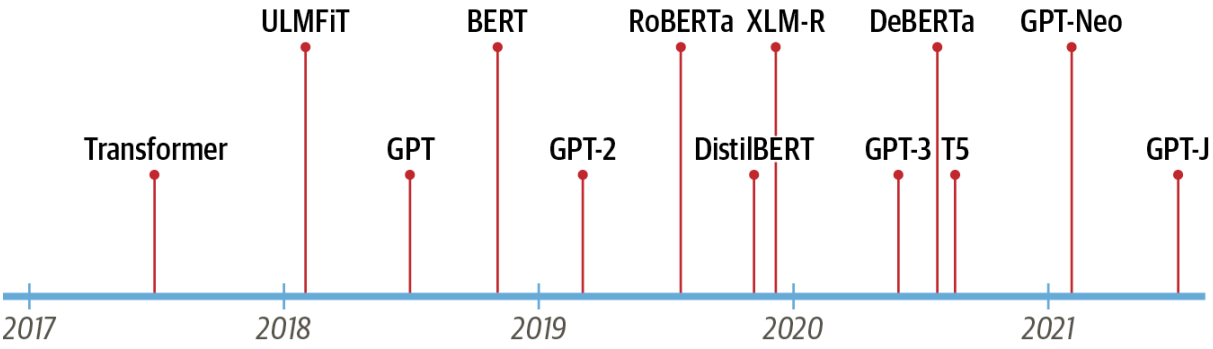
1주차 - 트랜스포머 소개 (1 1절~2 2절) 발표

1. 트랜스포머 소개

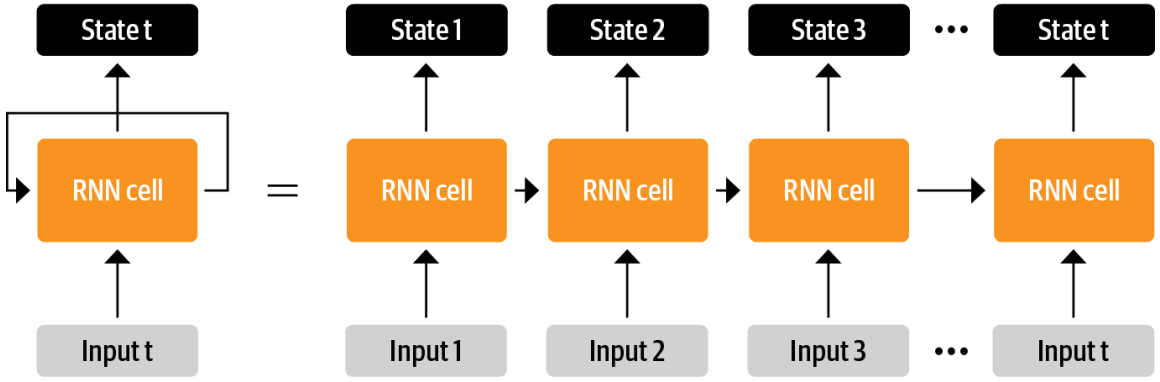
- 트랜스 포머 : 구글의 연구원들이 제안한 RNN의 성능을 능가하는 아키텍쳐.
- 트랜스포머 기반 모델들



- 이전 방법론 대비 트랜스포머의 새로운 점
  1. 인코더-디코더 프레임워크
  2. 어텐션 메커니즘
  3. 전이 학습

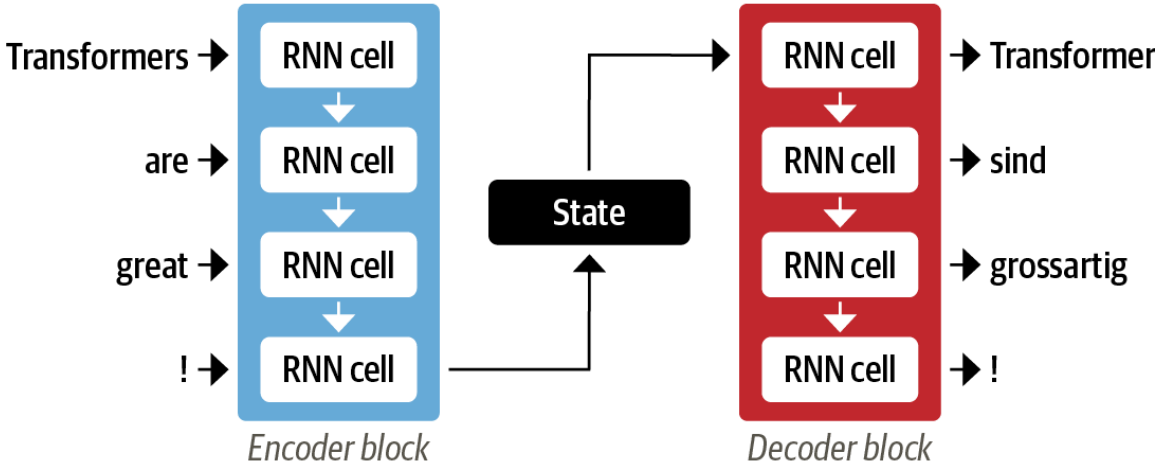
1.1 인코더-디코더 프레임 워크

- RNN?



- **현재 시점의 input 정보**와 **이전 시점의 모델이 출력한 정보**를 함께 받아서, 다음 시점의 모델이 출력할 정보를 만들어 냄.

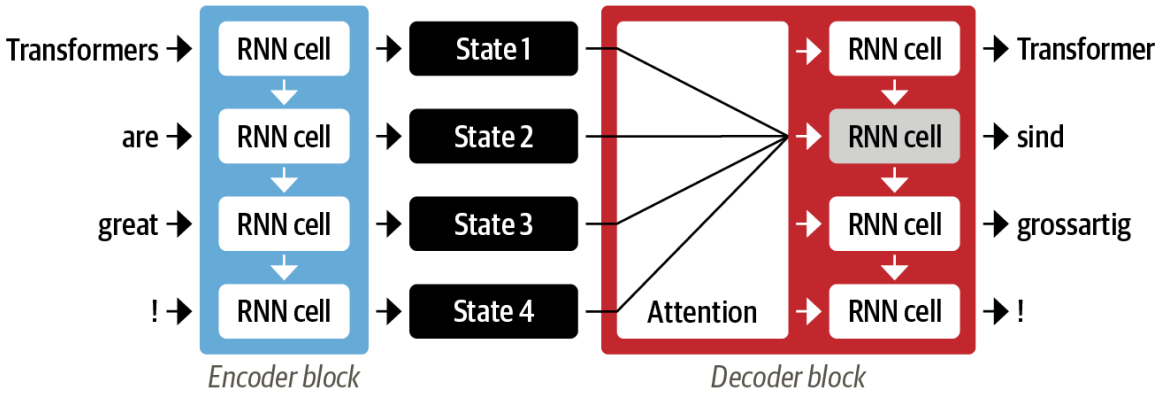
- 일반적으로 기계 번역 task를 할때 아래와 같이 **인코더-디코더 구조**를 사용함. 인코더에서 입력 정보를 인코딩하여 그 인코딩한 정보를 디코더에 넘겨주는 방식.



