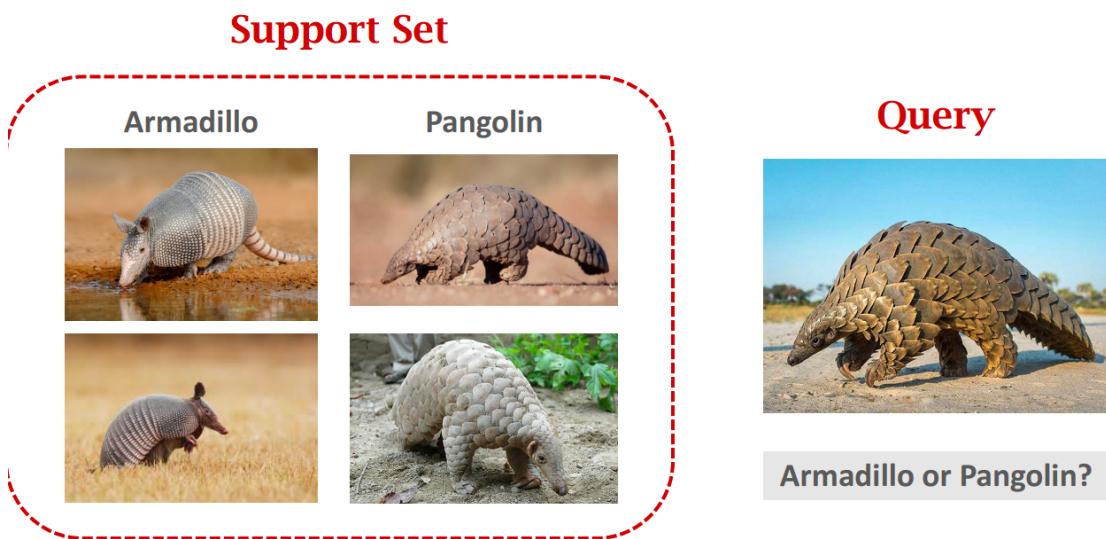


# 9장 : 레이블 부족 문제 다루기

- 제로샷 학습(zero-shot learning) : 모델이 특정 클래스의 훈련 예제 없이 새로운 개체나 클래스를 인식하도록 훈련되는 기계 학습 기술, 이 모델은 속성 또는 의미 관계와 같은 다른 클래스에 대한 정보를 사용하여 훈련되며, 이 정보를 사용하여 새 클래스의 특성을 추론한다.
- 퓨샷 학습(few-shot learning) : 말 그대로 'Few'한 데이터도 잘 분류할 수 있다는 것, 하지만 Few한 데이터로 학습을 한다는 것이 아님, 매우 제한된 훈련 예제를 사용하여 새로운 객체나 클래스를 인식하도록 훈련되는 기계 학습 기술이다.

## ▼ 퓨샷 학습

Meta Learning



왼쪽의 아르마딜로 사진 두 장과 천산갑 사진 두장을 주고 오른쪽의 Query가 아르마딜로인지 천산갑인지 분류하라고 할 때 모두 천산갑이라고 분류를 잘 할 것이다.

전통적인 딥러닝 모델은 이와 같이 각 클래스별 사진 두장을 가지고 Query 이미지를 맞출 수 있는가?

→ 아니다, 많은 데이터가 필요했을 것이다.

그렇다면 어떻게 이렇게 쉽게 답할 수 있는가?

→ 구분을 하는 방법을 배웠기 때문 (이러한 학습 시도를 Meta Learning = Learn to Learn), 하지만 이러한 방법을 배우는 과정에서 수많은 학습이 있었을 것이다. (사자와 호랑이가 다른 것, 토끼와 고양이가 다른 것.. 등등)

Few-shot learning은 이러한 점에 착안한 Meta-learning의 한 종류이다. '배우는 법을 배우기'는 결국 많은 데이터가 필요하고 아래와 같은 데이터들로 학습이 될 것이다. 다만 다른 점은 구분하려는 문제 (아르마딜로인지 천산갑인지)는 Training set에 없어도 된다.

훈련 데이터를 통하여 유사점과 차이점을 배우고 새롭게 적은 양의 support data만 받더라도 예측을 수행할 수 있는 것.(support set은 적은 양의 새롭게 예측해야 하는 데이터)

## Few-Shot Learning

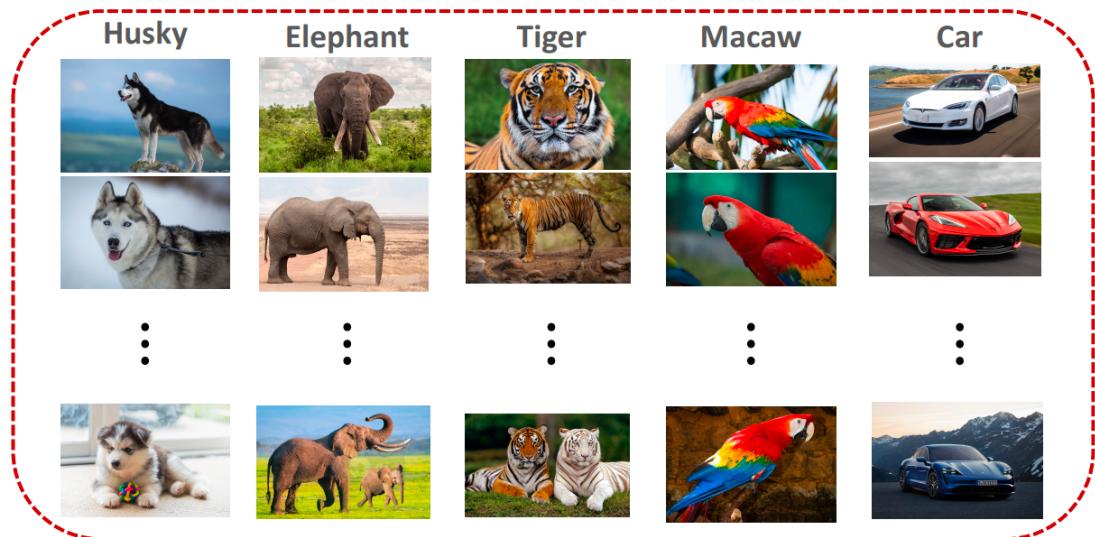
Query:



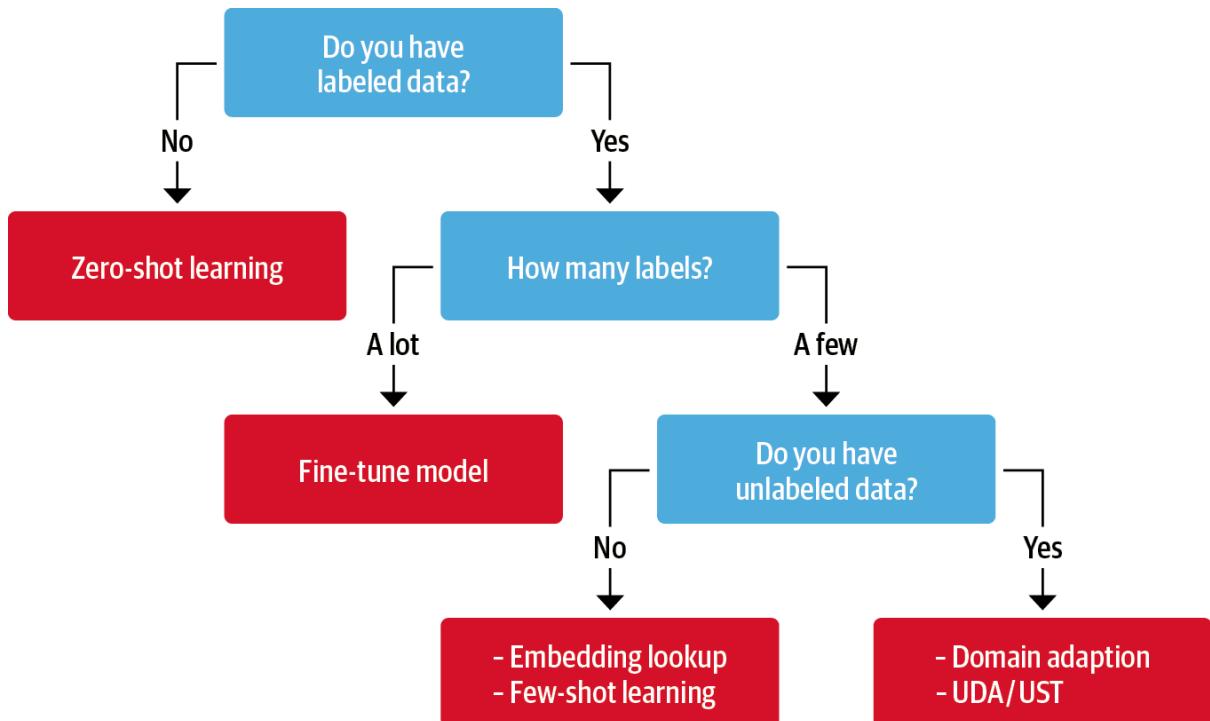
Support Set:



Training Set



우리는 Few shot learning을 위해 Training set, Support set, Query image가 필요하다는 점을 이해할 수 있다. 한마디로 정리하면, Training set을 통해 “구분하는 법을 배우”고, Query image가 들어왔을 때 이 Query image가 Support set 중 어떤 것과 같은 종류인지를 맞추는 일을 하는 것이다. 즉, Query image가 어떤 클래스에 “속하느냐”的 문제를 푸는 것이 아니라 어떤 클래스와 “같은 클래스냐”的 문제를 푼다고 생각하면 이해하기 쉽다.



### 1. 레이블링 된 데이터가 있나요?

레이블링 된 데이터가 전혀 없다면 종종 강력한 기준점을 세워주는 제로샷 학습을 추천.

### 2. 레이블링된 데이터가 얼마나 있나요?

레이블링 된 데이터가 있다면, 다음으로는 데이터의 양이 중요하다. 가용된 훈련 데이터가 많다면 표준적인 미세 튜닝 방식을 추천.

### 3. 레이블링되지 않은 데이터가 있나요?

레이블링된 샘플이 조금 있고, 레이블링되지 않은 데이터가 많이 있다면, 분류기를 훈련하기 전에 레이블링되지 않은 데이터를 사용해 해당 도메인에서 언어 모델을 미세 튜닝하는 방법, 비지도 데이터 증식(unsupervised data augmentation : UDA), 인지 자기 훈련(uncertainty-aware self-training : UST)같이 복잡한 방법을 사용 가능하다.

레이블링되지 않은 데이터가 전혀 없다면, 데이터를 레이블링하여 늘릴 수 없다. 이 경우 퓨샷 학습, 사전 훈련된 언어 모델의 임베딩을 사용해 최근접 이웃 검색(nearest neighbor search)으로 룩업(lookup)을 수행할 수 있다.

▼ 깃허브 이슈 태거 만들기

The screenshot shows a GitHub issue page for the repository `huggingface / transformers`. The issue is titled `[2D Parallelism] Tracking feasibility #9931`. The main content of the issue discusses the feasibility of 2D parallelism for large models, mentioning existing parallelism techniques like ZeRO-DP and PP, and the challenges of implementing 3D parallelism. It also tracks the status of different implementations (DeepSpeed, FairScale, PyTorch) and their inter-operability.

**Description**

**Background**

ZeRO-DP (Zero Data Parallel) and PP (Pipeline Parallelism) provide each a great memory saving over multiple GPUs. Each 1D allows for a much more efficient utilization of the gpu memory, but it's still not enough for very big models - sometimes not even feasible with any of the existing hardware. e.g. a model that's 45GB-big with just model params (t5-11b) can't fit even on a 40GB GPU.

The next stage in Model Parallelism that can enable loading bigger models onto smaller hardware is 2D Parallelism. That's combining Pipeline Parallelism (PP) with ZeRO-DP.

3D Parallelism is possible too and it requires adding a horizontal MP (ala [Megatron-LM](#), but we don't quite have any way to implement that yet. Need to study Megatron-LM first. So starting with a relatively low hanging fruit of 2D.

**Tracking**

We have 3 implementations that provide the required components to build 2D Parallelism:

1. DeepSpeed (DS)
2. FairScale (FS)
3. PyTorch (native) (PT)

and the purpose of this issue is to track the feasibility/status/inter-operability in each one of them. And also which parts have been back-ported to PyTorch core.

Plus it tracks the status of where transformers models are at with regards to the above 3 implementations.

The 2 main questions are:

1. native 2D: how do we integrate a native PP with native ZeRO-DP (sharded) (e.g. can fairscale PP work with fairscale ZeRO-DP)
2. inter-operability 2D: is there a chance one implementation of PP/ZeRO-DP could work with one or both others ZeRO-DP/PP (e.g. can fairscale PP work with DeepSpeed ZeRO-DP).

**Tags**

Labels: DeepSpeed, Model Parallel, Pipeline Parallel, WIP

Projects: None yet

Milestone: No milestone

Linked pull requests: Successfully merging a pull request may close this issue.

Notifications: Customize, Subscribe

You're not receiving notifications from this thread.

2 participants

다음과 같이 깃허브 이슈는 제목, 설명, 이슈의 특징을 나타내는 일련의 태그 또는 레이블을 담고 있어 자연스럽게 지도 학습으로 구성 할 수 있다. 이슈의 제목과 설명이 주어지면 한 개 이상의 레이블을 예측하는 식.

이슈에 여러 개의 레이블을 할당할 수 있으므로 **다중 레이블 분류** 문제이다.

<데이터 다운로드>

<데이터 준비하기>

	url	https://api.github.com/repos/huggingface/trans...
	repository_url	https://api.github.com/repos/huggingface/trans...
	labels_url	https://api.github.com/repos/huggingface/trans...
	comments_url	https://api.github.com/repos/huggingface/trans...
	events_url	https://api.github.com/repos/huggingface/trans...
	html_url	https://github.com/huggingface/transformers/is...
	id	849568459
	node_id	MDU6SXNzdWU4NDk1Njg0NTk=
	number	11046
	title	Potential incorrect application of layer norm ...
	user	{'login': 'sougata-ub', 'id': 59206549, 'node_...}
	labels	[]
	state	open
	locked	False
	assignee	None
	assignees	[]
	milestone	NaN
	comments	0
	created_at	2021-04-03 03:37:32
	updated_at	2021-04-03 03:37:32
	closed_at	NaT
	author_association	NONE
	active_lock_reason	None
	body	In BlenderbotSmallDecoder, layer norm is appl...
	performed_via_github_app	NaN
	pull_request	None
	split	unlabeled
	text	Potential incorrect application of layer norm ...

데이터의 형태이다.

```
df_issues.loc[2]["labels"]

->
[{'id': 2659267025,
 'node_id': 'MDU6TGFiZWwyNjU5MjY3MDI1',
 'url': 'https://api.github.com/repos/huggingface/transformers/labels/DeepSpeed',
 'name': 'DeepSpeed',
 'color': '4D34F7',
 'default': False,
 'description': ''}]
```

labels 데이터를 불러들여옴, 목적상 각 레이블의 name만 필요함.

```
df_issues["labels"] = (df_issues["labels"]
                        .apply(lambda x: [meta["name"] for meta in x]))
df_issues[["labels"]].head()
```

	labels
0	[]
1	[]
2	[DeepSpeed]
3	[]
4	[]

```
df_issues["labels"].apply(lambda x : len(x)).value_counts().to_frame().T
```

	0	1	2	3	4	5
labels	6440	3057	305	100	25	3

데이터 레이블의 개수이다.

```
df_counts = df_issues["labels"].explode().value_counts()
print(f"레이블 개수: {len(df_counts)}")
# 상위 8개 레이블을 출력합니다.
df_counts.to_frame().head(8).T
```

레이블 개수: 65

	wontfix	model card	Core: Tokenization	New model	Core: Modeling	Help wanted	Good Issue
labels	2284	649	106	98	64	52	50

데이터 레이블을 열로 변환하여 레이블의 등장 횟수를 카운팅 하였을 때, 상위 8개의 레이블이다.

