 DB 성능

*색인 : 벡터를 저장하는 과정
*재현율 : 실제로 가장 가까운 K개의 정답 데이터 중 몇 개를 찾았는지

벡터 데이터베이스 검색_KNN

- 실습을 통해 알아보는 KNN의 한계
  예제 데이터 : 100만개의 128차원 임베딩 데이터, SIFT1M
  예제 :
  데이터를 20만개에서 100만개까지 점차적으로 늘리면서
  메모리 사용량, 색인 시간, 검색 시간이 어떻게 변하는 지 확인한다.
  추가로, 차원이 커질 수록 검색속도가 매우 커진다

```
k=1
d = xq.shape[1]
nq = 1000
xq = xq[:nq]

for i in range(1, 10, 2):
    start_memory = get_memory_usage_mb()
    start_indexing = time.time()
    index = faiss.IndexFlatL2(d)
    index.add(xb[:(i+1)*100000])
    end_indexing = time.time()
    end_memory = get_memory_usage_mb()

    t0 = time.time()
    D, I = index.search(xq, k)
    t1 = time.time()
    print(f"데이터 {(i+1)*100000}개:")
    print(f"색인: {(end_indexing - start_indexing) * 1000 :.3f} ms ({end_memory - start_memory:.3f} MB)
    검색: {(t1 - t0) * 1000 / nq :.3f} ms")
```

표 12.1 데이터양에 따른 검색 지표 확인

데이터 수	색인 시간(ms)	메모리 사용량(MB)	검색 시간(ms)
200,000	83.861	97.859	1.467
400,000	173.872	195.382	2.576
600,000	335.873	293.003	8.445
800,000	355.941	390.616	4.523
1,000,000	446.439	488.284	7.617

벡터 데이터베이스 검색_ANN

- **ANN** (Approximate Nearest Neighbor)
  - KNN의 검색 속도를 극복하기 위해 사용되는 알고리즘으로, 정확성은 떨어지나 검색 속도가 매우 빠른 방법

  장점 : 매우 빠른 검색 속도
  단점 : 모든 벡터를 조사하는 것이 아니기 때문에, 정확도는 약간 떨어진다.
  의의 : 실제 사례에서는 데이터 셋의 크기가 크고 고차원의 데이터를 다루는 경우가 많다.

  따라서, ANN은 임베딩 벡터를 빠르게 탐색할 수 있는 구조를 띈다

벡터 데이터베이스 검색_ANN_IVF

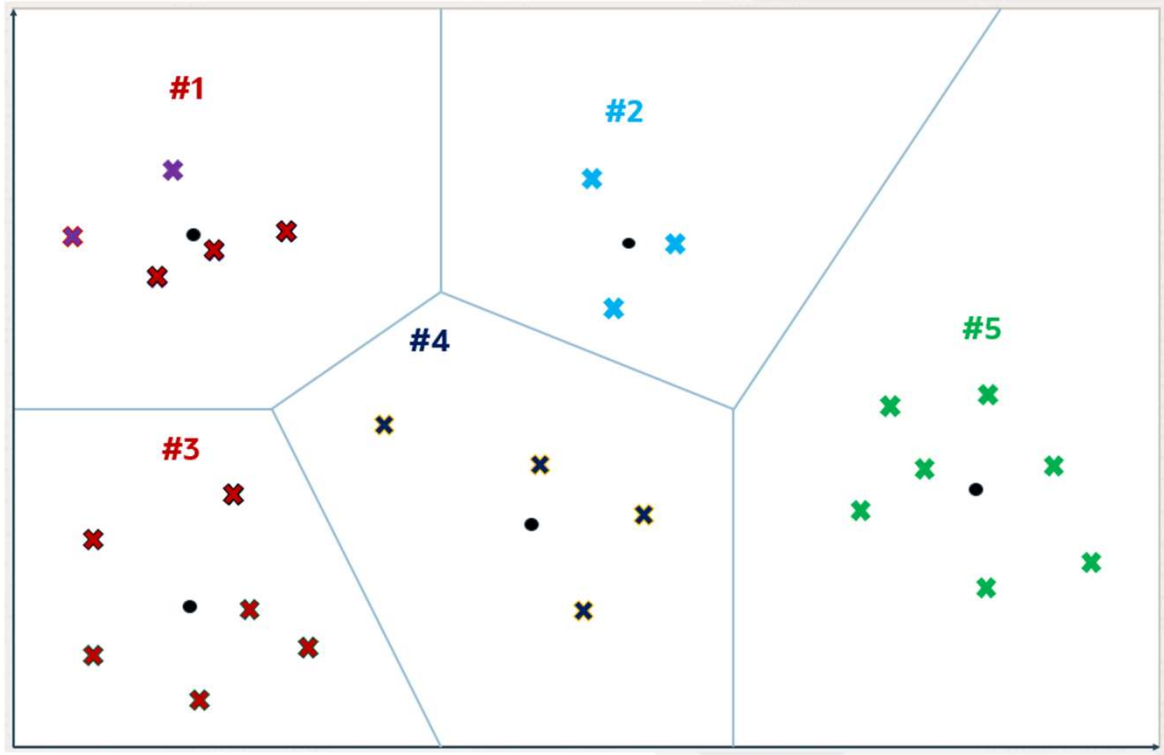
서강대학교 SOGANG UNIVERSITY

- **IVF** (Inverted File Index)
  - 검색 공간을 제한하기 위해 데이터셋 벡터들을 클러스터로 그룹화 하는 ANN 알고리즘

