NLP 4주차 스터디

작성일시	@2024년 2월 8일 오후 11:06
텍스트	6.2장 + 7.1장
복습	☐

6.2 텍스트 요약 파이프라인
- NLTK 패키지
- 6.2.1 요약 기준 모델
- 6.2.2 GPT-2
  - TL;DR
- 6.2.3 T5
- 6.2.4 BART
- 6.2.4 PEGASUS

7.1 리뷰 기반 QA시스템 구축하기
- 7.1.1 데이터셋
  - SubjQA 데이터셋의 열 이름과 설명
  - 스탠퍼드 질문 답변 데이터셋 (SQuAD)
- 7.1.2 텍스트에서 답 추출하기
  - 범위 분류
  - QA를 위한 텍스트 토큰화
  - 긴 텍스트 다루기
- 7.1.3 헤이스택을 사용해 QA 파이프라인 구축하기
  - 배경
  - 리트리버
  - 리더
  - 헤이스택 라이브러리
- 프로토타입 QA 파이프라인 구축하는 방법 알아보기
  - 문서 저장소 초기화하기
  - 리트리버 초기화하기
  - 리더 초기화하기
  - 모두 합치기

6.2 텍스트 요약 파이프라인

요약 작업에 많이 사용되는 트랜스포머 모델 살펴보기

```
sample_text = dataset["train"][1]["article"][:2000]
# 딕셔너리에 각 모델이 생성한 요약을 저장
summaries = {}
```

살펴볼 모델 구조는 최대 입력 크기는 다르지만 동일한 입력을 사용하고 출력을 비교하기 위해

입력 텍스트 2000자 제한

NLTK 패키지

요약에서는 관례적으로 요약 문장을 **줄바꿈**으로 나눔

마침표마다 그 뒤에 줄바꿈 토큰을 추가하면 U.S. 같은 문자열을 처리 XX

따라서 **NLTK** 패키지에 있는 구두점을 구별하는 더 정교한 알고리즘 사용

▼ NLTK

```
import nltk
from nltk.tokenize import sent_tokenize
# NLTK의 punkt는 구두점에 기반한 토크나이저로 사전 훈련된 모델
nltk.download("punkt")

string = "The U.S. are a country. The U.N. is an organizat
# 주어진 문자열을 문장으로 토큰화하여 문장의 리스트를 반환
sent_tokenize(string)
```

```
['The U.S. are a country.', 'The U.N. is an organization.']
```

: 이쁘게 잘 구분해줌 ~~

6.2.1 요약 기준 모델

기사를 **요약**하는 일반적인 기준 모델은 단순히 기사에서 **맨 처음 문장 세 개**를 선택하는 것

▼ NLTK 토크나이저로 구현

```
def three_sentence_summary(text):
    return "\n".join(sent_tokenize(text)[:3])
# summaries 딕셔너리의 baseline 키에 sample_text의 세 문장 요약
summaries["baseline"] = three_sentence_summary(sample_text
```

이런 기준 모델은 NLTK 문장 토크나이저 >> `sent_tokenize` 함수로 쉽게 구현할 수 있음

6.2.2 GPT-2

TL;DR

GPT-2 모델의 입력 텍스트 뒤에 **TL;DR**을 추가하면 요약을 생성

too long; didn't read의 약어임 ㅋㅋ

레딧 같은 사이트에서 긴 포스트를 짧게 요약할 때 종종 사용

🤗 트랜스포머스의 pipeline() 함수로 원본 논문 방식 재현하며 요약 작업 해보기

▼ **텍스트 생성 파이프라인 만들고 GPT-2 모델 로드**

```
from transformers import pipeline, set_seed

# 랜덤 시드를 설정하여 재현성 보장
set_seed(42)

# 코랩의 경우 gpt2-xl을 사용하면 메모리 부족 에러가 발생 >> gpt2-la
# 텍스트 생성 파이프라인 설정
pipe = pipeline("text-generation", model="gpt2-large")

# 샘플 텍스트에 TLDR 추가
gpt2_query = sample_text + "\nTL;DR:\n"
pipe_out = pipe(gpt2_query, max_length=512, clean_up_token
# 텍스트를 다시 문장으로 분할하고 gpt2 키에 요약 추가
summaries["gpt2"] = "\n".join(
    sent_tokenize(pipe_out[0]["generated_text"][len(gpt2_q
```

나중의 비교를 위해 출력에서 입력 텍스트의 다음 부분을 요약으로 추출해 딕셔너리에 저장

- `clean_up_tokenization_spaces=True` : 생성된 텍스트의 토큰화 공백을 정리

6.2.3 T5

- NLP에서 포괄적인 전이 학습 연구를 수행해 모든 작업을 텍스트-투-텍스트 작업으로 구성하는 범용의 트랜스포머 아키텍처를 만듦

- T5의 체크포인트는 요약을 포함해 여러 작업에서 (마스킹된 단어를 재구성하기 위한) 비지도 학습 데이터와 학습 데이터를 **섞은** 데이터로 훈련 됨

- 따라서 미세 튜닝 없이 체크포인트를 사전 훈련에 썼던 것과 동일한 프롬프트를 사용해 바로 요약에 사용할 수 있음

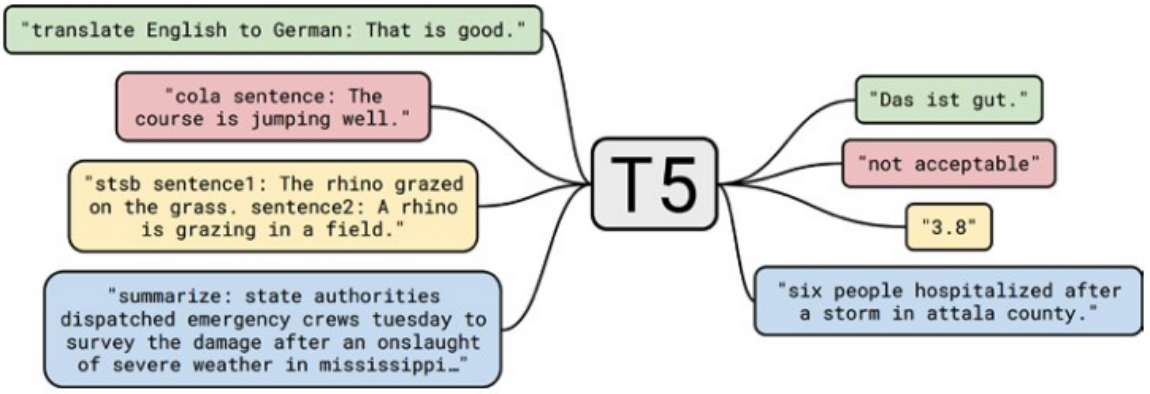


그림 6-1 T5의 텍스트–투–텍스트 프레임워크(Colin Raffel 제공). 번역과 요약 외에 CoLA linguistic acceptability와 STSB semantic similarity 작업이 있습니다.

- 문서 요약에 사용할 모델의 입력 포맷 : “summarize: <ARTICLE>” (그림에서 파랑이)
- 번역에 사용할 모델의 입력 포맷 : “translate English to German: <TEXT>” (그림에서 초록이)

▼ 요약을 위해 pipeline() 함수로 T5 로드

```
pipe = pipeline("summarization", model="t5-large")
pipe_out = pipe(sample_text)
summaries["t5"] = "\n".join(sent_tokenize(pipe_out[0]["sum
```

얘는 입력을 **텍스트-투-텍스트** 형식으로 제공하므로 앞에 `summarize`를 붙일 필요가 없음

단순히 요약하고자 하는 텍스트만 입력으로 제공하면 됨!

6.2.4 BART

인코더-디코더 구조 사용하는 모델로 문법적으로 손상된 문장을 모델에 제공하고,

모델이 이를 올바르게 이해하고 보완할 수 있는지를 평가하는 식의 훈련 진행

: BERT와 GPT-2의 사전 훈련 방식 결합 사용

▼ **BART**

```
pipe = pipeline("summarization", model="facebook/bart-larg
pipe_out = pipe(sample_text)
summaries["bart"] = "\n".join(sent_tokenize(pipe_out[0]["s
```

6.2.4 PEGASUS

BART와 동일한 인코더-디코더 트랜스포머로

여러 문장으로 구성된 텍스트에서 마스킹된 문장을 예측하는 것을 사전 훈련 목표로 훈련됨

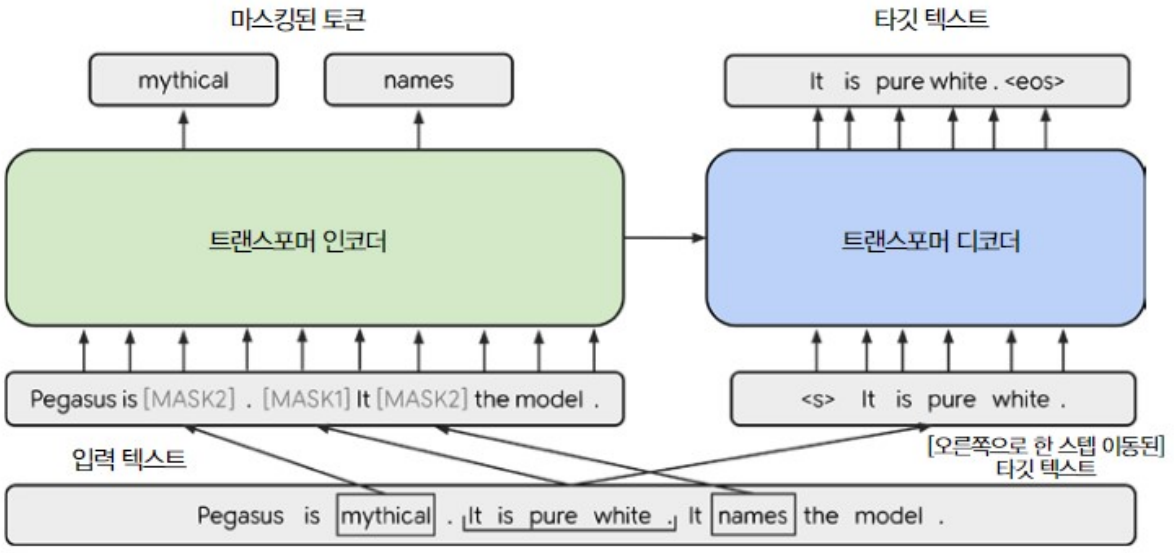


그림 6-2 PEGASUS 아키텍처(Jingqing Zhang 등 제공)

일반적인 언어 모델링보다 **요약에 특화**된 사전 훈련 목표를 찾기 위해 대규모 말뭉치에서

중복을 측정하는 요약 평가 지표를 사용해 주변 문단의 내용을 대부분 담은 문장을 **자동**으로 식별.