- 위 그림과 같이 인코더에서 입력 정보를 **last hidden state**라고 부르는 정보로 인코딩 한다. 위 그림에서는 (State)를 의미. **representation 정보**라고도 한다. 이 인코딩 정보는 디코더로 전달된다. 디코더는 이 정보를 받아 번역 등의 task를 수행.
- **문제점** : 이렇게 인코딩 하는 과정에서 정보 병목 현상이 발생하는 문제가 있다. → 입력 정보를 모두 하나의 state에 압축하는 과정에서 정보가 손실될 수 있다. **이 문제를 Attention 방식을 이용해서 해결할 수 있다.**

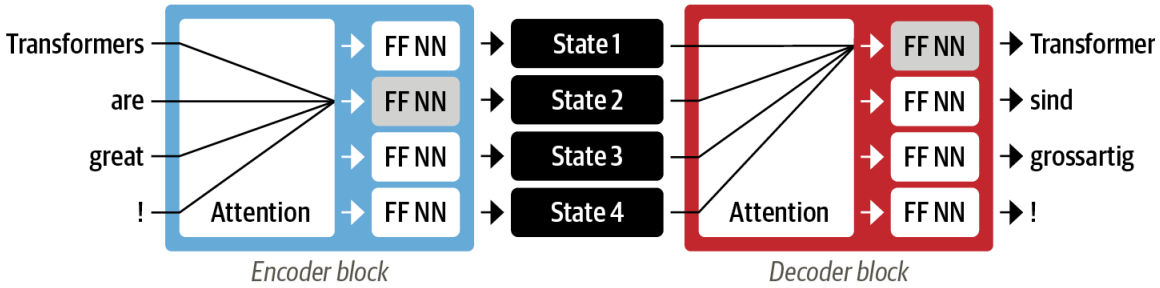
1.2 어텐션 메커니즘

- 어텐션



- 어텐션은 위와 같이 동작한다. RNN과 달리 각 timestep(cell)에서 다음 상태로 넘겨주기 위한 hidden state가 아닌, 디코더에서 참고할 hidden state를 만들어낸다.
- 디코더는 이 정보들을 활용하여 매 timestep에 올 적절한 단어를 예측하게 된다. 이렇게 하면 RNN의 **하나의 정보에 압축되어 발생하는 정보 소실 문제**를 해결할 수 있다.
- 문제점 : 그림에서도 알 수 있듯이 RNN의 순환구조를 가져가서, 계산이 순차적으로 진행되어 시간이 오래 걸린다는 문제는 아직 남아 있다. → 행렬 연산 불가능

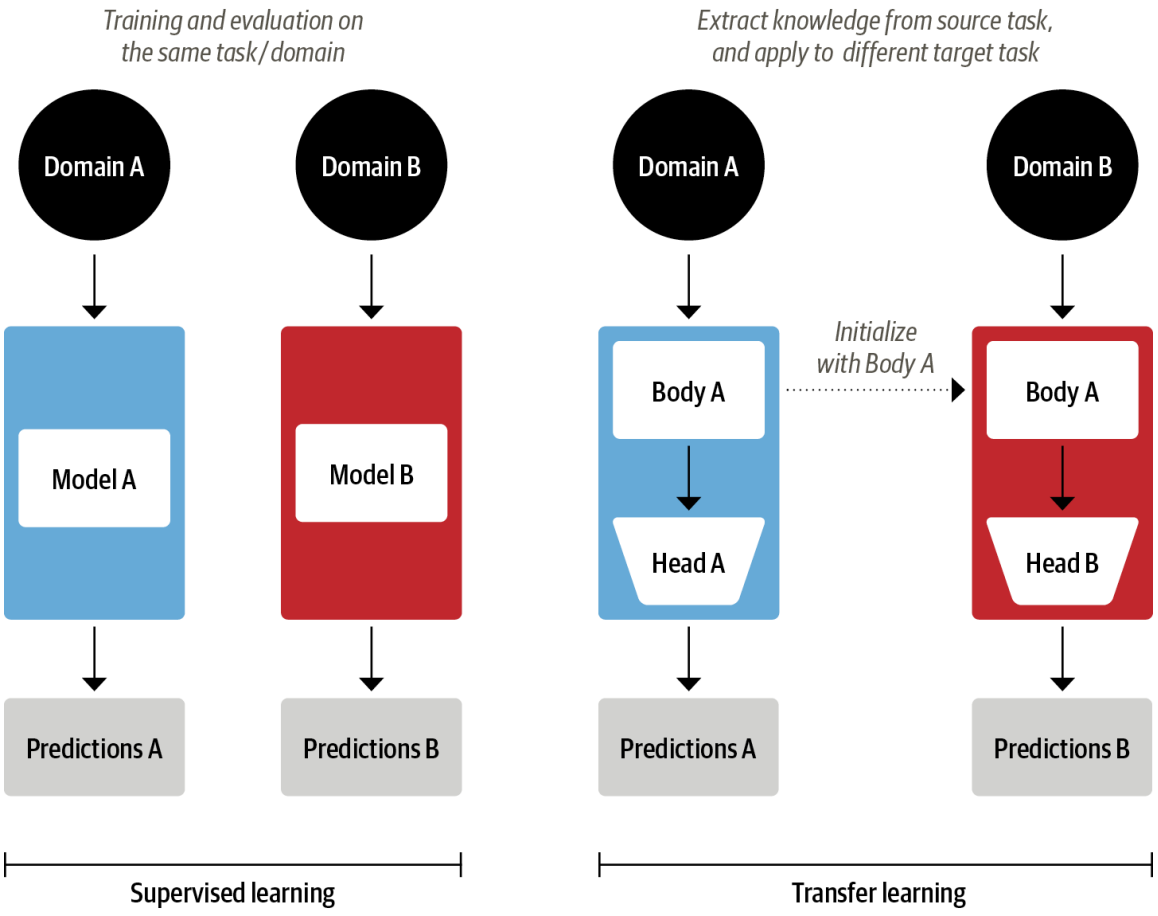
- 셀프 어텐션



- 순환적인 요소를 전부 제거함. 구체적인 동작은 3장에서 다시 다룰 예정.
- 속도가 매우 빨라져 훨씬 많은 양의 데이터로 훈련가능하게 되었다.

