BERT(Bidirectional Encoder Representations from Transformers) : 문맥을 양방향으로 이해해서 숫자로 바꿔준다. (decoder는 왼쪽에서 오른쪽으로, encoder가 양방향)

문장에서 단어를 하나씩 읽어가면서 학습한다.

입력 값 :

1. Token Embeddings

WordPiece Embedding을 사용해 문장을 token단위로 분리

1. Segment Embeddings

두개의 문장이 입력될 경우 각각의 문장에 서로 다른 숫자를 더 해주는 것(두개의 다른 문장이 있다는 걸 구분)

1. Position Embeddings

token들의 상대적 위치를 알려줌(sin, cos함수 사용 3가지 이유 1. Sin, cos의 출력값은 입력값에 따라 달라져 상대적인 위치로 사용가능. 2. Sin, cos의 출력값은 규칙적으로 증가 또는 감소 즉 쉽게 계산 가능. 3. 무한대의 입력값도 상대적인 위치를 출력 가능 1과 -1 사이 값. 절대적 위치를 쓰지않는 이유는 최장길이 문장을 입력해야함. 학습시에 사용한 최장길이의 문장을 초과하면 받을 수가 없어 상대적 위치를 선호.)

Fine Tuning : BERT는 GPT와 다르게 상대적으로 작은 크기로 사전 학습되어 있어서 사용자의 목적에 맞게(1. 문장 한 개 분류(스팸 메일 찾기, 감성 분류), 2. 문장 두개의 관계 분류(두 문장이 주어졌을 때 같은 뜻인지 반대되는 뜻인지 등), 3. 문장내 들어있는 단어나 문장의 문법 품사 예측, 4.묻고 답하기(챗봇)) 관련 데이터를 주고 학습시키는 것.

※ WordPiece Embedding : Texting -> (Text, ##ing) 이렇게 의미를 나눔으로서 Googling 같은 생소한 단어나 오탈자도 잘 예측할 수 있음

**편집거리를 이용한 철자 오류 교정**

1. 통계적인 방식을 통해 철자오류를 교정하기 위 해서는 철자오류가 포함되어 있지 않은 문서를 수집하여 말뭉 치 사전을 구축할 필요가 존재

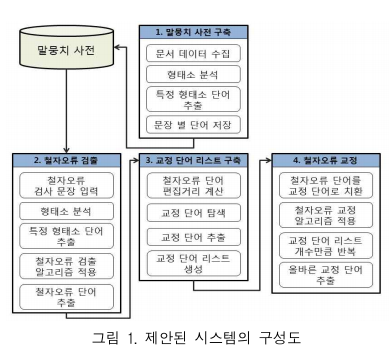
2. 말뭉치 사전 구축 이후 철자오류를 교정하기 위해 일차적으로 문장 내에 서 철자오류 존재 유무를 검사

3. 철자오류 단어가 존재할 시 철자오류 단어를 추출

4. 철자오류 단어와 편집거리가 작은 단어들을 말뭉치 사 전에서 추출하여 교정 단어 리스트를 제작

5. 교정 단어 리스트의 단어들을 입력 받은 문장 의 철자오류 단어와 치환한 다음 철자오류 교정 알고리즘을 수행함으로서 치환한 교정 단어의 알맞음 여부를 검사

6. 가장 수치가 높은 단어를 문장에 삽입하는 것으로 철자오류 교정을 끝마침.

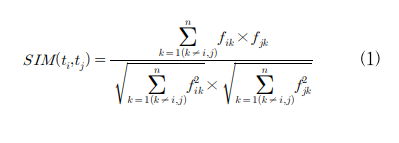


철자오류 검출에 수행되는 품사는 동사, 형용사, 명사 같은 품사 단어들을 중점으로

모든 품사의 단어들을 통계적인 방식을 통해 교정을 수행하게 되면 시간이 많이 들기 때문에

코사인 유사도를 이용하여 문장에 등장하는 단어들의 동시 등장 빈도를 구함.

