**Wrap Up Report**

**P – Stage 2**

**문장 내 개체간 관계 추출**

**부스트캠프 AI 테크 P-Stage**

**선재우 T1098**

목차

[**1.** **최종 점수 및 순위** 4](#_Toc70079373)

[**2.** **목표** 4](#_Toc70079374)

[**[BaseLine 작성]**  4](#_Toc70079375)

[**[Data Processing]**  4](#_Toc70079376)

[**[Model]** 5](#_Toc70079377)

[**[Training]** 5](#_Toc70079378)

[**[Deploy]** 5](#_Toc70079379)

[**3.** **기술적인 도전** 6](#_Toc70079380)

[**[Data Processing]** 6](#_Toc70079381)

[1) Exploratory Data Analysis (EDA) 6](#_Toc70079382)

[2) 한국어 토크나이징 10](#_Toc70079383)

[3) Tokenizer에서 sentence 1, sentence 2를 주는 방법 13](#_Toc70079384)

[4) 다양한Truncation 방법 15](#_Toc70079385)

[5) Entity token 추가 15](#_Toc70079386)

[6) pororo 사용한 tokenizer dataset 수정 17](#_Toc70079387)

[7) Pororo NER 사용법 변경 20](#_Toc70079388)

[**[Model]** 21](#_Toc70079389)

[1) BERT모델 21](#_Toc70079390)

[2) BERT에 Entity Layer 추가 23](#_Toc70079391)

[3) xlm-roberta-large 사용 28](#_Toc70079392)

[**[Training]** 30](#_Toc70079393)

[1) Custom Trainer 작성 30](#_Toc70079394)

[2) Hard Votes Ensemble 사용 32](#_Toc70079395)

[**4. 평가** 35](#_Toc70079396)

[1) 평가 지표 Table 35](#_Toc70079397)

[2) Entity token 추가 성능 비교 37](#_Toc70079398)

[3) Pororo를 사용한 tokenizer dataset 수정 성능 비교 38](#_Toc70079399)

[4) XLM-Roberta-Large 모델 & Tokenizer(Fast) 성능 비교 40](#_Toc70079400)

[5) 최적의 Hyperparameter 찾기 43](#_Toc70079401)

[6) Hard votes ensemble 성능 비교 44](#_Toc70079402)

[7) Pororo NER 변경 성능 비교 45](#_Toc70079403)

[**5.** **학습과정에서의 교훈** 45](#_Toc70079404)

[**6.** **마주한 한계와 도전숙제** 46](#_Toc70079405)

# **최종 점수 및 순위**





# **목표**

## **[BaseLine 작성] (새로운 baseline code)**

## **[Data Processing] - Exploratory Data Analysis (기간 : 4/12 ~ 4/12 ) - Cross-validation 사용 - 데이터 불균형 해소 (Huggingface Trainer – Label smoothing factor) - 한국어 전처리 (기간 : 4/13 ~ 4/13 ) - 새로운 tokenizer 사용 (기간 : 4/13 ~ 4/22 )  - 형태소 분류기 -> BERT wordpiece - [ENT][/ENT] tag를 추가해서 train 돌리기 (기간 : 4/20 ~ 4/22 ) - Pororo 사용해보기 (기간 : 4/20 ~ 4/21)**

## **[Model] - BERT 모델 사용 ( 기간 : 4/14 ~ 4/20 ) - Roberta 모델 사용 ( 기간 : 4/20 ~ 4/21 ) - xlm-roberta 모델 사용 (기간 : 4/21 ~ 4/22 ) - GPT 모델 사용 (취소 : 생성모델이라 대회 목적과 다름) - KoELECTRA 모델 사용 - KoBERT 모델 사용 (기간 : 4/14 ~ 4/16 )**

## **[Training] - 앙상블 시도 ( 기간 : 4/22 ~ 4/22 )** **- Hyperparameter 변경 ( 기간 : 4/21 ~ 4/22 ) - Learning Schedular 사용 (HuggingFace Trainer) - 좋은 위치에서 Checkpoint 만들기 (huggingface trainer) - Wandb (Auto ML) 사용**

## **[Deploy] - Python 모듈화 (새로운 baseline code)**

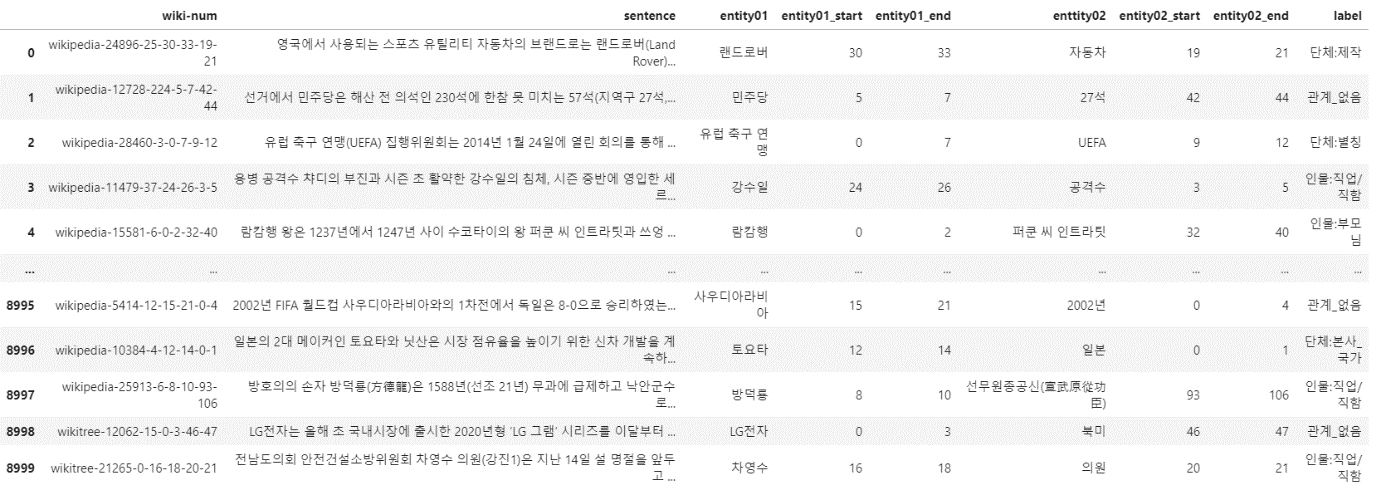
# **기술적인 도전**

## **[Data Processing]**

### 1) Exploratory Data Analysis (EDA)

train.tsv 데이터를 받아오면 다음과 같습니다.

trainfile = pd.read\_csv("/opt/ml/input/data/train/train.tsv", sep='\t', names=["wiki-num", "sentence", "entity01", "entity01\_start", "entity01\_end", "enttity02", "entity02\_start", "entity02\_end", "label"])



entity01과 entity02의 상관관계를 구하는 것이 이번 프로젝트의 목표입니다.

label들을 확인하면 **class들의 분포가 불균형**하다는 것을 알 수 있습니다.

>>> trainfile['label'].value\_counts()

관계\_없음 4432

단체:구성원 815

인물:소속단체 683

인물:직업/직함 553

단체:상위\_단체 335

단체:별칭 231

인물:출신성분/국적 209

인물:동료 186

단체:제작 171

인물:부모님 164

단체:본사\_도시 135

단체:본사\_국가 115

인물:별칭 103

인물:배우자 92

인물:자녀 84

단체:하위\_단체 79

단체:창립일 67

인물:기타\_친족 58

인물:제작 56

인물:형제/자매/남매 52

단체:창립자 45

인물:사망\_일시 45

단체:모회사 44

인물:출생\_일시 36

인물:거주\_국가 27

인물:거주\_도시 26

단체:본사\_주(도) 23

단체:정치/종교성향 18

단체:해산일 15

인물:종교 15

인물:거주\_주(도) 12

인물:용의자 11

단체:주주 11

인물:학교 9

단체:구성원\_수 9

인물:출생\_국가 8

인물:사망\_원인 7

단체:자회사 6

인물:나이 5

인물:출생\_도시 4

인물:사망\_도시 3

인물:사망\_국가 1

Name: label, dtype: int64

따라서 Matplotlib를 사용해 그래프를 그리면 다음과 같습니다.

plt.figure(figsize=(20, 3))

plt.title("Histogram", fontsize=15)

frq, bins, fig = plt.hist(trainfile["label"], bins=10, alpha=.8, color='red')