이런 문장을 재구성하도록 PEGASUS 모델을 사전 훈련해 최고 수준의 텍스트 요약 모델을 얻음

▼ **PEGASUS**

```
pipe = pipeline("summarization", model="google/pegasus-cnn
pipe_out = pipe(sample_text)

# <n>이 줄바꿈하는 특수토큰
# .<n>을 실제 줄바꿈 문자 \n 으로 대체시켜서 딕셔너리에 할당
summaries["pegasus"] = pipe_out[0]["summary_text"].replace
```

이 모델은 줄바꿈하는 특수 토큰이 있으므로 `sent_tokenize()` 함수를 사용할 필요 X

7.1 리뷰 기반 QA시스템 구축하기

가장 일반적인 QA 방법은 문서에 있는 텍스트 일부를 질문의 답으로 추출하는 **추출적 QA**

[QA 시스템의 기초]

1. 관련된 문서 추려내기
2. 그 문서에서 원하는 답 추출하기

: **리뷰**를 통해 어떻게 트랜스포머가 텍스트에서 의미를 추출하는 **독해 모델**로 동작하는지 알아보기

7.1.1 데이터셋

- 10,000여개의 영어 고객 리뷰로 구성된 SubjQA
- 트립어드바이저, 음식점, 영화, 책, 전자 제품, 식료품 분야로 구성

📣 핵심은 이 데이터셋의 대부분의 질문과 답이 **주관적**이라는 것

: 모든 리뷰가 사용자의 개인 경험에 의존함

따라서 사실 여부가 명확한 질문의 답을 찾는 작업보다 어려움

▼ 예제

Product: Nokia Lumia 521 RM-917 8GB

Query: Why is the camera of poor quality?

Review: Item like the picture, fast deliver 3 days well packed, good quality for the price. The camera is decent (as phone cameras go), There is no flash though ...

그림 7-2 제품에 관한 질문과 해당 리뷰(밑줄 부분이 답입니다)

1. 나쁜 품질에 대한 쿼리 입력

   : 나쁜 품질의 기준은 주관적!

2. 쿼리의 중요 부분이 리뷰에 전혀 나타나지 않음

: 이는 키워드 검색이나 입력 질문을 재구성하는 간단한 방식으로는 답을 찾지 못한다는 의미

따라서 SubjQA 데이터셋은 리뷰 기반 QA 모델을 벤치마킹하기 좋다

QA 시스템은 일반적으로 쿼리에 응답할 때 참조하는 데이터의 **도메인**으로 분류

- 클로즈드 도메인 QA : **단일** 카테고리 같은 좁은 주제에 대한 질문을 다룸
- 오픈 도메인 QA : **전체** 제품 카탈로그 같은 모든 주제에 대한 질문을 다룸

주로 클로즈드 도메인 QA가 오픈 도메인 QA보다 검색하는 문서의 개수가 적음

▼ 어떤 서브셋을 사용할 수 있는지 확인

```
from datasets import get_dataset_config_names

domains = get_dataset_config_names("subjqa")
domains
```

```
['books', 'electronics', 'grocery', 'movies', 'restaurants', 'tripadvisor']
```

▼ 위의 서브셋 중 전자 제품용 QA 시스템 구축하기

```
from datasets import load_dataset

subjqa = load_dataset("subjqa", name="electronics")
```

`load_dataset()` 함수의 `name` 매개변수에 `electronics` 값을 전달

▼ answers 열에 있는 행 하나 확인

```
print(subjqa["train"]["answers"][1])
```

허브에 있는 여느 질문 답변 데이터셋과 마찬가지로

SubjQA는 각 질문의 답을 **중첩된 딕셔너리**에 저장 (아래 참고)

```
dataset = {
    "question1": {
        "text": "What is the capital of France?",
        "answer": "Paris"
    },
    "question2": {
        "text": "Who wrote 'Romeo and Juliet'?",
        "answer": "William Shakespeare"
    }
}
```

```
{'text': ['Bass is weak as expected', 'Bass is weak as expected, even with EQ adjusted up'], 'answer_start': [1302, 1302], 'answer_subj_level': [1, 1], 'ans_subj_score': [0.5083333253860474, 0.5083333253860474], 'is_ans_subjective': [True, True]}
```

- 답은 `text` 필드에 저장됨
- `answer_start`에 시작 문자의 인덱스 [1302, 1302]가 있음

▼ 데이터셋을 쉽게 탐색하기 위해 각 분할을 DataFrame으로 변환

```
import pandas as pd
```

```
# dfs 딕셔너리 만들기 / 키 >> 데이터셋의 split, 값 >> 해당 split의
dfs = {split: dset.to_pandas() for split, dset in subjqa.f

# id 열을 기준으로 고유한 질문 수 카운트
for split, df in dfs.items():
  print(f"{split}에 있는 질문의 개수: {df['id'].nunique()}")
```

- `flatten()` 메서드로 중첩된 열을 펼치기

```
train에 있는 질문의 개수: 1295
test에 있는 질문의 개수: 358
validation에 있는 질문의 개수: 255
```

샘플은 총 1,908개로 비교적 작은 데이터셋이다

▫ SubjQA 데이터셋의 열 이름과 설명

열 이름	설명
title	각 제품에 연관된 ASIN(Amazon Standard Identification Number
question	질문
answers.text	사람이 레이블링한 리뷰 텍스트 일부
answers.answer_start	답이 시작하는 문자 인덱스
context	고객 리뷰

- ASIN : Amazon standard Identification Number

▼ 위에서 소개한 열을 기준으로 훈련 샘플 살펴보기

```
qa_cols = ["title", "question", "answers.text",
           "answers.answer_start", "context"]
# 임의로 2개 행을 선택하여 sample_df로 만들기
```

```
sample_df = dfs["train"][qa_cols].sample(2, random_state=7
sample_df
```

- `sample()` 메소드를 사용해 랜덤하게 샘플 선택하기
- `random_state` 는 처음에 `set_seed`랑 같은 역할 >> 재현 가능한 난수 생성

	title	question	answers.text	answers.answer_start	context
791	B005DKZTMG	Does the keyboard lightweight?	[this keyboard is compact]	[215]	I really like this keyboard. I give it 4 star...
1159	B00AAIPT76	How is the battery?	[]	[]	I bought this after the first spare gopro batt...

1. 질문이 문법적으로 틀림
2. 빈 `answers.text` 항목에는 리뷰에서 답을 찾지 못해 답변이 불가능한 질문이 담김
3. 시작 인덱스와 답변 길이를 사용해 리뷰에서 답변에 해당하는 텍스트 추출 가능

▼ 해당 코드

```
start_idx = sample_df["answers.answer_start"].iloc[0
# 시작 인덱스 + 해당하는 답변의 텍스트 길이 >> 끝 인덱스
end_idx = start_idx + len(sample_df["answers.text"].
# 답변에 해당하는 부분 문자열 추출
sample_df["context"].iloc[0][start_idx:end_idx]
```