→ wontfix와 model card가 가장 자주 등장하며 Good First Issue나 Help Wanted 같은 일부 레이블은 이슈 설명만으로는 예측하기가 매우 어렵다. model card 같은 레이블은 허깅페이스 허브에 모델 카드가 추가된 때를 감지하는 간단한 규칙으로 분류 할 수 있다.

```
label_map = {"Core: Tokenization": "tokenization",
             "New model": "new model",
             "Core: Modeling": "model training",
             "Usage": "usage",
             "Core: Pipeline": "pipeline",
             "TensorFlow": "tensorflow or tf",
             "PyTorch": "pytorch",
             "Examples": "examples",
             "Documentation": "documentation"}

def filter_labels(x):
    return [label_map[label] for label in x if label in label_map]

df_issues["labels"] = df_issues["labels"].apply(filter_labels)
all_labels = list(label_map.values())

df_counts = df_issues["labels"].explode().value_counts()
df_counts.to_frame().T
```

	tokenization	new model	model training	usage	pipeline	tensorflow or tf	pytorch
labels	106	98	64	46	42	41	37

앞으로 다를 레이블만 남긴 후 레이블을 읽기 쉽게 변환하고 분포를 확인한다.

```
df_issues["split"] = "unlabeled"
mask = df_issues["labels"].apply(lambda x: len(x) > 0)
df_issues.loc[mask, "split"] = "labeled"
df_issues["split"].value_counts().to_frame()
```

split	
unlabeled	9489
labeled	441

레이블의 유무를 확인한다.

```
for column in ["title", "body", "labels"]:
    print(f"{column}: {df_issues[column].iloc[26][:500]}\n")

->
title: Add new CANINE model

body: # 🌟 New model addition

## Model description

Google recently proposed a new **C**haracter **A**rchitecture with **N**o tokenization **I**n **N**eural **E**ncoders architecture
> Pipelined NLP systems have largely been superseded by end-to-end neural modeling, yet nearly all commonly-used models still requ
```

```
labels: ['new model']
```

해당 샘플에서는 새로운 모델 구조를 제안하기 때문에 new model 태그가 적절하다.

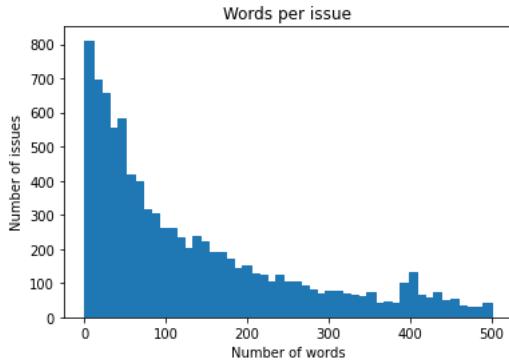
```
len_before = len(df_issues)
df_issues = df_issues.drop_duplicates(subset="text")
print(f"삭제된 중복 이슈: {(len_before - len(df_issues)) / len_before:.2%}")

-> 삭제된 중복 이슈: 1.88%
```

이후 중복된 데이터를 삭제한다.

```
import numpy as np
import matplotlib.pyplot as plt

(df_issues["text"].str.split().apply(len)
 .hist(bins=np.linspace(0, 500, 50), grid=False, edgecolor="c0"))
plt.title("Words per issue")
plt.xlabel("Number of words")
plt.ylabel("Number of issues")
plt.show()
```



텍스트에 있는 단어를 확인해보면 롱테일 특징이 있다. 대부분의 텍스트는 매우 짧지만 500단어가 넘는 이슈도 있다. 하지만 대부분의 트랜스포머 모델의 문맥 크기가 512개 토큰이나 그 이상이기 때문에 일부 긴 이슈를 잘라내는 것이 전체 성능에 영향을 미칠 것 같아 보이지 않는다.

### <훈련 세트 만들기>

- MultiLabelBinarizer 클래스를 사용한다 → 레이블 이름의 리스트를 받고 레이블에 해당하는 위치는 1, 나머지는 0인 벡터를 생성한다.

```
from sklearn.preprocessing import MultiLabelBinarizer

mlb = MultiLabelBinarizer()
mlb.fit([all_labels])
mlb.transform([["tokenization", "new model"], ["pytorch"]])

->array([[0, 0, 0, 1, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0]])
```

첫 번째 행은 두 개의 1을 포함하는데, 각각 new model과 tokenization 레이블에 해당한다.

```
from sklearn.model_selection import train_test_split

df_clean = df_issues[["text", "labels", "split"]].reset_index(drop=True).copy()
df_unsup = df_clean.loc[df_clean["split"] == "unlabeled", ["text", "labels"]]
df_sup = df_clean.loc[df_clean["split"] == "labeled", ["text", "labels"]]

np.random.seed(0)
df_train, df_tmp = balanced_split(df_sup, test_size=0.5)
df_valid, df_test = balanced_split(df_tmp, test_size=0.5)
```

해당 코드를 통하여 text, labels, split 부분만 들고 와 labels의 여부에 따라 지도학습과 비지도 학습으로 나눈다. 이 후, Trainer와 통합하기 위해 모든 분할을 담은 DatasetDict 객체를 만든다.

```
from datasets import Dataset, DatasetDict

ds = DatasetDict({
    "train": Dataset.from_pandas(df_train.reset_index(drop=True)),
    "valid": Dataset.from_pandas(df_valid.reset_index(drop=True)),
    "test": Dataset.from_pandas(df_test.reset_index(drop=True)),
    "unsup": Dataset.from_pandas(df_unsup.reset_index(drop=True))}

->
DatasetDict({
    train: Dataset({
        features: ['label_ids', 'labels', 'text'],
        num_rows: 223
    })
    valid: Dataset({
        features: ['label_ids', 'labels', 'text'],
        num_rows: 106
    })
    test: Dataset({
        features: ['label_ids', 'labels', 'text'],
        num_rows: 111
    })
    unsup: Dataset({
        features: ['label_ids', 'labels', 'text'],
        num_rows: 9303
    })
})
```

## <훈련 슬라이스 만들기>

```
np.random.seed(0)
all_indices = np.expand_dims(list(range(len(ds["train"]))), axis=1)
indices_pool = all_indices
labels = mlb.transform(ds["train"]["labels"])
train_samples = [8, 16, 32, 64, 128]
train_slices, last_k = [], 0

print(labels.shape)
for i, k in enumerate(train_samples):
    # 다음 슬라이스 크기를 채우는데 필요한 샘플을 분할합니다
    indices_pool, labels, new_slice, _ = iterative_train_test_split(indices_pool, labels, (k-last_k)/len(labels))
    last_k = k
    if i==0: train_slices.append(new_slice)
    else: train_slices.append(np.concatenate((train_slices[-1], new_slice)))

# 마지막 슬라이스를 포함하면 코랩의 경우 메모리 부족이 발생합니다.
# 대신 코랩 프로(https://colab.research.google.com/signup)를 사용하세요.
# 코랩을 사용하려면 다음 라인을 주석 처리하세요.
train_slices.append(all_indices), train_samples.append(len(ds["train"]))

train_slices = [np.squeeze(train_slice) for train_slice in train_slices]
```

