11장

> 자신의 데이터에 맞춘 임베딩 모델 만들기: RAG 개선하기

▼ 10장 내용 되짚기

문장의 유사도를 비교하는 두가지 방식인 `교차 인코더`, `바이 인코더`.

- **교차 인코더**

  교차인코더는 비교하려는 두 문장을 입력으로 받아서, 직접 비교

  → (장점) 유사도를 더 정확히 계산할 수 있다

  → (단점) 느리고 확장성이 떨어진다.

- **바이 인코더**

  바이 인코더는 입력 문장에 대해 독립적인 문장 임베딩을 생성, 임베딩끼리 코사인 유사도 같은 가벼운 거리 계산 방식으로 유사도를 계산

  → (장점) 큰 데이터셋에서도 빠르게 가장 비슷한 문장을 찾을 수 있다.

  ⇒ 문장 임베딩을 생성하는 대표적인 방식으로 사용

  ⇒ 검색 증강 생성(RAG)의 필수 구성 요소로 사용

  → (단점) 교차 인코더에 비해 떨어지는 유사도 정확성

11장 내용 미리보기

- **느리지만 정확한 교차 인코더**와 **빠르지만 덜 정확한 바이 인코더**를 결합해 RAG의 검색 성능을 높이는 방법
  - 교차 인코더는 느리기 때문에, 대규모 데이터의 유사도를 계산하는데 사용하지 않는다.

    → 바이 인코더를 통해 선별된 소수의 데이터를 대상으로 더 정확한 유사도 계산을 위해 사용.

    → `re-rank` : 교차 인코더가 미리 필터링된 문장 사이의 유사도를 계산하고, 순위를 변경하는 것을 `순위 재정렬(re-rank)` 라고 한다.
- 이미 학습된 문장 임베딩 모델이 아니라, BERT 같은 인코더 모델을 문장 임베딩 모델로 학습하는 방법
- 우리가 사용하려는 실습 데이터에 맞춰 미세 조정함으로써 검색 성능을 높이는 방법

11.1 검색 기능을 높이기 위한 두가지 방법

교차인코더

- 비교하려는 두 문장을 직접 입력으로 받아 비교

  → 따라서 유사도를 더 정확하게 계산

  → BUT 유사도를 계산하려는 조합의 수만큼 모두 BERT와 같은 트랜스포머 인코더 연산(무거운 연산)을 수행해야하기 때문에 확장성이 떨어진다.

바이 인코더

- 독립적인 문장 임베딩 사이의 유사도를 가벼운 벡터 연산을 통해 계산

  → 빠른 검색이 가능

  → BUT 교차 인코더만큼 정확하게 유사도를 계산하기 어렵다.

바이 인코더와 교차 인코더를 결합해서 사용하기

- `바이 인코더` | 대규모의 문서에서 검색 쿼리와 유사한 소수의 문서(예: 상위 100개)를 선별
- `교차 인코더` | 선별한 소수의 문서는 유사도를 더 정확히 계산할 수 있는 교차 인코더를 사용해 유사한 순서대로 재정렬

- 정리하자면

  ⇒ `교차 인코더`는 계산량이 많아 빠른 계산이 불가능하므로,

  ⇒ `바이 인코더`를 활용해 소수의 문서를 추려내어 교차 인코더가 계산할 문서의 양 자체를 줄여버림

- 바이 인코더와 교차 인코더를 결합해 검색 성능이 향상된 검색 과정
  1. **바이 인코더를 추가 학습해 검색 성능 높이기**
     - 문장 임베딩 모델도 학습 데이터와 유사한 입력 데이터에 대해 더 잘 작동한다
     - 따라서, 문장 임베딩 모델을 사용하려는 데이터셋으로 바이 인코더를 추가 학습해 검색 성능을 높일 수 있다.
  2. **교차 인코더를 추가해서 검색 성능 높이기**
     - 바이 인코더만 사용해 검색을 구현하는 경우, 유사도의 정확도가 떨어지므로, 관련성이 떨어지는 결과도 포함되는 단점.
     - 여기에 `검색 쿼리 문장`과 `검색 대상 문장`을 함께 입력으로 받는 교차 인코더를 추가하면
       - 바이 인코더를 사용해 좁힌 소수의 후보군에서
       - 더 관련성이 있는 높은 문서를 상위에 올릴 수 있다.
     - RAG(검색 증강 생성)에서는 검색된 문서 중 상위 몇 개의 입력만 프롬프트에 추가한다.
       - 따라서 검색 쿼리와 관련이 높은 문서가 상위 검색 결과에 포함되어야 검색 증강 생성이 효과적으로 작동한다.