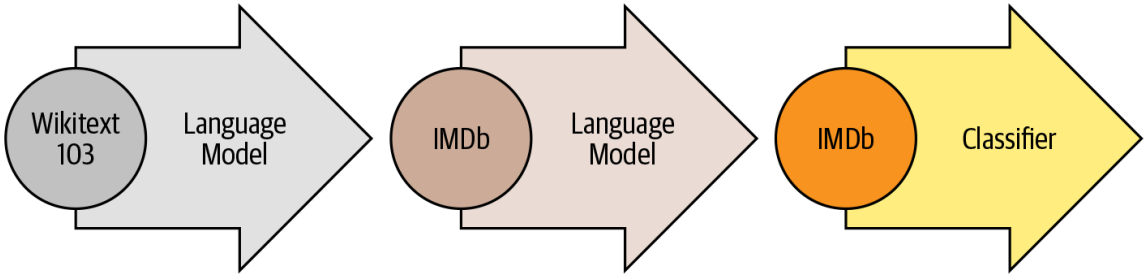
1.3 NLP의 전이 학습

- 보통 body와 head로 나뉘는데, 원래의 도메인에서 많은 양의 데이터로 body를 학습시키고(pre-train), 자기가 원하는 downstream task에 맞게 상대적으로 적은양의 데이터를 사용해 fine tuning하여 성능을 끌어 올리는 방법.



- 일반적으로 비전 분야에서는 ImageNet과 같은 데이터 셋에서 body를 훈련시키고 downstream task에 맞게 fine tuning 한다.
- NLP에서는 전이 학습이 잘 안되고 있었음.

- OpenAI에서 NLP의 전이학습을 수행하는 새로운 방식인 **ULMFiT** (**Universal Language Model Fine-tuning for Text Classification**) 제안. 3가지 단계!



1. **사전 훈련** : **이전 단어로 다음 단어를 예측**하는 방식 사용 → Language Modeling → 주로 인터넷 상의 매우 많은 양의 데이터로 사전 훈련함.
2. **도메인 적응 (Domain adaptation)** : 언어 모델을 대규모 말뭉치에서 사전 훈련한 후, 다음 단계로 도메인 내의 있는 말뭉치에 적응시킴. 위 그림에서는 IMDb라는 영

화 리뷰 말뭉치에 적응시킴. → 여전히 Language Modeling 방식 사용 (다음 단어 예측)

3. **미세 튜닝 (fine tuning)** : 그림처럼 Classifier 같은 모듈 하나 달아서 원하는 task 수행 (영화리뷰 감성분석 등)

- 위 ULMFiT 와 셀프어텐션을 활용하여 GPT와 BERT 가 탄생.
  - **GPT**는 트랜스포머의 디코더만 사용, ULMFiT 같은 언어 모델링 방식 사용
  - **BERT**는 트랜스포머의 인코더를 사용하여 Masked language modeling을 활용한 언어 모델링 사용. → I am a student에서, I am a [MASK] 처럼 마스크를 주고 원래 단어인 student를 예측하도록 모델을 학습.
- 이러한 다양한 NLP 모델이 나오면서 서로 호환되는 프레임워크가 필요했기에 huggingface transformers가 생겨났다. → 단일화된 API 구축!

1.4 허깅페이스 트랜스포머스

- 하나의 새로운 머신러닝 아키텍쳐를 새로운 작업에 적용하기 위해 아래와같은 작업 필수적
  1. 모델 아키텍처를 코드로 구현
  2. 서버로부터 사전 훈련된 가중치 로드
  3. 입력 데이터 전처리 후, 모델에 전달. 그다음 해당 작업에 맞는 사후처리 수행.
  4. 데이터 로더, 로스 등 학습에 필요한 것들 구현
- **허깅페이스의 Transformers**가 위 작업을 수행하기 위한 하나의 표준화된 인터페이스를 제공해준다.

1.5 트랜스포머스 어플리케이션 둘러보기

1.5.1 텍스트 분류

- 예시 : 고객 피드백 텍스트의 감성을 분류
- 트랜스포머스는 아래와 같이 API를 제공하여 편리하게 모델을 가져올 수 있음. pipeline 사용.

```
from transformers import pipeline

classifier = pipeline("text-classification")
```

- 위 처럼 코드 작성하면 허깅페이스 허브에 저장된 가중치를 자동으로 다운해 옴.

- pipeline은 텍스트 문자열을 받아서 예측 리스트를 반환한다.

```
import pandas as pd

outputs = classifier(text)
pd.DataFrame(outputs)
```

	label	score
0	NEGATIVE	0.901546

- 매우 높은 확률로 해당 Text를 부정적(Negative)으로 판단.

1.5.2 개체명 인식

- **개체명**? : 제품, 장소, 사람 같은 실제 객체를 의미.
- 개체명 인식 : 개체명을 실제로 Text에서 추출하는 task. (Named Entity Recognition) → NER
- 고객의 피드백이 특정 제품과 서비스 중 무엇인지 알고 싶을때 사용 가능.
- 역시 아래와 같이 pipeline을 통해 쉽게 구현 가능

```
ner_tagger = pipeline("ner", aggregation_strategy="simple"
outputs = ner_tagger(text)
pd.DataFrame(outputs)
```

- 모델 예측에 따라 단어를 그룹화 하기 위해 **aggregation_strategy(?)** 사용. Optimus Prime이 두 단어로 구성되지만 하나의 카테 고리로 할당.

	entity_group	score	word	start	end
0	ORG	0.879010	Amazon	5	11
1	MISC	0.990859	Optimus Prime	36	49
2	LOC	0.999755	Germany	90	97
3	MISC	0.556570	Mega	208	212
4	PER	0.590257	##tron	212	216
5	ORG	0.669692	Decept	253	259
6	MISC	0.498350	##icons	259	264
7	MISC	0.775361	Megatron	350	358
8	MISC	0.987854	Optimus Prime	367	380
9	PER	0.812096	Bumblebee	502	511

- ORG : 조직, LOC : 위치, PER : 사람, MISC : 기타
- score는 얼마나 모델이 개체명을 확신하는지 나타냄.

