임베딩 모델로 데이터 의미 압축하기

개요

9장에서 한 것

→ RAG, LLM 캐시, 데이터 검증, 데이터 로깅

→ 상업용 임베딩 모델인 OpenAI의 text-embedding-ada-002를 사용하기

10장에서 할 것

→ 텍스트를 숫자로 바꾸려던 다양한 시도 살펴보기

→ Sentence-Transformers 사용해 보기

→ 의미 검색, 키워드 검색, 하이브리드 검색

10.1 텍스트 임베딩 이해하기

임베딩 Embedding : 텍스트를 저차원의 임베딩 벡터로 변환하는 것

왜 해야 하는가? 단어나 문장 사이의 관계를 계산할 수 있기 때문

```
from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity

smodel = SentenceTransformer('snunlp/KR-SBERT-V40K-klueNLI-
augSTS') # 데이터 받아 오기
dense_embeddings = smodel.encode(['학교', '공부', '운동']) #
세 단어에 대해 임베딩 진행하기
cosine_similarity(dense_embeddings) # 단어간 유사도 계산하기

# array([[1.        , 0.59507453, 0.32537565],
#        [0.59507453, 1.0000001 , 0.54595685],
```

```
#        [0.32537565, 0.54595685, 1.0000002 ]], dtype=float
32)
```

원핫 인코딩

- 머신 러닝 분류 태스크에서 정답 데이터가 단어로 이루어져 있을 때 이를 숫자로 바꾸기 위해 사용
- 임베딩이 아니라 인코딩
- 단어의 개수만큼 차원을 두고, 자신의 차원을 1로, 나머지를 0으로 두는 방식

```
학교 = [1, 0, 0]
공부 = [0, 1, 0]
운동 = [0, 0, 1]
```

- 단점
  - 단어의 개수만큼 차원이 많아짐 → 공간 낭비
  - 단어간의 관계를 나타낼 수 없음 why? 코사인 유사도 값이 항상 0이기 때문

$$\text{cosine\_similarity}(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

백오브워즈

- 비슷한 단어가 많이 나오면 비슷한 문장 또는 문서라는 가정에 입각
- 특정 단어의 빈도 수로 그 문장이나 문서의 의미를 예측하는 방식
- 문제점
  - 한국어의 경우 은,는,이,가 와 같은 조사가 항상 많이 등장해서 이들의 빈도가 높아짐

- 여러 문서에 공통적인 단어가 등장하면 문서의 의미를 예측하기 어려워짐

TF-IDF

- Term Frequency-Inverse Document Frequency
- 백오브워즈의 단점을 극복하기 위해 등장
- 특정 단어가 여러 문서에 걸쳐서 등장하는 경우, 그에 대한 중요도를 줄이는 방식

$$\text{TF-IDF}(w) = \text{특정 단어 w가 등장한 횟수} \times \log \frac{\text{문서의 개수 N}}{\text{특정 단어 w가 등장한 문서의 수}}$$

- 단점
  - 단어의 의미를 살릴 수 있으나 여전히 차원의 문제를 해결하지 못함
  - 문서가 많거나 단어가 많을수록 공간의 낭비가 심해짐

→ 여기까지의 방식을 희소 임베딩 (Sparse Embedding) 이라고 함.

워드투벡 Word2Vec

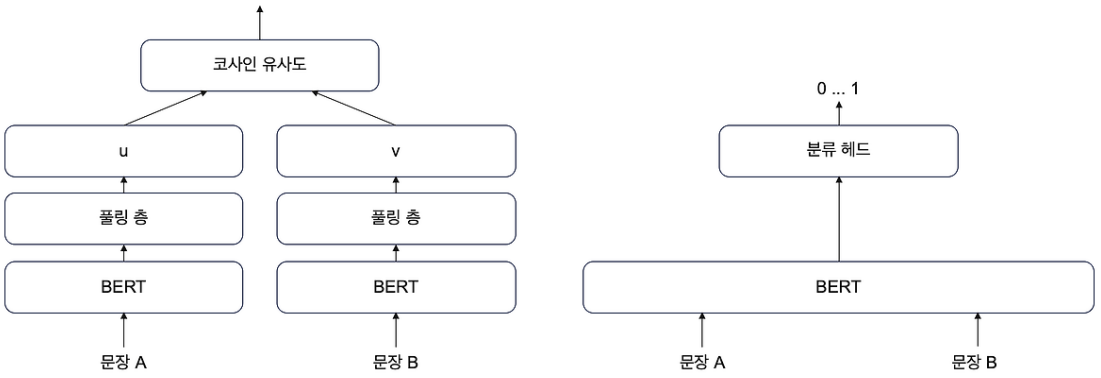
- 밀집 임베딩 (Dense Embedding) 등장
- 토큰의 개수와 관계 없이 벡터의 차원을 100에서 1000 사이로 둘 수 있음
- 방식
  - CBoW: 빈칸에 들어갈 토큰을 예측하는 학습
  - 스킵그램: 토큰 근처에 있을 다른 토큰을 예측하는 학습