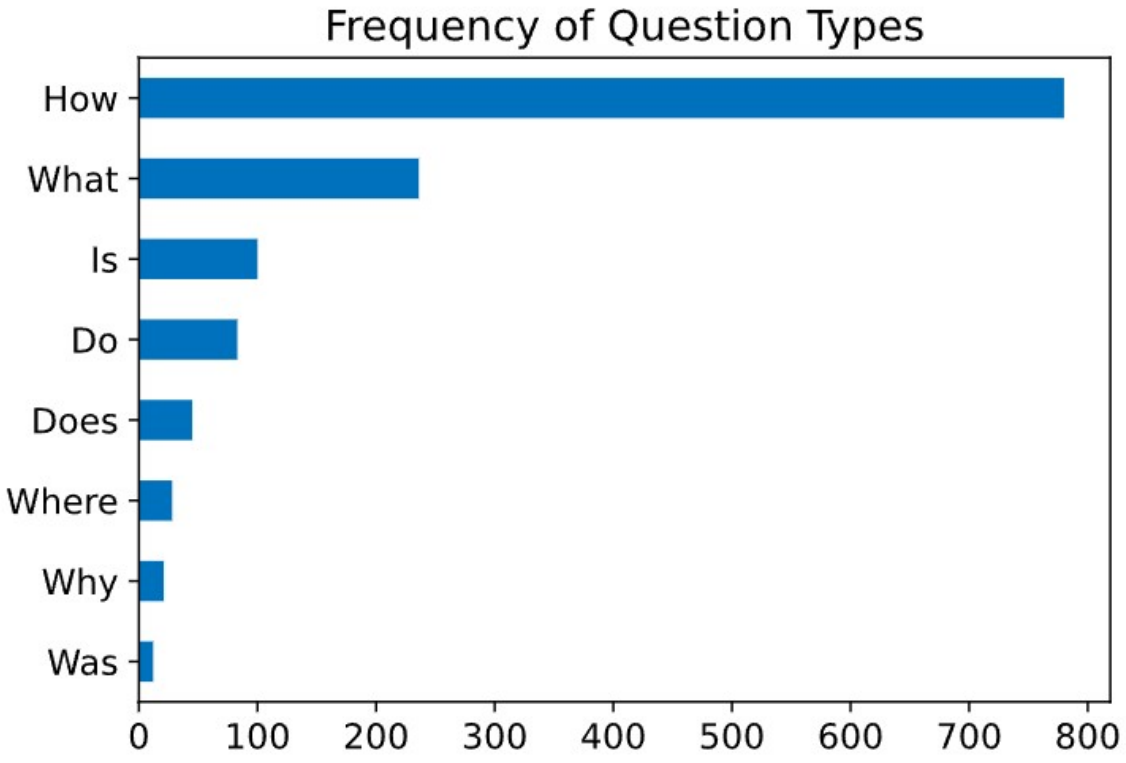
```
'this keyboard is compact'
```

▼ 훈련 세트에 대략 어떤 종류의 질문이 있는지 알아보기

```
# 흔한 단어로 시작하는 질문의 개수 카운트
counts = {}
question_types = ["What", "How", "Is", "Does", "Do", "Was"
```

```
for q in question_types:
    counts[q] = dfs["train"]["question"].str.startswith(q)

pd.Series(counts).sort_values().plot.barh()
plt.title("Frequency of Question Types")
plt.show()
```



How, What, Is 순으로 가장 많음

▼ 해당 질문들 자세히 확인해보기

```
for question_type in ["How", "What", "Is"]:
    for question in (
        dfs["train"][dfs["train"].question.str.startswith(
        .sample(n=3, random_state=42)['question']):
        print(question)
```

```
How is the camera?
How do you like the control?
How fast is the charger?
What is direction?
What is the quality of the construction of the bag?
What is your impression of the product?
Is this how zoom works?
Is sound clear?
Is it a wireless keyboard?
```

스탠퍼드 질문 답변 데이터셋 (SQuAD)

- SubjQA의 형식인 **(질문, 리뷰, [답변])**은 추출적 데이터셋에서 널리 사용되는데,
  이는 스탠퍼드 질문 답변 데이터셋에서 처음 사용된 방식임
- 컴퓨터가 텍스트 문단을 읽고 관련 질문에 답변이 가능한지 테스트할 때 많이 사용
- 수백 개의 위키피디아 영어 문서에서 샘플링하고 크라우드 소싱을 통해
  각 문단에서 일련의 질문과 답을 생성해 만듦
- 최초 버전은 각 질문의 답이 해당 구절 안에 반드시 존재했는데,
  작업의 난도를 높이기 위해 주어진 텍스트와 관련되지만 텍스트만으로는 답변할 수 없는
  적대적인 질문으로 SQuAD 1.1을 보강해 SQuAD 2.0 만듦

7.1.2 텍스트에서 답 추출하기

QA 시스템에서는 먼저 리뷰에 있는 텍스트에서
답변에 사용할 만한 부분을 식별해낼 방법을 찾아야 함

📣 [방법]

- 지도 학습 문제로 구성하기
- QA 작업을 위해 텍스트를 토큰화하고 인코딩하기
- 모델의 최대 문맥 크기를 초과하는 긴 텍스트 다루기

>> 문제를 구성하는 방법부터 알아보자!

범위 분류

문제를 범위 분류 작업으로 구성하는 방법은

텍스트에서 답을 추출하는 가장 일반적인 방법

그림 7-4 QA 작업을 위한 범위 분류 헤드

- 이 작업에서 모델이 예측해야 하는 레이블은 답변 범위에 해당하는

시작 토큰과 종료 토큰임

- 훈련 세트에는 비교적 적은 1,295개 샘플만 있으므로 SQuAD 같은 대규모 QA 데이터셋에서 미세 튜닝한 언어 모델로 시작하는 편이 좋음

이는 이전 장처럼 사전 훈련된 모델을 사용해 작업에 특화된 **헤드를** 미세 튜닝하는 일반적인 방식과 다름
2장에서는 클래스 개수가 현재 데이터셋과 **달라서** 분류 헤드를 미세 튜닝해야 했지만 추출적 QA는 레이블 구조가 데이터셋에 따라 달라지지 않기 때문에 미세 튜닝한 모델로 시작해도 무방!

🤗 허브에서 모델 탭에 'squard'를 검색해 추출적 QA 목록 확인하면

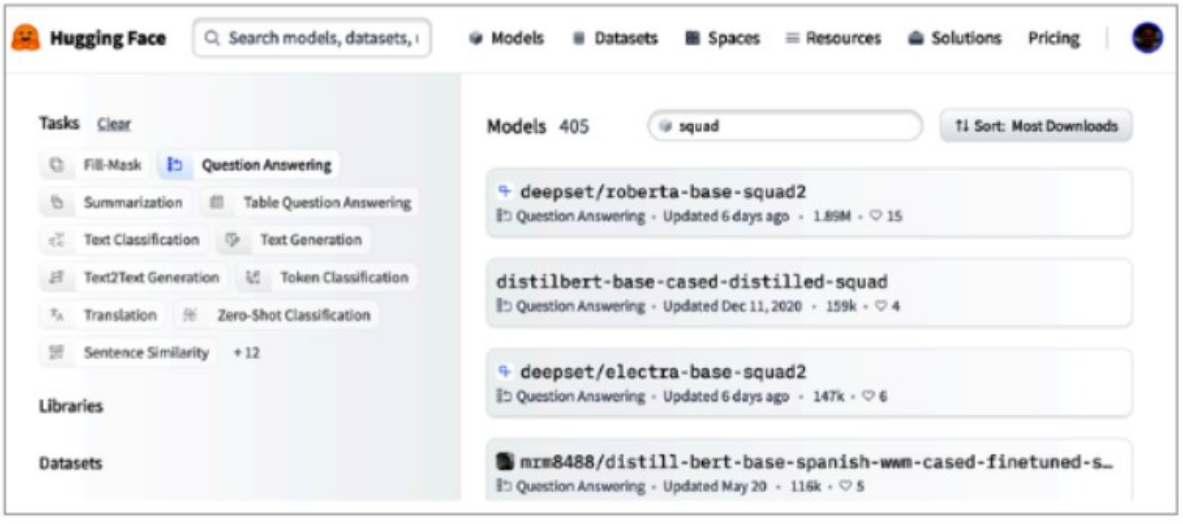


그림 7-5 허깅페이스 허브에서 추출적 QA 모델 고르기

350개가 넘는 QA 모델을 볼 수 있음!

이중 어떤 모델을 선택해야 할까?

1. 주어진 말뭉치가 단일 언어인지 다중 언어인지에 따라

2. 제품 환경에서 모델을 실행하는 제약 조건에 따라

등등 여러 요인에 따라 달라진다

표 7-2 SQuAD 2.0에서 미세 튜닝된 트랜스포머 모델

트랜스포머	설명	파라미터 개수	SQuAD 2.0에서 F_1-점수
MiniLM	99% 성능을 유지하면서 두 배 빠른 BERT 기반의 압축 버전입니다.	66M	79.5
RoBERTa-base	RoBERTa 모델이 BERT보다 성능이 더 높고 단일 GPU를 사용해 대부분의 QA 데이터셋에서 미세 튜닝이 가능합니다.	125M	83.0
ALBERT-XXL	SQuAD 2.0에서 최고의 성능을 내지만 계산 집약적이고 배포가 어렵습니다.	235M	88.1
XLM-RoBERTa-large	강력한 제로샷zero shot 성능을 내며 100개 언어를 위한 다중 언어 모델입니다.	570M	83.8

이 장에서는 훈련 속도가 빠른 미세 튜닝한 **MiniLM** 모델 사용

QA를 위한 텍스트 토큰화

늘 그렇듯 텍스트를 인코딩할 토크나이저가 먼저 필요하겟지

▼ 허브에서 MiniLM 모델의 체크포인트 로드

```
from transformers import AutoTokenizer

model_ckpt = "deepset/minilm-uncased-squad2"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
```

▼ 모델 작동을 위해 텍스트에서 답 추출하기

```
question = "How much music can this hold?"
context = """An MP3 is about 1 MB/minute, so about 6000 ho
file size."""

# 두 값을 모두 토크나이저에 전달
inputs = tokenizer(question, context, return_tensors="pt")
```

- 추출적 QA 작업에서는 입력을 (질문, 문맥) 쌍 형태로 제공

  >> 두 값을 모두 토크나이저에 전달

- 파이토치 Tensor 객체가 반환되므로 이를 사용해 정방향 패스 실행

[토큰화된 입력 표]

input_ids	101	2129	2172	2189	2064	2023	_	5834	2006	5371	2946	1012	102
token_type_ids	0	0	0	0	0	0	_	1	1	1	1	1	1
attention_mask	1	1	1	1	1	1	_	1	1	1	1	1	1

- `token_type_ids` 텐서: 입력에서 어떤 부분이 질문과 문맥에 해당하는지 나타냄 (0은 질문 토큰, 1은 문맥 토큰)

(참고) `token_type_ids` 텐서가 모든 트랜스포머 모델에 있는 것은 아님!