**철자오류 단어 검출 과정(상세)**



식 (1)은 문장에서 등장하는 i번째 단어 와 j번째 단어 를 쌍으로 묶어 두 단어의 동시 출현 확률을 구하는 수식

k는 i나 j가 되지 않기 때문에 두 단어 와 의 동시 등장 빈도, 즉 절대빈도를 이용하지 않는다

코사인 유사도는 0 부터 1까지의 값을 가질 수 있으며, 유사도가 0에 가까울수록 두 객체는 서로 독립적 이라는 의미.(위의 식이 0에 가까울수록 독립적)

수식 (1)을 통해서 나온 두 단어의 동시 등장 확률이 0에 가까울수록 수식 (1)에 적용한 두 단어 중 한 단어가 철자오류가 발생한 단어

수식 (1)만 가지고는 수식에 적용한 두 단어 중 어느 단어가 철자오류가 발생한 단어인지 알 수 없다. 그렇기 때문에 특정 문장에서 등장하는 단어 이 철자오류 단어인지 아니면 정상 단어인지 여부를 가리기 위해 단어 을 제외한 문장에 등장하는 다른 단어들인 부터 문장에 등장하는 마지막 단어 까지의 코사인 유사도를 계산한 다음 평균값을 추출하여 그 값이 가장 낮으며 통시에 임계값 이하인 단어를 철자오류로 검출

**교정단어 리스트 구축 및 철자오류 교정(상세)**

철자오류 검출이 성공적으로 수행이 완료 되면, 검출된 단어를 교정해야하는데 교정단어를 다른 단어로 치환후 다시 좌우 문맥 단어와 동시 등장 빈도를 구하는 것은 시간이 크게 증가됨으로 사전에 존재하는 단어들 중 철자오류가 발생한 단어와 편집거리가 작은 단어들만 추출하여 좌우문맥 단어들과 등시 등장 확률을 구하게 되면 철자오류 교정이 걸리는 시간이 줄어든다.

한글 편집거리 알고리즘을 사용하여 철자오류 단어와 편집거리가 작은(몇으로 할지 정의해야 됨) 단어들을 말뭉치 사전에서 추출함으로서 철자오류 교정을 수행.(교정할 때 식(1)을 사용해 가장 낮은 평균값을 기록한 단어를 오류 단어라고 판단했다면, 교정은 리스트들을 만들고 식(1)을 사용해 가장 높은 값을 기록한 단어를 올바른 교정 단어라 판단)

**Transformers(Bert) 라이브러리 정리.**

무엇을 할 수 있는가? 참고(<https://huggingface.co/transformers/quicktour.html>)

* Sentiment analysis: is a text positive or negative?(감정 분석)
* Text generation (in English): provide a prompt and the model will generate what follows.(model에 prompt를 주고 다음에 나올 text 생성)
* Name entity recognition (NER): in an input sentence, label each word with the entity it represents (person, place, etc.) (주어진 문장에서 각 단어가 표현하는 종류 인식)
* Question answering: provide the model with some context and a question, extract the answer from the context.(문단과 질문을 model에 주고 문단에서 답 추출하기.)
* Filling masked text: given a text with masked words (e.g., replaced by [MASK]), fill the blanks. (text에서 특전 단어에 [MASK]를 씌우고 [MASK]를 맞추기)
* Summarization: generate a summary of a long text. (긴 text 요약!)
* Translation: translate a text in another language.(text에서 다른 언어 번역)
* Feature extraction: return a tensor representation of the text.( text의 표현 tensor? 반환.)

위 과정(task)들을 어떻게 할 수 있을까?

* Pipelines: very easy-to-use abstractions, which require as little as two lines of code.

(transformers 라이브러리에 있는 pipeline함수 이용.

-> 장점 : 추상적이라 쉽게 사용가능. 단점: 유연한 코드 불가)

* Direct model use: Less abstractions, but more flexibility and power via a direct access to a tokenizer (PyTorch/TensorFlow) and full inference capacity.

(학습하거나 불러온 model 직접사용

-> 장점 : 유연한 코드 작성 및 tokenizer에 직접 접근해서 강력하다?

단점 : 각 task들에 결과값을 내기 위해 예측결과를 내기위한 코드를 직접 코딩해야함.