220개 정도의 샘플로 구성되었으며 전이 학습으로도 어려운 문제이다. 해당 장에서는 적은 양의 레이블링된 데이터에서 얼마나 좋은 성능을 내는지 확인하기 위해 샘플 개수가 더 적은 훈련 데이터의 슬라이스도 만든다.

```

슬라이스 : [ 0  2  9 11 18 19 45 52 54 92] 크기 : 10

슬라이스 : [ 0  2  9 11 18 19 45 52 54 92  1  6 10 12 41 56 63 88
              110] 크기 : 19

슬라이스 : [ 0  2  9 11 18 19 45 52 54 92  1  6 10 12 41 56 63 88
              110  3  4 14 16 22 28 40 55 58 60 65 89 97 118 122 132 133] 크기 : 36

슬라이스 : [ 0  2  9 11 18 19 45 52 54 92  1  6 10 12 41 56 63 88
              110  3  4 14 16 22 28 40 55 58 60 65 89 97 118 122 132 133
              5 13 15 20 23 25 27 30 31 33 35 36 44 57 64 69 73 74
              78 80 90 91 99 105 123 136 141 152 180 184 190 197] 크기 : 68

슬라이스 : [ 0  2  9 11 18 19 45 52 54 92  1  6 10 12 41 56 63 88
              110  3  4 14 16 22 28 40 55 58 60 65 89 97 118 122 132 133
              5 13 15 20 23 25 27 30 31 33 35 36 44 57 64 69 73 74
              78 80 90 91 99 105 123 136 141 152 180 184 190 197 8 17 26 32
              37 38 42 46 48 50 61 62 67 70 72 75 77 82 83 85 94 95
              96 100 101 106 111 112 120 121 126 127 129 130 135 137 139 140 143 145
              147 149 153 154 157 163 165 167 168 170 174 175 176 178 183 185 194 199
              200 203 205 210 214 216 218 220] 크기 : 134

슬라이스 : [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
              18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
              36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53
              ...
              180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197
              198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215

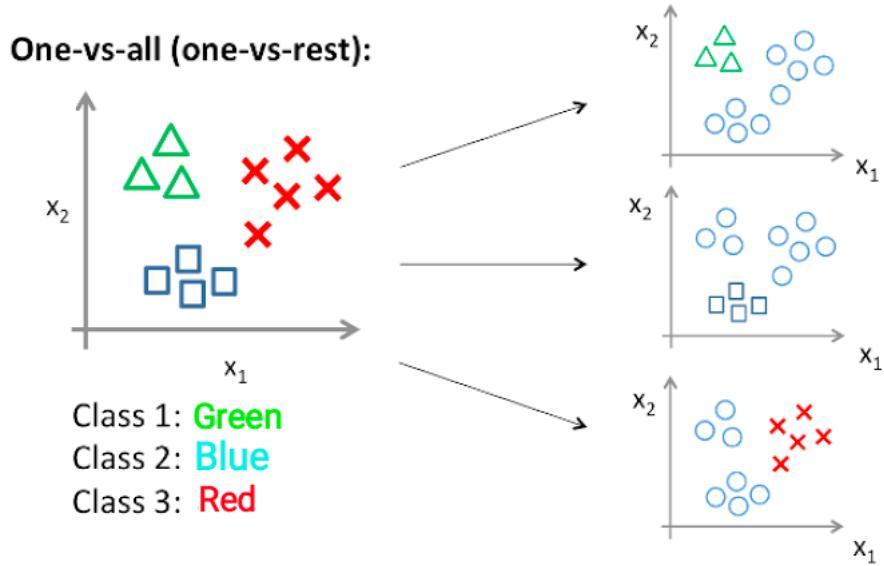
```

원하는 크기에 근사하게 샘플을 나누었다.

#### ▼ 나이브 베이즈 모델 만들기

강력한 기준 모델을 구현하는 것은 늘 유익하다.

1. 매우 간단한 모델을 기반으로 하는 기준 모델이 실제로 문제를 잘 해결하기도 한다. 이 경우 트랜스포머와 같은 해머를 집어들 이유가 없다. 트랜스포머 같은 모델은 배포, 유지가 힘들다.
  2. 복잡한 모델을 탐색할 때 기준 모델이 간단한 검증 역할을 한다. BERT-large 모델을 훈련하고 검증 세트에서 80%의 정확도를 얻었는데 로지스틱 회귀 같은 간단한 분류기가 95% 정확도를 달성했다면 모델에 버그가 있는지 의심하고 디버깅을 하게 된다.
- 기준 모델을 만든다. 이 때의 좋은 기준점은 **나이브 베이즈 분류기**이다.
- 나이브 베이즈 분류기란?
    - 간단하고 훈련 속도가 빠르며 입력의 변동에 매우 안정적이다. 기본적으로 다중 레이블 분류를 지원하지 않지만 사이킷런 라이브러리를 활용하여 해당 문제를 OvR(One VS rest)분류 작업으로 바꿀 것이다. 이 방식은 L개의 레이블을 위해 L개의 이진 분류기를 훈련한다.
  - OvR(One vs Rest)?
    - 다중 클래스 문제를 여러 개의 이진 분류 문제로 바꾸는데 이 때 하나의 이진 분류기는 한 클래스를 1, 나머지 클래스를 0으로 취급하여 학습한다.



예를 들어 A, B, C의 세 가지 클래스가 있는 데이터 세트가 있다고 가정해 보겠습니다. One-vs-Rest 기술을 적용하려면 세 개의 개별 분류기를 훈련합니다. 하나는 클래스 A와 다른 클래스를 구별하고 다른 하나는 클래스 B와 다른 클래스를 구별하고, 클래스 C와 다른 클래스를 구별하기 위한 세 번째.

분류를 위해 새 인스턴스가 제공되면 훈련된 각 분류자를 통해 전달하고 분류자가 가장 높은 예측 확률을 반환하는 클래스에 할당 한다.

```
print(ds)

->
DatasetDict({
    train: Dataset({
        features: ['text', 'labels', 'label_ids'],
        num_rows: 223
    })
    valid: Dataset({
        features: ['text', 'labels', 'label_ids'],
        num_rows: 106
    })
    test: Dataset({
        features: ['text', 'labels', 'label_ids'],
        num_rows: 111
    })
    unsup: Dataset({
        features: ['text', 'labels', 'label_ids'],
        num_rows: 9303
    })
})
```

다음과 같이 label\_ids를 매핑한다.

```
from collections import defaultdict

macro_scores, micro_scores = defaultdict(list), defaultdict(list)

print(macro_scores)
print(micro_scores)

->
defaultdict(<class 'list'>, {'Naive Bayes': [0.23288166214995487, 0.21006897585844955, 0.24086240556828795, 0.257305008182201,
defaultdict(<class 'list'>, {'Naive Bayes': [0.3604651162790698, 0.30208333333333337, 0.41081081081083, 0.4435483870967742
```

마이크로 점수 - 자주 등장하는 레이블에 대한 성능을 추적

매크로 F1 점수 - 빈도를 무시하고 모든 레이블에 대한 성능을 평균

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report
from skmultilearn.problem_transform import BinaryRelevance
from sklearn.feature_extraction.text import CountVectorizer

for train_slice in train_slices:
    # 훈련 슬라이스와 테스트 데이터를 준비합니다
    ds_train_sample = ds["train"].select(train_slice)
    y_train = np.array(ds_train_sample["label_ids"])
    y_test = np.array(ds["test"]["label_ids"])
    # 간단한 CountVectorizer를 사용해 텍스트를 토큰 카운트로 인코딩합니다
    count_vect = CountVectorizer()
    X_train_counts = count_vect.fit_transform(ds_train_sample["text"])
    X_test_counts = count_vect.transform(ds["test"]["text"])
    # 모델을 만들고 훈련합니다!
    classifier = BinaryRelevance(classifier=MultinomialNB())
    classifier.fit(X_train_counts, y_train)
    # 예측을 생성하고 평가합니다
    y_pred_test = classifier.predict(X_test_counts)
    clf_report = classification_report(
        y_test, y_pred_test, target_names=mlb.classes_, zero_division=0,
        output_dict=True)
    # 평가 결과를 저장합니다
    macro_scores["Naive Bayes"].append(clf_report["macro avg"]["f1-score"])
    micro_scores["Naive Bayes"].append(clf_report["micro avg"]["f1-score"])
```

CountVectorizer : 문서 집합에서 단어 토큰을 생성하고 각 단어의 수를 세어 BOW 인코딩 벡터를 만듭니다

BoW(Bag of Words) : 단어들의 순서는 전혀 고려하지 않고, 단어들의 출현 빈도에만 집중하는 텍스트 데이터의 수치화 표현 방법.

```
from sklearn.feature_extraction.text import CountVectorizer

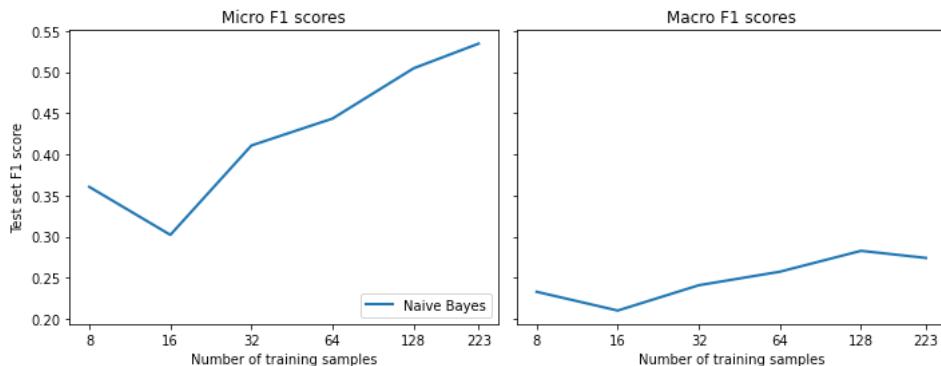
corpus = ['you know I want your love. because I love you.']
vector = CountVectorizer()