1.5.3 질문 답변 (QA)

- 질문을 던지면 모델은 답변을 반환.

```
reader = pipeline("question-answering")
question = "What does the customer want?"
outputs = reader(question=question, context=text)
pd.DataFrame([outputs])
```

	score	start	end	answer
0	0.631292	335	358	an exchange of Megatron

- “추출적 질문 답변” 방식을 통해 답변을 줌. → 문자 index에 해당하는 start와 end 도 준다. 7장에서 자세히 배울 예정.

1.5.4 요약

- 주어진 텍스트 요약

```
summarizer = pipeline("summarization")
outputs = summarizer(text, max_length=60, clean_up_tokeniz
print(outputs[0]['summary_text'])
```

```
 Bumblebee ordered an Optimus Prime action figure from your online store in
Germany. Unfortunately, when I opened the package, I discovered to my horror
that I had been sent an action figure of Megatron instead. As a lifelong enemy
of the Decepticons, I hope you can understand
```

1.5.5 번역

- 다른 언어로 번역.

```
translator = pipeline("translation_en_to_de",
                      model="Helsinki-NLP/opus-mt-en-de")
outputs = translator(text, clean_up_tokenization_spaces=Tr
print(outputs[0]['translation_text'])
```

```
Sehr geehrter Amazon, letzte Woche habe ich eine Optimus Prime Action Figur aus
Ihrem Online-Shop in Deutschland bestellt. Leider, als ich das Paket öffnete,
entdeckte ich zu meinem Entsetzen, dass ich stattdessen eine Action Figur von
Megatron geschickt worden war! Als lebenslanger Feind der Decepticons, Ich
hoffe, Sie können mein Dilemma verstehen. Um das Problem zu lösen, Ich fordere
einen Austausch von Megatron für die Optimus Prime Figur habe ich bestellt.
Eingeschlossen sind Kopien meiner Aufzeichnungen über diesen Kauf. Ich erwarte,
von Ihnen bald zu hören. Aufrichtig, Bumblebee.
```

1.5.6 텍스트 생성

- 예시

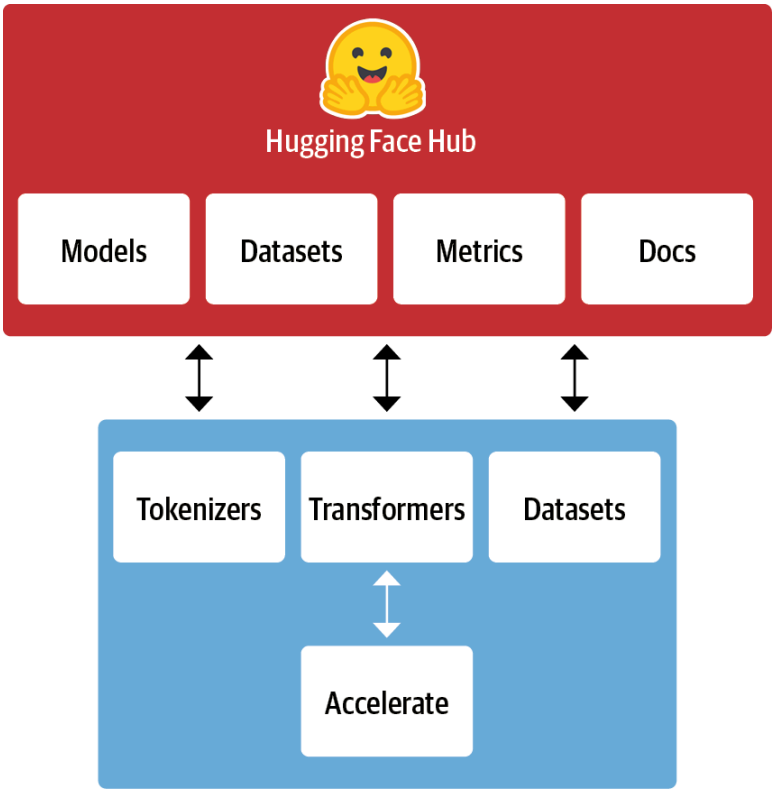
```
generator = pipeline("text-generation")
response = "Dear Bumblebee, I am sorry to hear that your o
prompt = text + "\n\nCustomer service response:\n" + respo
outputs = generator(prompt, max_length=200)
print(outputs[0]['generated_text'])
```

- 효율적으로 데이터를 처리하고, 결과를 동료와 공유하고, 작업을 재현가능하게 하기 위해 트랜스포머스를 활용해야 한다.

- 결론적으로 1.5에서 하고싶은 말은, **NLP의 다양한 task를 huggingface의 Transformers를 이용해서 매우 간편하게 구현**할 수 있음을 보여준다. 위 모델들은 이미 **임의의 데이터 셋**으로 **각 task에 맞게 파인튜닝** 되었지만, **자신의 데이터에 맞게 파인튜닝**을 할 수도 있다. 이는 다음 장에서 배운다.