참고 : <https://huggingface.co/transformers/task_summary.html>

)

**Models**

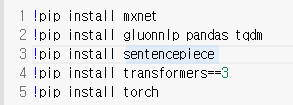
Autoregressive model(자기회귀) : 고전적 언어 모델링 작업. 이전 토큰을 모두 읽고 다음 토큰을 추측(이전 내용만 볼 수 있고 이후 내용은 볼 수 없음.) Text 생성이 대표적 예이고 관련 모델은 GPT.

Autoencoder model:

**학습 예제.**

예제 1. Skt-kobert tutorial

참고 <https://colab.research.google.com/github/SKTBrain/KoBERT/blob/master/scripts/NSMC/naver_review_classifications_pytorch_kobert.ipynb>



여기서 transformers==3 해줘야함.. 최신 버전으로 하면 마지막 예측할 때 “Input: must be Tensor, not str” 에러 발생!!!!



: 데이터 파일 불러오기(위 train, 아래 test, 데이터 내용은 아래와 같음)

id document label

9976970 아 더빙.. 진짜 짜증나네요 목소리 0

3819312 흠...포스터보고 초딩영화줄....오버연기조차 가볍지 않구나 1

10265843 너무재밓었다그래서보는것을추천한다 0

7156791 액션이 없는데도 재미 있는 몇안되는 영화 1

(네이버 영화리뷰를 가져왔으며, id값에 따라 리뷰평과 label은 부정(0), 긍정(1) 나타냄. )



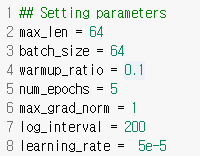
Train data와 test data 구분

field\_indices의 기본값이 [0, 1] ,

(위 예제는 텍스트가 1, 레이블이 2이기 때문에 위처럼 설정)

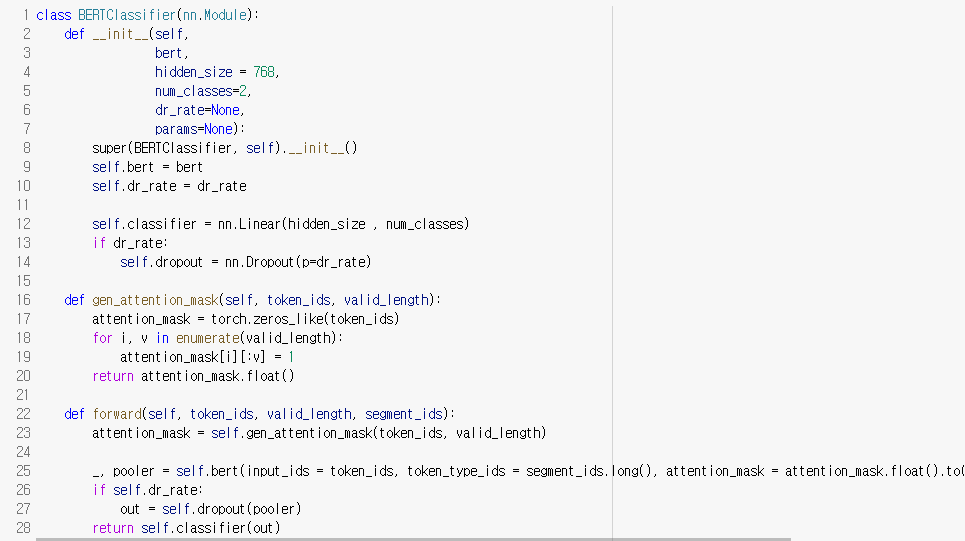
num\_discard\_samples는 데이터 상위 몇 개 row 제거할지 설정

(위 예제는 id document label이 적혀 있어서 1줄 삭제.)