10.2 문장 임베딩 방식

- 10.1 에서 다뤘던 것은 단어, 토큰과 같이 작은 단위를 임베딩하는 방식
- 하지만 우리는 단어가 아닌, 문장을 임베딩해야 함
- Why? 우리가 하려는 것은 문장으로 요청 → 문장과 관련된 정보 검색이기 때문
- 단어로 하면 문장 전체의 의미를 담지 못한다! 다른 의미로 쓰일 수 있는 단어의 문맥 파악도 불가하다!

문장 임베딩의 방식 → 그냥 트랜스포머 인코더 설명을 다시 보면 됨 (2장)

바이 인코더, 교차 인코더



바이 인코더

- 문장 각각을 BERT에 넣고 나온 임베딩 벡터끼리의 유사도를 비교
- 효율적임. 미리 문장별 임베딩 벡터를 구해 놓으면 코사인 유사도만 계산해주면 됨. 확장이 편함
- 상호작용 고려의 어려움

교차 인코더

- 문장 두개를 잇고 BERT에 넣어 나온 값을 분류 헤드에 넣어 0 ~ 1 사이의 값으로 변환

- 새로운 문장이 들어올 때마다 모든 관계를 연산해야 하는 비효율성
- 상호작용 고려는 확실함

```
from sentence_transformers import SentenceTransformer, models
# 사용할 BERT 모델
word_embedding_model = models.Transformer('klue/roberta-base')
# 풀링 층 차원 입력하기
pooling_model = models.Pooling(word_embedding_model.get_word_embedding_dimension())
# 두 모듈 결합하기
model = SentenceTransformer(modules=[word_embedding_model, pooling_model])

# SentenceTransformer(
#   (0): Transformer({'max_seq_length': 512, 'do_lower_case': False}) with Transformer model: RobertaModel
#   (1): Pooling({'word_embedding_dimension': 768, 'pooling_mode_cls_token': False, 'pooling_mode_mean_tokens': True, 'pooling_mode_max_tokens': False, 'pooling_mode_mean_sqrt_len_tokens': False, 'pooling_mode_weightedmean_tokens': False, 'pooling_mode_lasttoken': False, 'include_prompt': True})
# )
```

풀링 모드

- BERT 가 내뱉은 각 토큰별 임베딩 벡터를 조합하여 문장 전체를 대표하는 하나의 벡터를 생성하는 것
- [batch_size, seq_length, hidden_size]
- 문장의 개수, 문장 내 토큰 개수, 벡터의 차원

- 종류
  - CLS
    - 원래 BERT는 CLS 토큰에 문장 전체의 정보를 요약하도록 학습함
    - 문장이 너무 길면 정확도 떨어짐
  - Mean
    - 모든 토큰의 벡터를 평균 내기
    - 패딩 토큰의 정보 왜곡 가능성
  - Max
    - 각 차원별 최댓값을 선택하여 벡터를 생성
    - 사소한 정보를 잃을 수 있음 why? 너무 특정 정보만 가져오려 하기 때문

```
# 평균 모드
def mean_pooling(model_output, attention_mask):
    token_embeddings = model_output[0]
    input_mask_expanded = attention_mask.unsqueeze(-1).expa
nd(token_embeddings.size()).float()
    sum_embeddings = torch.sum(token_embeddings * input_mas
k_expanded, 1)
    sum_mask = torch.clamp(input_mask_expanded.sum(1), min=
1e-9)
    return sum_embeddings / sum_mask

# 최대 모드
def max_pooling(model_output, attention_mask):
    token_embeddings = model_output[0]
    input_mask_expanded = attention_mask.unsqueeze(-1).expa
nd(token_embeddings.size()).float()
    token_embeddings[input_mask_expanded == 0] = -1e9
    return torch.max(token_embeddings, 1)[0]
```

실습

```
from sentence_transformers import SentenceTransformer, util

model = SentenceTransformer('snunlp/KR-SBERT-V40K-klueNLI-augSTS')

embs = model.encode(['잠이 안 옵니다',
                     '졸음이 옵니다',
                     '기차가 옵니다'])

cos_scores = util.cos_sim(embs, embs)
print(cos_scores)

#   warnings.warn(
# tensor([[1.0000, 0.6410, 0.1887],
#         [0.6410, 1.0000, 0.2730],
#         [0.1887, 0.2730, 1.0000]])
```