1.6 허깅페이스 생태계

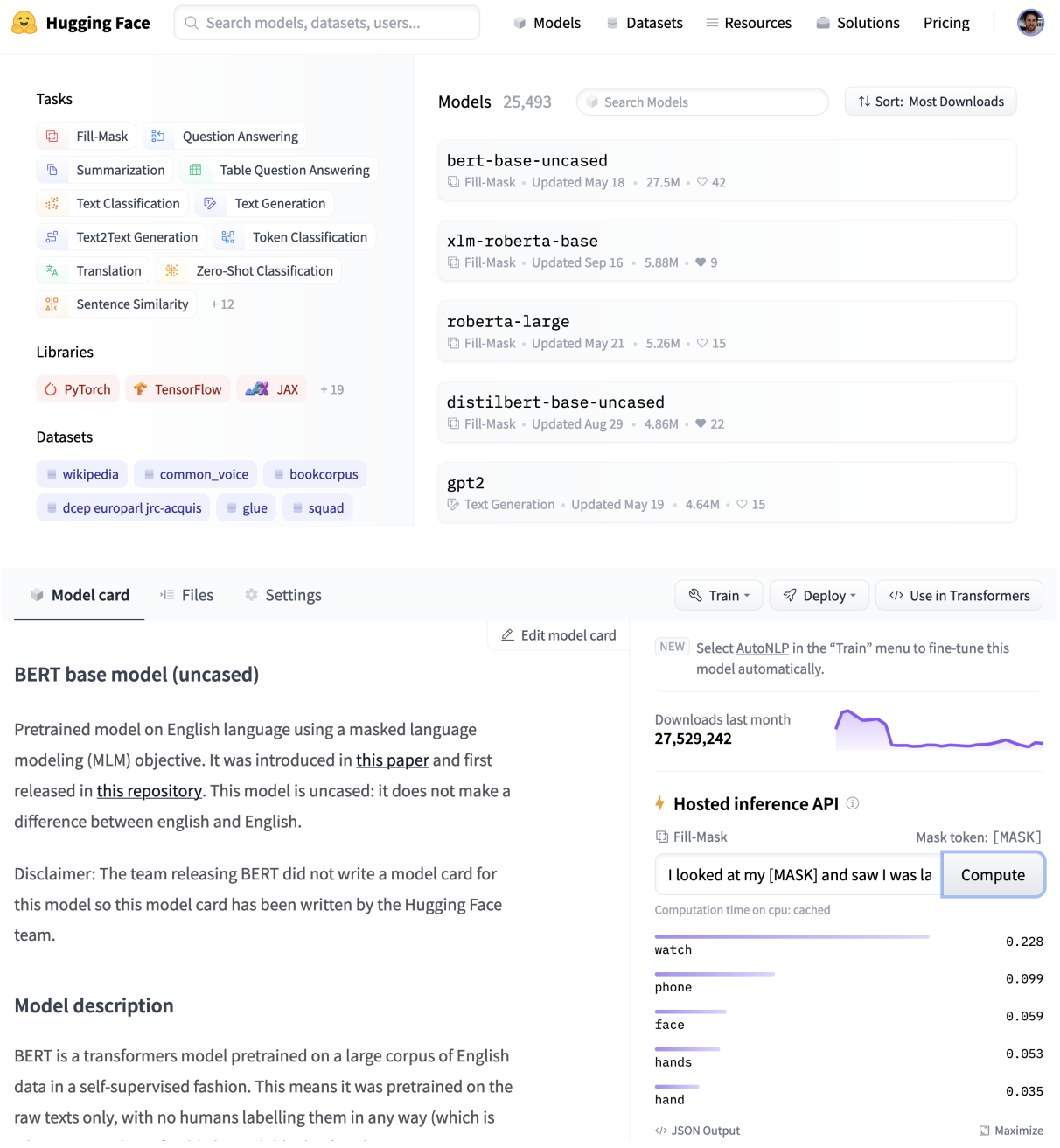
- 허깅 페이스는 크게 라이브러리와 허브로 구성된다. **라이브러리는 코드를 제공**하고, **허브는 사전 훈련된 모델 가중치, 데이터셋, 평가지표를 위한 스크립트 등을 제공**한다.



1.6.1 허깅페이스 허브

- 사전 훈련된 수많은 모델들이 존재(프레임워크, 데이터셋 등으로 필터링하여 원하는 모델 탐색 가능). 모델 외에도 데이터셋과 평가 지표를 위한 스크립트 등도 호스팅 하고 있음.

- 모델과 데이터셋 내용을 문서화한 **모델 카드와 데이터셋 카드**도 제공함.



1.6.2 허깅페이스 토크나이저

- 토큰화 : 텍스트를 더 작은 단위로 분할
- 각 모델마다 잘 맞는 토크나이저가 다른데, 이를 AutoTokenizer 같은 것을 활용하여 바로 받아올 수가 있음.

1.6.3 허깅페이스 데이터셋

- 데이터 로드, 처리, 저장이 번거로운 과정임. 허깅페이스 데이터셋이 이를 편하게 해줌.

- 스마트한 캐싱을 제공하고 메모리 매핑이라는 특별한 매커니즘을 사용하여 램 부족을 피함.
  메모리 매핑은 파일 내용을 가상 메모리에 저장하고 여러 개의 프로세스로 더 효율적으로 파일을 수정함.
- NLP 평가지표의 구현은 다 조금씩 다를 수 있지만, **허깅페이스 데이터셋은 평가 지표를 위한 스크립트를** 제공하여 실험의 재현 가능선과 신뢰성을 높임!
- 10장에서 훈련 루프를 미세하게 조정할 필요가 있는데, 이를 위해 허깅페이스 엑셀러레이트가 필요하다! (multi GPU 사용)

1.6.4 허깅페이스 엑셀러레이트

- 허깅페이스 엑셀러레이트는 사용자 정의 로직을 처리하는 일반적인 훈련 루프에 훈련 인프라에 필요한 추상화 층을 추가함. 필요한 인프라 전환을 단순화해 **워크플로우를 가속화함**. (예시 : multi GPU 사용)

1.7 트랜스포머의 주요 도전 과제

- 언어
  - 영어 외의 언어에 대해서 데이터 부족으로 인해 모델을 찾는 데에 어려움이 많다. → 4장에서 자세히 살펴봄.
- 데이터 가용성
  - 모델에 필요한 레이블링된 훈련 데이터의 양은 많다(전이 학습을 사용하더라도.) → **레이블링 된 데이터가 없거나 소량일때의 경우**는 9장에서 다룸.
- 긴 문서 처리
  - 셀프 어텐션은 텍스트 길이가 문단정도일때 잘 작동함. 길이가 매우 길어지면 비용이 많이듬. → 11장에서 이를 완화하는 방법 배움.
- 불투명성
  - **모델이 왜 그러한 선택을 했는지 설명하기 어려움**. 2장과 4장에서 모델의 오류를 조사하는 방법 배움.
- 편향
  - **인터넷의 텍스트 데이터를 사용했기에 인종차별, 성차별의 데이터가 존재할 수있어 편향이 존재.** → 10장에서 다룸

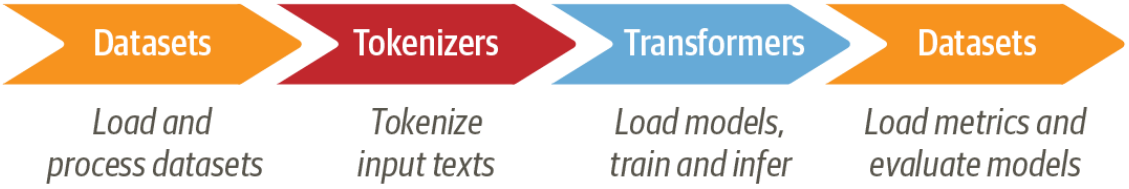
- 전 DPO 같은 방법으로 모델을 훈련 시키면 이런 편향이 사라진다고 알고 있습니다 !

2. 텍스트 분류

스팸 메일을 걸러내거나, 글의 감성을 분석하는 task 가 있음.