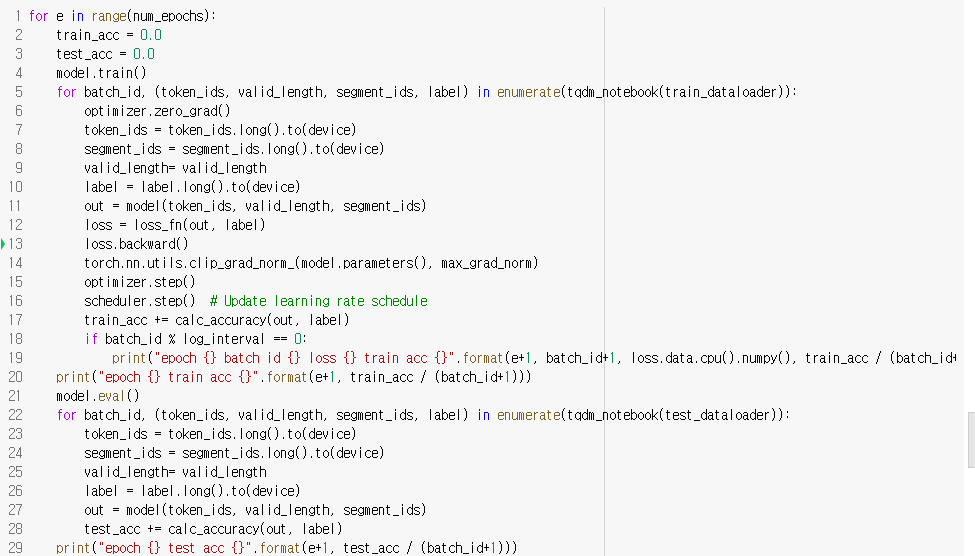
파라미터 설정

(max\_len : 텍스트 데이터 최대 길이, num\_epochs : 훈련 반복 횟수)

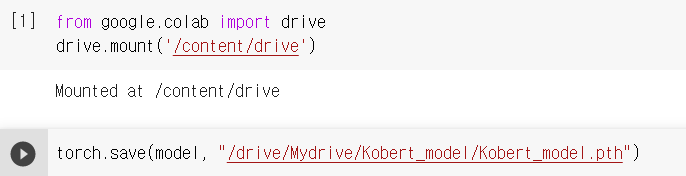


분류기 설정

Num\_class = 2 (2가지로 분류하겠다는 뜻.. 이진 분류기)



Train 및 test

학습시킨 model google drive에 저장!

불러올 때는 아래처럼…

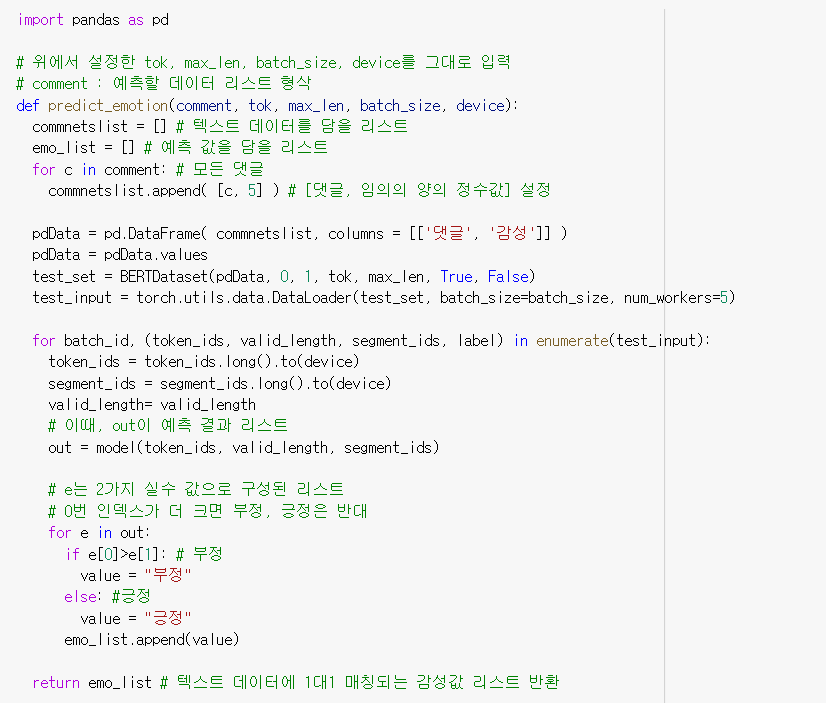
*# 모델 클래스는 어딘가에 반드시 선언되어 있어야 합니다*

**model** **=** **torch.load(“/drive/Mydrive/Kobert\_model/Kobert\_model.pth”)**

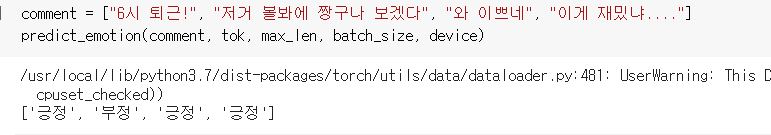
불러와서 결과 보기



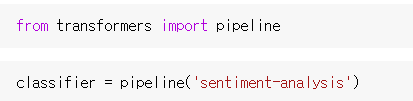
모델에서 예측 결과 값 보기위해 만든 함수 (밑에 pipeline과 비교해보면..)