1 단계

N개의 데이터를 K개의 클러스터링

*일반적으로, K은 N의 제곱근으로 설정

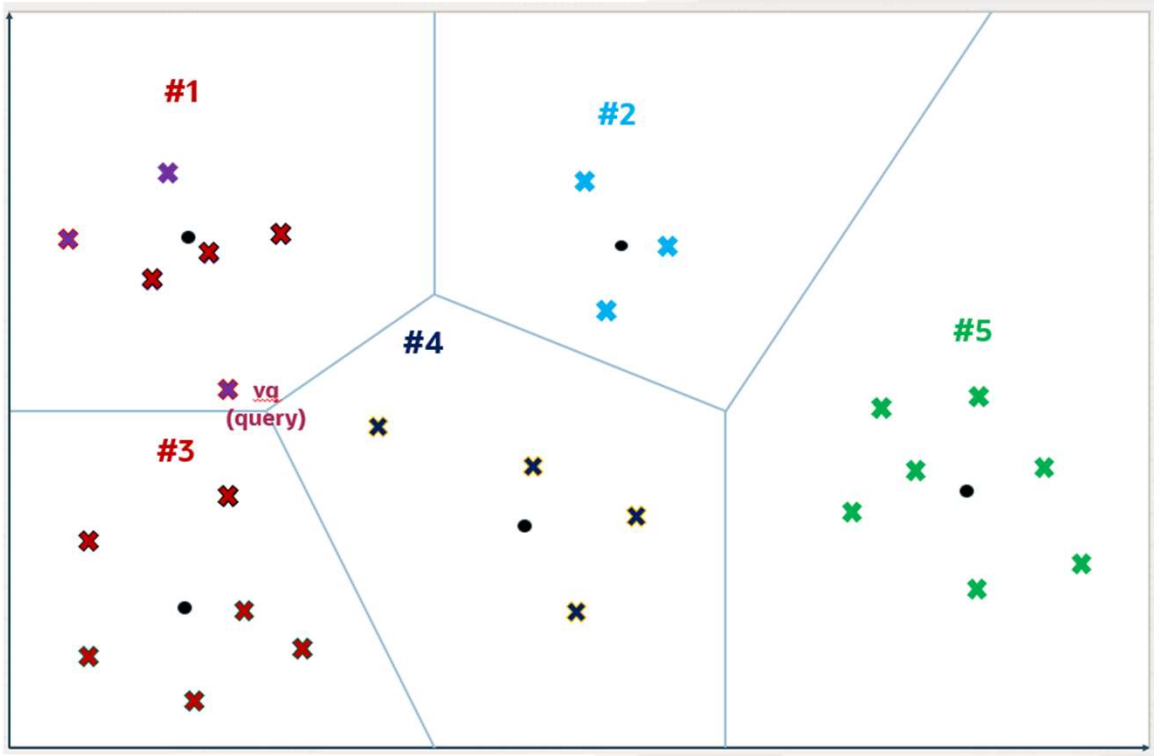


벡터 데이터베이스 검색_ANN_IVF

- **IVF** (Inverted File Index)
  - 검색 공간을 제한하기 위해 데이터셋 벡터들을 클러스터로 그룹화 하는 ANN 알고리즘