이 장에서는 DistilBERT라는 경량화 모델을 사용해서 텍스트 분류 진행.

아래 사진과 같이, 허깅페이스를 이용하여 텍스트 분류 task 를 직접 수행하자.



2.1 데이터셋

- 트위터 메시지 감정 분석을 연구한 논문의 데이터 셋 사용. → 분노, 기쁨, 등의 총 6가지의 감정으로 나뉜다.

2.1.1 허깅페이스 데이터셋 처음 사용하기

- list_datasets() 함수로 사용가능한 데이터셋 목록 출력 가능함. 그렇구나!

```
from huggingface_hub import list_datasets

all_datasets = [ds.id for ds in list_datasets()]
print(f"현재 허브에는 {len(all_datasets)}개의 데이터셋이 있습니다.
print(f"처음 10개 데이터셋: {all_datasets[:10]}")
```

- load_dataset() 함수로 emotion 데이터 로드.

```
from datasets import load_dataset

emotions = load_dataset("emotion")
```

- 출력

```
emotions
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 16000
    })
    validation: Dataset({
        features: ['text', 'label'],
        num_rows: 2000
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 2000
    })
})
```

- Dataset 객체

```
train_ds = emotions["train"]
train_ds
```

```
Dataset({
    features: ['text', 'label'],
    num_rows: 16000
})
```

  - **python의 리스트처럼 동작**함.
  - **하나의 행이 하나의 dictionary 형태로 표현**된다.

```
train_ds[0]
```

```
{'text': 'i didnt feel humiliated', 'label': 0}
```

  - 키는 column의 이름임.

```
train_ds.column_names
```

```
['text', 'label']
```

- 아래 함수로 **데이터 type이나 label**을 알 수 있음.

```
print(train_ds.features)
```

```
{'text': Value(dtype='string', id=None), 'label': ClassLabel(names=['sadness',
'joy', 'love', 'anger', 'fear', 'surprise'], id=None)}
```

- 슬라이스 연산자 : 를 사용하면 각 행이 **별개의 딕셔너리가 아닌,하나의 리스트로 표현**된다.

```
print(train_ds[:5])
```

```
{'text': ['i didnt feel humiliated', 'i can go from feeling so hopeless to so
damned hopeful just from being around someone who cares and is awake', 'im
grabbing a minute to post i feel greedy wrong', 'i am ever feeling nostalgic
about the fireplace i will know that it is still on the property', 'i am feeling
grouchy'], 'label': [0, 0, 3, 2, 3]}
```

허브에 필요한 데이터셋이 없다면?

- **대부분은 회사나 연구실의 데이터를 사용하여 작업한다. 아래와 같이 url을 입력하여 사용**.

```
dataset_url = "https://huggingface.co/datasets/transformer
emotions_remote = load_dataset("csv", data_files=dataset_u
                               names=["text", "label"])
```

```
emotions_remote
```

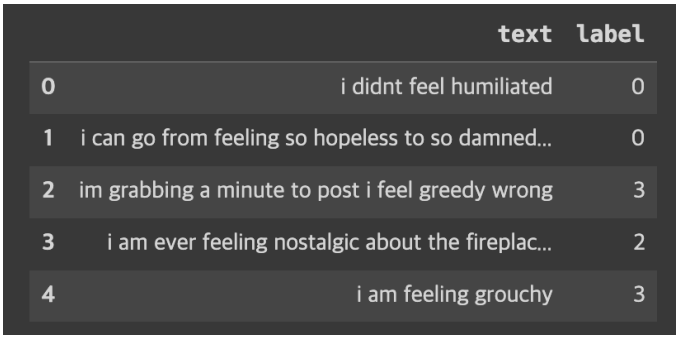
```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 16000
    })
})
```

2.1.2 데이터셋에서 데이터프레임으로

- **데이터 특성이나 분포를 파악하기 위해서 데이터 프레임으로 변경하자**.
- pandas를 사용. set_format() 메서드로 포맷 바꿀 수 있음. 이는 내부 데이터 포맷은 바꾸지 않으므로 나중에 필요에 따라 다른 포맷으로 변환이 가능하다.

```
import pandas as pd

emotions.set_format(type="pandas")
df = emotions["train"][:]
df.head()
```



	text	label
0	i didnt feel humiliated	0
1	i can go from feeling so hopeless to so damned...	0
2	im grabbing a minute to post i feel greedy wrong	3
3	i am ever feeling nostalgic about the fireplac...	2
4	i am feeling grouchy	3

- label이 숫자로 표시 되어 있으므로 아래와 같이 문자로 변경가능.

```
def label_int2str(row):
    return emotions["train"].features["label"].int2str(row
# str2int() 함수도 있음.
df["label_name"] = df["label"].apply(label_int2str)
df.head()
```

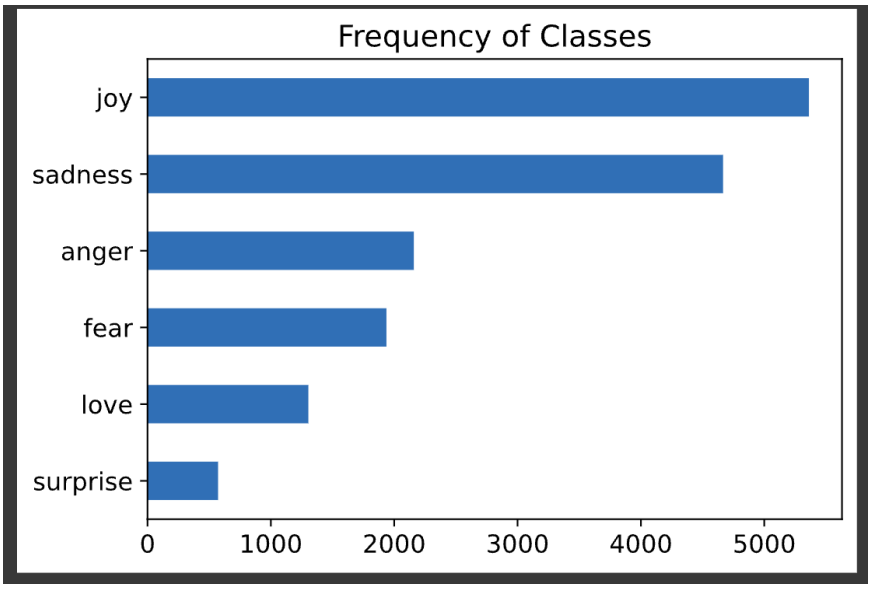
- **좋은 모델을 훈련하려면 데이터 셋을 더 자세히 둘러볼 필요가 있음.**