11.2 언어 모델을 임베딩 모델로 만들기

- `Sentence-Transformers` 라이브러리 사용해 문장 임베딩 모델을 쉽게 활용 가능
- 문장 임베딩 모델의 구성 (2개의 레이어로 구성)
  1. 대량의 텍스트 데이터로 사전학습한 언어 모델 (예: BERT, RoBERTa)
  2. 풀링 층
     - 입력 문장의 길이에 따라 달라질 수 있는 **출력 차원을 고정된 차원으로 맞추는 역할**
     - 대표적인 풀링 방법
       - 클래스 모드: 언어 모델 출력의 첫번째 토큰(`[CLS]`)을 사용 하는 풀링 모드
       - 평균 모드: 언어 모델의 출력을 평균 내 사용하는 풀링 모드 (일반적으로 많이 사용)
       - 최대 모드: 언어 모델의 출력 중 가장 큰 값들을 모아 사용하는 풀링 모드
- 문장 임베딩 모델 만들기
  - `사전 학습 언어 모델 불러오기` + `모델 위에 풀링 층을 추가` + `문장의 의미를 잘 담을 수 있도록 학습`

▼ 🕯️실습

▼ 평가 데이터 준비

- 한국어 벤치마크 데이터셋, `KLUE` 중 `STS 데이터셋` 사용
  - `STS 데이터셋`: 2개의 문장이 얼마나 유사한지 점수를 매긴 데이터셋
  - 구성
    - `두 개의 문장`: sentence1, sentence2 컬럼
    - `labels`: 두 문장이 얼마나 유사한지 나타내는 다양한 형식의 레이블
      - `label`(소수점 첫째), `real-label`(반올림 하지 않은 전체), `binary-label`(0 혹은 1)이 존재하고
      - 이 중, `label` 사용
    - 학습(train) 데이터와 테스트(test) 데이터 분리되어 저장

- 평가 데이터 전처리

1. 학습 데이터의 일부를 검증(eval) 데이터셋으로 분리 (즉, train dataset, eval dataset, test dataset 준비)
2. 유사도 점수 0 ~ 1 사이 값으로 정규화
3. torch.utils.data.Dataloader로 배치 데이터로 만들기

KLUE STS 데이터 예시

```
from datasets import load_dataset

# KLUE sts 데이터셋은 학습(train) 데이터와 테스트(validation) 데이터로 나누어 저장되어 있다.
klue_sts_train = load_dataset('klue', 'sts', split='train')
klue_sts_test = load_dataset('klue', 'sts', split='validation')

# sts 데이터 구성: 2개의 문장, 두 문장의 유사도에 관련된 지표들
klue_sts_train[0]

# {'guid': 'klue-sts-v1_train_00000',
#  'source': 'airbnb-rtt',
#  'sentence1': '숙소 위치는 찾기 쉽고 일반적인 한국의 반지하 숙소입니다.',
#  'sentence2': '숙박시설의 위치는 쉽게 찾을 수 있고 한국의 대표적인 반지하 숙박시설입니다.',
#  'labels': {'label': 3.7, 'real-label': 3.714285714285714, 'binary-label': 1}}
```

평가 데이터 전처리

```
# 전처리 1. 학습 데이터셋의 10%를 검증 데이터셋으로 구성한다.
klue_sts_train = klue_sts_train.train_test_split(test_size=0.1, seed=42)
klue_sts_train, klue_sts_eval = klue_sts_train['train'], klue_sts_train['test']
```

```
from sentence_transformers import InputExample

# 전처리 2. 유사도 점수를 0~1 사이로 정규화 하고 InputExample 객체에 담는다.
# 예: label 점수 3.7 -> (정규화) 3.7 / 5 = 0.74

def prepare_sts_examples(dataset):
    examples = []
    for data in dataset:
        examples.append(
            InputExample(
                texts=[data['sentence1'], data['sentence2']],
                label=data['labels']['label'] / 5.0)
            )
    return examples

train_examples = prepare_sts_examples(klue_sts_train)
eval_examples = prepare_sts_examples(klue_sts_eval)
test_examples = prepare_sts_examples(klue_sts_test)
```

```
# 전처리 3. torch.utils.data.Dataloader로 배치 데이터로 만들기
from torch.utils.data import DataLoader

train_dataloader = DataLoader(train_examples, shuffle=True, batch_size=16)
```