# 코퍼스로부터 각 단어의 빈도수를 기록
print('bag of words vector :', vector.fit_transform(corpus).toarray())

# 각 단어의 인덱스가 어떻게 부여되었는지를 출력
print('vocabulary :', vector.vocabulary_)

->
bag of words vector : [[1 2 1 2 1]]
vocabulary : {'you': 4, 'know': 1, 'want': 3, 'your': 5, 'love': 2, 'because': 0}
```

CountVectorizer는 단지 띄어쓰기만을 기준으로 하기 때문에 한국어에 적용하면 조사 등의 이유로 BoW가 제대로 만들어지지 않음.



훈련 샘플 개수의 증가에 따라 마이크로와 매크로 F1-점수가 모두 향상되었다.

#### ▼ 레이블링된 데이터가 없는 경우

- 첫 번째로 고려할 방법은 레이블링된 데이터가 전혀 없는 상황에 적합한 제로샷 분류(zero-shot classification)
  - 제로샷 분류의 목표는 작업별 말뭉치에서 추가로 미세 튜닝하지 않고 사전 훈련된 모델을 사용하는 것.

- 이 방식을 이해하기 위해서는 BERT 같은 언어의 모델이 대규모 데이터에서 텍스트에 있는 마스킹된 토큰을 예측하도록 사전 훈련했다는 사실을 알고 있어야 한다. 누락된 토큰을 잘 예측하려면 모델은 문맥에 있는 주제를 인식하여야 한다.

```
from transformers import pipeline
pipe = pipeline("fill-mask", model="bert-base-uncased")
```

fill-mask 파이프라인 : 마스크드 언어 모델을 사용하여 마스킹된 토큰 내용을 예측한다.

```
movie_desc = "The main characters of the movie madagascar \
are a lion, a zebra, a giraffe, and a hippo. "
prompt = "The movie is about [MASK]."

output = pipe(movie_desc + prompt)
for element in output:
    print(f"토큰 {element['token_str']}: \t{element['score']:.3f}%")

->
토큰 animals: 0.103%
토큰 lions: 0.066%
토큰 birds: 0.025%
토큰 love: 0.015%
토큰 hunting: 0.013%
```

해당 프롬프트는 모델이 분류하도록 안내하는 역할을 한다.

```
output = pipe(movie_desc + prompt, targets=["animals", "cars"])
for element in output:
    print(f"토큰 {element['token_str']}: \t{element['score']:.3f}%")

->
토큰 animals: 0.103%
토큰 cars: 0.001%
```

cars와 animals를 타깃으로 하여 파이프라인에 전달한 결과이다.

```
movie_desc = "In the movie transformers aliens \
can morph into a wide range of vehicles."

output = pipe(movie_desc + prompt, targets=["animals", "cars"])
for element in output:
    print(f"토큰 {element['token_str']}: \t{element['score']:.3f}%")

->
토큰 cars: 0.139%
토큰 animals: 0.006%
```

자동차에 관한 설명을 넣었을 때에 토큰 cars에 확률이 높은 것을 보아 잘 작동한다.

텍스트 분류에 좀 더 가까운 작업인 자연어 추론에서 미세 튜닝된 모델을 적용하면 결과가 더 좋은지 알아보도록 하자.

- 텍스트 함의(text entailment) : 모델이 두 개의 텍스트 구절이 서로 연결되는지 아니면 모순되는지 판단
  - 해당 데이터셋에 있는 샘플들은 전제(premise), 가설(hypothesis), Label(entailment, neutral, contradiction)로 구성되며 Label은 세 개 중 하나이다.
    - entailment : 전제 조건하에서 가설 텍스트가 참일 때 할당
    - contradiction : 전제 조건하에서 가설 텍스트가 거짓이거나 부적할 때
    - neutral : 두 경우 모두 해당되지 않을 때

표 9-1 MNLI 데이터셋에 있는 세 가지 클래스

전제	가설	레이블
His favourite color is blue.	He is into heavy metal music.	neutral
She finds the joke hilarious.	She thinks the joke is not funny at all.	contradiction
The house was recently built.	The house is new.	entailment

MNLI(Multi genre NLI) 데이터셋에서 훈련한 모델을 사용하여 어떤 레이블도 필요하지 않은 분류기를 만들 수 있다.

"This example is about {label}"

label에는 클래스 이름을 넣는다.

```
sample = ds["train"][0]
print(f"레이블: {sample['labels']}")
output = pipe(sample["text"], all_labels, multi_label=True)
print(output["sequence"][:400])
print("\n예측:")
for label, score in zip(output["labels"], output["scores"]):
    print(f"{label}, {score:.2f}")

->

레이블: ['new model']
Add new CANINE model

# 🌟 New model addition

## Model description

Google recently proposed a new **C**haracter **A**rchitecture with **N**o tokenization **I**n **N**eural **E**ncoders architecture
> Pipelined NLP systems have largely been superseded by end-to-end neural modeling, yet nearly all commonly-used models still req
예측:
new model, 0.98
tensorflow or tf, 0.37
examples, 0.34
usage, 0.30
pytorch, 0.25
documentation, 0.24
model training, 0.23
tokenization, 0.17
pipeline, 0.16
```

multi\_label = True로 지정하면 단일 레이블 분류를 위한 최댓값이 아니라 모든 점수가 반환된다.

텍스트의 주제가 'new model'이라고 강하게 확신하지만 다른 레이블에도 비교적 높은 점수를 할당한다.

```
def zero_shot_pipeline(example):
    output = pipe(example["text"], all_labels, multi_label=True)
    example["predicted_labels"] = output["labels"]
    example["scores"] = output["scores"]
    return example

ds_zero_shot = ds["valid"].map(zero_shot_pipeline)
```

```
print(ds_zero_shot[2])
1 ✓ 0.3s
{'text': "[example scripts] inconsistency around eval vs val\n  'val' == validation set (split)\n  'eval' == evaluation (mode)\n  those two are orthogonal to each other - one is a split, another is a model's run mode.\n  the trainer args and the scripts are inconsistent around when it's 'val' and when it's 'eval' in variable names and metrics.\n  examples: 'eval_dataset' but 'validation_file'\n  'eval_*' metrics key for validation dataset - why the prediction is 'test_*' metric keys?\n  'data_args.max_val_samples' vs 'eval_dataset' in the same line\n  the 3 parallels:\n    - 'train' is easy - it's both the process and the split\n    - 'prediction' is almost never used in the scripts it's all 'test'_var names and metrics and args\n    - 'eval' vs 'val' vs 'validation' is very inconsistent. when writing tests I'm never sure whether I'm looking up 'eval_*' or 'val_*' key. And one could run evaluation on the test dataset.\n  Perhaps asking a question would help and then a consistent answer is obvious!\n  Are metrics reporting stats on a split or a mode? \n  split - rename all metrics keys to be 'train|val|test'\n  mode - rename all metrics keys to be 'train|eval|predict'\n  Thank you.\n  @patrickvonplaten ", 'labels': ['examples'], 'label_ids': [0, 1, 0, 0, 0, 0, 0, 0], 'predicted_labels': ['examples', 'model training', 'tensorflow or tf', 'pytorch', 'usage', 'tokenization', 'documentation', 'pipeline', 'new model'], 'scores': [0.879645586013794, 0.6277863383293152, 0.4844728112220764, 0.47425973415374756, 0.36496955156326294, 0.214844632229805, 0.15436385105686188, 0.093257836997589], 'pred_label_ids': [0, 1, 0, 0, 0, 0, 0, 0]}
```

어떤 레이블을 할당할지 정한다.

- 임곗값을 정의하고 이 임곗값을 초과한 모든 레이블을 선택한다.
- 점수가 높은 순으로 상위 k개 레이블을 선택한다.

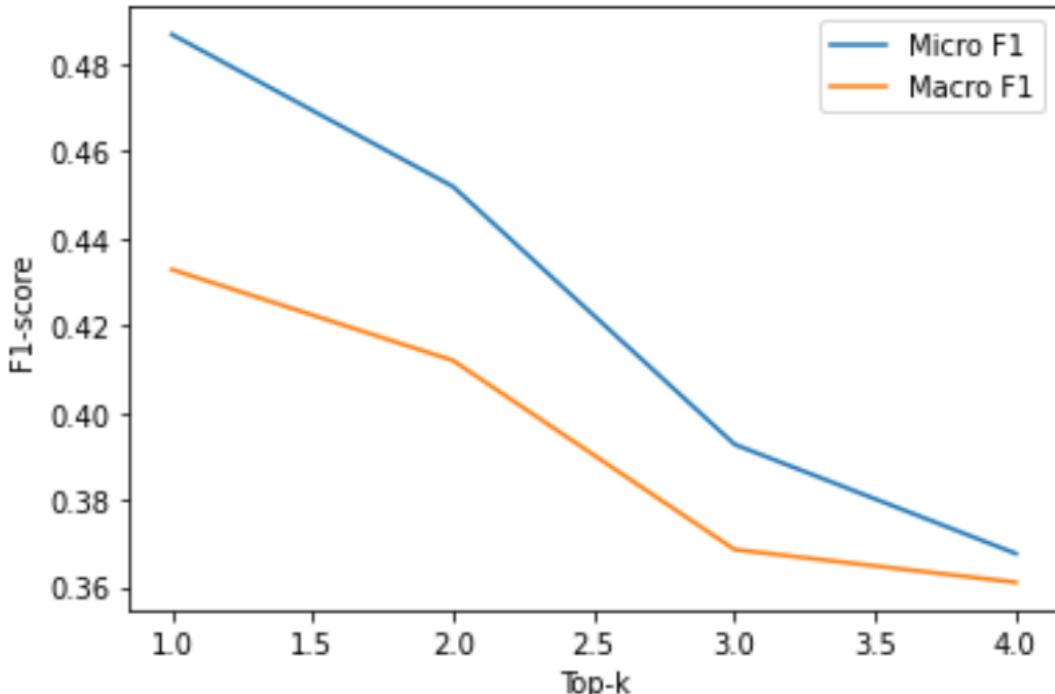
어떤 방식이 최선인지 두 가지 방법 모두 확인해보자.