실습 2 멀티모달

```
from PIL import Image
from sentence_transformers import SentenceTransformer, util

model = SentenceTransformer('clip-ViT-B-32')

img_embs = model.encode([Image.open('dog.jpg'), Image.open('cat.jpg')])
text_embs = model.encode(['A dog on grass', 'Brown cat on yellow background'])

cos_scores = util.cos_sim(img_embs, text_embs)
print(cos_scores)
```

```
# tensor([[0.2771, 0.1509],
#         [0.2071, 0.3180]])
```

10.3 의미 검색 구현하기

- 의미 검색: 의미를 고려한 검색 (코사인 유사도, 유클리드 거리 등 활용)

```
from datasets import load_dataset
from sentence_transformers import SentenceTransformer

klue_mrc_dataset = load_dataset('klue', 'mrc', split='trai
n') # 데이터셋 불러오기
sentence_model = SentenceTransformer('snunlp/KR-SBERT-V40K-
klueNLI-augSTS') # 모델 불러오기
```

```
klue_mrc_dataset = klue_mrc_dataset.train_test_split(train_
size=1000, shuffle=False)['train']
embeddings = sentence_model.encode(klue_mrc_dataset['contex
t'])
embeddings.shape
# 출력 결과
# (1000, 768)
```

- 데이터셋에서 1000개의 데이터 가져오기
- 데이터셋 context를 임베딩하기
- 결과: 임베딩 벡터의 차원

```
import faiss
# 인덱스 만들기
index = faiss.IndexFlatL2(embeddings.shape[1])
# 인덱스에 임베딩 저장하기
index.add(embeddings)
```

- 768 차원을 가진 인덱스 (데이터를 저장할 테이블) 생성
- 아까 가져온 embeddings 를 인덱스에 추가

```
query = "이번 연도에는 언제 비가 많이 올까?"
query_embedding = sentence_model.encode([query])
distances, indices = index.search(query_embedding, 3)

for idx in indices[0]:
  print(klue_mrc_dataset['context'][idx][:50])

# 출력 결과
# 올여름 장마가 17일 제주도에서 시작됐다. 서울 등 중부지방은 예년보다 사
나흘 정도 늦은   (정답)
# 연구 결과에 따르면, 오리너구리의 눈은 대부분의 포유류보다는 어류인 칠성
장어나 먹장어, 그 (오답)
# 연구 결과에 따르면, 오리너구리의 눈은 대부분의 포유류보다는 어류인 칠성
장어나 먹장어, 그 (오답)
```

- 쿼리를 선언하고,
- 쿼리도 임베딩 변환 진행 후에,
- 인덱스에서 검색하여 상위 3개를 찾는다

```
query = klue_mrc_dataset[3]['question'] # 로버트 헨리 딕이 194
6년에 매사추세츠 연구소에서 개발한 것은 무엇인가?
query_embedding = sentence_model.encode([query])
```

```
distances, indices = index.search(query_embedding, 3)

for idx in indices[0]:
  print(klue_mrc_dataset['context'][idx][:50])

# 출력 결과
# 태평양 전쟁 중 뉴기니 방면에서 진공 작전을 실시해 온 더글러스 맥아더
장군을 사령관으로 (오답)
# 태평양 전쟁 중 뉴기니 방면에서 진공 작전을 실시해 온 더글러스 맥아더
장군을 사령관으로 (오답)
# 미국 세인트루이스에서 태어났고, 프린스턴 대학교에서 학사 학위를 마치고
1939년에 로체스 (정답)
```

- 같은 코드를 다른 쿼리로 진행
- 만약 가장 유사한 게 별로 안 유사하다면 이런 결과가 발생

→ 키워드를 맞추면 해결되지 않을까? 그래서 나온 것이 BM25!