> 임베딩 모델의 성능을 평가하기 위해 `EmbeddingSimilarityEvaluator` 클래스 사용

```
from sentence_transformers.evaluation import EmbeddingSimilarityEvaluator

eval_evaluator = EmbeddingSimilarityEvaluator.from_input_examples(eval_examples)
test_evaluator = EmbeddingSimilarityEvaluator.from_input_examples(test_examples)
```

▼ 기존에 사전 학습된 언어 모델을 불러와 그대로 임베딩 모델로 사용하기

> 사전 학습된 언어 모델을 사용해 임베딩 모델 정의

```
from sentence_transformers import SentenceTransformer, models

# 사전 학습된 언어 모델
transformer_model = models.Transformer('klue/roberta-base')

# 평균 풀링층
pooling_layer = models.Pooling(
    transformer_model.get_word_embedding_dimension(),
    pooling_mode_mean_tokens=True
)

# 최종 임베딩 모델
embedding_model = SentenceTransformer(modules=[transformer_model, pooling_layer])
```

- 임베딩 모델의 구성
  - `klue/roberta-base` 모델 + `평균 풀링층`
  - 언어 모델의 출력을 단순히 평균 내어 고정된 차원의 벡터로 만든다

  → 즉, 해당 임베딩 모델은 테스트 데이터와 유사한 문장에 대한 정보가 없는 상태로 입력 문장을 임베딩 벡터로 만드는 것.

> 기존에 사전 학습된 언어 모델을 불러와 그대로 임베딩 모델에 대한 평가

```
# EmbeddingSimilarityEvaluator 클래스를 사용해 정의한 test_evaluator에
# - embedding_model을 인자로 넣어주면,
# - 평가 데이터셋의 정답 유사도 점수와 embedding model을 사용해 예측한 유사도 점수를 비교
# - 비교를 통해 나온 수치로 문장 임베딩 모델이 얼마나 잘 작동하는지 평가한다

test_evaluator(embedding_model)
# 0.36460670798564826 (낮은 평가 수치)
```

▼ 참고: `EmbeddingSimilarityEvaluator` 의 작동 원리

1. **두 문장 유사도 계산**:
   - 두 문장을 각각 임베딩하고, 지정된 유사도 함수를 통해 벡터 간 유사도를 계산
   - 예: Cosine Similarity, Euclidean Distance, Manhattan Distance 등 옵션 존재
2. **지표 측정**:
   - 모델이 계산한 유사도 점수와 정답 유사도 점수 간 **Pearson 상관계수** 및 **Spearman 상관계수**를 계산.
   - 이 중, 특정 지표를 선택하지 않는다면 **Spearman 상관계수**가 기본값으로 사용

3. **출력 결과 예시**

```
# sts-dev dataset에 대한 EmbeddingSimilarityEvaluator 결과
EmbeddingSimilarityEvaluator: Evaluating the model on the sts-dev dataset:

# 두 문장의 유사도를 Cosine-Similarity로 측정했을 때
Cosine-Similarity :  Pearson: 0.8806 Spearman: 0.8810
Manhattan-Similarity :  Pearson: 0.8503 Spearman: 0.8601
```

▼ Pearson 상관계수와 Spearman 상관계수의 의미와 두 수치를 확인하는 이유

1. **Pearson 상관계수란?**
   - 두 값의 **선형적 관계**를 측정
   - 쉽게 말해, **값들이 비례적으로 증가하거나 감소하는 정도**를 보는 수치
     예)
     - 참값(정답) 유사도가 높을수록 모델이 계산한 유사도도 높다면, Pearson 상관계수가 높다.
     - 만약 참값은 높아지는데 모델 점수는 엉뚱하게 낮아진다면, 값이 낮아진다.

   👉 **값의 변화가 얼마나 일치하는지** 보는 척도

---

1. **Spearman 상관계수란?**
   - 점수 간의 **순위 관계**를 측정
   - 점수가 실제로 얼마나 가까운지는 보지 않고, **순서만 일치하는지** 확인하는 수치
     예)
     - 참값 순서: 문장 A > 문장 B > 문장 C
     - 모델 점수 순서: 문장 A > 문장 B > 문장 C
       → Spearman 상관계수 높음.
     - 모델 점수 순서: 문장 B > 문장 A > 문장 C
       → Spearman 상관계수 낮음.