```
def get_preds(example, threshold=None, topk=None):
    preds = []
    if threshold:
        for label, score in zip(example["predicted_labels"], example["scores"]):
            if score >= threshold:
                preds.append(label)
    elif topk:
        for i in range(topk):
            preds.append(example["predicted_labels"][i])
    else:
        raise ValueError("`threshold` 또는 `topk`로 지정해야 합니다.")
    return {"pred_label_ids": list(np.squeeze(mlb.transform([preds])))}
```

```
def get_clf_report(ds):
    y_true = np.array(ds["label_ids"])
    y_pred = np.array(ds["pred_label_ids"])
    return classification_report(
        y_true, y_pred, target_names=mlb.classes_, zero_division=0,
        output_dict=True)
```

사이킷런의 분류 리포트 결과를 반환한다.

Tok-K 방식은 특정 기준에 따라 데이터 집합에서 상위 k개 요소를 선택하는 데 사용되는 방법이다. 예를 들어 추천 시스템에서는 tok-k는 사용자가 가장 관심을 가질만한 상위 k개 항목을 선택하는 것을 의미할 수 있다.



그래프를 확인해보면 샘플당 점수가 가장 높은 레이블(top-1)을 선택하는 방식이 최상의 결과를 냈다.

→ 데이터셋의 샘플 대부분이 레이블이 하나 있기 때문이다.

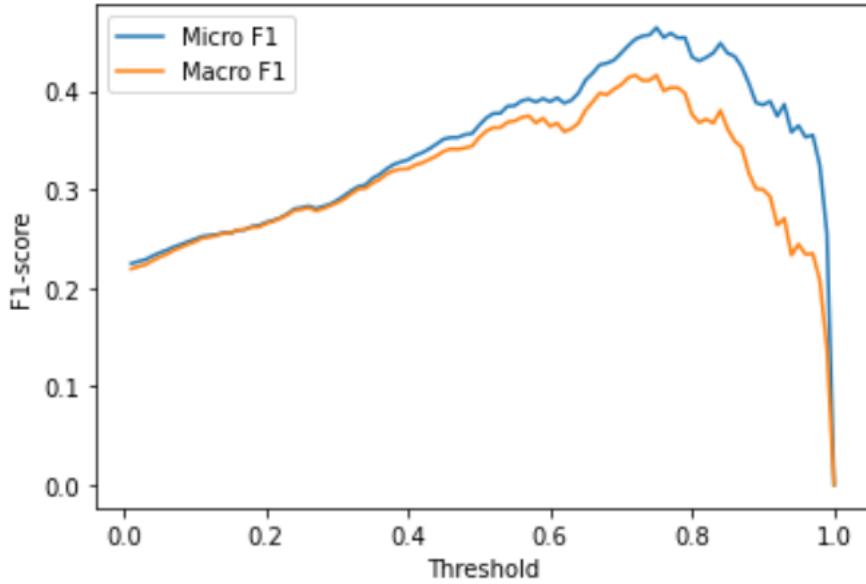
```
macros, micros = [], []
thresholds = np.linspace(0.01, 1, 100)
for threshold in thresholds:
    ds_zero_shot = ds_zero_shot.map(get_preds,
                                    fn_kwarg={"threshold": threshold})
    clf_report = get_clf_report(ds_zero_shot)
```