10.4 키워드 검색 방식 이해하기

키워드 검색

- 동일한 키워드가 많이 포함될수록 유사도를 높게 평가하는 검색 방식
- 하지만 키워드가 포함되지 않고 유사한 경우에 정확도가 떨어지는 단점이 존재
- 대표적인 키워드 검색 방식: BM25

BM25

- 키워드 검색 방식
- TF-IDF를 뛰어넘음 How? 문서의 길이에 대한 가중치 추가를 통해

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

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

- IDF(q) : 특정 단어 q가 전체 문서에서 얼마나 자주 등장했는지
  - n(q)가 크다면? → IDF가 작아짐 → 중요도가 작아짐
  - n(q)가 작다면? → IDF가 커짐 → 중요도가 커짐
- f(q,D): 특정 문서 D에 토큰 q가 등장하는 횟수
  - f(q,D)가 크다면? → k + 1로 포화됨 → 최댓값 k + 1
- 문서 길이가 길다? → 분모가 커진다 → 중요도 작아짐
- 문서 길이가 짧다? → 분모가 작아진다 → 중요도 커짐

상호 순위 조합

- 의미 검색과 키워드 검색을 둘 다 고려했을 때 순위를 어떻게 매길지
- Score = 1/(자신의 순위 + 임의의 상수 K)
- BM25 와 의미 검색 Score의 합 = 최종 점수 → 이 점수로 순위를 매김

10.5 하이브리드 검색 구현하기

BM25 구현하기

```
import math
import numpy as np
from typing import List
```

```
from transformers import PreTrainedTokenizer
from collections import defaultdict

class BM25:
  def __init__(self, corpus:List[List[str]], tokenizer:PreT
rainedTokenizer):
    self.tokenizer = tokenizer
    self.corpus = corpus
    self.tokenized_corpus = self.tokenizer(corpus, add_spec
ial_tokens=False)['input_ids']
    self.n_docs = len(self.tokenized_corpus)
    self.avg_doc_lens = sum(len(lst) for lst in self.tokeni
zed_corpus) / len(self.tokenized_corpus)
    self.idf = self._calculate_idf()
    self.term_freqs = self._calculate_term_freqs()

    # 각 토큰이 몇개의 문서에 등장하는지 집계
  def _calculate_idf(self):
    idf = defaultdict(float)
    # 토큰화된 문서들의 집합에서
    for doc in self.tokenized_corpus:
        # 고유한 토큰들만 추출하여 해당 토큰을 가진 문서가 몇개인지 저
장
      for token_id in set(doc):
        idf[token_id] += 1
    # 그 정보를 바탕으로 idf 계산
    for token_id, doc_frequency in idf.items():
      idf[token_id] = math.log(((self.n_docs - doc_frequenc
y + 0.5) / (doc_frequency + 0.5)) + 1)
    return idf

    # 특정 문서에서 특정 토큰의 등장 횟수 저장
  def _calculate_term_freqs(self):
    term_freqs = [defaultdict(int) for _ in range(self.n_do
cs)]
    for i, doc in enumerate(self.tokenized_corpus):
      for token_id in doc:
        term_freqs[i][token_id] += 1
```

```
    return term_freqs

    # 위 두 메서드를 바탕으로 BM25값 계산
  def get_scores(self, query:str, k1:float = 1.2, b:float=
0.75):
    query = self.tokenizer([query], add_special_tokens=Fals
e)['input_ids'][0]
    scores = np.zeros(self.n_docs)
    for q in query:
      idf = self.idf[q]
      for i, term_freq in enumerate(self.term_freqs):
        q_frequency = term_freq[q]
        doc_len = len(self.tokenized_corpus[i])
        score_q = idf * (q_frequency * (k1 + 1)) / ((q_freq
uency) + k1 * (1 - b + b * (doc_len / self.avg_doc_lens)))
        scores[i] += score_q
    return scores

    # 상위 점수 k개 반환하는 코드
  def get_top_k(self, query:str, k:int):
    scores = self.get_scores(query)
    top_k_indices = np.argsort(scores)[-k:][::-1]
    top_k_scores = scores[top_k_indices]
    return top_k_scores, top_k_indices
```

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained('klue/roberta-base'

bm25 = BM25(['안녕하세요', '반갑습니다', '안녕 서울'], tokenizer)
bm25.get_scores('안녕')
# array([0.44713859, 0.        , 0.52354835])
```

- 예상대로 안녕이라는 키워드가 포함되면 유사도가 생기고,
- 아닌 경우에 유사도가 0이 됨.