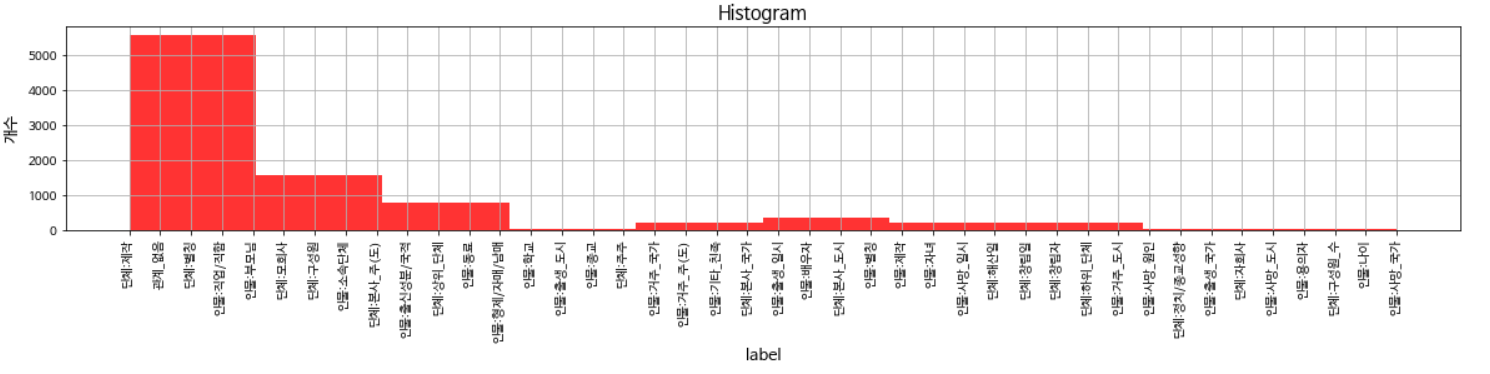
plt.ylabel("개수", fontsize=13)

plt.xlabel("label", fontsize=13)

plt.xticks(rotation=90)

plt.grid()

plt.show()



**entity01의 분포**는 3개로 나눠집니다.

* 단계
* 관계(없음)
* 인물

plt.figure(figsize=(12, 3))

plt.title("Histogram", fontsize=15)

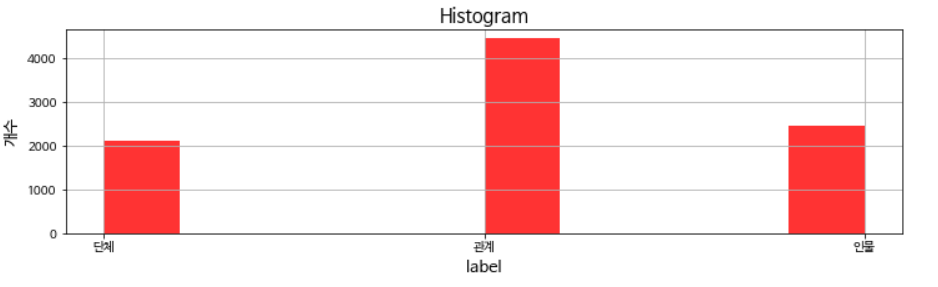
frq, bins, fig = plt.hist([x[:2] for x in trainfile['label']], bins=10, alpha=.8, color='red')

plt.ylabel("개수", fontsize=13)

plt.xlabel("label", fontsize=13)

plt.grid()

plt.show()



**entity02의 분포**는 전체 label의 개수와 비슷합니다.

plt.figure(figsize=(20, 3))

plt.title("Histogram", fontsize=15)

frq, bins, fig = plt.hist([x[3:] for x in trainfile['label']], bins=10, alpha=.8, color='red')

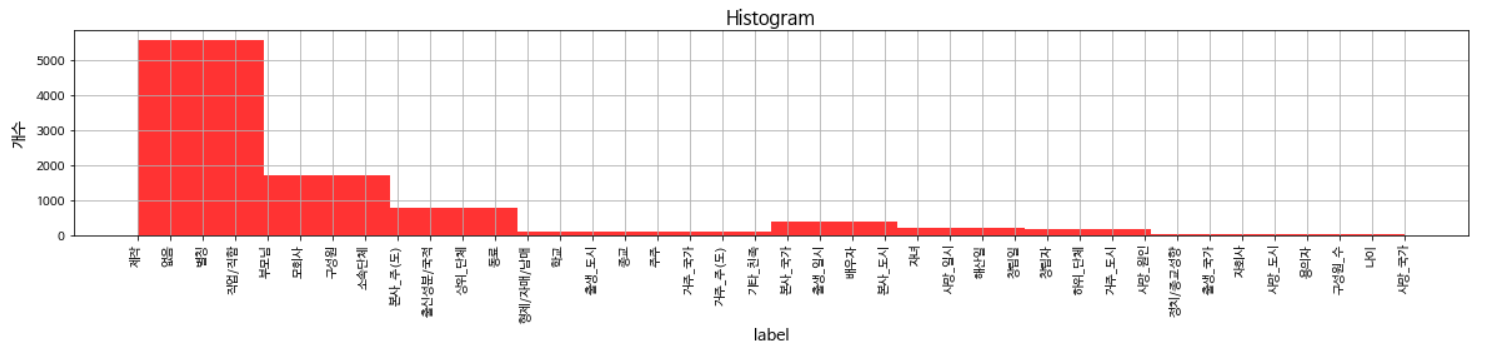
plt.ylabel("개수", fontsize=13)

plt.xlabel("label", fontsize=13)

plt.xticks(rotation=90)

plt.grid()

plt.show()



### 2) 한국어 토크나이징

토크나이징의 목적은 크게 두 가지입니다.

1. 의미를 지닌 단위로 자연어를 분절
2. Model의 학습 시, 동일한 size로 입력

따라서 token의 개수가 부족할 때는 [padding] 처리를 해주고, 개수가 많을 때는 token을 잘라서 반환해주어야 합니다.

max\_seq\_length = 10

*# padding*

tokenized\_text += ["[PAD]"] \* (max\_seq\_length - len(tokenized\_text))

>>> print(tokenized\_text)

['이순신은', '조선', '중기의', '무신이다.', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]']

* **어절 단위 tokenizing**모든 단어를 띄어쓰기 단위로 분리

text = "이순신은 조선 중기의 무신이다."

tokenized\_text = text.split(" ")

>>> print(tokenized\_text)

['이순신은', '조선', '중기의', '무신이다.']

* **형태소 단위 tokenizing**형태소 분석기를 사용해서 분리

!pip install konlpy

!bash <(curl -s https://raw.githubusercontent.com/konlpy/konlpy/master/scripts/mecab.sh)

from konlpy.tag import Mecab

mecab = Mecab()

text = "이순신은 조선 중기의 무신이다."

tokenized\_text = [lemma[0] for lemma in mecab.pos(text)]

>>> print(tokenized\_text)

['이순신', '은', '조선', '중기', '의', '무신', '이', '다', '.']

* **음절 단위 tokenizing**자연어를 한 글자씩 분리

text = "이순신은 조선 중기의 무신이다."

tokenized\_text = list(text)

>>> print(tokenized\_text)

['이', '순', '신', '은', ' ', '조', '선', ' ', '중', '기', '의', ' ', '무', '신', '이', '다', '.']

* **자소 단위 tokenizing**초성, 중성, 종성 총 3개의 자소로 분리

!pip install hgtk

import hgtk

text = "이순신은 조선 중기의 무신이다."

tokenized\_text = list(hgtk.text.decompose(text))

>>> print(tokenized\_text) *# ᴥ는 음절과 음절 단위를 구분해주는 구분자*

['ㅇ', 'ㅣ', 'ᴥ', 'ㅅ', 'ㅜ', 'ㄴ', 'ᴥ', 'ㅅ', 'ㅣ', 'ㄴ', 'ᴥ', 'ㅇ', 'ㅡ', 'ㄴ', 'ᴥ', ' ', 'ㅈ', 'ㅗ', 'ᴥ', 'ㅅ', 'ㅓ', 'ㄴ', 'ᴥ', ' ', 'ㅈ', 'ㅜ', 'ㅇ', 'ᴥ', 'ㄱ', 'ㅣ', 'ᴥ', 'ㅇ', 'ㅢ', 'ᴥ', ' ', 'ㅁ', 'ㅜ', 'ᴥ', 'ㅅ', 'ㅣ', 'ㄴ', 'ᴥ', 'ㅇ', 'ㅣ', 'ᴥ', 'ㄷ', 'ㅏ', 'ᴥ', '.']