2 단계

쿼리가 들어오면, vq 벡터로 임베딩



벡터 데이터베이스 검색_ANN_IVF

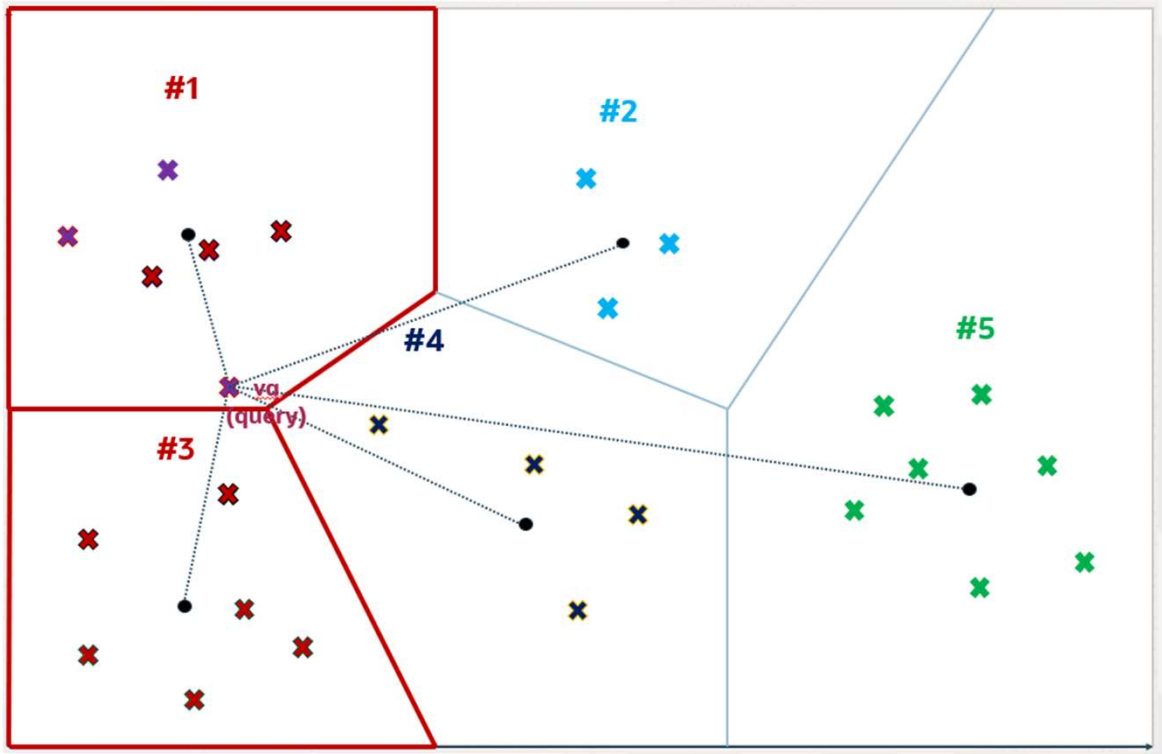
서강대학교 SOGANG UNIVERSITY

- **IVF** (Inverted File Index)
  - 검색 공간을 제한하기 위해 데이터셋 벡터들을 클러스터로 그룹화 하는 ANN 알고리즘

3 단계

Vq 벡터와 가장 가까운 클러스터를
클러스터 중심점 기준으로 i개 검색

*기본적으로 i는 k의 제곱근으로 설정



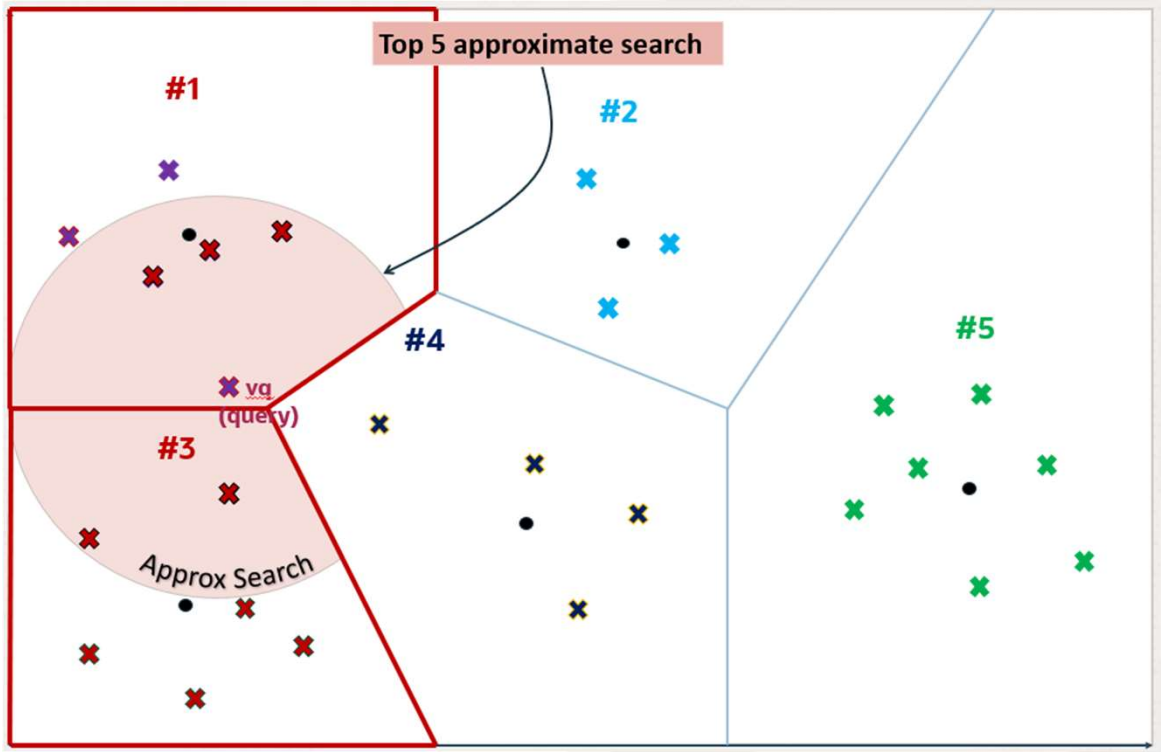
벡터 데이터베이스 검색_ANN_IVF

- **IVF** (Inverted File Index)
  - 검색 공간을 제한하기 위해 데이터셋 벡터들을 클러스터로 그룹화 하는 ANN 알고리즘