MiniLM같은 BERT 유사 모델의 경우 사전 훈련하는 동안

다음 문장 예측 작업을 통합하기 위해? 이 텐서를 사용하기도 함

>> 세그먼트 임베딩을 말하는 듯

질문 문장이랑 문맥 문장을 함께 처리할 때 시작과 끝을 구분하려고 세그먼트 임베딩 부여

▼ 토크나이저가 QA 작업에서 입력을 포맷팅하는 방법 이해하기

```
# input_ids 텐서 디코딩하기
print(tokenizer.decode(inputs["input_ids"][0]))
```

QA 샘플마다 다음 포맷으로 입력이 구성된다

```
[CLS] how much music can this hold? [SEP] an mp3 is about 1 mb / minute, so
about 6000 hours depending on file size. [SEP]
```

[CLS] 질문 토큰 [SEP] 문맥 토큰 [SEP]

- 첫번째 [SEP] 토큰의 위치는 `token_type_ids`에 의해 결정

  >> 질문이니까 그 값이 0이면 된다는 것임

▼ QA 헤드와 함께 모델 객체를 초기화하고 입력을 정방향 패스에 통과시킴

```
import torch
from transformers import AutoModelForQuestionAnswering

model = AutoModelForQuestionAnswering.from_pretrained(mode

with torch.no_grad():
    outputs = model(**inputs)
print(outputs)
```

```
QuestionAnsweringModelOutput(loss=None, start_logits=tensor([[-0.9862, -4.7750,
-5.4025, -5.2378, -5.2863, -5.5117, -4.9819, -6.1880,
         -0.9862,  0.2596, -0.2144, -1.7136,  3.7806,  4.8561, -1.0546, -3.9097,
         -1.7374, -4.5944, -1.4278,  3.9949,  5.0391, -0.2018, -3.0193, -4.8549,
         -2.3108, -3.5110, -3.5713, -0.9862]]), end_logits=tensor([[-0.9623,
-5.4733, -5.0326, -5.1639, -5.4278, -5.5151, -5.1749, -4.6233,
         -0.9623, -3.7855, -0.8715, -3.7745, -3.0161, -1.1780,  0.1758, -2.7365,
          4.8934,  0.3046, -3.1761, -3.2762,  0.8937,  5.6606, -0.3623, -4.9554,
         -3.2531, -0.0914,  1.6211, -0.9623]]), hidden_states=None,
attentions=None)
```

: QA 헤드는 `QuestionAnsweringModelOutput` 객체를 출력

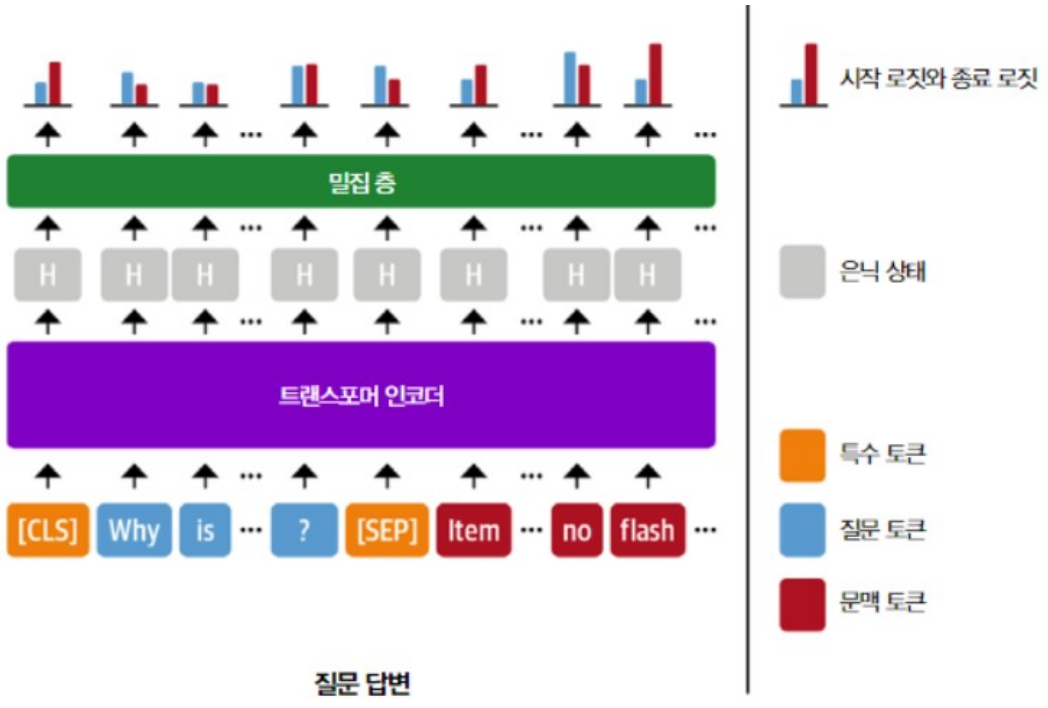


그림 7-4 QA 작업을 위한 범위 분류 헤드

- QA헤드는 인코더의 은닉 상태를 받아 시작과 종료 범위의 로짓을 계산하는 **선형층**에 해당
- 이는 QA 작업을 4장 개체명 인식과 비슷하게 토큰 분류 형태로 다룬다는 의미

▼ 출력을 답의 범위로 변환하기 위해 시작과 종료 토큰의 로짓 가져오기

```
start_logits = outputs.start_logits
end_logits = outputs.end_logits
```

▼ 로짓의 크기를 입력 ID와 비교하기

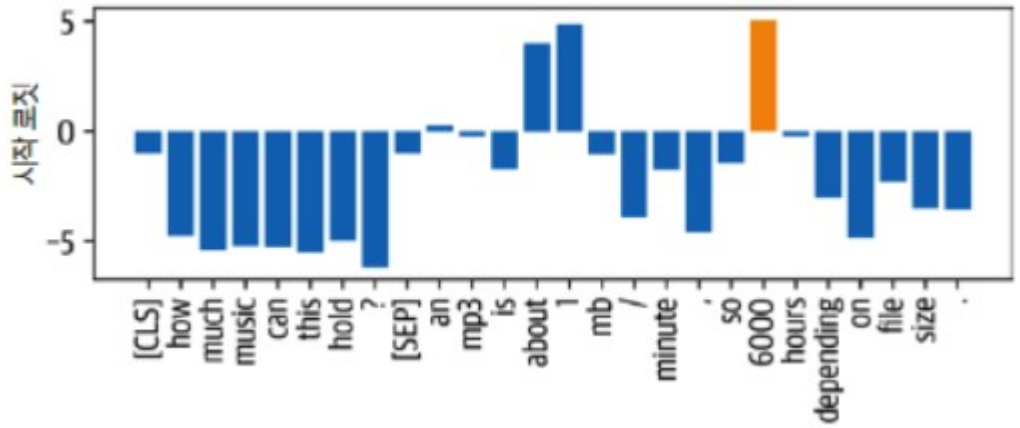
```
print(f"입력 ID 크기: {inputs.input_ids.size()}")
print(f"시작 로짓 크기: {start_logits.size()}")
print(f"종료 로짓 크기: {end_logits.size()}")
```

```
입력 ID 크기: torch.Size([1, 28])
시작 로짓 크기: torch.Size([1, 28])
종료 로짓 크기: torch.Size([1, 28])
```

- 입력 ID 크기: 이는 모델에 주어진 텍스트의 길이
- 시작 및 종료 로짓 : 각 토큰이 답변의 시작과 끝에 해당하는 지를 나타내는 확률
- 입력 ID 크기와 시작 및 종료 로짓 크기가 같다

  >> 모델이 각 토큰에 대해 답변의 시작과 끝을 예측할 수 있도록 하는 것

아래 사진에서 큰 양수 로짓은 가능성이 높은 시작과 종료 토큰 후보에 해당



- 모델이 1과 6000에 가장 높은 시작 토큰 로짓을 할당
  - 질문이 어떤 양 (How much music can this hold?)에 관한 것이기 때문

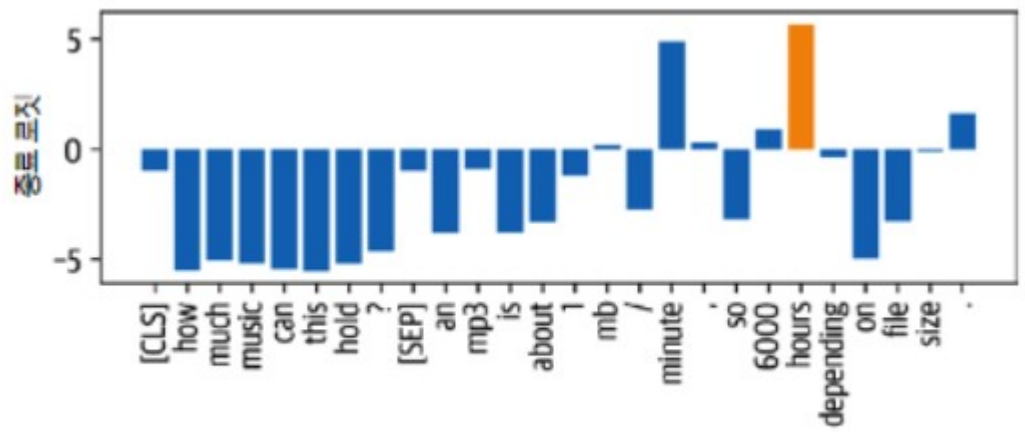


그림 7-6 시작 토큰과 종료 토큰에 대한 예측 로짓. 오렌지색 토큰이 점수가 가장 높은 토큰입니다.