2.1.3 클래스 분포 살펴보기

- matplotlib 사용

```
import matplotlib.pyplot as plt

df["label_name"].value_counts(ascending=True).plot.barh()
plt.title("Frequency of Classes")
plt.show()
```

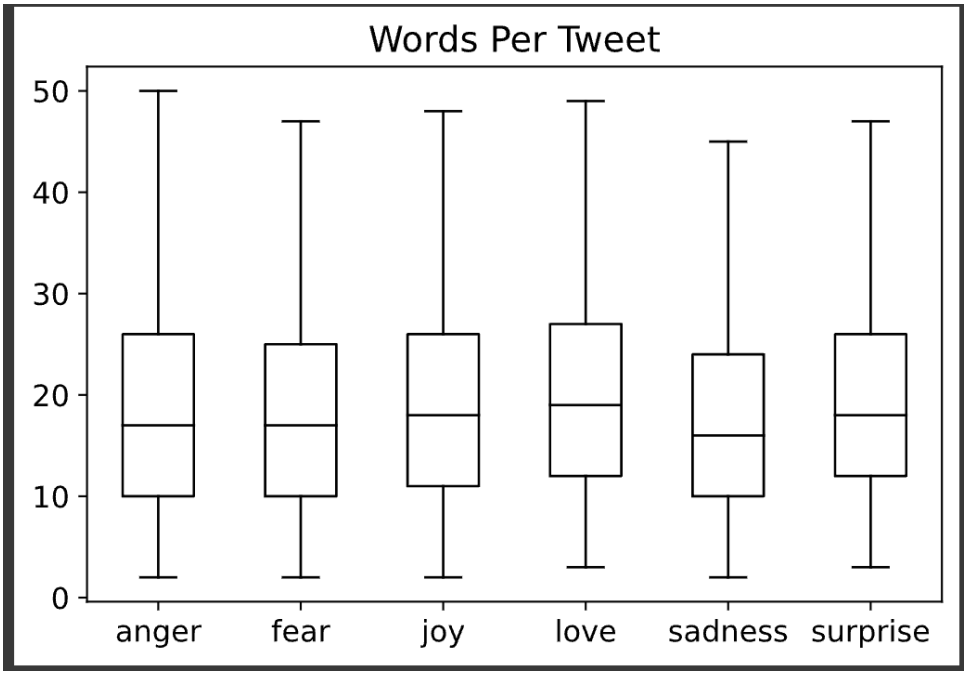


- 위와 같이 불균형이 심한 경우. 오버샘플링, 언더샘플링, 데이터 추가 수집 등의 방법 사용 가능. **train, test 분리전에는 샘플링 전략 사용하지 않기. (분포 보존하며 샘플링이 안될 수 있으니)**

2.1.4 트윗 길이 확인

- 트랜스포머 모델은 최대 문맥 크기 (maximum context size)라는 최대 입력 시퀀스가 있다. DistilBERT는 512토큰이다. 아래코드는 대충 단어 개수로 확인 (실제 토큰개수는 토크나이저에 따라 다름)

```
df["Words Per Tweet"] = df["text"].str.split().apply(len)
df.boxplot("Words Per Tweet", by="label_name", grid=False,
           color="black")
plt.suptitle("")
plt.xlabel("")
plt.show()
```



- 데이터 분포 확인 하면 이제 DataFrame 포맷 필요 없으니 데이터셋의 출력 포맷 초기화

```
emotions.reset_format()
```

2.2 텍스트에서 토큰으로

- 트랜스포머 모델들은 텍스트를 바로 입력으로 받지 못하고, 토큰화되어 인코딩 된 수치 벡터를 받을 수 있다.
  토큰화 방법은 여러가지가 있는데, 단어를 부분 단위 로 나누는 최적 분할 방법은 일반적으로 말뭉치에서 학습이 된다.
- 여기에서는 먼저 극단적으로 **문자 토큰화와 단어 토큰화**를 소개. → 현재는 사용하지 않는 방법임.

2.2.1 문자 토큰화

- text = "Tokenizing text is a core task of NLP" . 가있을때 아래와 같이 index를 매긴다.
  - {' ': 0, '.': 1, 'L': 2, 'N': 3, 'P': 4, 'T': 5, 'a': 6, 'c': 7, 'e': 8, 'f': 9,
    'g': 10, 'i': 11, 'k': 12, 'n': 13, 'o': 14, 'r': 15, 's': 16, 't': 17, 'x': 18,
    'z': 19}

- 위처럼 index로 토큰을 labeling하면 순서가 생기므로 좋지 않음. 이 문제를 해결하기 위해 원-핫 벡터 사용 변경한다. 아래는 T의 경우 예시.

```
토큰: T
텐서 인덱스: 5
원-핫 인코딩: tensor([0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

- 철자 오류나 희귀한 단어를 처리하는데 유용하지만, 문자 하나를 토큰으로 사용하면서 단어의 의미, 문장의 의미를 파악하기가 어렵다. → 이 의미를 학습하기 위해 상단한 비용이 들 수 있음.

  또한 각 문자별로 모두 원핫벡터로 변경하는 것은 모든 문자들 간에 거리가 동일하므로 매우 안좋은 벡터 표현법임. 따라서 NLP 분야에서는 텍스트의 일부 구조를 유지하는 방법 사용.

2.2.2 단어 토큰화

- 단어 토큰화는 단어 전체를 토큰으로 사용 (띄어쓰기 기준으로 토큰화)
- text = "Tokenizing text is a core task of NLP” . 가 있을때 아래와 같이 토큰화
  - ‘Tokenizing', 'text', 'is', 'a', 'core', 'task', 'of', 'NLP.’
- 너무 다양한 단어들이 존재하기 때문에 어휘사전이 너무 커질 수 있다. → 모델의 파라미터가 너무 커지기 때문에 문제. 존재하는 단어가 100만개이고, 첫번채 층에서 1000차원의 벡터로 압축한다고 하면 10억개의 파라미터를 가짐. (원핫벡터일때)

2.2.3 부분단어 토큰화

- 문자 토큰화, 단어 토큰화의 절충안.
- BERT와 DistilBERT에서 사용되는 토크나이저인 WordPiece를 알아보자.
- AutoTokenizer로 해당 모델과 연관된 토크나이저 로드 (가장 잘 작동하는 토크나이저)

```
from transformers import AutoTokenizer

model_ckpt = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
```

- 두번째 호출하면 ~/.cache/huggigface에 저장된 경로에서 토크나이저를 로드한다.
  → 이 경로는 허깅페이스에서 받은 대부분의 것들을 저장하는 경로이다.
- 아래와같이 지정해서 특정 클래스를 로드할 수 있다.