   👉 **값의 순위가 얼마나 일치하는지** 보는 척도

결론

1. **두 값이 높을수록** (1에 가까울수록) 모델이 데이터를 잘 이해하고 있는 것
   - **Pearson**: 점수의 선형적 관계가 잘 맞는다 → 모델이 참값 유사도를 잘 따라감.
   - **Spearman**: 점수의 순위가 잘 맞는다 → 모델이 상대적인 우선순위를 잘 예측함.
2. **왜 두 개를 사용하나요?**
   - Pearson은 값의 절대적 차이까지 신경 쓰고,
   - Spearman은 값의 **순서 관계만** 보니까, 둘을 함께 보면 모델의 전반적인 성능을 더 잘 알 수 있다.

👉 두 수치가 **높을수록 모델이 잘하고 있는 것**

▼ 유사한 문장 데이터로 학습된 임베딩 모델 사용하기

- 학습 조건
  - 학습 시킬 기본 언어 모델: `klue/roberta-base`
  - 학습 손실 함수: `CosineSimilarityLoss`
  - 학습 데이터를 문장 임베딩으로 변환 → 그 후, 두 문장 사이의 `코사인 유사도`와 `정답 유사도`를 비교해 학습 수행
    - 따라서, 모델은 두 문장의 코사인 유사도가 정답 유사도와 비슷해지도록 임베딩하는 모델로 학습될 것.(그래야 학습 Loss가 낮아질테니까)

> 임베딩 모델 학습

```
from sentence_transformers import losses

num_epochs = 4
model_name = 'klue/roberta-base'
model_save_path = 'output/training_sts_' + model_name.replace("/", "-")
train_loss = losses.CosineSimilarityLoss(model=embedding_model)

# 임베딩 모델 학습
embedding_model.fit(
    train_objectives=[(train_dataloader, train_loss)],  # 평가 데이터와 유사한 문장 데이터로 구성된 학습 데이터로 임베딩 모델 학습
    evaluator=eval_evaluator,
    epochs=num_epochs,
    evaluation_steps=1000,
    warmup_steps=100,
    output_path=model_save_path
)
```

> 학습된 임베딩 모델의 성능 평가

```
trained_embedding_model = SentenceTransformer(model_save_path)
test_evaluator(trained_embedding_model)
# 0.8965595666246748
```

⇒ 동일한 평가 데이터셋에 대해 `0.36460670798564826` 점이 나왔던 기존 임베딩 모델에 비해 크게 향상된 결과

11.3 임베딩 모델 미세 조정하기

검색 증강 생성(RAG)의 임베딩 모델 활용

- `검색 쿼리와 관련된 문서를 찾아` LLM 프롬프트에 맥락 데이터로 추가할 때 임베딩 모델을 활용
  - 따라서, **좋은 임베딩 모델**은
    - 검색 쿼리와 관련이 있는 문서는 유사도가 높게 (1에 가깝게)
    - 검색 쿼리와 관련이 없는 문서는 유사도가 낮게 (0에 가깝게) 나와야 한다.
- **더 좋은 임베딩 모델을 만들기 위해, KLUE의 MRC 데이터셋으로 추가 학습 진행 (미세 조정 학습)**

▼ 🕯️실습

▼ 학습 데이터 준비

- 한국어 벤치마크 데이터셋, `KLUE` 중 `MRC 데이터셋` 사용
  - `MRC 데이터셋`: 기사 본문 및 해당 기사와 관련된 질문을 수집한 데이터셋
  - 구성
    - `title`: 기사 제목
    - `context`: 기사 본문
    - `question`: 기사 내용 관련 질문

> KLUE MRC 데이터 예시

```
from datasets import load_dataset
klue_mrc_train = load_dataset('klue', 'mrc', split='train')
klue_mrc_train[0]
```

```
# {'title': '제주도 장마 시작 … 중부는 이달 말부터',
#  'context': '올여름 장마가 17일 제주도에서 시작됐다. 서울 등 중부지방은 예년보다 사나흘 정도 늦
#  'news_category': '종합',
#  'source': 'hankyung',
#  'guid': 'klue-mrc-v1_train_12759',
#  'is_impossible': False,
#  'question_type': 1,
#  'question': '북태평양 기단과 오호츠크해 기단이 만나 국내에 머무르는 기간은?',
#  'answers': {'answer_start': [478, 478], 'text': ['한 달가량', '한 달']}}
```