* **WordPiece tokenizing**NLP 모델이 학습한 단어 조각으로 분리

!pip install transformers

!mkdir wordPieceTokenizer

from tokenizers import BertWordPieceTokenizer

*# Initialize an empty tokenizer*

wp\_tokenizer = BertWordPieceTokenizer(

clean\_text=True, *# [이순신, ##은, ' ', 조선]에서 띄어쓰기 ' '를 지우고 싶다면 True // bert model은 모두 clean\_text가 true*

handle\_chinese\_chars=True, *# 본문 내에 있는 한자가 음절 단위로 분리*

strip\_accents=False, *# True: [YepHamza] -> [Yep, Hamza] // 대문자가 나오는 기준으로 분리*

lowercase=False, *# 모두 소문자로 변경 // False로 하는 것이 성능이 더 좋음*

)

*# And then train*

wp\_tokenizer.train(

files="[FILE\_PATH]",

vocab\_size=10000, *# 내가 만들고 싶은 vocab의 size이며, 크게 만들수록 음절 단위로 잘림*

min\_frequency=2, *# 2개 이하로 등장하면 vocab으로 만들지 않음*

show\_progress=True,

special\_tokens=["[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]"],

limit\_alphabet=1000,

wordpieces\_prefix="##" *# prefix // ##은 앞에 연결된 음절이 있다는 의미*

)

*# Save the files*

wp\_tokenizer.save\_model("wordPieceTokenizer", "my\_tokenizer")

text = "이순신은 조선 중기의 무신이다."

tokenized\_text = wp\_tokenizer.encode(text)

>>> print(tokenized\_text)

Encoding(num\_tokens=10, attributes=[ids, type\_ids, tokens, offsets, attention\_mask, special\_tokens\_mask, overflowing])

>>> print(tokenized\_text.tokens)

['이', '##순', '##신은', '조선', '중', '##기의', '무', '##신이', '##다', '.']

>>> print(tokenized\_text.ids)

[708, 1340, 7612, 1999, 756, 2601, 452, 8478, 1016, 16]

### 3) Tokenizer에서 sentence 1, sentence 2를 주는 방법

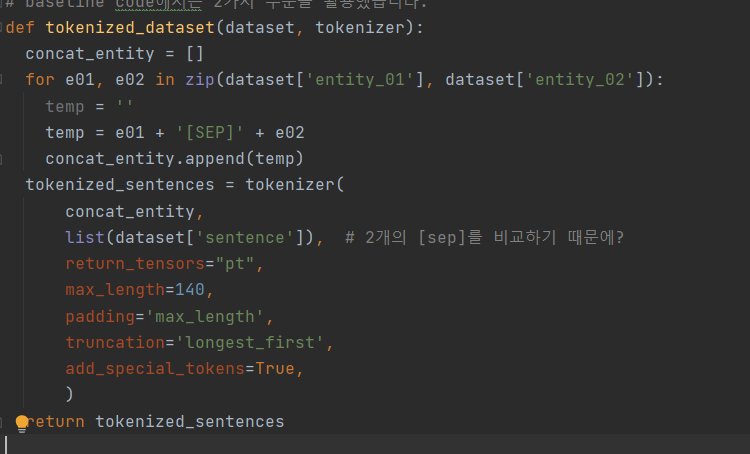
tokenizer에서 Sentence를 이어붙이는 방법에는 여러가지가 있습니다.

* **Sentence 1 + '[SEP]' + Sentence 2**

하지만 다음과 같이 sentence가 길어질 때는 tokenizer의 첫 번째, 두 번째 parameter를 사용할 수 있습니다.

* **entity01 [SEP] entity02 [SEP] 전체문장**

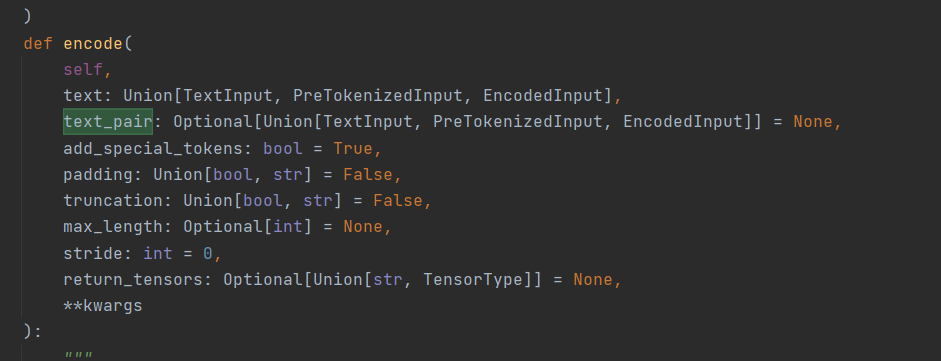
아래와 같이 tokenizer에서 concat\_entity, list(dataset['sentence'])를 첫 번째 parameter와 두 번째 parameter에 넣어서 사용할 수 있습니다.



원래는 tokenizer에 하나의 parameter에만 넣어서 사용했지만, 두 개의 parameter에 sentence가 들어가 있는 것을 보고 혼란이 와서 찾아보았습니다.

아래 코드와 같이 encode에는 text와 text\_pair로 구성되어 있어서 두 개의 parameter에 동시에 넣는 것이 가능합니다.

text와 text\_pair는 [SEP]로 나눠서 들어갑니다. (text[SEP]text\_pair)



### 4) 다양한Truncation 방법

Tokenizer에서 Truncation을 사용할 때 총 4가지의 값을 넣어줄 수 있습니다.

* **True** 혹은 **'longest\_first'** : sentence 1과 sentence 2 중 더 긴 문장을 자름
* **'only\_first'** : sentence 1을 무조건 자름
* **'only\_second'** : sentence 2를 무조건 자름
* **False** 혹은**'do\_not\_truncate'** : 자르지 않음

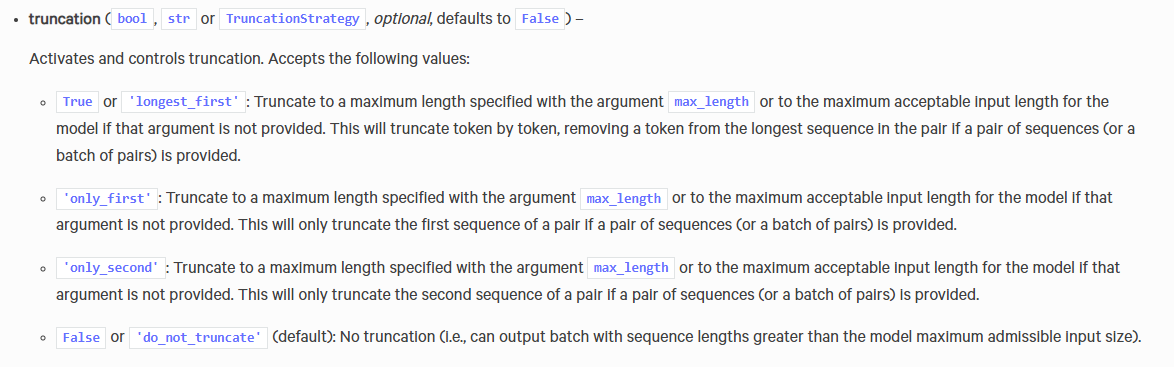


Figure 1. https://huggingface.co/transformers/main\_classes/tokenizer.html

### 5) Entity token 추가

Entity token을 추가해서 각 entity의 위치에 넣어주는 방법입니다. 따라서 [ENT] token과 [/ENT] token 두 개를 tokenizer에 speical token으로 넣습니다.

special\_tokens\_dict = {'additional\_special\_tokens': ['[ENT]', '[/ENT]']}

num\_added\_toks = tokenizer.add\_special\_tokens(special\_tokens\_dict)

이후 model에 token embedding의 크기를 늘려줍니다.

model.resize\_token\_embeddings(len(tokenizer))

tokenized 하기 전 dataset에서 해당 index에 [ENT], [/ENT] token을 넣어줍니다.

token들을 넣게되면 index의 위치가 변함으로 entity01 index가 먼저인지, entity02 index가 먼저인지를 확인 후 token을 넣어줍니다.

def tokenized\_dataset(dataset, tokenizer):

concat\_entity = []

for e01, e02 in zip(dataset['entity\_01'], dataset['entity\_02']):

temp = ''

temp = e01 + '[SEP]' + e02

concat\_entity.append(temp)

sentence\_list = []

for column\_name, sentence in dataset['sentence'].iteritems():

\_sentence = list(sentence)

if dataset['entity\_01\_idx0'][column\_name] < dataset['entity\_02\_idx0'][column\_name]:

\_sentence.insert(int(dataset['entity\_01\_idx0'][column\_name]), "[ENT]")

\_sentence.insert(int(dataset['entity\_01\_idx1'][column\_name])+2, "[/ENT]")

\_sentence.insert(int(dataset['entity\_02\_idx0'][column\_name])+2, "[ENT]")

\_sentence.insert(int(dataset['entity\_02\_idx1'][column\_name])+4, "[/ENT]")

else:

\_sentence.insert(int(dataset['entity\_02\_idx0'][column\_name]), "[ENT]")

\_sentence.insert(int(dataset['entity\_02\_idx1'][column\_name])+2, "[/ENT]")

\_sentence.insert(int(dataset['entity\_01\_idx0'][column\_name])+2, "[ENT]")

\_sentence.insert(int(dataset['entity\_01\_idx1'][column\_name])+4, "[/ENT]")

sentence\_list.append(''.join(\_sentence))

tokenized\_sentences = tokenizer(

concat\_entity,

sentence\_list,

return\_tensors="pt",

max\_length=160,

padding='max\_length',

truncation='longest\_first',

add\_special\_tokens=True,

)

return tokenized\_sentences

실험 결과는 [**4.평가 / 2) Entity token 추가 성능 비교**](#_2)_Entity_token) 부분에 있습니다.