- 마찬가지로 로짓이 가장 높은 종료 토큰은 시간과 관련된 minute과 hours

▼ 최종 답을 얻고자 시작, 종료 토큰의 로짓에 argmax 함수 적용 후 입력에서 이 범위 슬라이싱

```
import torch

# 시작 로짓에서 가장 높은 확률을 갖는 위치 찾기
start_idx = torch.argmax(start_logits)
# 토큰의 인덱스이므로 실제 답변에 1을 더해줌 >> 특수 토큰 등의 이유
end_idx = torch.argmax(end_logits) + 1
answer_span = inputs["input_ids"][0][start_idx:end_idx]
# 토큰을 문자열로 디코딩
answer = tokenizer.decode(answer_span)
print(f"질문: {question}")
print(f"답변: {answer}")
```

```
질문: How much music can this hold?
답변: 6000 hours
```

성공성공. 🤗 트랜스포머는 모든 전처리 단계와 후처리 단계가

전용 파이프라인 안에 들어있음

▼ 파이프라인 초기화

```
from transformers import pipeline
# 토크나이저와 미세 튜닝된 모델 전달해 파이프라인 초기화
pipe = pipeline("question-answering", model=model, tokeniz
# 초기화된 파이프라인을 사용하여 질문 답변 수행
pipe(question=question, context=context, topk=3)
```

: 답변 후보들을 포함하는 딕셔너리를 반환

```
[{'score': 0.2651616334915161, 'start': 38, 'end': 48, 'answer': '6000 hours'},
 {'score': 0.22082962095737457,
  'start': 16,
  'end': 48,
  'answer': '1 MB/minute, so about 6000 hours'},
 {'score': 0.10253521054983139,
  'start': 16,
  'end': 27,
  'answer': '1 MB/minute'}]
```

- 이 파이프라인은 답변 외에도 로짓에 소프트맥스를 적용해

  모델이 추정한 확률을 `score` 필드로 제공함

  - 이는 한 문맥에서 여러 답을 비교할 때 편리함

- `topk` 매개변수를 사용하면 모델이 여러 개의 답을 예측 >> 여기서는 3개!

▼ 답변 불가능한 질문

```
pipe(question="Why is there no data?", context=context,
     handle_impossible_answer=True)
```

- `answers.answer.start` 가 비어있는 답변이 불가능한 질문의 경우,

[CLS] 토큰에 높은 시작 점수와 종료 점수를 할당

```
{'score': 0.9068413972854614, 'start': 0, 'end': 0, 'answer': ''}
```

: 파이프 라인은 이 출력을 **빈 문자열**로 매핑

이 예시에서 해당 로짓에 argmax를 적용해 시작과 종료 인덱스를 얻음.
그러나 이 방법은 문맥 대신 **질문**에 속한 토큰을 선택해서
범위가 **벗어난** 답을 생성할 때도 있음! 실전에서는 범위 내에 있는지,
시작 인덱스가 종료 인덱스 앞에 있는지 등의 다양한 제약 조건을 따라
파이프라인이 최상의 시작 인덱스와 종료 인덱스의 조합을 계산함!

긴 텍스트 다루기

독해 모델의 결점 하나는 종종 문맥에 있는 토큰이 모델의 최대 시퀀스 길이를 초과한다는 것

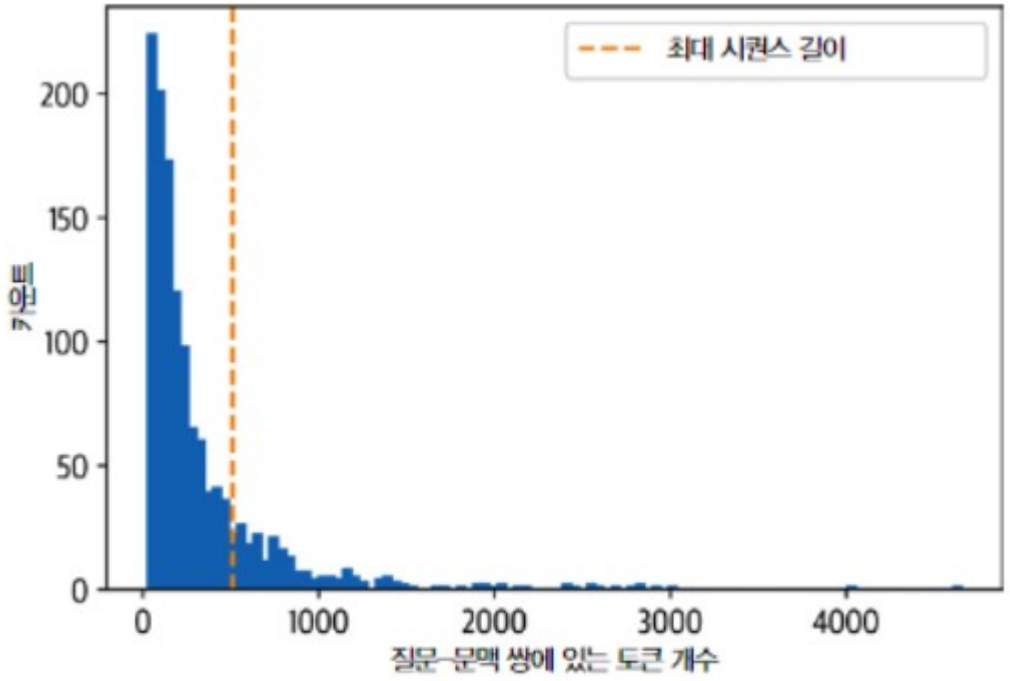


그림 7-7 SubjQA 훈련 세트에 있는 질문-문맥 쌍의 토큰 분포

- 그림을 보면 훈련 세트의 상당 부분이 MiniLM의 문맥 크기인
  512 토큰에 맞지 않는 질문-문맥 쌍을 가진다

텍스트 분류에서는 정확한 예측을 생성하기 위해 [CLS] 토큰 임베딩에
충분한 정보가 담겼다고 가정하고 그냥 긴 텍스트를 잘랐음

그러나, QA 작업에서는 이런 전략이 문제를 일으킨다
질문의 답이 문맥의 끝에 있으면 텍스트를 잘랐을 때 답이 삭제됨
이 문제를 다루기 위해 등장한 방법이 **슬라이딩 윈도**

- **슬라이딩 윈도**

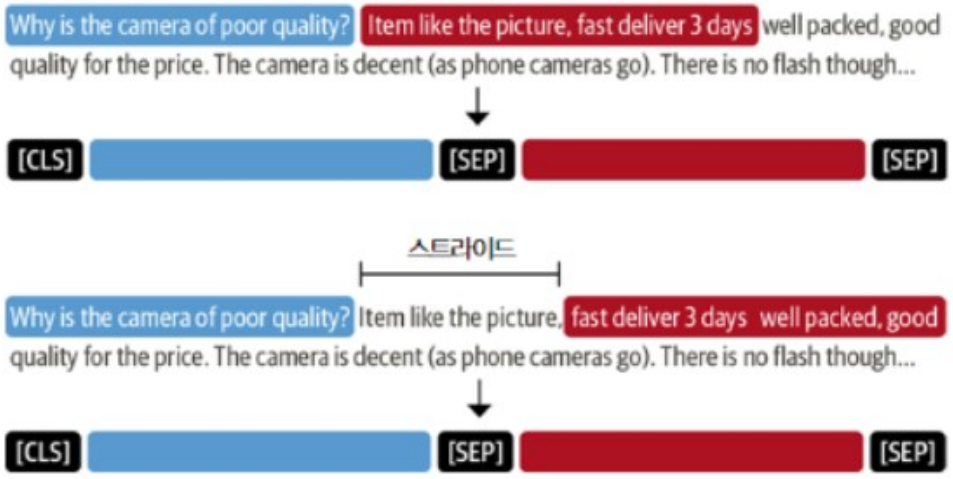


그림 7-8 긴 문서에서 슬라이딩 윈도가 여러 개의 질문–문맥 쌍을 만드는 방법. 첫 번째(파란색) 막대는 질문에 해당하고 두 번째(붉은색) 막대는 각 윈도에서 캡처한 문맥입니다.