- KLUE의 MRC 데이터셋을 활용해 사용자의 질문에 적절히 응답하는 챗봇을 만드는 경우
  - 검색 증강 생성을 통해 사용자의 질문과 관련된 기사 본문 찾고 → 이렇게 찾은 기사 본문을 LLM이 참고하도록 프롬프트에 추가
  - 이 때, 검색 증강 생성이 잘 되려면?
    - 질문과 관련된 기사 본문이 검색 결과의 상위에 나타나야 한다.
      - 그리고, 이를 위해서는 관련 있는 질문-내용 쌍에는 높은 유사도를, 관련 없는 질문-내용 쌍에는 낮은 유사도를 줘야 한다.
- 11.2 내용에서 확인한 것처럼,
  - **MRC 데이터셋에 임베딩 모델을 활용하려는 경우에, 해당 임베딩 모델을 MRC 데이터셋으로 미세조정을 해주어야**
  - 관련 있는 질문-내용 쌍에는 높은 유사도를, 관련 없는 질문-내용 쌍에는 낮은 유사도를 출력하는 **좋은 임베딩 모델**을 사용할 수 있다.

- 임베딩 모델 학습에 사용할 MRC 데이터 만들기

> KLUE/MRC 데이터 불러와서 전처리

```
from datasets import load_dataset
klue_mrc_train = load_dataset('klue', 'mrc', split='train')
klue_mrc_test = load_dataset('klue', 'mrc', split='validation')

# 데이터 프레임으로 변환
df_train = klue_mrc_train.to_pandas()
df_test = klue_mrc_test.to_pandas()

# 필요한 3개의 컬럼(title, question, context)만 남기기
df_train = df_train[['title', 'question', 'context']]
df_test = df_test[['title', 'question', 'context']]
```

> MRC 데이터셋은 기사 본문과 그에 관련된 질문으로 구성된 데이터셋이기 때문에 서로 유사한 질문-내용 쌍으로 이루어진 데이터셋만 존재한다.
> 따라서, 질문과 관련이 없는 임의의 본문을 짝지어 유사하지 않은 데이터(`irrelevant_context`)도 만들어준다.

```

from sentence_transformers import InputExample

examples = []
for idx, row in df_test_ir[:100].iterrows():
  examples.append(
```

```
      InputExample(texts=[row['question'], row['context']], label=1)  # question과
context가 관련이 있는 경우, label=1 으로 지정
  )
  examples.append(
      InputExample(texts=[row['question'], row['irrelevant_context']], label=0) #
question과 context가 관련이 없는 경우, label=0 으로 지정
  )
```

기본 임베딩 모델 불러오기

- 참고: 여기에서 사용된 기본 임베딩 모델은, `sts 데이터셋` 을 학습시킨 사전 학습 언어 모델 (≠ `klue/roberta-base`)

```
from sentence_transformers import SentenceTransformer
sentence_model = SentenceTransformer('shangrilar/klue-roberta-base-klue-sts')
```

기본 임베딩 모델에 대한 `MRC 평가 데이터` 에 대한 모델 성능

```
from sentence_transformers.evaluation import EmbeddingSimilarityEvaluator
evaluator = EmbeddingSimilarityEvaluator.from_input_examples(
    examples
)
evaluator(sentence_model)
# 0.8151553052035344
```

MNR(Multiple Negatives Ranking) 손실

- MRC 데이터셋같이 데이터셋에 서로 고나련이 있는 문장만 있는 경우 사용하기 좋은 손실 함수
- **하나의 배치 데이터 안에서 다른 데이터의 기사 본문을 관련이 없는(=negative) 데이터로 사용해 모델을 학습시킨다.**
- 즉, `irrelevant_context` 를 만들어준 것처럼 관련이 없는 데이터를 따로 만들 필요가 없이 → 서로 관련된 데이터만으로 학습 데이터를 구성하면 된다.