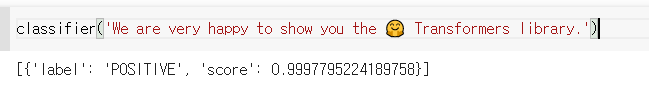


예측 결과



위에 코드를 pipeline에 있는 예제 사용했을 경우





굉장히 간단해짐.



이런식으로 원하는(학습된) 모델을 들고 와서 사용가능!

(저기 있는 모델은 huggingface에 있는 모델 )

FILL-MASK 프로세스 동작 원리.

**1. 체크포인트 이름에서 tokenizer와 모델을 인스턴스화하고 식별된 Bert모델에서 저장된 가중치를 로드.**

**2. 마스킹된 token을 가지는 시퀀스를 정의하고 단어 대신 tokenizer.mask\_token배치**

**3. 해당 시퀀스를 ID목록으로 인코딩(파이토치용으로) 해당 목록에서 mask된 token의 position을 찾는다.**

**4. 마스크 token의 인덱스에서 예측값을 검색(이 tensor는 어휘와 동일한 크기를 가지며 값은 각 token에 부여된 점수. 모델은 해당 컨텍스트에서 가능성이 있다고 간주되는 token에 더 높은 점수부여)**

**5. topk 메서드를 사용하여 상위 5개 token을 검색**

**6. mask\_token을 위 상위 5개 token으로 교체하고 결과 print**

**TODO LIST**

**1. 'fill-mask'를 위한 task 모델로 ‘fill-mask’동작 수행**

**2. 'fill-mask'를 위한 task 모델이 아닌 다른 모델을 가져와서 'fill-mask'수행**

**3. 2.모델에 'fill-mask'위한 courpus들을 pretrained에 추가 학습시켜본다.**

**4. etri kobert, klue/roberta-large 를 가져와서 'fill-mask'를 시험해본다.**

**5. 포지션 임베딩이랑 다른 임베딩 입력으로 들어가는거 분석해보기.**

**1. 'fill-mask'를 위한 task 모델로 ‘fill-mask’동작 수행**

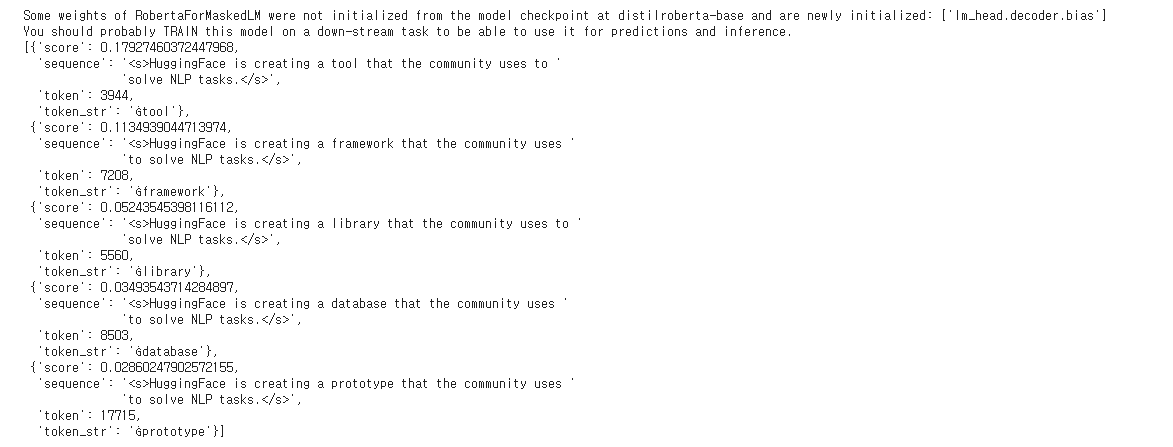
입력

unmasker = pipeline("fill-mask")

from pprint import pprint # 데이터 이쁘게 출력 보통 json파일같은 복잡한 객체 출력할 때 사용!

pprint(unmasker(f"HuggingFace is creating a {unmasker.tokenizer.mask\_token} that the community uses to solve NLP tasks."))