입력에 슬라이딩 윈도를 적용하면 각 윈도는 모델의 문맥 크기에 맞는

토큰 리스트가 된다

- window : 중심 단어 앞 뒤로 몇 개의 단어를 볼 것인지에 대한 범위

▼ 슬라이딩 윈도의 작동 방식

```
example = dfs["train"].iloc[0][["question", "context"]]
tokenized_example = tokenizer(example["question"], example
                              return_overflowing_tokens=Tr
                              stride=25)
```

- 🤗 트랜스포머스는 토크나이저에 `return_overflowing_tokens=True`를 설정해 슬라이딩 윈도를 만든다.
- 슬라이딩 윈도의 크기는 `max_seq_length` 매개변수로 조정
- 스트라이드의 크기는 `doc_stride` 로 조정

  왜 이걸로 조절 안햇지? >> 해보니까 오류 남

▼ 각 윈도의 토큰 개수 확인하기

```
for idx, window in enumerate(tokenized_example["input_ids"
    print(f"#{idx} 윈도에는 {len(window)}개의 토큰이 있습니다.")
```

이 경우 윈도마다 input_ids의 리스트를 하나씩 얻는다

```
#0 윈도에는 100개의 토큰이 있습니다.
#1 윈도에는 88개의 토큰이 있습니다.
```

▼ input_ids를 디코딩해 두 윈도가 어디서 겹치는지 확인

```
for window in tokenized_example["input_ids"]:
    print(f"{tokenizer.decode(window)} \n")
```

```
[CLS] how is the bass? [SEP] i have had koss headphones in the past, pro 4aa and
qz - 99. the koss portapro is portable and has great bass response. the work
great with my android phone and can be " rolled up " to be carried in my
motorcycle jacket or computer bag without getting crunched. they are very light
and don't feel heavy or bear down on your ears even after listening to music
with them on all day. the sound is [SEP]

[CLS] how is the bass? [SEP] and don't feel heavy or bear down on your ears even
after listening to music with them on all day. the sound is night and day better
than any ear - bud could be and are almost as good as the pro 4aa. they are "
open air " headphones so you cannot match the bass to the sealed types, but it
comes close. for $ 32, you cannot go wrong. [SEP]
```

>> 0 윈도 마지막 줄이랑 1 윈도 첫번째 부분이 겹침

7.1.3 헤이스택을 사용해 QA 파이프라인 구축하기

엔드-투-엔드 QA 파이프라인을 만드는데 필요한 다른 구성 요소 알아보기

▫ 배경

앞에서는 질문과 문맥을 모두 모델에 제공했으나 실제 사용자는 **질문**만 제공함

따라서 말뭉치에 있는 전체 리뷰 중 관련된 텍스트를 선택할 방법이 필요!

보통은 해당 제품 리뷰를 모두 연결해 하나의 긴 문맥으로 만들어 모델에 주입

간단하지만 문맥이 너무 길어져 사용자 쿼리에 대한 **레이턴시**를 수용하지 못한다는 단점이 있음

- 레이턴시 : 지연 시간

최신 QA 시스템은 **리트리버-리더** 구조를 기반으로 문제를 처리

▫ 리트리버

쿼리에서 관련된 문서를 추출하는 친구

- 희소 리트리버 : 단어 빈도를 사용해 각 문서와 쿼리를 희소 벡터로 표현

  희소 벡터는 대부분 원소가 0이고 이 벡터의 내적을 계산해 쿼리와 문서의 **관련성**을 결정

- 밀집 리트리버 : 트랜스포머 같은 인코더를 사용해 쿼리와 문서를 문맥화된 임베딩으로 표현

  이런 임베딩이 **의미**를 인코딩하므로 밀집 리트리버는 쿼리의 **내용**을 이해해 검색 정확도를 향상

▫ 리더

리트리버가 제공한 문서에서 답을 추출

리더는 대개 독해 모델이지만 이 장의 끝에서 자유 형식의 답변을 생성하는 모델도 살펴볼 것임

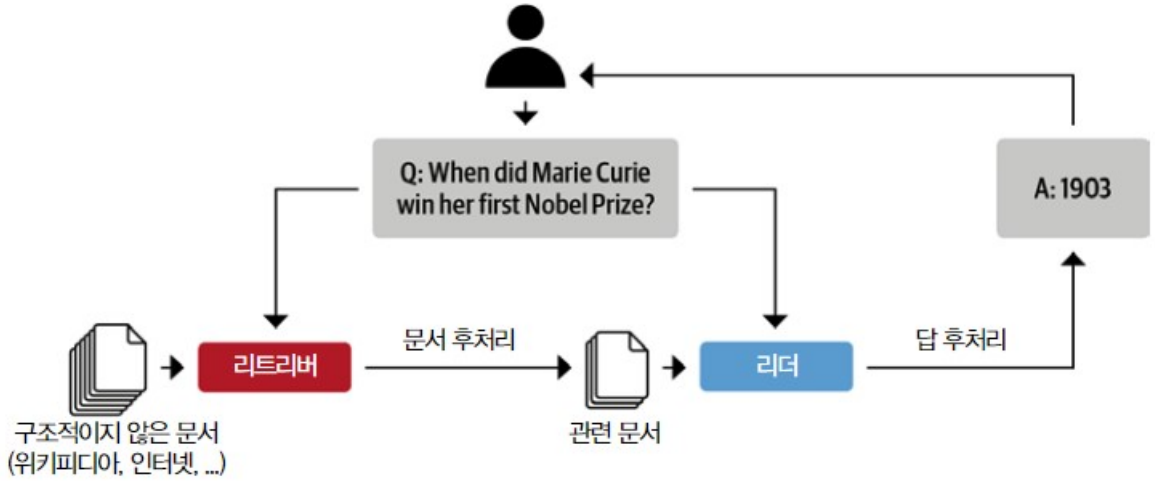


그림 7-9 최신 QA 시스템의 리트리버–리더 아키텍처

리트리버가 추출한 문서나 리더가 추출한 답에 **후처리**를 적용하는 구성 요소도 있음

▼ 예시

> 📣 예를 들어 리더에 혼동을 일으키는 노이즈나 관련 없는 요소를 제외하기 위해 추출된 문서를 다시 랭킹하는 작업을 진행한다. 마찬가지로 긴 문서에 있는 여러 구절에서 정답이 나오면 종종 리더의 답을 후처리할 필요가 있음

▫ 헤이스택 라이브러리

헤이스택은 리트리버-리더 아키텍처 기반으로

헤이스택으로 QA 파이프라인을 만들 때는 두 가지 구성 요소가 더 필요

문서 저장소

쿼리 시점에 리트리버로 제공하는 문서와 메타 데이터를 저장하는 문서 전용 데이터베이스

파이프라인

사용자 쿼리가 잘 흘러가도록 QA 시스템의 모든 구성 요소를 결합하고

여러 리트리버에서 추출한 문서를 합치는 등의 기능을 함

프로토타입 QA 파이프라인 구축하는 방법 알아보기

▫ 문서 저장소 초기화하기

헤이스택에서 사용 가능한 문서 저장소는 다양, 저장소마다 조합할 수 있는 전용 리트리버가 있음

표 7-3 헤이스택 리트리버와 문서 저장소의 호환성

	메모리	Elasticsearch	FAISS	Milvus
TF-IDF	Yes	Yes	No	No
BM25	No	Yes	No	No
Embedding	Yes	Yes	Yes	Yes
DPR	Yes	Yes	Yes	Yes

- 희소 리트리버 : TF-IDF, BM25
- 밀집 리트리버 : Embedding, DPR

이 장에서는 희소, 밀집 리트리버를 모두 살펴보므로 양쪽에 호환되는 **Elasticsearch** 사용

Elasticsearch

- 텍스트, 수치, 구조적, 비구조적 데이터를 포함해 다양한 데이터 타입을 처리하는 검색 엔진
- 대용량 데이터를 저장하고 전체 텍스트 검색으로 빠르게 필터링하므로

  특히 QA 시스템 개발과 잘 맞음
- 인프라 분석을 위한 업계 표준이라는 이점이 있음

>> 회사에서는 이미 일래스틱서치 클러스터가 있을 가능성이 높다

▼ 문서 저장소를 **초기화**하기 위해 일래스틱 서치 다운로드하고 설치하기

```
url = """https://artifacts.elastic.co/downloads/elasticsea
elasticsearch-7.9.2-linux-x86_64.tar.gz"""

# wget으로 최신 리눅스용 릴리스 다운로드하기
!wget -nc -q {url}
# tar 셸 명령으로 압축 풀기
!tar -xzf elasticsearch-7.9.2-linux-x86_64.tar.gz
```

▼ `Popen()` 함수를 이용해 새로운 프로세스 시작

```
import os
from subprocess import Popen, PIPE, STDOUT

# 백그라운드 프로세스로 일래스틱서치를 실행합니다
!chown -R daemon:daemon elasticsearch-7.9.2
es_server = Popen(args=['elasticsearch-7.9.2/bin/elasticse
                  stdout=PIPE, stderr=STDOUT, preexec_fn=l
# 일래스틱서치가 시작할 때까지 기다립니다
!sleep 30
```

- 이 서브프로세스를 백그라운드에서 실행하기 위해 `chown` 셸 명령을 사용
- `Popen()` 함수에서 `args`로 실행할 프로그램 지정
- `stderr=STDOUT`로 지정하면 동일한 파이프로 에러가 수집됨
- `preexec_fn` 매개변수에 사용할 서브프로세스의 아이디를 지정

▼ localhost로 HTTP 요청을 보내서 연결 테스트

```
!curl -X GET "localhost:9200/?pretty"
```

기본적으로 일래스틱서치는 포트 9200으로 로컬에서 실행됨


```
{
  "name" : "6ce9cd7ae9bd",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "67Q3FjSRTr0SpRKJel-DJw",
  "version" : {
    "number" : "7.9.2",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "d34da0ea4a966c4e49417f2da2f244e3e97b4e6e",
    "build_date" : "2020-09-23T00:45:33.626720Z",
    "build_snapshot" : false,
    "lucene_version" : "8.6.2",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```


일래스틱서치 서버 설치 및 실행 완.

▼ 문서 저장소 객체 초기화