### 6) pororo 사용한 tokenizer dataset 수정

캠퍼분이 올려주신 새로운 Entity 구분 방법 연구에 대해서 올려주셔서 따라해보았습니다.

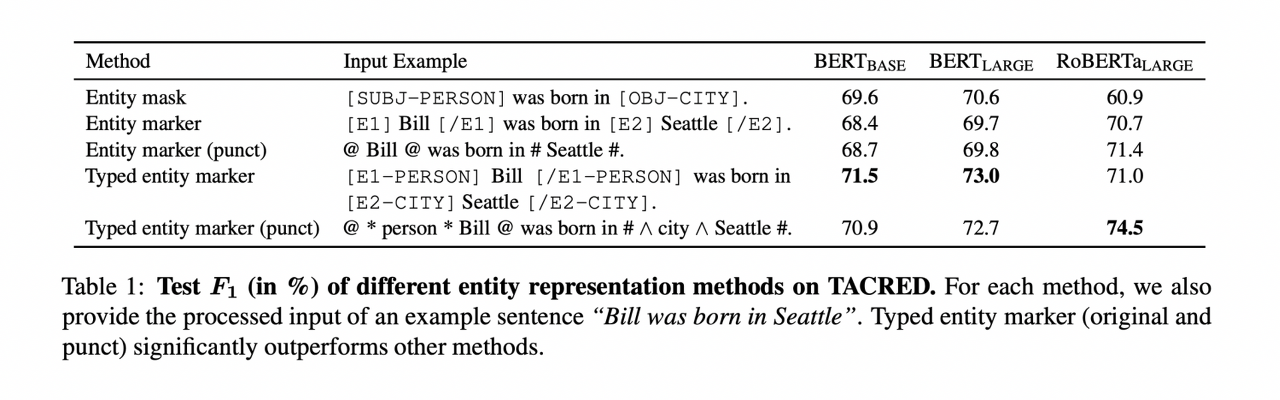


Figure 2. [Wenxuan Zhou et al., arXiv 2021] (링크 : https://arxiv.org/pdf/2102.01373.pdf)

즉, 이 방식은 **@ \* entity01의 개체명 \* entity01@**과 **#^ entity02의 개체명^ entity02 #** 형태로 넣어줍니다. #과 @는 entity를 감싸고, \*,^은 entity가 나타내는 개체를 감싸서 만드는 방식입니다.

https://blog.kakaocdn.net/dn/lJZ3i/btq26ppUwnH/HLUJ4Gnf7jy1pMFMwF8YCk/img.png

Figure 3. [Wenxuan Zhou et al., arXiv 2021]

따라서 tokenized\_dataset에서 형태를 변경해서 넣어주었습니다.

concat\_entity는 기존 방식이며, sentence\_list는 새롭게 #,@,\,^를 사용해 전체 sentence에서 entity와 개체명을 구분해주었습니다.

pororo를 import합니다.

from pororo import Pororo

ner은 한 문장만 추가하면 사용이 가능합니다.

*# pororo ner*

ner = Pororo(task="ner", lang="ko")

tokenized\_dataset을 작성합니다.

def tokenized\_dataset(dataset, tokenizer, ner):

concat\_entity = []

for e01, e02 in zip(dataset['entity\_01'], dataset['entity\_02']):

temp = ''

temp = '#'+e01+' # ' + '[SEP]' + '@' + e02 + ' @ '

concat\_entity.append(temp)

sentence\_list = []

for sent, ent01, ent02, start1, end1, start2, end2 in tqdm(zip(dataset['sentence'], dataset['entity\_01'], dataset['entity\_02'],\

dataset['entity\_01\_idx0'], dataset['entity\_01\_idx1'], dataset['entity\_02\_idx0'], dataset['entity\_02\_idx1']), total=len(dataset['sentence'])):

ner\_01 = ' ₩ '+ner(ent01)[0][1].lower()+' ₩ '

ner\_02 = ' ^ '+ner(ent02)[0][1].lower()+' ^ '

start1, end1 = int(start1), int(end1)

start2, end2 = int(start2), int(end2)

if start1 < start2:

sent = sent[:start1]+'#'+ner\_01+sent[start1:end1+1]+' # '+sent[end1+1:start2]+\

'@'+ner\_02+sent[start2:end2+1]+ ' @ '+sent[end2+1:]

else:

sent = sent[:start2] + '@' + ner\_02 + sent[start2:end2 + 1] + ' @ ' + sent[end2 + 1:start1] + \

'#' + ner\_01 + sent[start1:end1 + 1] + ' # ' + sent[end1 + 1:]

sentence\_list.append(sent)

tokenized\_sentences = tokenizer(

concat\_entity,

sentence\_list,

*# list(dataset['sentence']), # 2개의 [sep]를 비교하기 때문에?*

return\_tensors="pt",

max\_length=200,

padding='max\_length',

truncation='longest\_first',

add\_special\_tokens=True,

)

return tokenized\_sentences

이후, model에 special token을 추가해주었습니다.

special\_tokens\_dict = {'additional\_special\_tokens': ["#", "@", '₩', '^']}

num\_added\_toks = tokenizer.add\_special\_tokens(special\_tokens\_dict)

실험 결과는 [**4.평가 / 3) Pororo를 사용한 tokenizer dataset 수정 성능 비교**](#_3)_Pororo를_사용한) 부분에 있습니다.

### 7) Pororo NER 사용법 변경

어떤 캠퍼분이 올려주신 Pororo NER에서 tagging 방식을 수정해서 성능향상이 있었다고 하길래 관련 내용을 가지고 테스트 해보게 되었습니다.

def return\_tag(tagging\_list, is\_first):

tag = ''

if len(tagging\_list) != 1:

tagging = [tag[1] for tag in tagging\_list if tag[1] != 'O']

if tagging:

tag = ' '.join(list(set(tagging)))

else:

tag = 'o'

else:

tag = tagging\_list[0][1]

assert tag!='', 'tagging이 빔'

if is\_first:

return ' ₩ ' + tag.lower() + ' ₩ '

else:

return ' ^ ' + tag.lower() + ' ^ '

sentence\_list = []

for sent, ent01, ent02, start1, end1, start2, end2 in tqdm(zip(dataset['sentence'], dataset['entity\_01'], dataset['entity\_02'],\

dataset['entity\_01\_idx0'], dataset['entity\_01\_idx1'], dataset['entity\_02\_idx0'], dataset['entity\_02\_idx1']), total=len(dataset['sentence'])):

ner\_01 = return\_tag(ner(ent01), True)

ner\_02 = return\_tag(ner(ent02), False)



이 방식은 tagging\_list에서 하나를 선택해주는 것이 아닌, 'o'를 제외하고 하나의 문자열로 합쳐서 entity의 개체명으로 사용하는 방법입니다.

실험 결과는 [**4.평가 / 7) Pororo NER 변경 성능 비교**](#_7)_Pororo_NER_1) 부분에 있습니다.

## **[Model]**

### 1) BERT모델

huggingface의 transformers를 설치하고 불러옵니다.

!pip install transformers

from transformers import AutoModel, AutoTokenizer, BertTokenizer

104개의 언어를 통째로 학습한 모델 multi-lingual bert model를 사용합니다.

MODEL\_NAME = "bert-base-multilingual-cased" *# 104개 언어를 통째로 학습한 모델*

model = AutoModel.from\_pretrained(MODEL\_NAME)

tokenizer = AutoTokenizer.from\_pretrained(MODEL\_NAME)

Multi-lingual bert model은 총 vocab size는 119,547개 정도되며, 그 중 한국어는 8000개 정도된다고 합니다.

>>> print(tokenizer.vocab\_size) *# 한국어는 8000개 정도의 vocab*

119547

tokenizer를 하게되면 bert model은 총 3개의 값들을 출력합니다.

tokenized\_input\_text = tokenizer(text, return\_tensors="pt") *# return\_tensors는 pytorch로 반환*

for key, value in tokenized\_input\_text.items():

print("{}:\n\t{}".format(key, value))

input\_ids: *# token -> vocab의 ID*

tensor([[ 101, 9638, 119064, 25387, 10892, 59906, 9694, 46874, 9294,

25387, 11925, 119, 102]])

token\_type\_ids: *# Sentence 구분 (sentence2는 1로 초기화됨)*

tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])

attention\_mask: *# special token (pad, cls, sep) 구분 / padding은 0으로 초기화*

tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])

여기서 tokenizer.tokenize를 사용하면 입력된 texts를 tokenizing해서 명시적으로 보여줍니다.

tokenized\_text = tokenizer.tokenize(text)

>>> print(tokenized\_text)

['이', '##순', '##신', '##은', '조선', '중', '##기의', '무', '##신', '##이다', '.']

text를 encode하면 앞 뒤로 [CLS], [SEP] special token이 자동으로 붙습니다.