4 단계

I개의 클러스터의 데이터는 모두 거리 계산 하여, 가장 가까운 벡터를 검색

(여기서는 5개)



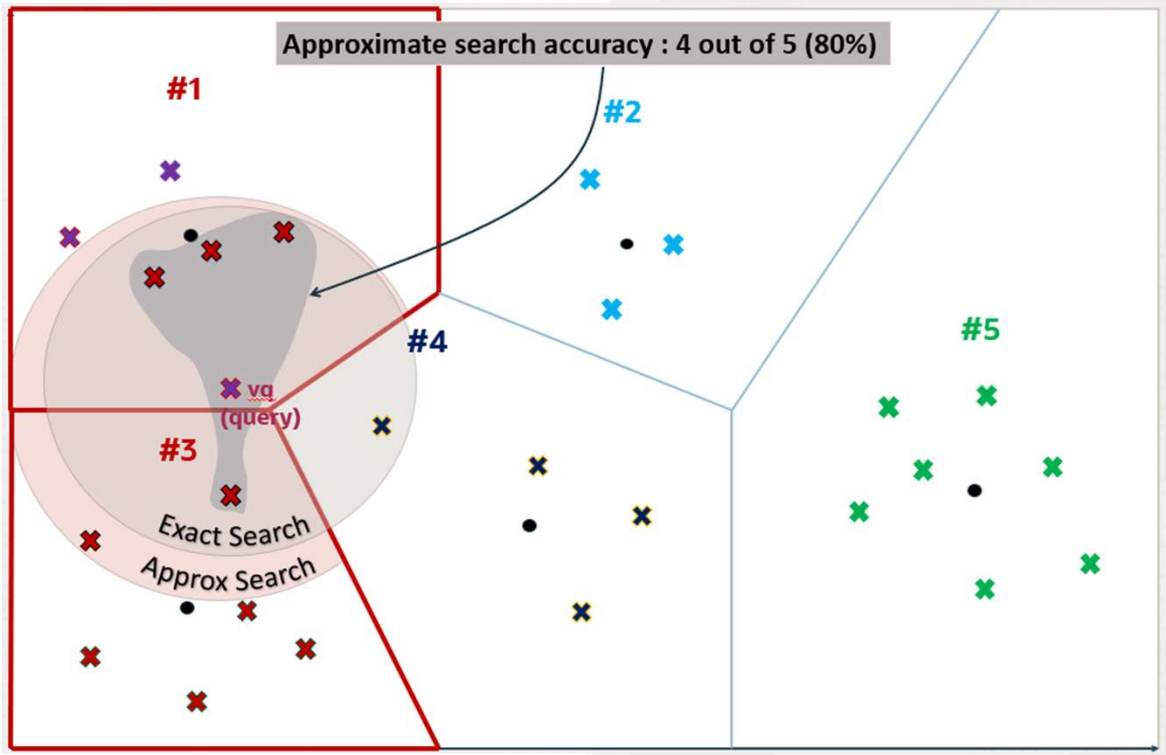
벡터 데이터베이스 검색_ANN_IVF

- **IVF** (Inverted File Index)
  - 검색 공간을 제한하기 위해 데이터셋 벡터들을 클러스터로 그룹화 하는 ANN 알고리즘

결과

KNN으로 검색했을 때는 100퍼센트 정확도 비교해보면 재현율 80퍼센트

(여기서는 5개 중 4개 정답)



벡터 데이터베이스 검색_HNSW

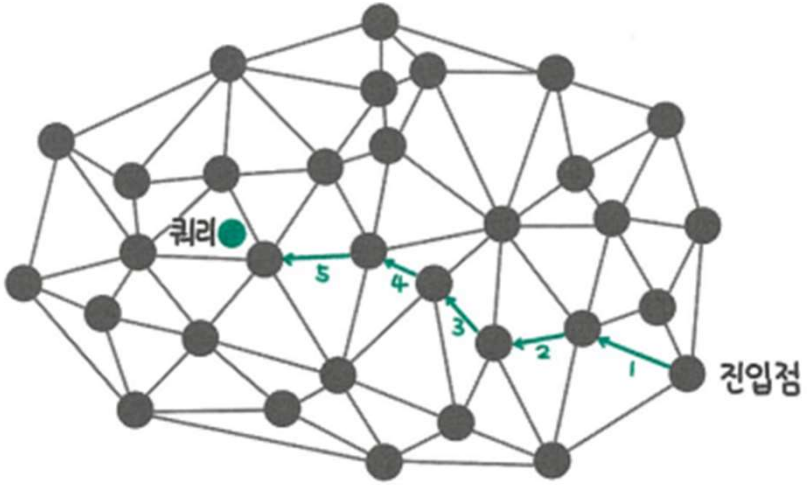
- **HNSW** (Hierarchical Navigable Small Worlds)
  - 네트워크 이론 중 하나인 6단계 법칙의 적용으로,
  그래프가 Regular한 상태에서 Random한 Edge를 추가하여
  한 Node에서 다른 Node까지 평균적으로 도달하기 위한 총 Edge수를
  크게 감소 시키는 그래프 구조