MNR 손실을 사용한다면, 긍정 데이터만으로 학습 데이터를 구성해도 된다.

```
train_samples = []
for idx, row in df_train_ir.iterrows():
    train_samples.append(InputExample(
        texts=[row['question'], row['context']]
    ))
```

```
from sentence_transformers import datasets

batch_size = 16

loader = datasets.NoDuplicatesDataLoader(
    train_samples, batch_size=batch_size)
```

```
from sentence_transformers import losses

loss = losses.MultipleNegativesRankingLoss(sentence_model)
```

▼ **임베딩 모델 학습 with MNR 손실**

> 임베딩 모델 학습 with MNR 손실

```
epochs = 1
save_path = './klue_mrc_mnr'

sentence_model.fit(
    train_objectives=[(loader, loss)], # loss: MultipleNegativesRankingLoss
    epochs=epochs,
    warmup_steps=100,
    output_path=save_path,
    show_progress_bar=True
)
```

> 미세 조정한 모델 성능 평가

```
evaluator(sentence_model)
# 0.8600968992433692
```

⇒ 미세 조정 전, 0.815 였던 성능에 비해 0.05 정도 향상된 것을 알 수 있음.

11.4 검색 품질을 높이는 순위 재정렬

교차 인코더의 특징

- 2개의 문장을 입력으로 받고
- 문장 사이의 관계를 더 직접적으로 계산하는 구조
- 장점: 두 문장 사이의 유사도를 더 정확히 계산
- 단점: 속도가 느리다.

교차 인코더의 활용

- 교차 인코더는 문장 분류 모델을 활용
- 2개의 문장에 대해 “관련이 있는지(1)”, “관련이 없는지(0)”로 이진 분류 시행
- 교차 인코더는 (바이 인코더가 선별한 소수의) 유사도 기반 관련 문서들의 순위를 재정렬하는데 사용된다.

교차 인코더 미세 조정

> 교차 인코더로 사용할 사전 학습 모델 불러오기

- 교차 인코더는 많은 계산을 해야 하므로, 파라미터 수가 적은 SMALL 모델 사용
- 분류 헤드는 랜덤으로 초기화

```
from sentence_transformers.cross_encoder import CrossEncoder
cross_model = CrossEncoder('klue/roberta-small', num_labels=1)
```

미세 조정하지 않은 교차 인코더의 성능 평가 결과

```
from sentence_transformers.cross_encoder.evaluation import CECorrelationEvaluator

ce_evaluator = CECorrelationEvaluator.from_input_examples(examples) # CECorrelationEvaluat
or: 교차 인코더 평가시 사용
ce_evaluator(cross_model)
# 0.003316821814673943
```

⇒ 미세 조정하지 않은 (=랜덤으로 초기화된 분류 헤드를 가진) 교차 인코더는 성능이 좋지 않음

교차 인코더 학습 데이터셋 준비

- 교차 인코더는 관련 있는 질문-내용 쌍과 관련이 없는 질문-내용 쌍을 구분 해야 한다.
- 따라서 학습 내용에 두 부류의 데이터가 모두 포함되어 있어야 하므로, 만들어준다.

```
train_samples = []
for idx, row in df_train_ir.iterrows():
    train_samples.append(InputExample(
        texts=[row['question'], row['context']], label=1
    ))
    train_samples.append(InputExample(
        texts=[row['question'], row['irrelevant_context']], label=0
    ))
```

교차 인코더 학습 수행

```
train_batch_size = 16
num_epochs = 1
model_save_path = 'output/training_mrc'

train_dataloader = DataLoader(train_samples, shuffle=True, batch_size=train_batch_size)

cross_model.fit(
    train_dataloader=train_dataloader,
    epochs=num_epochs,
    warmup_steps=100,
    output_path=model_save_path
)
```

▼ 참고: 아니, 학습에 사용되는 loss가 없는데? → `cross_model`의 학습 Loss 관련

- 🔗: https://github.com/UKPLab/sentence-transformers/blob/07fac06d7af8d92791d9386cce9bf970d3d7dbd3/sentence_transformers/cross_encoder/Cros
- `loss_fct: Which loss function to use for training. If None, will use nn.BCEWithLogitsLoss() if self.config.num_labels == 1 else nn.CrossEntropyLoss(). Defaults to None.`
- 사용자가 별도로 `loss_fct`를 제공하지 않을 경우 (`None`일 경우),
  - `self.config.num_labels == 1`: 예측할 라벨의 개수가 1개(즉, 이진 분류 또는 회귀)라면, `nn.BCEWithLogitsLoss()`가 기본값으로 사용됩니다.
    - `nn.BCEWithLogitsLoss()`: 이진 분류 문제에 적합한 손실 함수로, 로지트 값에 `sigmoid`를 적용한 뒤, Binary Cross-Entropy를 계산합니다.