따라서 101 (CLS), 102 (SEP)이 붙습니다.

input\_ids = tokenizer.encode(text)

>>> print(input\_ids)

[101, 9638, 119064, 25387, 10892, 59906, 9694, 46874, 9294, 25387, 11925, 119, 102]

### 2) BERT에 Entity Layer 추가

BERT의 Embedding layer에 Entity embedding layer를 추가해서 사용하는 방법을 구상했습니다.

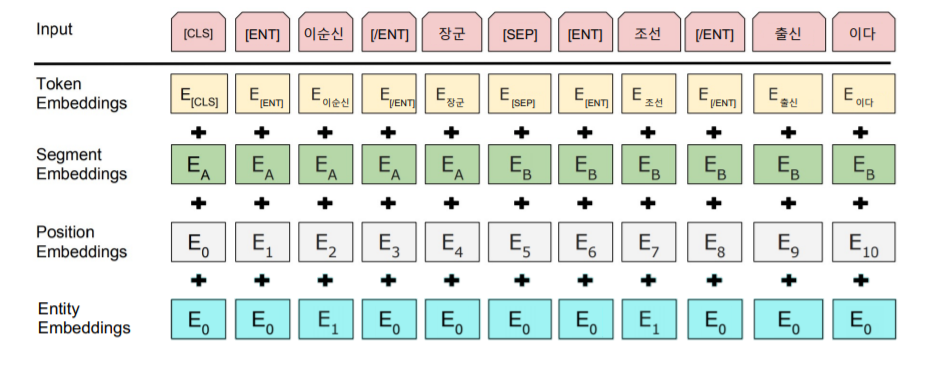


Figure 4. https://www.edwith.org/bcaitech1

Huggingface docs를 살펴보니 BertEmbeddings 클래스를 약간 수정해서 layer를 추가할 수 있을 것 같았습니다.

아래 사진은 huggingface에 있는 BertEmbeddings class입니다.



Figure 5. https://huggingface.co/transformers/\_modules/transformers/models/bert/modeling\_bert.html#BertModel

여기서 entity\_embeddings를 추가해줍니다.

물론 여기서 input size, hidden size를 모두 token\_type\_embedding 형태를 그대로 따라 했고 필요에 따라 수정해야 합니다.

class BertEmbeddings(nn.Module):

"""Construct the embeddings from word, position and token\_type embeddings."""

def \_\_init\_\_(self, config):

super().\_\_init\_\_()

self.word\_embeddings = nn.Embedding(config.vocab\_size, config.hidden\_size, padding\_idx=config.pad\_token\_id)

self.position\_embeddings = nn.Embedding(config.max\_position\_embeddings, config.hidden\_size)

self.token\_type\_embeddings = nn.Embedding(config.type\_vocab\_size, config.hidden\_size)

self.entity\_embeddings = nn.Embedding(config.type\_vocab\_size, config.hidden\_size) *# entity\_embedding 추가*

*# self.LayerNorm is not snake-cased to stick with TensorFlow model variable name and be able to load*

*# any TensorFlow checkpoint file*

self.LayerNorm = nn.LayerNorm(config.hidden\_size, eps=config.layer\_norm\_eps)

self.dropout = nn.Dropout(config.hidden\_dropout\_prob)

*# position\_ids (1, len position emb) is contiguous in memory and exported when serialized*

self.register\_buffer("position\_ids", torch.arange(config.max\_position\_embeddings).expand((1, -1)))

self.position\_embedding\_type = getattr(config, "position\_embedding\_type", "absolute")

def forward(

self, input\_ids=None, token\_type\_ids=None, position\_ids=None, inputs\_embeds=None, past\_key\_values\_length=0

):

if input\_ids is not None:

input\_shape = input\_ids.size()

else:

input\_shape = inputs\_embeds.size()[:-1]

seq\_length = input\_shape[1]

if position\_ids is None:

position\_ids = self.position\_ids[:, past\_key\_values\_length : seq\_length + past\_key\_values\_length]

if token\_type\_ids is None:

token\_type\_ids = torch.zeros(input\_shape, dtype=torch.long, device=self.position\_ids.device)

if inputs\_embeds is None:

inputs\_embeds = self.word\_embeddings(input\_ids)

token\_type\_embeddings = self.token\_type\_embeddings(token\_type\_ids)

entity\_embeddings = self.entity\_embeddings(token\_type\_ids)

embeddings = inputs\_embeds + token\_type\_embeddings + entity\_embeddings

if self.position\_embedding\_type == "absolute":

position\_embeddings = self.position\_embeddings(position\_ids)

embeddings += position\_embeddings

embeddings = self.LayerNorm(embeddings)

embeddings = self.dropout(embeddings)

return embeddings

그리고 BertConfig를 가지고 와서 새롭게 정의한 BertEmbeddings 클래스에 넣어줍니다.

configuration = BertConfig()

custom\_bertembeddings = BertEmbeddings(configuration)

저는 BertForSequenceClassification을 사용하기 때문에 BERT backbone 모델과 classification layer를 동시에 사용합니다.

따라서 custom\_bertembeddings를 넣으려면 다음과 같이 넣어주어야 합니다.

model.bert.embeddings = custom\_bertEmbeddings

이후, model에서 input\_ids, attention\_mask, labels와 함께 entity를 추가로 넣을 수 있게 Dataset을 수정했습니다.

([ENT]를 만나면 10이 들어가고 [/ENT] 일 경우는 0이 들어감)

class RE\_Dataset(torch.utils.data.Dataset):

def \_\_init\_\_(self, tokenized\_dataset, labels):

self.tokenized\_dataset = tokenized\_dataset

self.labels = labels

def \_\_getitem\_\_(self, idx):

item = {key: torch.tensor(val[idx]) for key, val in self.tokenized\_dataset.items()}

item['labels'] = torch.tensor(self.labels[idx])

entity = []

flag = False

for value in item['input\_ids']:

value = int(value)

if value == 119548:

flag = False

if flag:

entity.append(10)

else:

entity.append(0)

if value == 119547:

flag = True

item['entity'] =torch.tensor(entity)

return item

하지만 model에서 forward에 들어가는 parameter와 여러 가지 상속이 되어있는 복잡한 코드 때문에 test는 못했습니다.

### 3) xlm-roberta-large 사용

transformer에서 xlm-roberta 관련 값들을 가지고 옵니다.

from transformers import XLMRobertaTokenizer, XLMRobertaForSequenceClassification, XLMRobertaTokenizerFast

XLMRobertaTokenizer는 Tokenizer와 TokenizerFast가 있는데 이 두 개의 차이점은 **SentencePiece와 BPE** 방식입니다.

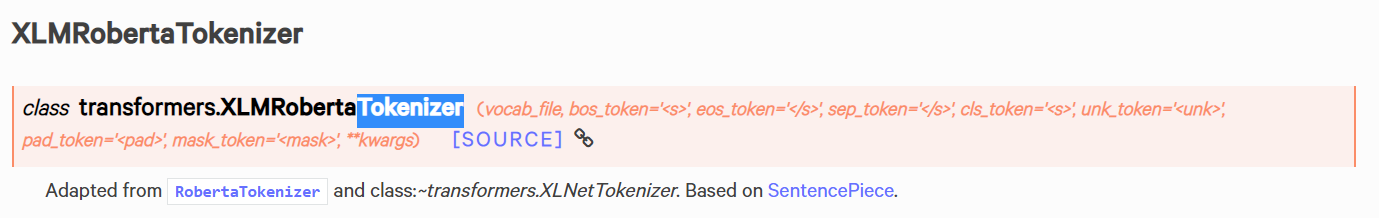


Figure 6. https://huggingface.co/transformers/model\_doc/xlmroberta.html

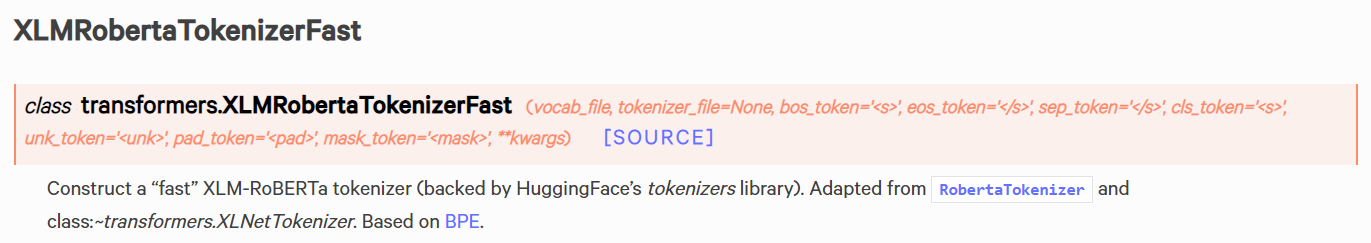


Figure 7. https://huggingface.co/transformers/model\_doc/xlmroberta.html

두 개 모두 사용해보려고 합니다.

tokenizer = XLMRobertaTokenizer.from\_pretrained(MODEL\_NAME)

tokenizer = XLMRobertaTokenizerFast.from\_pretrained(MODEL\_NAME)

모델은 다음과 같이 불러올 수 있습니다.