그림 12.10 랜덤한 그래프(a)와 규칙적인 그래프(c) 사이의 작은 세계(b)
(출처: https://dl.acm.org/doi/fullHtml/10.1145/3611450.3611467)

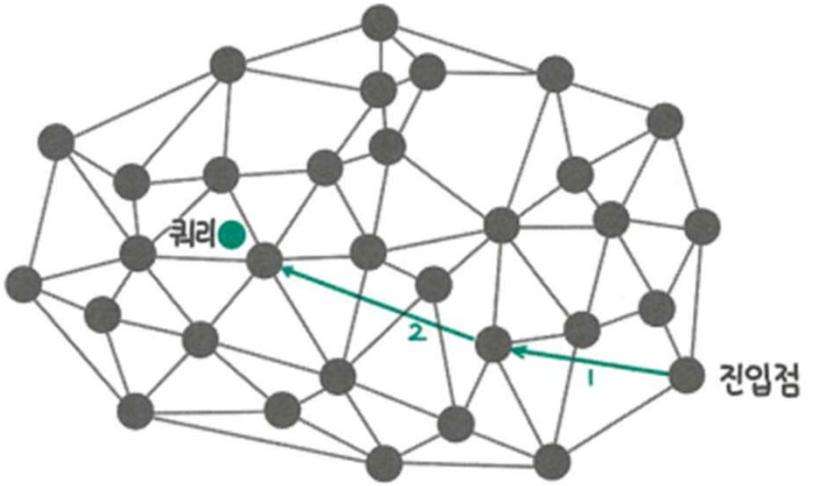
벡터 데이터베이스 검색_HNSW

- **HNSW** (Hierarchical Navigable Small Worlds)
  - 네트워크 이론 중 하나인 6단계 법칙의 적용으로,
  그래프가 Regular한 상태에서 Random한 Edge를 추가하여
  한 Node에서 다른 Node까지 평균적으로 도달하기 위한 총 Edge수를
  크게 감소 시키는 그래프 구조

Regular 상태에서의 탐색



Random Edge 추가한 상태에서의 탐색

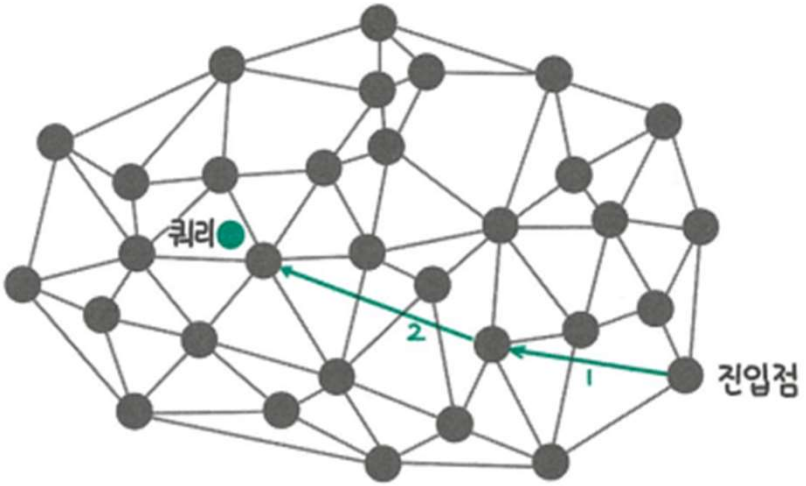


벡터 데이터베이스 검색_HNSW

- **HNSW** (Hierarchical Navigable Small Worlds)
  - 기존 NSW에서의 한계점 : Local Minimum
  이유 : 검색은 entry 부터 시작하여, entry와 연결된 정점을
  query와 비교하여 query와 가장 가까운 정점으로 이동
  이동한 정점에서 다시 비교하여 가장 가까운 정점으로 이동

(Greedy Algorithm. 이 과정을 반복하다보면 어느 순간 현재 정점보다 쿼리벡터에 더 가까운 정점을 찾을 수 없게 되어 local minimum에 빠진다. 모든 루트를 검색하지 못하는 것)

Random Edge 추가한 상태에서의 탐색



Random Edge 추가한 상태에서의 문제점 _LOCAL MINIMUM

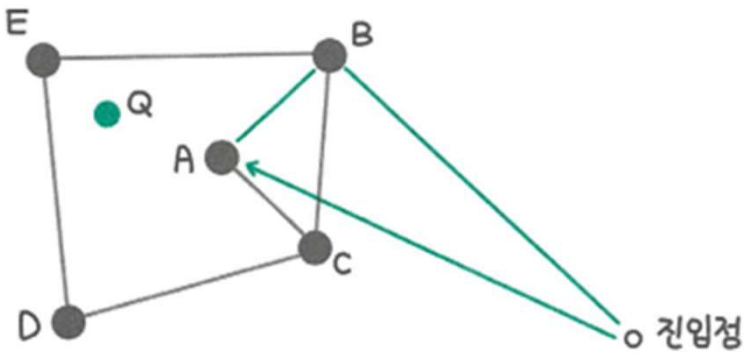


그림 12.13 작은 세계에서 지역 최솟값에 빠지는 문제

벡터 데이터베이스 검색_HNSW

- **HNSW** (Hierarchical Navigable Small Worlds)
  - 기존 NSW에서의 한계점 극복 방법 : Skip list 구조 도입