- `self.config.num_labels > 1`: 예측할 라벨의 개수가 2개 이상(즉, 다중 클래스 분류)이라면, `nn.CrossEntropyLoss()`가 기본값으로 사용됩니다.
    - `nn.CrossEntropyLoss()`: 다중 클래스 분류 문제에 적합한 손실 함수로, 소프트맥스와 함께 사용되어 클래스 확률을 비교합니다.

> 학습한 교차 인코더 평가 결과

```
ce_evaluator(cross_model)
# 0.8650250798639563
```

⇒ 학습 전에는 거의 0(0.003)이었던 성능이 0.865로 크게 향상

11.5 바이 인코더와 교차 인코더로 개선된 RAG 구현하기

- MRC 데이터셋 중 검증(validation) 데이터를 활용
- 질문(question) 컬럼을 입력했을 때, 검색된 상위 10개의 기사 본문(context)에 정답이 있는 비율(HitRate@10)을 성능 지표로 사용
- 전체 검색을 수행하는데 걸리는 시간 비교

▼ 🕯️실습

▼ 데이터 준비

> 평가를 위한 데이터셋을 불러와 1,000개만 선별

```
from datasets import load_dataset
klue_mrc_test = load_dataset('klue', 'mrc', split='validation')
klue_mrc_test = klue_mrc_test.train_test_split(test_size=1000, seed=42)['test']
```

▼ 필요한 함수 구현

> 임베딩을 저장하고 검색하는 함수 정의

```
import faiss

# 변환된 문장 임베딩 벡터들을 인덱스에 저장
def make_embedding_index(sentence_model, corpus):
    embeddings = sentence_model.encode(corpus)
    index = faiss.IndexFlatL2(embeddings.shape[1])
    index.add(embeddings)
    return index

# 인덱스에서 검색 쿼리와 유사한 k개의 문서를 검색
def find_embedding_top_k(query, sentence_model, index, k):
    embedding = sentence_model.encode([query])
    distances, indices = index.search(embedding, k)
    return indices
```

> 교차 인코더를 활용한 순위 재정렬 함수 정의

```
def make_question_context_pairs(question_idx, indices):
  return [[klue_mrc_test['question'][question_idx], klue_mrc_test['context'][idx]] fo
r idx in indices]

def rerank_top_k(cross_model, question_idx, indices, k):
  input_examples = make_question_context_pairs(question_idx, indices)
  relevance_scores = cross_model.predict(input_examples)
  reranked_indices = indices[np.argsort(relevance_scores)[::-1]]
  return reranked_indices
```

> 성능 지표(히트율)와 평가에 걸린 시간을 반환하는 함수 정의

- **히트율**
  - 각 질문별로 k개의 유사 문서를 검색
  - 해당 검색 결과 내에 정답 데이터가 포함돼 있는 경우 정답을 맞췄다고 계산
  - 전체 평가 데이터 중 맞춘 데이터의 수와 평가에 걸린 시간 결과로 반환

```
import time

def evaluate_hit_rate(datasets, embedding_model, index, k=10):
  start_time = time.time()
  predictions = []
  for question in datasets['question']:
    predictions.append(find_embedding_top_k(question, embedding_model, index, k)[0])
  total_prediction_count = len(predictions)
  hit_count = 0
  questions = datasets['question']
  contexts = datasets['context']

  for idx, prediction in enumerate(predictions):
    for pred in prediction: # prediction: 1 question에 대한 관련 문서라고 예측한 10개(k개)
문서들의 인덱스들이 담겨 있다.
      if contexts[pred] == contexts[idx]:  # 10개의 문서를 하나씩(하나의 문서 인덱스: pred)
돌아가며 context(기사 본문)이 동일한지 확인.
        hit_count += 1  # 그 중 동일한 것이 발견되면, hit_count에 1을 더하고, 다음 question에
대한 문서들로 넘어간다 (break)
        break
  end_time = time.time()
  return hit_count / total_prediction_count, end_time - start_time
```

▼ 각 케이스 별 성능 비교

▼ 전체 비교 (표)

	기본 임베딩 모델	미세 조정한 임베딩 모델	미세 조정한 임베딩 모델 + 순위 재정렬
히트율	88%	94.6%	97.3%
시간	0.013초	0.014초	1.1초

- 기본 임베딩 모델 → 미세 조정한 임베딩 모델
  - 처리 시간: 거의 동일한 시간 (처리 시간 성능 우수)
  - 성능 지표: 6.6% 성능 향상
- 미세 조정한 임베딩 모델 → 임베딩 모델 + 교차 인코더를 모두 사용해 순위 재정렬