```
# document_store --> document_stores
from haystack.document_stores.elasticsearch import Elastic

# 밀집 리트리버에서 사용할 문서 임베딩을 반환
document_store = ElasticsearchDocumentStore(return_embeddi
```

기본적으로 `ElasticsearchDocumentStore`은 일래스틱서치에 두 개의 인덱스를 만든다

- `document` : 문서를 저장
- `label` : 답의 범위를 저장

여기서는 SubjQA 리뷰로 document 인덱스를 채울 것임

```
{
    "text": "<the-context>",
    "meta": {
        "field_01": "<additional-metadata>",
        "field_02": "<additional-metadata>",
        ...
    }
}
```

- 헤이스택 문서 저장소는 위와 같이 **text**와 **meta** 키를 가진 딕셔너리의 리스트를 가짐

▼ SubjQA 모든 리뷰를 document 인덱스에 저장하기

```
for split, df in dfs.items():
    # 중복 리뷰를 제외시키기
    docs = [{"content": row["context"], "id": row["review_
             "meta":{"item_id": row["title"], "question_id
                     "split": split}}
        for _,row in df.drop_duplicates(subset="context").
    document_store.write_documents(documents=docs, index="

print(f"{document_store.get_document_count()}개 문서가 저장되
```

1. `meta`에 있는 필드를 사용해 검색 과정에서 필터 적용하기
2. 제품과 질문 ID로 필터링 할 수 있도록 SubjQA의 item_id와 id 열,

   그리고 해당되는 분할(split)이름 포함시키기
3. 각 DataFrame에 있는 샘플을 순회하면서 `write_documents()`메소드로 인덱스에 추가하기

```
1615개 문서가 저장되었습니다
```

모든 리뷰를 인덱스에 저장 완!

▫ 리트리버 초기화하기

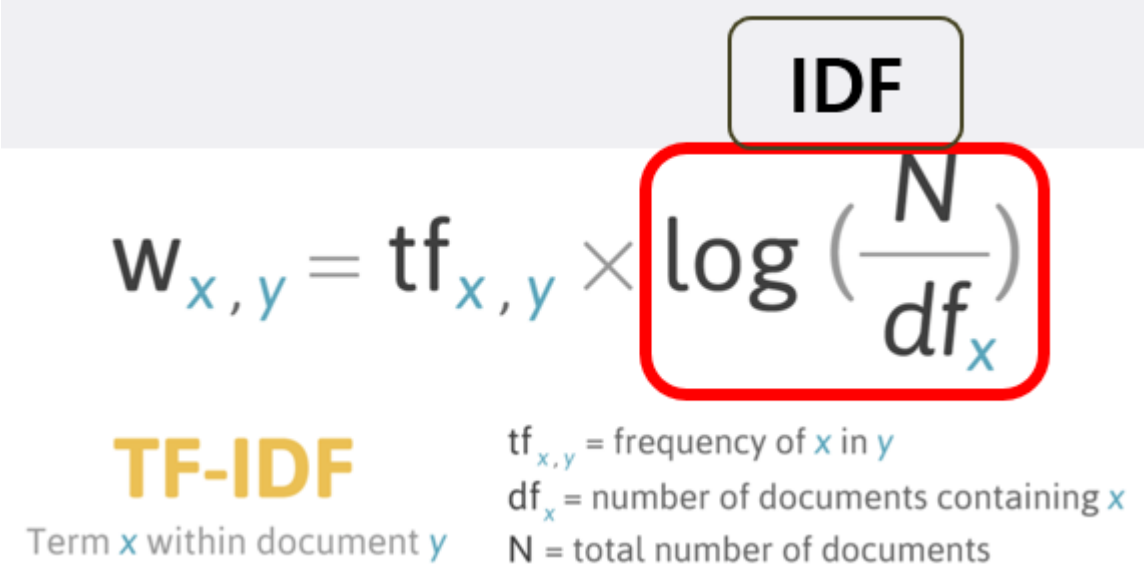
인덱스를 검색하려면 리트리버가 필요하므로 일래스틱서치를 위한 리트리버 만드는 법 알아보기

BM25

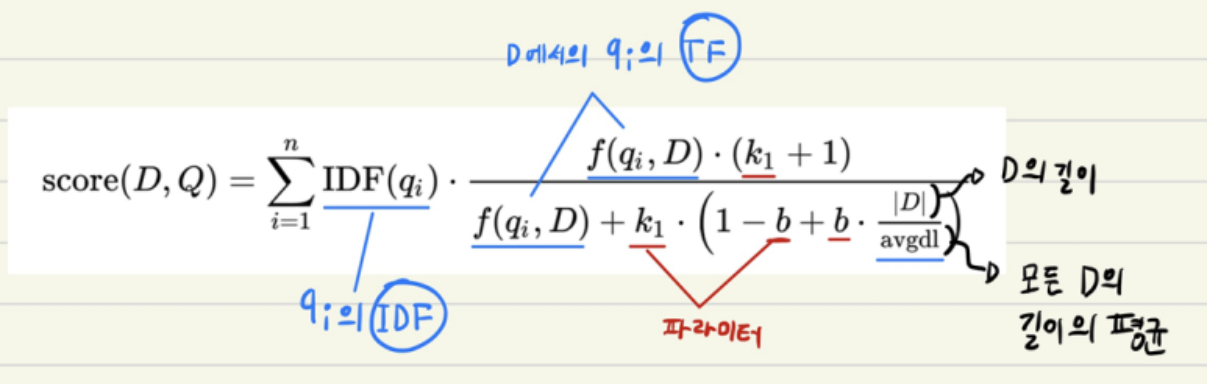
- Best Match 25의 약자
- BM25는 고전적인 TF-IDF 알고리즘을 개선한 버전

  일래스틱서치에서 효율적으로 검색할 수 있는 희소 벡터로 질문과 문맥을 표현함
- BM25 점수는 검색 쿼리에서 얼마나 많은 텍스트가 일치하는지 측정
- 그다음 TF 값을 빠르게 포화시키고 짧은 문서가 긴 문서보다 선호되도록

  문서 길이로 정규화해 TF-IDF를 개선



- TF - 문서의 빈도, IDF - 문서의 역빈도
- TF-IDF는 TF와 IDF를 곱한 값
- TF-IDF는 단어의 빈도수에 따른 중요성을 나타내면서 동시에
  전체 문서 집합에서의 희귀성을 고려하여 가중치를 부여



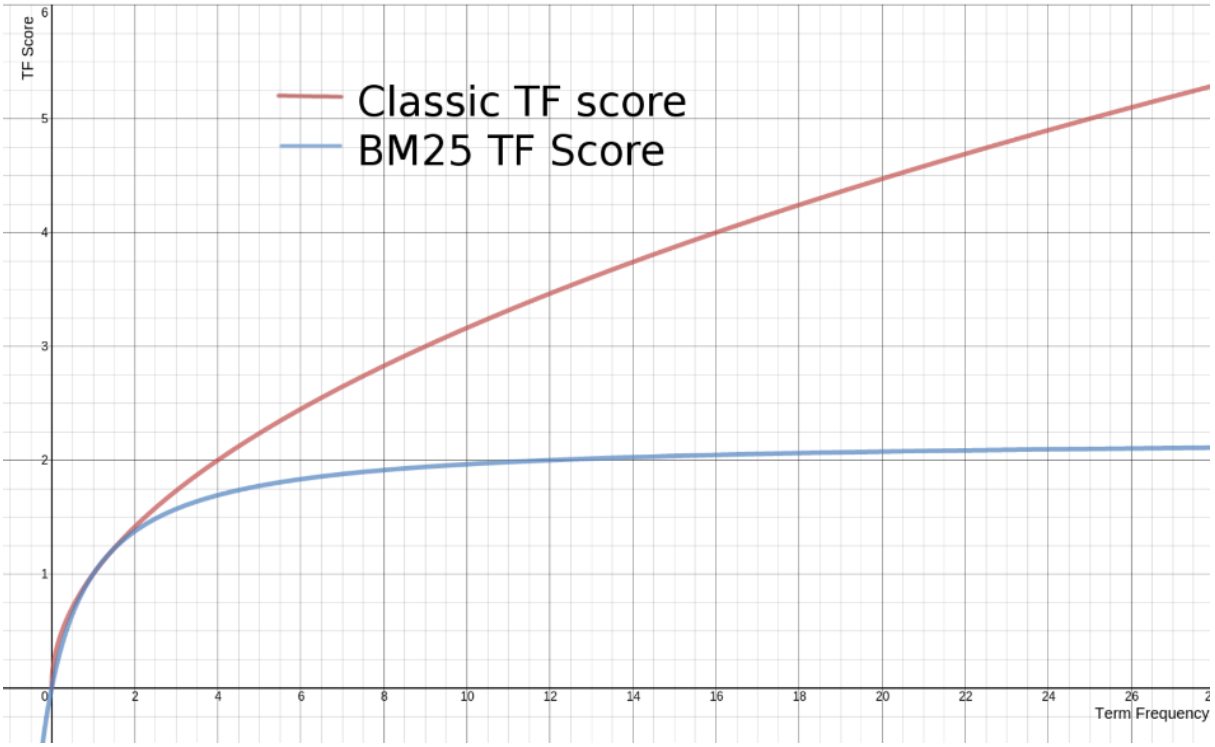
$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i,D) \cdot (k_1+1)}{f(q_i,D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

- **qi** : 쿼리에서의 i번째 토큰
- **f(qi, D)** : 해당 토큰이 문서 D에서 얼마나 자주 나타났는가, 즉 TF
- **k1** : 쿼리에 똑같은 토큰이 여러 개 등장했을 때, 어느 정도의 패널티를 줄 것인가
- **b** : 문서의 길이에 따르는 패널티

문서에 해당 토큰이 많이 등장할수록 점수를 많이 주되,

토큰이 너무 자주 등장하면 적당히 컷해주고,

비교하는 문서가 다른 문서들에 너무 길다 싶으면 또 패널티를 줘라!!



- BM25의 TF가 늘어나도 일정 빈도 이상 부터 score가 높아지지 않는다

▼ 검색하려는 문서 저장소(일래스틱서치)를 지정해 BM25 클래스의 객체 만들어보기