- Skip list 구조 도입

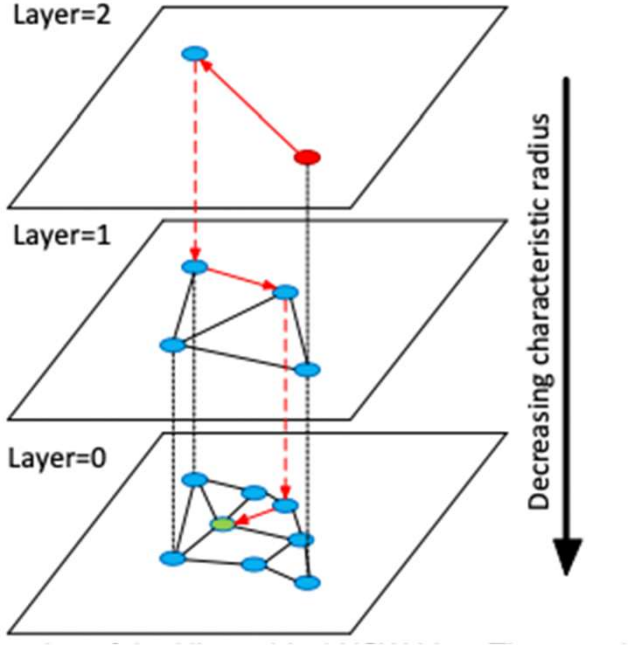
핵심 요약 : 91 찾기 위해, 기존 리스트는 8번 이동
반면에 Skip list는 3번 이동

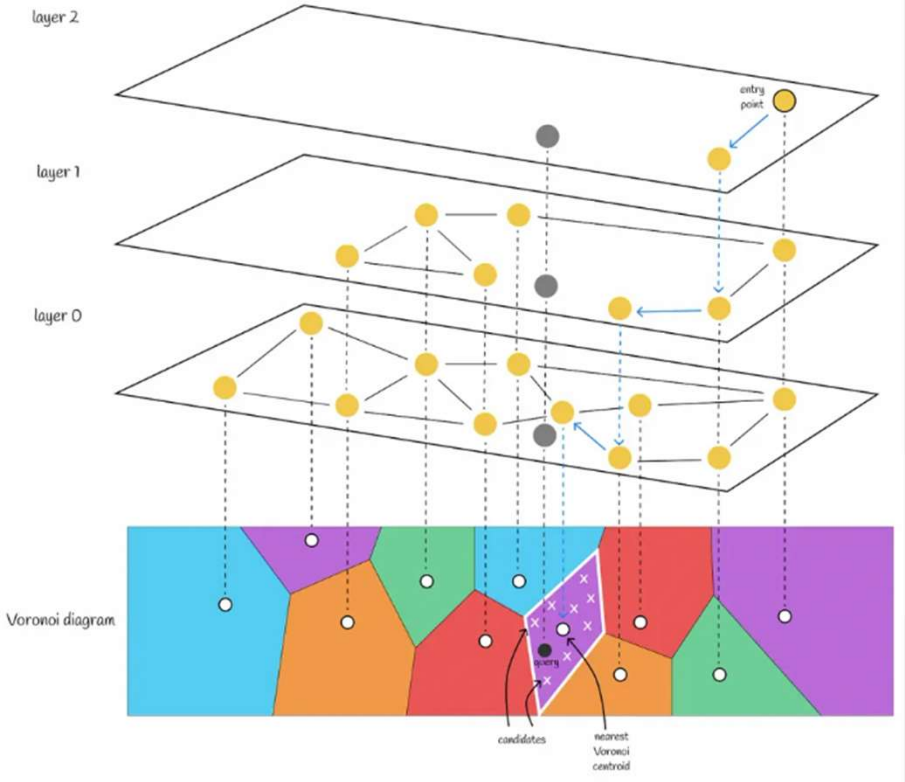
그림 12.15 스킵 리스트 작동 원리

벡터 데이터베이스 검색_HNSW

- **HNSW** (Hierarchical Navigable Small Worlds)
  - 기존 NSW에서의 한계점 극복 방법 : Skip list 구조 도입

핵심 요약 : NSW 를 계층화 하여, Edge를 길이에 따라 다른 계층으로 분류한다.
이를 통해 네트워크 크기에 어느정도 독립적으로 Edge를 평가할 수 있게 된다.
과정 요약 : 가장 긴 Edge를 가진 층인 최상위 계층에서 시작하여,Greedy 하게 local minimum 에 도달할 때까지 노드 순회.
도달하면 하위 계층으로 내려간다