단일 분류인 XLMRobertaForSequenceClassification을 사용하겠습니다.

bert\_config = XLMRobertaConfig.from\_pretrained(MODEL\_NAME)

model = XLMRobertaForSequenceClassification.from\_pretrained(MODEL\_NAME, config=bert\_config)

실험 결과는 [**4.평가 / 3) Pororo를 사용한 tokenizer dataset 수정 성능 비교**](#_3)_Pororo를_사용한) 부분에 있습니다.

## **[Training]**

### 1) Custom Trainer 작성

Huggingface에서 제공하는 trainer를 사용하지 않고, Epoch과 Optimizer, 그리고 여러 변수들을 추가하기 위해 새롭게 Trainer를 다시 작성했습니다.

device = torch.device('cuda') if torch.cuda.is\_available() else torch.device('cpu')

train\_loader = DataLoader(RE\_train\_dataset, batch\_size=16, shuffle=True, num\_workers=8)

valid\_loader = DataLoader(RE\_valid\_dataset, batch\_size=16, shuffle=True, num\_workers=8)

optim = AdamW(model.parameters(), lr=5e-5)

*# loss\_fn = LabelSmoothingLoss()*

model.train()

EPOCHS, print\_every = 3, 1

for epoch in range(EPOCHS):

loss\_val\_sum = 0

for batch in tqdm(train\_loader):

optim.zero\_grad()

input\_ids = batch['input\_ids'].to(device)

attention\_mask = batch['attention\_mask'].to(device)

labels = batch['labels'].to(device)

outputs = model(input\_ids, attention\_mask=attention\_mask, labels=labels)

loss = outputs[0]

loss.backward()

optim.step()

loss\_val\_sum += loss

loss\_val\_avg = loss\_val\_sum / len(train\_loader)

if ((epoch % print\_every) == 0 or epoch == (EPOCHS - 1)):

train\_accr = func\_eval(model, train\_loader, device)

valid\_accr = func\_eval(model, valid\_loader, device)

print ("epoch:[%d] loss:[%.3f] train\_accr:[%.3f] valid\_accr:[%.3f]."%(epoch,loss\_val\_avg,train\_accr,valid\_accr))

valid\_data에 대해 accur, loss 측정을 위해 func\_eval function을 정의합니다.

def func\_eval(model, data\_iter, device):

with torch.no\_grad():

n\_total, n\_correct = 0, 0

model.eval() *# evaluate*

for batch in tqdm(data\_iter):

input\_ids = batch['input\_ids'].to(device)

attention\_mask = batch['attention\_mask'].to(device)

labels = batch['labels'].to(device)

outputs = model(input\_ids, attention\_mask=attention\_mask, labels=labels)

logits = outputs[1]

y\_pred = torch.argmax(logits, axis=-1)

n\_correct += (y\_pred==labels).sum().item()

n\_total += len(y\_pred)

val\_accr = (n\_correct / n\_total)

model.train() *# back to train mode*

return val\_accr

Huggingface의 Bert를 사용하면서 편한 점은 Fine-tuning된 모델들을 자유롭게 사용할 수 있다는 점이었습니다.

CLS token만 검사하는 단일 분류 BertForSequenceClassification을 사용했는데 **loss와 logits**값을 return해주니 train code를 편리하게 짤 수 있습니다.

### 2) Hard Votes Ensemble 사용

Ensemble을 사용하기 위해 기존에 만들어 두었던 CSV를 사용했습니다.

(Soft votes나 K-fold cross-validation을 사용하고 싶었지만 시간관계상 부족했습니다. 또한, 모델의 크기도 너무 커서 각각의 모델을 가지고 있기에 어려움이 있었습니다.)

Hard votes는 상위 5개의 csv파일을 가지고 와서 Ensemble했습니다.

각각의 성능은 78.5%, 78%, 77.6%, 77%, 76.3%의 csv파일을 가지고 왔습니다.

import pandas as pd

from collections import Counter

import operator

MULTI\_LABEL\_CLASSES = 42

\_multi\_list = [ *# 앙상블하고 싶은 파일 path*

'/content/output (4).csv',

'/content/output (5).csv',

'/content/output (6).csv',

'/content/output (7).csv',

'/content/output (8).csv',

]

multi = []

for idx, path in enumerate(\_multi\_list):

multi.append(pd.read\_csv(path))

submission = {'pred':[]}

for j in range(len(multi[0]['pred'])):

temp = []

for i in range(len(multi)):

temp.append(multi[i]['pred'][j])

c = Counter(temp)

max\_key = max(c.items(), key=operator.itemgetter(1))[0] *# 가장 votes가 많이 나온 key값*

submission['pred'].append(max\_key)

pd\_submission = pd.DataFrame.from\_dict(submission)

pd\_submission.to\_csv('submission.csv')

실험 결과는 [**4.평가 / 6) Hard votes ensemble 성능 비교**](#_6)_Hard_votes) 부분에 있습니다.

# **4. 평가**

## **1) 평가 지표 Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **날짜** | Data processing & Tokenizer | Model | Training | Time | Accuracy |
| 4/13 | - EDA - |  |  | 1h | - |
|  | " |  | - new baseline code - | 30m | 59.3000% |
| 4/14 | " | - KoBERT - | - epoch : 20 max\_len : 128 batch\_size : 32 - | 23m | 72.0000% |
| 4/17 | " | - "bert-base-multilingual-cased" BERT - | - epoch : 4 batch\_size : 16 - | 14m | 72.8000% |
| 4/19 | - "entity01[SEP]entity02" + "[ENT]가 들어간 전체 문장" - | " | " | 16m | 71.2000% |
|  | - "[ENT]가 들어간 전체 문장" - | " | " | 17m | 52.5000% |
| 4/20 | - pororo NER - "#,@,\,^를 사용해서 구분한 전체 문장" - | " | - epoch : 10 checkpoint : 5000 - | 50m | 70.8000% |
|  | - "#entity01 # [SEP] @entity02 @" + "#,@,\,^ 전체 문장" - | " | - checkpoint : 2500 - | 16m | 71.6000% |
|  | **-** "# \ entity01개체명 \ entity01 # [SEP] @ ^ entity02 개체명 ^ entity02 @" + "#,@,\,^ 전체 문장" - | " | - epoch : 5 checkpoint : 1500 - | 12m | 74.5000% |
| 4/21 | - "entity01[SEP]entity02" + "[ENT]가 들어간 전체 문장" - xlmrobertaTokenizer (sentencePiece) - | - xlm-roberta-large - | " | 31m | 74.6000% |
|  | **-** "# \ entity01개체명 \ entity01 # [SEP] @ ^ entity02 개체명 ^ entity02 @" + "#,@,\,^ 전체 문장" - | " | - epoch : 10 checkpoint : 2400 batch\_size : 32 label\_smoothing\_factor : 0.5 - | 51m | 76.3000% |
|  | " | " | - checkpoint : 2000 batch\_size : 16 label\_smoothing\_factor:0.3 - | 31m | 77.6000% |
|  | " | " | - checkpoint : 2600 batch\_size : 20 label\_smoothing\_factor : 0.4 | 46m | 78.5000% |
| 4/22 | " | " | - checkpoint-2200,  batch\_size-25 - | 34m | 77.0000% |
|  | " | " | - checkpoint-2600,  batch\_size-20,  label\_smoothing\_factor : X - | 33m | 76.2000% |
|  | " | " | - Hard Votes Ensemble - | 1m | 78.4000% |
|  | - Pororo NER 수정 - | " | - checkpoint-2400,  batch\_size-20,  label\_smoothing\_factor : 0.4 - | 46m | 79.3000% |

## **2) Entity token 추가 성능 비교**

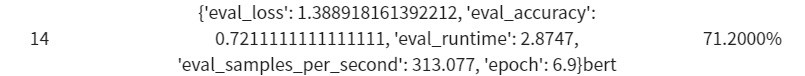
실험 과정은 [**3. 기술적인 도전 / [Data Porcessing] / 5) Entity token추가**](#_5)_Entity_token) 에 있습니다.

다음은 시도한 방법 및 결과입니다.

* **(기존방식)** **"entity01[SEP]entity02" + "전체 문장"  
    
  - 하이퍼파라미터 : epoch-4, checkpoint-2000, batch\_size-16, Model-"bert-base-multilingual-cased"**

https://blog.kakaocdn.net/dn/coXyXL/btq22GyLEYJ/k54CoZFqkajqyKR1hmzQ5K/img.png

* **(시도 1) "entity01[SEP]entity02" + "[ENT]가 들어간 전체 문장"**  
    
  - **하이퍼파라미터 :** epoch-7, checkpoint-3500, batch\_size-16, Model-"bert-base-multilingual-cased"  
    
  - Public score를 비교해보았을 때 성능이 약간 떨어지는 것을 알 수 있습니다.