- 처리 시간: 처리 시간은 많이 증가
- 성능 지표: 2.7%의 성능 향상
  - 미세 조정 임베딩 모델만 사용했을 때, 실패한 5.4%의 데이터 중
  - 절반인 2.7%를 추가로 정답으로 포함 (오답률을 절반으로 줄였다는 측면에서 우수한 성능 향상)

▼ 기본 임베딩 모델 케이스 (바이 인코더)

> 기본 임베딩 모델 평가 (`klue-roberta-base-klue-sts`)

```
from sentence_transformers import SentenceTransformer
base_embedding_model = SentenceTransformer('shangrilar/klue-roberta-base-klue-sts')
base_index = make_embedding_index(base_embedding_model, klue_mrc_test['context'])
evaluate_hit_rate(klue_mrc_test, base_embedding_model, base_index, 10)
# (0.88, 13.216430425643921)
```

⇒ 88%의 데이터에서 정답

⇒ 1000개 평가 시 13.21초 소요. 즉, 1개 데이터 당 0.013초 처리 시간 소요

▼ 미세 조정한 임베딩 모델 케이스 (바이 인코더)

> 미세 조정한 임베딩 모델 평가

```
finetuned_embedding_model = SentenceTransformer('shangrilar/klue-roberta-base-klue-st
finetuned_index = make_embedding_index(finetuned_embedding_model, klue_mrc_test['cont
evaluate_hit_rate(klue_mrc_test, finetuned_embedding_model, finetuned_index, 10)
# (0.946, 14.309881687164307)
```

⇒ 94.6%의 데이터에서 정답

⇒ 1000개 평가 시 14.30초 소요. 즉, 1개 데이터 당 0.014초 처리 시간 소요

- 임베딩 모델의 상위 N개 결과를 받아 교차 인코더가 순위를 유사도 순으로 재정렬한 후, 상위 K개(N보다 작은 수, K) 를 추출

  ⇒ 임베딩 모델을 통해 유사도가 높은 상위 K개를 바로 뽑았을 때보다 성능을 높일 수 있음

  ⇒ 또한, 상위 N개만을 대상으로 교차 인코더 계산을 진행하므로, 속도 측면도 보완 가능

▼ 미세 조정한 임베딩 모델 (바이 인코더) + 순위 재정렬 케이스 (크로스 인코더)

> 순위 재정렬을 포함한 평가 함수

- `bi_k` : 임베딩 모델의 상위 몇 개의 문서를 추출하는가 (바이 인코더를 활용한 임베딩 모델)
- `cross_k` : 교차 인코더가 재정렬해서 추출하는 문서의 수
  - 즉, `bi_k` 인자로 들어온 수 만큼의 문서 개수 중에 교차 인코더가 재정렬 → 그 중 상위 몇 개의 문서를 추출할지에 대한 인자

```
import time
import numpy as np
from tqdm.auto import tqdm

def evaluate_hit_rate_with_rerank(datasets, embedding_model, cross_model, index, bi
_k=30, cross_k=10):
  start_time = time.time()
  predictions = []
```

```
  for question_idx, question in enumerate(tqdm(datasets['question'])):
    indices = find_embedding_top_k(question, embedding_model, index, bi_k)[0]
    predictions.append(rerank_top_k(cross_model, question_idx, indices, k=cross_k))
  total_prediction_count = len(predictions)
  hit_count = 0
  questions = datasets['question']
  contexts = datasets['context']
  for idx, prediction in enumerate(predictions):
    for pred in prediction:
      if contexts[pred] == contexts[idx]:
        hit_count += 1
        break
  end_time = time.time()
  return hit_count / total_prediction_count, end_time - start_time, predictions
```

임베딩 모델(바이 인코더)과 교차 인코더를 조합해 성능 평가

```
hit_rate, cosumed_time, predictions = evaluate_hit_rate_with_rerank(klue_mrc_test,
finetuned_embedding_model, cross_model, finetuned_index, bi_k=30, cross_k=10)
hit_rate, cosumed_time
# (0.973, 1103.055629491806)
```

⇒ 97.3%의 데이터에서 정답

⇒ 1000개 데이터 처리 시, 1103초. 즉 한 개의 데이터 당 약 1.1초 처리 소요

- 참고: 실습에서는 T4 GPU를 사용