벡터 데이터베이스 검색_HNSW

서강대학교 SOGANG UNIVERSITY

- **HNSW** (Hierarchical Navigable Small Worlds)
  - 파라미터 설명
    - M(Minimum) : 하나의 노드가 가지는 이웃 노드의 수
      - 값이 클수록 그래프가 촘촘해진다
      ➔ 메모리 공간 차지 많아짐, 검색 속도 느려짐, 검색 정확도 높아짐

    - ef_construction : 그래프 제작 시, 기준 노드로부터 고려하는 이웃 노드를 저장하는 큐의 크기(그래프 구성 시 참조하는 이웃의 범위)
      - 그래프 신뢰도에 영향을 미침
      값이 커질수록
      ➔ 검색 결과가 좋아진다 (검색 속도 & 검색 정확도)
      그래프 빌드 시간이 늘어난다. (단, 더 먼 노드에도 연결 가능)

    - ef_search : 검색 시, 탐색하는 이웃 노드 수
      - 값이 클수록 더 많은 이웃을 탐색한다
      ➔ 검색 시간 증가, 정확도 향상
      (꼼꼼하게 많은 책을 비교 선택 / 빠르게 둘러보고 비슷한 책 선택)

벡터 데이터베이스 검색_HNSW_실습

- 파라미터별 성능 비교

1. M의 변화에 따른 성능 정리

세팅 : (기본 설정_ construction : 40, search : 16)
기존 KNN을 사용했을 때는 7.6ms 검색속도 -> 64기준 30배 이상 빨라짐

```
k=1
d = xq.shape[1]
nq = 1000
xq = xq[:nq]

for m in [8, 16, 32, 64]:
    index = faiss.IndexHNSWFlat(d, m)
    time.sleep(3)
    start_memory = get_memory_usage_mb()
    start_index = time.time()
    index.add(xb)
    end_memory = get_memory_usage_mb()
    end_index = time.time()
    print(f"M: {m} - 색인 시간: {end_index - start_index} s, 메모리 사용량: {end_memory -
start_memory} MB")

    t0 = time.time()
    D, I = index.search(xq, k)
    t1 = time.time()

    recall_at_1 = np.equal(I, gt[:nq, :1]).sum() / float(nq)
    print(f"{(t1 - t0) * 1000.0 / nq:.3f} ms per query, R@1 {recall_at_1:.3f}")
```

표 12.2 파라미터 m의 변화에 따른 성능 정리 표

m	메모리 사용량(MB)	색인 시간(s)	검색 시간(ms)	재현율
8	553.8	155.1	0.07	0.692
16	625.4	182.8	0.09	0.792
32	737.4	351.6	0.164	0.905
64	981.0	456.7	0.237	0.932

벡터 데이터베이스 검색_HNSW_실습

- 파라미터별 성능 비교

2. ef_construction의 변화에 따른 성능 정리

40보다 80이 안좋아졌는데, 이는 그래프 생성 시 랜덤성이 들어가기 때문에 의도한 대로 항상 나오진 않는다

```
k=1
d = xq.shape[1]
nq = 1000
xq = xq[:nq]

for ef_construction in [40, 80, 160, 320]:
    index = faiss.IndexHNSWFlat(d, 32)
    index.hnsw.efConstruction = ef_construction
    time.sleep(3)
    start_memory = get_memory_usage_mb()
    start_index = time.time()
    index.add(xb)
    end_memory = get_memory_usage_mb()
    end_index = time.time()
    print(f"efConstruction: {ef_construction} - 색인 시간: {end_index - start_index} s, 메모리 사용
량: {end_memory - start_memory} MB")

    t0 = time.time()
    D, I = index.search(xq, k)
    t1 = time.time()

    recall_at_1 = np.equal(I, gt[:nq, :1]).sum() / float(nq)
    print(f"{(t1 - t0) * 1000.0 / nq:.3f} ms per query, R@1 {recall_at_1:.3f}")
```

표 12.3 ef_construction 변화에 따른 성능 결과

ef_construction	메모리 사용량(MB)	색인 시간(s)	검색 시간(ms)	재현율
40	740.0	349.8	0.165	0.894
80	736.3	425.8	0.129	0.866
160	736.3	861.2	0.147	0.904
320	736.6	1668.1	0.160	0.917

벡터 데이터베이스 검색_HNSW_실습

서강대학교 SOGANG UNIVERSITY

- 파라미터별 성능 비교

3. ef_search의 변화에 따른 성능 정리

기존 KNN을 사용했을 때는 7.6ms 검색속도 -> 32기준 10배 이상 빨라짐

```
for ef_search in [16, 32, 64, 128]:
    index.hnsw.efSearch = ef_search
    t0 = time.time()
    D, I = index.search(xq, k)
    t1 = time.time()

    recall_at_1 = np.equal(I, gt[:nq, :1]).sum() / float(nq)
    print(f"{(t1 - t0) * 1000.0 / nq:.3f} ms per query, R@1 {recall_at_1:.3f}")
```