* **(시도 2) "[ENT]가 들어간 전체 문장"**  
    
  - **하이퍼파라미터 :** epoch-7, checkpoint-2000, batch\_size-16, Model-"bert-base-multilingual-cased"  
    
  - Public score를 비교해보았을 떄 성능이 많이 떨어지는 것을 알 수 있습니다.

https://blog.kakaocdn.net/dn/ehOwrn/btq23uEQ9QZ/cZjl1ikLtK4QKH8jtrLAF0/img.png

## **3) Pororo를 사용한 tokenizer dataset 수정 성능 비교**

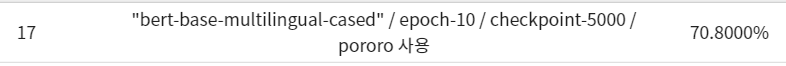
실험 과정은 [**3. 기술적인 도전 / [Data Processing] / 6) pororo 사용한 tokenizer dataset 수정**](#_6)_pororo_사용한)에 있습니다.

다음은 시도한 방법 및 결과입니다.

* **(기존방식) tokenized\_sentence : "entity01[SEP]entity02" + "전체 문장"  
    
  - 하이퍼파라미터 :**epoch-4, checkpoint-2000, batch\_size-16, Model-"bert-base-multilingual-cased"

https://blog.kakaocdn.net/dn/coKdgX/btq21pqXGC5/EyGGwjUnPkauWKfGDfRQY0/img.png

* **(시도 1) tokenized\_sentence : "#,@,\,^를 사용해서 구분한 전체 문장"**  
    
  - **하이퍼파라미터 :** epoch-10, checkpoint-5000, batch\_size-16, Model-"bert-base-multilingual-cased"  
    
  - Public score를 비교해보았을 때 성능이 약간 떨어지는 것을 알 수 있습니다.



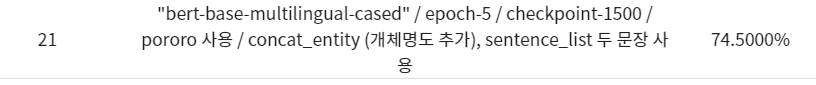
* **(시도 2) tokenized\_sentence : "#entity01 # [SEP] @entity02 @" + "#,@,\,^ 전체 문장"**  
    
  - **하이퍼파라미터 :** epoch-10, checkpoint-2500, batch\_size-16, Model-"bert-base-multilingual-cased"  
    
  - 기존 방식보다는 성능이 떨어지지만, 시도1보다는 약간의 성능이 오른 것을 알 수 있습니다.

https://blog.kakaocdn.net/dn/U6idt/btq232BTOCJ/tagBwhw16UQpcTYNnvKLLk/img.png

* **(시도 3) tokenized\_sentence : "#entity01 # [SEP] @entity02 @" + "#,@,\,^ 전체 문장"**  
    
  - **하이퍼파라미터 :** epoch-20, checkpoint-9000, batch\_size-16, Model-"bert-base-multilingual-cased"  
    
  - 시도 2에서 더 많은 epoch을 돌렸을 때의 결과입니다.  
    
  - overfitting을 체크하기 위해 돌려봤는데 시도 2보다 성능이 내려간 것을 알 수 있습니다.

https://blog.kakaocdn.net/dn/dBmj0s/btq23sVcvnG/lxWHvrk6LGjrjcx6u3tFo0/img.png

* **(시도 4) tokenized\_sentence : "# \ entity01개체명 \ entity01 # [SEP] @ ^ entity02 개체명 ^ entity02 @" + "#,@,\,^ 전체 문장"**  
    
  - **하이퍼파라미터 :** epoch-5, checkpoint-1500, batch\_size-16, Model-"bert-base-multilingual-cased"  
    
  - **가장 좋은 성능**을 보였으며, bert 모델임에도 불구하고 74.5% 성능이 나왔습니다.



pororo의 NER 기능은 정말 강력한 것 같습니다.

논문에서 이미 tokenizer의 여러 방향에 대해서 연구했고, 이 부분을 따라하는 것이 좋다고 생각합니다.

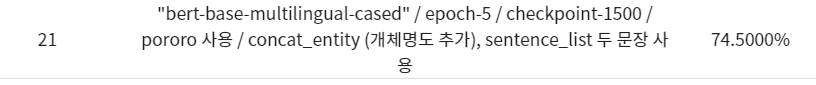
하지만 논문에서는 roberta를 사용했기 때문에 저도 차후에 roberta나 xlm-roberta로 성능을 평가해 보았습니다.

## **4) XLM-Roberta-Large 모델 & Tokenizer(Fast) 성능 비교**

실험 과정은 [**3. 기술적인 도전 / [Model] / 3) xlm-roberta-large 사용**](#_3)_xlm-roberta-large_사용) 에 있습니다.

 다음은 시도한 방법 및 결과입니다. (변경사항은 노란색 음영 표시)

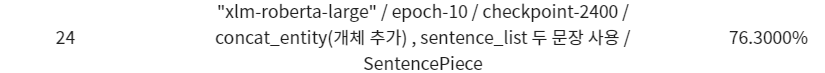
* **(기존방식)** **Bert 모델 + BertTokenizer**  
  - **하이퍼파라미터 :** epoch-5, checkpoint-1500, batch\_size-16, Model-"bert-base-multilingual-cased"  
    
  **- tokenized\_sentence :**"# \ entity01개체명 \ entity01 # [SEP] @ ^ entity02 개체명 ^ entity02 @" + "#,@,\,^ 전체 문장"



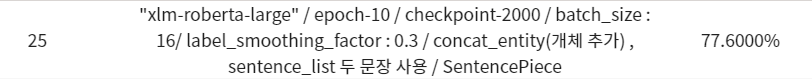
* **(시도 1) xlm-roberta 모델 + xlmrobertaTokenizer**  
    
  - **하이퍼파라미터 :** epoch-5, checkpoint-1500, batch\_size-16, Model-"xlm-roberta-large"  
    
  - **tokenized\_sentence :** **"entity01[SEP]entity02"+ "전체 문장"**- tokenized\_sentence를 다시 원래대로 바꾸니 전체적인 성능이 약간 밖에 안 올랐습니다.

https://blog.kakaocdn.net/dn/uP7ZM/btq3bVg8KXY/QxyZ2odUud2FgEBdQy3MFK/img.png

* **(시도 2) xlm-roberta 모델 + xlmrobertaTokenizer + tokenized\_sentence 변경**  
    
  - **하이퍼파라미터 :** epoch-10, checkpoint-2400, batch\_size-32, label\_smoothing\_factor : 0.5, Model-"xlm-roberta-large"  
    
  - **tokenized\_sentence : "# \ entity01개체명 \ entity01 # [SEP] @ ^ entity02 개체명 ^ entity02 @" + "#,@,\,^ 전체 문장"**  
    
  - tokenized\_sentence를 바꾸고 조금 더 많은 epoch을 돌리니 Accuracy가 2%정도 향상되었습니다.



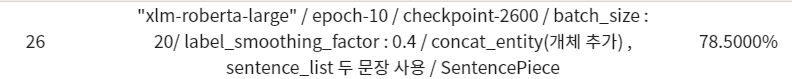
* **(시도 3) xlm-roberta 모델 + xlmrobertaTokenizer + 하이퍼파라미터 변경**  
    
  - **하이퍼파라미터 :** epoch-10, checkpoint-2000, batch\_size : 16, label\_smoothing\_factor : 0.3, Model-"xlm-roberta-large"  
    
  - **tokenized\_sentence :** "# \ entity01개체명 \ entity01 # [SEP] @ ^ entity02 개체명 ^ entity02 @" + "#,@,\,^ 전체 문장"  
    
  - batch\_size를 16으로 줄이고 checkpoint 2000에서 가장 좋은 eval 성능이 나왔길래 해당 값을 사용했습니다.



* **(시도 4) xlm-roberta 모델 + xlmrobertaTokenizerFast (BPE)**  
    
  - **하이퍼파라미터 :** epoch-10, checkpoint-2200, batch\_size-32, label\_smoothing\_factor: 0.3, Model-"xlm-roberta-large"  
    
  - **tokenized\_sentence :** "# \ entity01개체명 \ entity01 # [SEP] @ ^ entity02 개체명 ^ entity02 @" + "#,@,\,^ 전체 문장"  
    
  - PB에 제출은 안했지만 일반 Tokenizer보다 좋지 않은 성능을 보여 취소.

https://blog.kakaocdn.net/dn/CoIpc/btq26FAkaBu/YmFphe5OJYEjBtcILP7J40/img.png

* **(시도 5) xlm-roberta 모델 + xlmrobertaTokenizer + 하이퍼파라미터 변경**  
    
  - **하이퍼파라미터 :** epoch-10, checkpoint-2600, batch\_size-20, label\_smoothing\_factor: 0.4, Model-"xlm-roberta-large"  
    
  - **tokenized\_sentence :** "# \ entity01개체명 \ entity01 # [SEP] @ ^ entity02 개체명 ^ entity02 @" + "#,@,\,^ 전체 문장"  
    
  - eval accuracy가 가장 좋은 checkpoint를 inference하니 성능이 좋게 나왔습니다.