```

    micros.append(clf_report["micro avg"]["f1-score"])
    macros.append(clf_report["macro avg"]["f1-score"])

```

샘플당 하나 이상의 레이블을 예측할 수 있으므로, 이를 임계값 방식과 비교해보자.

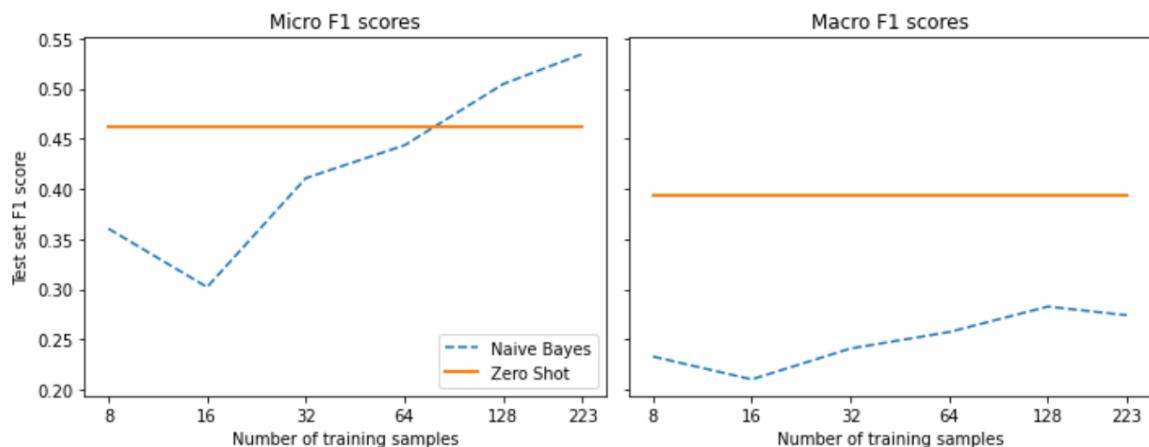


임계값(Threshold) : 애매한 값을 이분법으로 확실히 분류를 할 기준

ex. 로지스틱 회귀 모형에서 특정 이메일에 관해 스팸일 확률이 0.95가 반환 되었다면 이 이메일은 스팸일 가능성이 매우 높은 메일로 예측할 수 있다. 동일한 로지스틱 회귀 모형에서 예측 점수가 0.03인 이메일이라면 이 이메일은 스팸이 아닐 것이다. 하지만 스팸 확률이 0.6이라면? 여기서 기준을 잡기 위한 것이 임계값이다.

해당 방식의 결과는 tok-1 결과보다 나쁘지만, 그래프에 정밀도/재현율의 트레이드오프가 확실하게 나타난다.

임계값이 매우 낮게 설정하면 예측이 매우 많아져 정밀도가 낮아지고 임계값을 매우 높게 하면 예측하기 어려워져 재현율이 낮아진다. 한쪽이 커지면 한쪽이 작아지는 트레이드오프가 확실하게 나타난다.



테스트 세트에서 제로샷 분류와 나이브 베이즈를 비교한 그래프.

- 레이블링된 샘플이 50개보다 더 적다면 제로샷 파이프라인이 기존 모델의 성능을 쉽게 추월한다.
- 샘플이 50개 이상이라도 마이크로와 매크로 F1-score를 고려하면 제로샷 파이프라인이 더 낫다. 마이크로 F1-score 결과를 확인해보면, 기준 모델이 빈도가 높은 클래스에서 잘 작동하지만 제로샷 파이프라인은 학습할 샘플이 필요하지 않기 때문에 기준 모델보다 더 뛰어나다.

→ 성능향상 방법

- 파이프라의 작동 방식은 레이블 이름에 민감하다. 이름을 바꾸거나 여러 이름을 동시에 사용한 다음 후속 단계에서 결과를 집계하는 방식.
- 가설을 변경해도 성능은 높아진다. 기본적으로 가설은 hypothesis = "This is example is about {}"이지만 파이프라인에 다른 텍스트를 전달해도 된다.

▼ 레이블링된 데이터가 적은 경우

### <데이터 증식>

- 기존 샘플에서 새 훈련 샘플을 생성하는 데이터 증식(data augmentation) 기법을 사용.
- 역 번역(back traslation) : 원본 언어로 된 텍스트를 기계 번역을 사용해 하나 이상의 타깃 언어로 번역한다. 그 다음 이를 원본 언어로 다시 번역. 역 번역은 데이터가 많은 언어 또는 도메인 특화 언어가 매우 많지 않은 말뭉치에서 잘 동작한다.
  - 토큰 섞기(token perturbations) : 훈련 세트의 한 텍스트에서 동의어 교체, 단어 추가, 교환, 삭제 같은 간단한 변환을 임의로 선택하여 수행한다.

표 9-2 텍스트에 사용할 수 있는 여러 가지 데이터 증식 기법

데이터 증식	문장
없음	Even if you defeat me Megatron, others will rise to defeat your tyranny
동의어 교체	Even if you kill me Megatron, others will prove to defeat your tyranny
무작위 추가	Even if you defeat me Megatron, others humanity will rise to defeat your tyranny
무작위 교체	You even if defeat me Megatron, others will rise defeat to tyranny your
무작위 삭제	Even if you me Megatron, others to defeat tyranny
역 번역 (독일어)	Even if you defeat me, others will rise up to defeat your tyranny

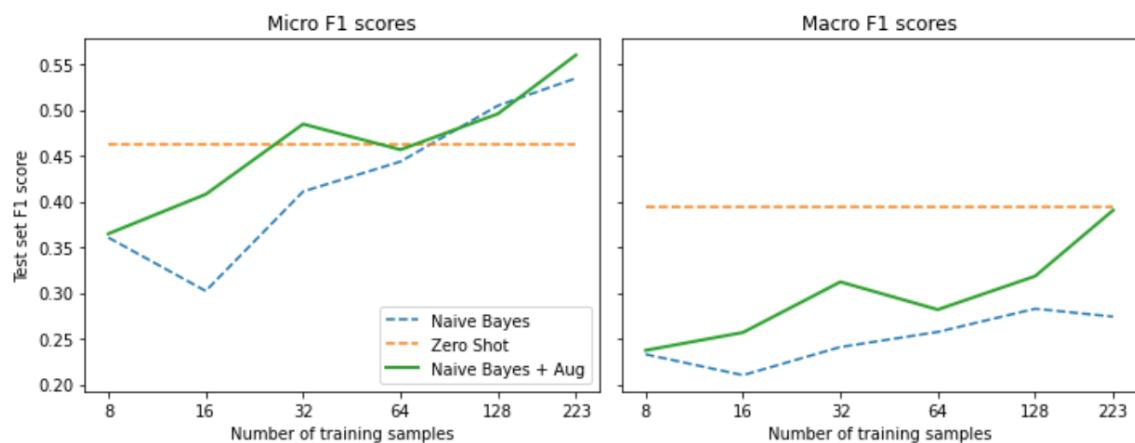
```
from transformers import set_seed
import nlpAug.augmenter.word as naw

set_seed(3)
aug = naw.ContextualWordEmbsAug(model_path="distilbert-base-uncased",
                                  device="cpu", action="substitute")

text = "Transformers are the most popular toys"
print(f"원본 텍스트: {text}")
print(f"증식된 텍스트: {aug.augment(text)}")

->
원본 텍스트: Transformers are the most popular toys
증식된 텍스트: ['transformers have the most available toys']
```

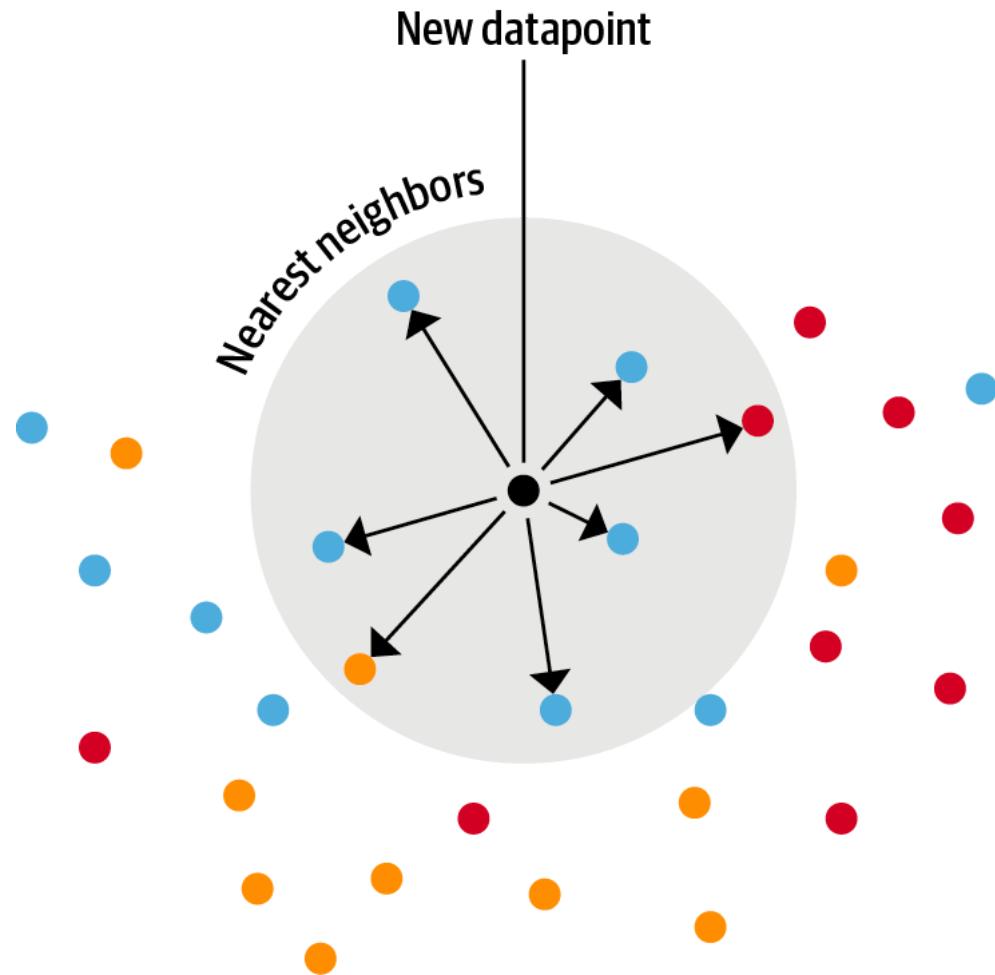
NLPAug의 ContextualWordEmbsAug 클래스로 DistilBERT의 문맥 단어 임베딩 사용.



약 5포인트 상승

### <임베딩을 룩업 테이블로 사용하기>

- GPT-3 같은 대규모 언어 모델은 제한된 데이터로 문제를 해결하는 데 뛰어나다. 유용한 텍스트 표현을 학습하였기 때문인데, 이런 표현은 감성, 토픽, 텍스트 구조 등의 많은 차원에 걸쳐 정보를 인코딩한다.
  - 해당 장에서는 Open AI 분류 엔드포인트를 본떠 텍스트 분류기를 만들 것인데 아래의 세 단계를 따름.
1. 언어 모델을 사용해 레이블링된 전체 텍스트를 임베딩한다.
  2. 저장된 임베딩에 최근접 이웃 검색을 수행한다.
  3. 최근접 이웃의 레이블을 수집해 예측을 만든다.



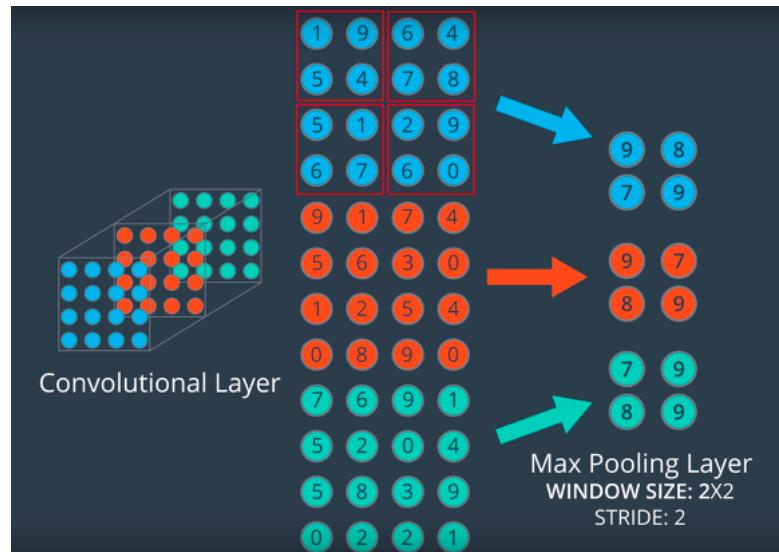
모델을 사용하여 레이블링된 데이터를 임베딩해서 레이블과 함께 저장하는 방법.

새로운 텍스트를 분류할 때 이를 임베딩한 후 최근접 이웃의 레이블을 기반으로 새로운 텍스트의 레이블을 부여한다. 탐색할 이웃의 개수를 조정하는 것이 중요한데 너무 적으면 잡음이 너무 많고 너무 많으면 이웃한 그룹이 혼합된다.

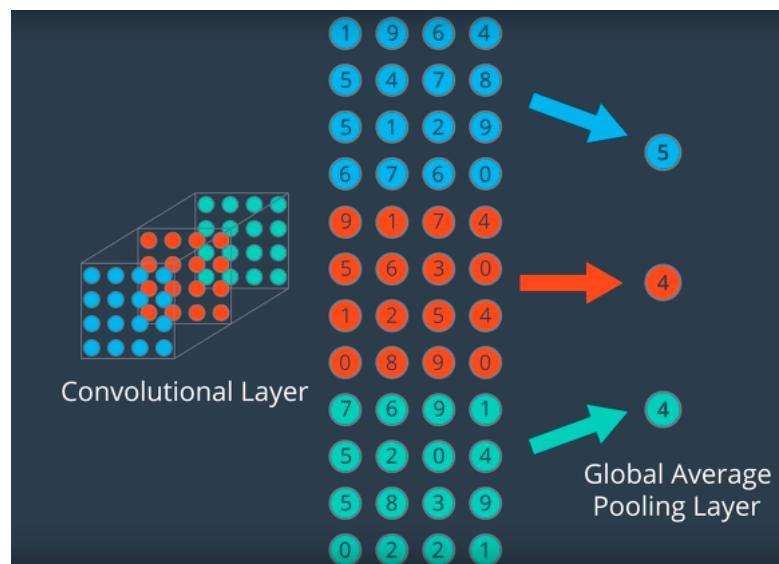
GPT-3는 OpenAI API를 통해서만 사용 가능하기 때문에 GPT-2를 사용해 이 기법을 테스트 할 예정.

GPT-2 같은 트랜스포머 모델은 토큰마다 하나의 임베딩 벡터를 반환한다. 하지만 실제로 필요한 것은 전체 문장에 대한 임베딩 벡터이다 → **풀링** 기법 사용

- 풀링 : 일반적으로 컨볼루션 계층의 출력으로 얻은 기능 맵의 공간 차원을 줄이는데 사용되는 기법.



Max Pooling : 문장에서 토큰 임베딩의 최대값을 가져온다.



Mean Pooling : 문장에서 토큰 임베딩의 평균 값을 취한다.

Sum Pooling : 문장에서 토큰 임베딩의 합을 가져온다.

Min Pooling : 문장에서 최솟값을 대푯값으로 뽑는다.

보통 feature 하나하나를 버리지 말고 평균을 내서 같이 가자 → Mean Pooling

분류하는거니까 대푯값만 필요하다 → Max Pooling

토큰 임베딩을 평균하는 평균 풀링을 사용한다. 주의사항은 평균에 패딩 토큰을 포함하면 안된다는 것이다.