```
from transformers import DistilBertTokenizer

dist_tokenizer = DistilBertTokenizer.from_pretrained(model
```

- 토큰화하면 아래와 같이 결과 나옴

```
encoded_text = tokenizer(text)
print(encoded_text)
```

```
{'input_ids': [101, 19204, 6026, 3793, 2003, 1037, 4563, 4708, 1997, 17953,
2361, 1012, 102], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

```
tokens = tokenizer.convert_ids_to_tokens(encoded_text.inpu
print(tokens)
```

```
['[CLS]', 'token', '##izing', 'text', 'is', 'a', 'core', 'task', 'of', 'nl',
'##p', '.', '[SEP]']
```

  - 여기에서 WordPiece 토크나이저의 결과를 살펴보자.
  - CLS는 문장의 시작, SEP은 문장의 끝을 알려주는 스페셜 토큰이며 nlp, tokenizing은 자주 나오는 단어가 아니므로 두 단어로 분리되었다. ##는 앞에 있는 문자열이 공백이 아님을 뜻한다.
- tokenizer 의 속성
  - tokenizer.vocab_size : 30522
  - tokenizer.model_max_length : 512
  - tokenizer.model_input_names : [’input_ids’, ‘attention_mask’] → model이 forward pass에서 기대하는 필드의 이름. attention_mask는 다음 절에서 설명.

2.2.4 전체 데이터셋 토큰화하기

- 전체 말뭉치를 토큰화하기 위해 DatasetDict 객체의 map() 메서드 사용한다.

- 먼저 샘플을 토큰화할 함수 정의.

```
def tokenize(batch):
    return tokenizer(batch["text"], padding=True, truncati
```

  - padding = Ture : 배치에 있는 가장 긴 샘플 크기에 맞춰서 0으로 패딩.
  - truncation = True : 모델의 최대 문맥 크기에 맞춰 샘플을 잘라냄.

- 샘플 2개 토큰화

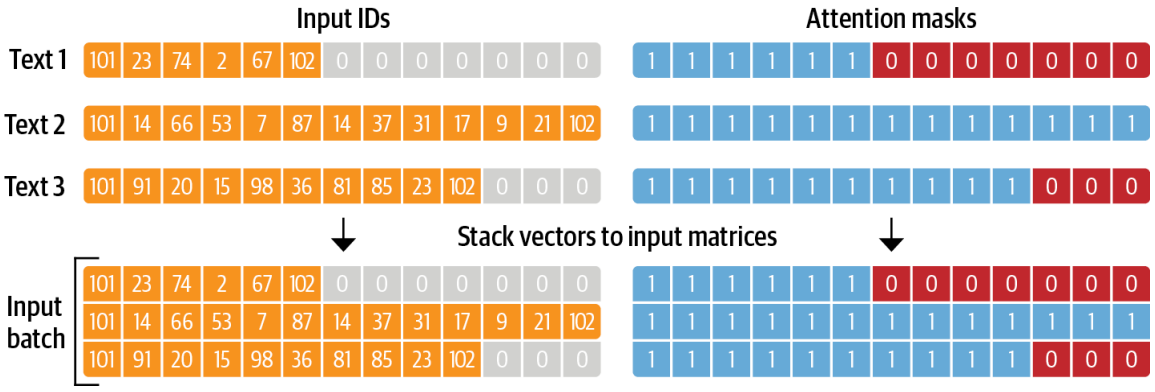
```
emotions["train"][:2]
```

```
{'text': ['i didnt feel humiliated',
  'i can go from feeling so hopeless to so damned hopeful just from being around someone who cares and is awake'],
 'label': [0, 0]}
```

```
print(tokenize(emotions["train"][:2]))
```

```
{'input_ids': [[101, 1045, 2134, 2102, 2514, 26608, 102, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0], [101, 1045, 2064, 2175, 2013, 3110, 2061, 20625, 2000,
2061, 9636, 17772, 2074, 2013, 2108, 2105, 2619, 2040, 14977, 1998, 2003, 8300,
102]], 'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1]]}
```

  - 첫 번째 문장이 두 번째 문장보다 작으므로 그 만큼 0으로 패딩. CLS와 SEP은 동일하게 붙어있는 것을 볼 수 있음. (CLS가 101, SEP이 102, PAD가 0임)
  - 토크나이저는 input_ids와 attention_mask 를 반환함, **attention_mask를 사용하여 패딩으로 처리된 부분을 무시하도록 만듬**. (모델에서 어텐션 계산할때 무시한다고 생각하면 된다.)



- map 함수

```
emotions_encoded = emotions.map(tokenize, batched=True, ba
```

  - map은 기본적으로 말뭉치에 있는 모든 샘플에 개별적으로 작용, **batch True를 적용해서 트윗을 배치로 인코딩.** batch_size=None으로 주어 전체 데이터셋이 하나의 배치로 tokenzie()함수에 적용됨. 이렇게하면 입력 텐서와 어텐션 마스크는 같은 크기로 생성되고, 데이터셋에 input_ids와 attention_mask를 추가한다. (원래는 text와 label정보 밖에 없었음)

```
{'text': ['i didnt feel humiliated'],
 'label': [0],
 'input_ids': [[101, 1045, 2134, 2102, 2514, 26608, 102]],
 'attention_mask': [[1, 1, 1, 1, 1, 1, 1]]}
```

  - 이렇게 하면 모든 단어들이 모든 샘플중 가장 긴 샘플만큼 패딩처리 되고, 그에 맞게 attention_mask도 가지게 된다.

- 이번 절 내용을 정리하면 허깅페이스 허브에 있는 데이터셋이나, 자신 만의 데이터셋을 load_dataset()을 이용하여 불러오고, Dataset의 포맷을 변경하여 분포를 살펴봤다.

  그리고
  모델에 넣어주기 위해서 토큰화를 해야하므로, 각 모델에 맞는 **토크나이저를 AutoTokenizer**를 이용해서 받아온다. 그 토크나이저에 데이터를 넣어 **각 샘플마다 input_ids와 attention_mask**를 구한다.
- 이제 만들어진 **토큰화된 데이터셋**을 가지고 2.3절에서 모델을 훈련시킬 수 있다.

2.3 텍스트 분류 모델 훈련하기