표 12.4 ef_search 변경에 따른 성능 결과

ef_search	검색 시간(ms)	재현율
16	0.188	0.917
32	0.286	0.971
64	0.471	0.986
128	0.883	0.992

실습 코드 첨부

https://drive.google.com/file/d/1cODRlUGqor755BbZL0dCRD75BcYPBrOe/view?usp=sharing

정리

- 벡터 데이터베이스 :

벡터 데이터베이스는 **벡터 사이의 거리 계산에 특화**된 데이터베이스

- 벡터 데이터베이스 검색_KNN :

KNN은 모든 데이터들과 거리 계산하며 가까운 K개 선택

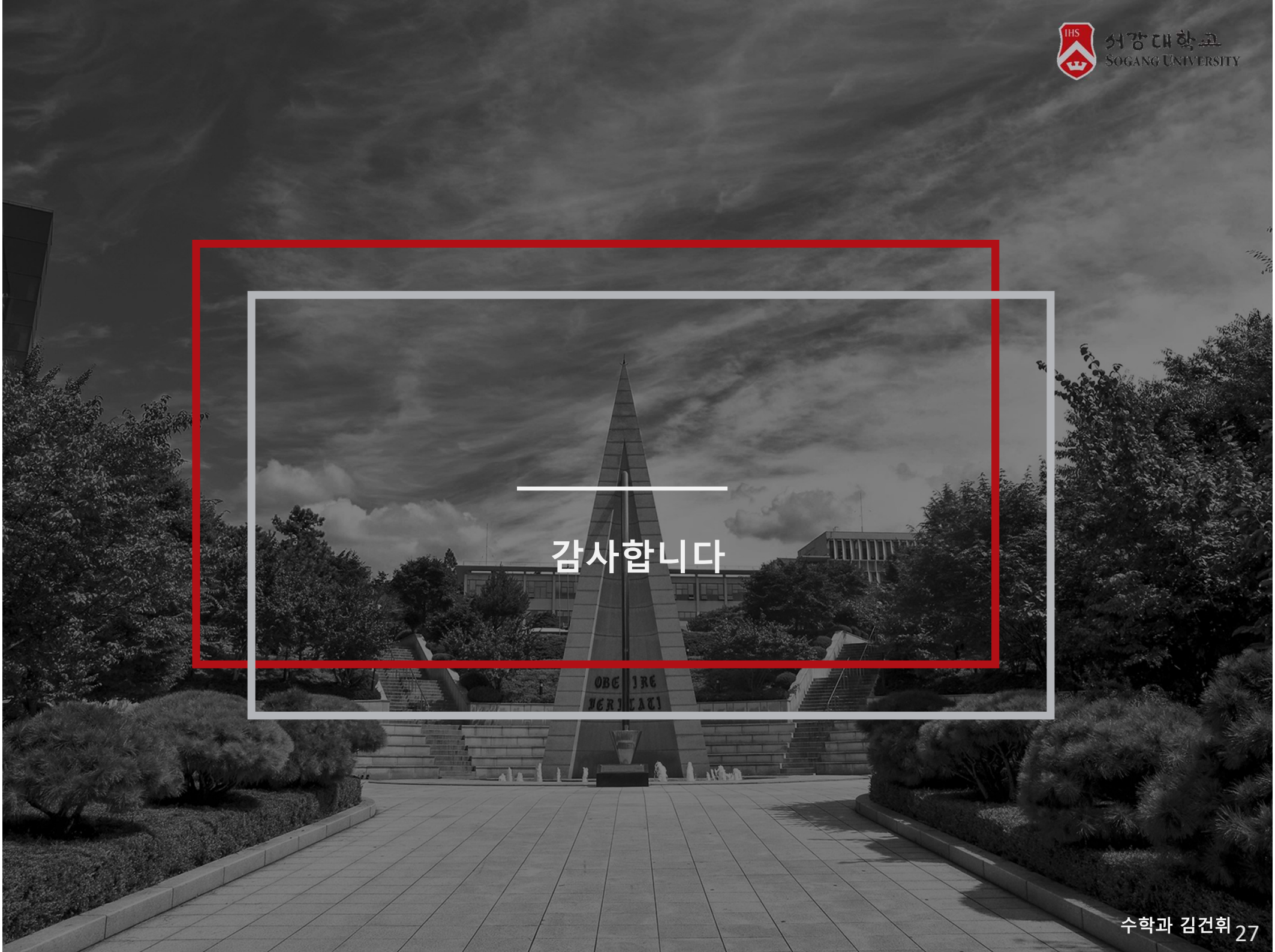
- 벡터 데이터베이스 검색_ANN_IVF

ANN은 KNN의 검색속도, 확장성 한계점 해결
IVF는 가까운 클러스터 찾고, K개 선택

- 벡터 데이터베이스 검색_ANN_HNSW

HNSW는 6단계 네트워크 이론 NSW + SKIP LIST 구조로,
빠르고 정확하게 가까운 벡터 찾는 알고리즘

- 벡터 데이터베이스인 파인콘을 사용하여 원본 이미지와 비슷한 이미지를 생성하는 실습 코드 첨부


감사합니다