```
# BM25 검색 준비
bm25 = BM25(klue_mrc_dataset['context'], tokenizer)

query = "이번 연도에는 언제 비가 많이 올까?"
_, bm25_search_ranking = bm25.get_top_k(query, 100)

for idx in bm25_search_ranking[:3]:
  print(klue_mrc_dataset['context'][idx][:50])

# 출력 결과
# 갤럭시S5 언제 발매한다는 건지언제는 “27일 판매한다”고 했다가 “이르면 26
# 인구 비율당 노벨상을 세계에서 가장 많이 받은 나라, 과학 논문을 가장 많이
# 올여름 장마가 17일 제주도에서 시작됐다. 서울 등 중부지방은 예년보다 사나
```

- 언제 가 들어갔다고 바로 오답 발생
- 이 경우에는 의미 검색이 더 높은 정확도를 보임

```
query = klue_mrc_dataset[3]['question']  # 로버트 헨리 딕이 19
46년에 매사추세츠 연구소에서 개발한 것은 무엇인가?
_, bm25_search_ranking = bm25.get_top_k(query, 100)

for idx in bm25_search_ranking[:3]:
  print(klue_mrc_dataset['context'][idx][:50])

# 출력 결과
# 미국 세인트루이스에서 태어났고, 프린스턴 대학교에서 학사 학위를 마치고
1939년에 로체스 (정답)
# ;메카동(メカドン)
(오답)
# :성우 : 나라하시 미키(ならはしみき)
# 길가에 버려져 있던 낡은 느티나
# ;메카동(メカドン)
(오답)
```

```
# :성우 : 나라하시 미키(ならはしみき)
# 길가에 버려져 있던 낡은 느티나
```

- 이 경우에는 키워드 검색이 더 높은 정확도를 보임

→ 이 둘을 합쳐 보자! 그러면 결과가 잘 나오겠지

상호 순위 조합 함수 구현 및 테스트

```
from collections import defaultdict

def reciprocal_rank_fusion(rankings:List[List[int]], k=5):
    rrf = defaultdict(float)
    for ranking in rankings:
        for i, doc_id in enumerate(ranking, 1):
            rrf[doc_id] += 1.0 / (k + i)
    return sorted(rrf.items(), key=lambda x: x[1], reverse=Tr

rankings = [[1, 4, 3, 5, 6], [2, 1, 3, 6, 4]]
reciprocal_rank_fusion(rankings)

# [(1, 0.30952380952380953),
#  (3, 0.25),
#  (4, 0.24285714285714285),
#  (6, 0.2111111111111111),
#  (2, 0.16666666666666666),
#  (5, 0.1111111111111111)]
```

하이브리드 검색 구현

```
# 의미 검색 반환
def dense_vector_search(query:str, k:int):
  query_embedding = sentence_model.encode([query])
  distances, indices = index.search(query_embedding, k)
  return distances[0], indices[0]

# 의미 검색 랭킹과 BM25 랭킹 반환
def hybrid_search(query, k=20):
  _, dense_search_ranking = dense_vector_search(query, 100)
  _, bm25_search_ranking = bm25.get_top_k(query, 100)

    # 상호 순위 조합 함수로 결과 출력
  results = reciprocal_rank_fusion([dense_search_ranking, b
m25_search_ranking], k=k)
  return results
```

결과 확인

```
query = "이번 연도에는 언제 비가 많이 올까?"
print("검색 쿼리 문장: ", query)
results = hybrid_search(query)
for idx, score in results[:3]:
  print(klue_mrc_dataset['context'][idx][:50])

print("=" * 80)
query = klue_mrc_dataset[3]['question'] # 로버트 헨리 딕이 194
6년에 매사추세츠 연구소에서 개발한 것은 무엇인가?
print("검색 쿼리 문장: ", query)

results = hybrid_search(query)
for idx, score in results[:3]:
  print(klue_mrc_dataset['context'][idx][:50])

# 출력 결과
```

```
# 검색 쿼리 문장:  이번 연도에는 언제 비가 많이 올까?
# 올여름 장마가 17일 제주도에서 시작됐다. 서울 등 중부지방은 예년보다 사나흘 정도 늦은  (정답)
# 갤럭시S5 언제 발매한다는 건지언제는 “27일 판매한다”고 했다가 “이르면 26일 판매한다  (오답)
# 연구 결과에 따르면, 오리너구리의 눈은 대부분의 포유류보다는 어류인 칠성장어나 먹장어, 그 (오답)
# ================================================================================
# 검색 쿼리 문장:  로버트 헨리 딕이 1946년에 매사추세츠 연구소에서 개발한 것은 무엇인가?
# 미국 세인트루이스에서 태어났고, 프린스턴 대학교에서 학사 학위를 마치고 1939년에 로체스 (정답)
# 1950년대 말 매사추세츠 공과대학교의 동아리 테크모델철도클럽에서 ‘해커’라는 용어가 처음 (오답)
# 1950년대 말 매사추세츠 공과대학교의 동아리 테크모델철도클럽에서 ‘해커’라는 용어가 처음 (오답)
```

- 두 검색 방식의 약점을 모두 보완할 수 있었음

END