 큰 모델을 사용하는 것이 성능향상에 도움이 되는 것을 깨달았습니다.

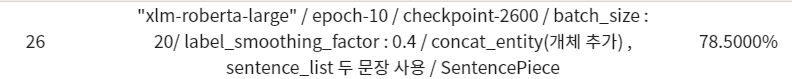
tokenizer를 아무리 해도 BERT로는 성능 향상에 한계가 있었는데, large 모델로 변경하니 성능이 눈에 띄게 좋아졌습니다.

## **5) 최적의 Hyperparameter 찾기**

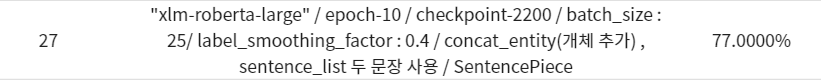
대회 마지막날이였기 때문에 최적의 Hyperparameter를 찾아 성능을 높이는 것에 주력했습니다.

다음은 시도한 방법 및 결과입니다. (변경사항은 노란색 음영 표시)

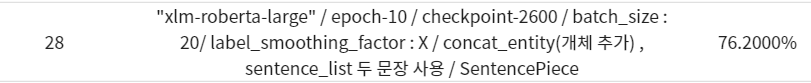
* **(기존 방식) xlm-roberta 모델 + xlmrobertaTokenizer**  
    
  - **하이퍼파라미터 :** epoch-10, checkpoint-2600, batch\_size-20, label\_smoothing\_factor: 0.4, Model-"xlm-roberta-large"  
    
  - **tokenized\_sentence :** "# \ entity01개체명 \ entity01 # [SEP] @ ^ entity02 개체명 ^ entity02 @" + "#,@,\,^ 전체 문장"



* **(시도 1) xlm-roberta 모델 + xlmrobertaTokenizer + 하이퍼파라미터 변경**  
    
  - **하이퍼파라미터 :** epoch-10, checkpoint-2200, batch\_size-25, label\_smoothing\_factor : 0.4, Model-"xlm-roberta-large"  
    
  - **tokenized\_sentence :**"# \ entity01개체명 \ entity01 # [SEP] @ ^ entity02 개체명 ^ entity02 @" + "#,@,\,^ 전체 문장"  
    
  - batch size를 늘려주니 오히려 성능 하락이 있었습니다.



* **(시도 2) xlm-roberta 모델 + xlmrobertaTokenizer + 하이퍼파라미터 변경**  
    
  - **하이퍼파라미터 :** epoch-10, checkpoint-2600, batch\_size-20, label\_smoothing\_factor : X, Model-"xlm-roberta-large"  
    
  - **tokenized\_sentence :**"# \ entity01개체명 \ entity01 # [SEP] @ ^ entity02 개체명 ^ entity02 @" + "#,@,\,^ 전체 문장"  
    
  - Label smoothing factor를 빼니 성능이 더 떨어졌습니다.



## **6) Hard votes ensemble 성능 비교**

실험 과정은 [**3. 기술적인 도전 / [Training] / 2) Hard Votes Ensemble 사용**](#_2)_Hard_Votes)에 있습니다.

다음은 시도한 방법 및 결과입니다.

* **(시도 1) hard votes ensemble**  
    
  - 현재 최고 성능인 78.5%보다 0.1% 떨어진 성능이 나왔습니다.  
    
  - 성능이 떨어지는 4개를 votes했기 때문에 약간 떨어지게 나왔던 것 같습니다. 만약 78%가 넘는 csv파일들을 모두 ensemble했다면 더 좋은 성능이 나왔을 것 같습니다.

https://blog.kakaocdn.net/dn/bjm7Xz/btq3i9GyEfl/JeuwRlSlnYPi3kfs916nTk/img.png

## **7) Pororo NER 변경 성능 비교**

실험 과정은 [**3. 기술적인 도전 / [Data Processing] / 7) Pororo NER 사용법 변경**](#_7)_Pororo_NER)에 있습니다.

이 방법에 대한 결과는 다음과 같습니다.

* **(시도 1) xlm-roberta 모델 + xlmrobertaTokenizer + Pororo tagging list 수정**  
    
  - **하이퍼파라미터 :** epoch-10, checkpoint-2600, batch\_size-20, label\_smoothing\_factor : 0.4, Model-"xlm-roberta-large"  
    
  - **tokenized\_sentence :**"# \ entity01개체명 \ entity01 # [SEP] @ ^ entity02 개체명 ^ entity02 @" + "#,@,\,^ 전체 문장"  
    
  - NER을 바꿔줘서 넣어주니 성능향상이 있었습니다.

https://blog.kakaocdn.net/dn/bifvRl/btq3caAD9FB/vHBAb9w6LtZkB5fduovUv1/img.png

# **학습과정에서의 교훈**

* 피어세션을 진행하면서 개인의 구글 Searching보다 집단 지성이 더 효과적이라는 것을 배웠습니다.   
    
  물론 기본적인 구글 Searching은 중요한 요소 중 하나지만, 피어세션과 토론 게시판을 통해 사람들과 함께 지식을 공유하고 소통하면서 전체적으로 더욱 성장하는 시간을 가졌습니다.
* 이론과 실제 프로젝트는 다릅니다.  
    
  프로젝트를 하면서 정말 많이 느낀 부분입니다. 이론적으로 “어떤 Model이 좋다”, “어떤 Loss가 좋다”라는 글을 많이 봤지만, 실제로 프로젝트를 진행하면서 큰 모델보다 작은 모델에서 성능이 좋게 나온 경우도 있었고 Focal Loss(Class imbalance 문제 해결 때문)보다 Cross Entropy가 성능이 더 높게 나온 경우도 있었습니다.
* 토론 게시판의 공유 문화가 좋았습니다.  
    
  피어님들이 사용한 데이터, 모델, 하이퍼파라미터 공유와 그에 따른 결과를 알게 되어, 프로젝트를 하면서 탄력을 받았던 부분들이 있었습니다. 또한, 자잘한 팁 공유를 통해 많은 도움을 받았던 것 같습니다.
* NLP분야는 Tokenizer에 따라 성능이 천차만별인 것 같습니다.  
    
  Pstage 1 Image Classfication에서는 Data Augmentation이 가장 큰 task 중 하나였다면, Pstage 2 KLUE문제 에서는 tokenizer를 어떻게 사용하냐에 따라 성능이 달라졌습니다. 또한 여러가지 창의적 방법들로 문제를 해결해 나갈 수 있었다는 점이 큰 도움이 되었던 것 같습니다.
* KLUE의 많은 Metric, Data, Model들에 대해 알게되는 계기가 되었습니다.

U stage에서는 Transformer의 구조와 GLUE에 대해서만 많이 배워서, 현업에서는 ‘한국어’에 대해 어떠한 연구와 개발이 되고있는지 궁금했습니다. P stage2를 통해서 SKT, NAVER, KAKAO에서 연구되고 배포하는 다양한 모델들과 다양한 ‘한국어’ 데이터셋들을 어떻게 다루어야 하는지에 대해 배울 수 있었던 좋은 기회인 것 같습니다.

# **마주한 한계와 도전숙제**

* **아쉬웠던 점들**

1. NLP 모델을 가져다 쓰기 때문에 직접 그 구조를 코드로 작성해보지 못한 것
2. 더 다양한 NER 방법을 논문으로 찾아보고 적용해보지 못한 것
3. K-Fold Validation을 통한 Ensemble을 못해본 것
4. 어떻게 동작하는지 파헤치기보다는 성능 향상에 급급해 모듈을 가져다 쓰는 경우가 많았던 것 같습니다. 가장 경계해야할 부분입니다.
5. 어떤 캠퍼분이 Huggingface 오픈 소스 오류 수정을 하신 것이 진짜 대단한 것 같습니다.
6. Entity layer을 추가한 모델의 성능을 TEST해보지 못한 것

* **한계/교훈을 바탕으로 다음 스테이지에서 새롭게 시도해볼 것**

1. 이론적인 내용을 정확히 알고 모듈을 갖다 쓰기
2. KLUE의 다양한 metric에 대해 공부하고 조사해보기
3. DST에서는 좀 더 심도있는 Language Model에 대해 공부, 실험할 것
4. (기회가 된다면) Huggingface 컨트리뷰터 도전해보기