```
from haystack.nodes.retriever import BM25Retriever

bm25_retriever = BM25Retriever(document_store=document_sto
```

: 헤이스택에서 `ElasticsearchRetriever`의 기본 리트리버는 `BM25`

> 📣 **주의!**
>
> 헤이스택 1.4에서 `ElasticsearchRetriever`가 `BM25Retriever`로 바뀌었음.
>
> 여전히 `ElasticsearchRetriever`를 사용할 수 있지만 향후 버전에서 삭제될 수 있음

이제 훈련 세트에 있는 한 전자 제품에 대한 간단한 쿼리 살펴보기

리뷰 기반 QA 시스템에서는 **단일 아이템으로 쿼리를 제한**하는 것이 중요함

그렇지 않으면 리트리버가 사용자 쿼리와 **무관한** 제품의 리뷰도 검색하기 때문

예를 들어 제품 필터링 없이 *'Is the camera quality any good?'* 같은 질문을 하면

사용자가 물어본 제품은 노트북인데 휴대폰 리뷰가 반환되는 경우가 생김.

데이터셋에 있는 ASIN 값은 암호처럼 보이지만

아마존 ASIN 같은 온라인 도구나 www.amazon.com/dp/ 주소 뒤에 item_id 값을 추가하면 해독 됨.

- ASIN : "Amazon Standard Identification Number"의 약자로
  아마존에서 판매되는 제품을 식별하기 위한 고유한 ID.

▼ 리트리버의 `retrieve()` 메소드를 사용해 이 제품이 독서에 유용한지 물어보기

```
# 아래 아이템 ID는 아마존의 파이어 태블릿 ID
item_id = "B0074BW614"
query = "Is it good for reading?"
retrieved_docs = bm25_retriever.retrieve(
    query=query, top_k=3, filters={"item_id":[item_id], "s
```

- `top_k` 매개변수로 얼마나 많은 문서를 반환할지 지정
- 문서의 `meta` 필드에 포함시킨 `item_id`와 `split` 키에 모두 필터를 적용

▼ 추출된 문서 하나 살펴보기

```
print(retrieved_docs[0])
```

- `retrieved_docs`의 각 원소는 문서를 나타내는데 사용하는 헤이스택의 Document 객체이고

  리트리버의 쿼리 점수와 그 외 메타데이터를 포함함


```
{'text': 'This is a gift to myself.  I have been a kindle user for 4 years and
this is my third one.  I never thought I would want a fire for I mainly use it
for book reading.  I decided to try the fire for when I travel I take my laptop,
my phone and my iPod classic.  I love my iPod but watching movies on the plane
with it can be challenging because it is so small. Laptops battery life is not
as good as the Kindle.  So the Fire combines for me what I needed all three to
do. So far so good.', 'score': 6.243799, 'probability': 0.6857824513476455,
'question': None, 'meta': {'item_id': 'B0074BW614', 'question_id':
'868e311275e26dbafe5af70774a300f3', 'split': 'train'}, 'embedding': None, 'id':
'252e83e25d52df7311d597dc89eef9f6'}
```


문서 텍스트 외에, 일래스틱서치가 쿼리와의 연관성을 계산한 score 필드가 출력됨

- 점수가 높을수록 매칭이 더 잘됐음을 의미

▫ 리더 초기화하기

헤이스택에 MiniLM 모델을 로드하는 방법 알아보기

헤이스택에는 두 종류의 리더가 있음

FARMReader

트랜스포머를 미세 튜닝하고 배포하는 딥셋의 FARM 프레임워크를 기반

TransformersReader

🤗 트랜스포머스의 QA 파이프라인을 기반

추론만 실행하는데 적합함

두 리더가 모델의 가중치를 같은 방식으로 처리하지만 예측을 변환해 답을 만드는 방식은 조금 다름

🤗 트랜스포머스에서 QA 파이프라인은 각 구절의 시작 로짓과 종료 로짓을 소프트맥스로 정규화.

따라서 확률의 합이 1이 되는 **같은 구절**에서 추출한 답의 점수를 **비교**할 때만 의미가 있음

예를 들어 한 구절에서 점수가 0.9인 답이 다른 구절에서 0.8을 얻은 답보다 반드시 더 좋지는 않음!

- FARM : 로짓을 정규화하지 않아 구절 간의 답변이 더 쉽게 비교됨

문맥이 길고 답이 중첩된 윈도에 놓인 경우

- Transformers : 같은 답을 다른 점수로 두 번 예측함.
- FARM : 이런 중복을 그냥 제거함

이 장 후반부에서 리더를 미세 튜닝하기 위해 여기서는 FARM 사용!

▼ 모델 불러오기

```
from haystack.reader.farm import FARMReader

model_ckpt = "deepset/minilm-uncased-squad2"
max_seq_length, doc_stride = 384, 128
reader = FARMReader(model_name_or_path=model_ckpt, progres
                    max_seq_len=max_seq_length, doc_stride
                    return_no_answer=True)
```

QA 특화된 매개변수와 함께 허깅페이스 허브에 있는 MiniLM 체크포인트 지정하기

- FARMReader에서 슬라이딩 윈도의 동작은 토크나이저에서 본 것과 같은 `max_seq_length`와 `doc_stride` 매개변수로 제어
- 여기서는 MiniLM 논문에 있는 값 사용

> 📣 **참고!**
>
> 🤗 트랜스포머스에서 독해 모델을 **직접** 미세 튜닝해 추론할 때는 TransformerReader에 로드하는 방법도 있음.

▼ 간단한 예로 리더의 동작 확인하기

```
print(reader.predict_on_texts(question=question, texts=[co
```

```
{'query': 'How much music can this hold?', 'no_ans_gap': 12.648089170455933,
'answers': [<Answer {'answer': '6000 hours', 'type': 'extractive', 'score':
0.5293056815862656, 'context': 'An MP3 is about 1 MB/minute, so about 6000 hours
depending on file size.', 'offsets_in_document': [{'start': 38, 'end': 48}],
'offsets_in_context': [{'start': 38, 'end': 48}], 'document_id':
'e344757014e804eff50faa3ecf1c9c75', 'meta': {}}>]}
```

잘 동작하는 중.

▫ 모두 합치기

헤이스택 파이프라인 중 하나를 사용해 구성 요소들을 모두 연결하기

▼ 파이프라인 불러오기

```
from haystack.pipeline import ExtractiveQAPipeline

pipe = ExtractiveQAPipeline(reader=reader, retriever=bm25_
```

현재는 답 추출에 관심이 있으므로 매개변수로

하나의 리트리버-리더 쌍을 받는 `ExtractiveQAPipeline` 사용!

▼ 3개의 추출된 답 반환

```
n_answers = 3
preds = pipe.run(query=query,
                 params={"Retriever": {"top_k": 3, "filter
                                       "Reader": {"top_k":

print(f"질문: {preds['query']} \n")
for idx in range(n_answers):
    print(f"답변 {idx+1}: {preds['answers'][idx].answer}")
    print(f"해당 리뷰 텍스트: ...{preds['answers'][idx].conte
    print("\n\n")
```

```
n_answers = 3
preds = pipe.run(query=query, top_k_retriever=3, top_k_reader=n_answers,
                 filters={"item_id": [item_id], "split":["train"]})
```

요 부분이 다름 git 코드는 딕셔너리 형태로 params 매개변수를 통해 구성 요소 전달

책에 있는 코드는 직접적인 매개변수로 전달

- 각 파이프라인은 쿼리가 어떻게 실행되어야 하는지 지정하는 `run()` 메소드가 있음
- ExtractiveQAPipeline의 경우 `query`, `top_k_retriever`로 추출할 문서 개수, `top_k_reader`로 문서에서 추출할 **답** 개수 전달
- 이 경우 아이템 ID에 대한 필터도 지정해야함
- 리트리버에서 한 것처럼 `filters` 매개변수를 사용해 처리

```
질문: Is it good for reading?

답변 1: it is great for reading books when no light is available
해당 리뷰 텍스트: ...ecoming addicted to hers! Our son LOVES it and it is great for
reading books when no light is available. Amazing sound but I suggest good
headphones t...


답변 2: I mainly use it for book reading
해당 리뷰 텍스트: ... is my third one.  I never thought I would want a fire for I
mainly use it for book reading.  I decided to try the fire for when I travel I
take my la...


답변 3:
해당 리뷰 텍스트: ...None...
```

아마존 제품 리뷰를 위한 엔드-투-엔드 QA 시스템 만들기 완.

성능을 더 높이려면 리트리버와 리더의 성능을 정량화할 지표가 필요 >> 다음 절에서!