출력



+)pipeline없이 수행

입력

from transformers import AutoModelWithLMHead, AutoTokenizer

import torch

# 체크포인트 이름에서 tokenizer와 모델을 인스턴스화하고 식별된 Bert모델에서 저장된 가중치를 로드.

tokenizer = AutoTokenizer.from\_pretrained("distilbert-base-cased")

model = AutoModelWithLMHead.from\_pretrained("distilbert-base-cased")

# 마스킹된 token을 가지는 시퀀스를 정의하고 단어 대신 tokenizer.mask\_token배치

sequence = f"Distilled models are smaller than the models they mimic. Using them instead of the large versions would help {tokenizer.mask\_token} our carbon footprint."

# 해당 시퀀스를 ID목록으로 인코딩(파이토치용으로) 해당 목록에서 mask된 token의 position을 찾는다.

input = tokenizer.encode(sequence, return\_tensors="pt")

mask\_token\_index = torch.where(input == tokenizer.mask\_token\_id)[1]

# 마스크 token의 인덱스에서 예측값을 검색(이 tensor는 어휘와 동일한 크기를 가지며 값은 각 token에 부여된 점수. 모델은 해당 컨텍스트에서 가능성이 있다고 간주되는 token에 더 높은 점수부여)

token\_logits = model(input)[0]

mask\_token\_logits = token\_logits[0, mask\_token\_index, :]

# topk 메서드를 사용하여 상위 5개 token을 검색

top\_5\_tokens = torch.topk(mask\_token\_logits, 5, dim=1).indices[0].tolist()

**2. 'fill-mask'를 위한 task 모델이 아닌 다른 모델을 가져와서 'fill-mask'수행**

from transformers import AutoModelWithLMHead, AutoTokenizer

import torch

# 체크포인트 이름에서 tokenizer와 모델을 인스턴스화하고 식별된 Bert모델에서 저장된 가중치를 로드.

tokenizer = AutoTokenizer.from\_pretrained("skt/kobert-base-v1")

model = AutoModelWithLMHead.from\_pretrained("skt/kobert-base-v1")

# 마스킹된 token을 가지는 시퀀스를 정의하고 단어 대신 tokenizer.mask\_token배치

sequence = f"기사는 중세부터 유럽에서 기마로 싸우는 {tokenizer.mask\_token} 에게 주는 명예 칭호 및 그로부터 파생한 계급을 가리킨다."

# 해당 시퀀스를 ID목록으로 인코딩(파이토치용으로) 해당 목록에서 mask된 token의 position을 찾는다.

input = tokenizer.encode(sequence, return\_tensors="pt")

mask\_token\_index = torch.where(input == tokenizer.mask\_token\_id)[1]

# 마스크 token의 인덱스에서 예측값을 검색(이 tensor는 어휘와 동일한 크기를 가지며 값은 각 token에 부여된 점수. 모델은 해당 컨텍스트에서 가능성이 있다고 간주되는 token에 더 높은 점수부여)

token\_logits = model(input)[0]

mask\_token\_logits = token\_logits[0, mask\_token\_index, :]

# topk 메서드를 사용하여 상위 5개 token을 검색

top\_5\_tokens = torch.topk(mask\_token\_logits, 5, dim=1).indices[0].tolist()

# mask\_token을 위 상위 5개 token으로 교체하고 결과 print

for token in top\_5\_tokens:

    print(sequence.replace(tokenizer.mask\_token, tokenizer.decode([token])))

**3. 2.모델에 'fill-mask'위한 courpus들을 pretrained에 추가 학습시켜본다.**

**5. 포지션 임베딩이랑 다른 임베딩 입력으로 들어가는거 분석해보기.**

**포지션 임베딩 관련** [**https://huggingface.co/transformers/glossary.html#position-ids**](https://huggingface.co/transformers/glossary.html#position-ids)