```

import torch
from transformers import AutoTokenizer, AutoModel

# 코랩에서 python-gpt2-large를 사용하면 메모리 부족이 발생합니다.
# 대신 코랩 프로(https://colab.research.google.com/signup)를 사용하세요.
# 코랩을 사용하려면 대신 python-gpt2-medium을 사용하세요.
model_ckpt = "miguelvictor/python-gpt2-large"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = AutoModel.from_pretrained(model_ckpt)

def mean_pooling(model_output, attention_mask):
    
```

```

# 토큰 임베딩 추출하기
token_embeddings = model_output[0]
# 어텐션 마스크 계산하기
input_mask_expanded = (attention_mask
    .unsqueeze(-1)
    .expand(token_embeddings.size())
    .float())
# 임베딩을 더하지만 마스킹된 토큰은 무시합니다
sum_embeddings = torch.sum(token_embeddings * input_mask_expanded, 1)
sum_mask = torch.clamp(input_mask_expanded.sum(1), min=1e-9)
# 하나의 벡터로 평균을 반환합니다
return sum_embeddings / sum_mask

def embed_text(examples):
    inputs = tokenizer(examples["text"], padding=True, truncation=True,
        max_length=128, return_tensors="pt")
    with torch.no_grad():
        model_output = model(**inputs)
    pooled_embeds = mean_pooling(model_output, inputs["attention_mask"])
    return {"embedding": pooled_embeds.cpu().numpy()}

```

패딩 토큰을 포함하지 말아야 하기 때문에 어텐션 마스크를 이용하여 이를 처리한다.

GPT 스타일의 모델은 패딩 토큰이 없으므로 앞의 코드에서 구현된 것처럼 배치 형식으로 임베딩을 구하기 위해서는 패딩 토큰을 추가해야 한다. EOS(end-of-string) 토큰 재사용. (ex. <EOS>, </S>)

```

tokenizer.pad_token = tokenizer.eos_token
embs_train = ds["train"].map(embed_text, batched=True, batch_size=16)
embs_valid = ds["valid"].map(embed_text, batched=True, batch_size=16)
embs_test = ds["test"].map(embed_text, batched=True, batch_size=16)

```

쿼리할 새로운 텍스트 임베딩과 훈련 세트에 있는 기존 임베딩 사이의 코사인 유사도를 계산하는 함수 사용

- 코사인 유사도 : 두 벡터 간의 코사인 각도를 이용하여 구할 수 있는 두 벡터의 유사도를 의미한다.



값이 1에 가까울 수록 유사도가 높다고 판단할 수 있다.

- Faiss : facebook에서 만든 vector 유사도를 효율적으로 측정하는 라이브러리이다.

```

embs_train.add_faiss_index("embedding")

->
Dataset({
    features: ['text', 'labels', 'label_ids', 'embedding'],
    num_rows: 223
})

```

해당 함수를 통하여 데이터셋의 기준 필드를 FAISS 인덱스로 만든다.

```

len(embs_train[0]["embedding"])
✓ 0.3s
1280

```

토큰 임베딩이 아닌 문장 임베딩으로 된 것을 확인할 수 있다.

```

i, k = 0, 3      # 첫 번째 쿼리와 3개의 최근점 이웃을 선택합니다
rn, nl = "\r\n\r\n\r\n", "\n"  # 간결한 출력을 위해 텍스트에서 줄바꿈 문자를 삭제합니다

```

```

query = np.array(embs_valid[i]["embedding"], dtype=np.float32)
scores, samples = embs_train.get_nearest_examples("embedding", query, k=k)

print(f"쿼리 레이블: {embs_valid[i]['labels']}")
print(f"쿼리 텍스트:\n{embs_valid[i]['text'][:200].replace(rn, nl)} [...]\n")
print("=="*50)
print(f"추출된 문서:")
for score, label, text in zip(scores, samples["labels"], samples["text"]):
    print("=="*50)
    print(f"텍스트:\n{text[:200].replace(rn, nl)} [...]")
    print(f"점수: {score:.2f}")
    print(f"레이블: {label}")

->
Output exceeds the size limit. Open the full output data in a text editor
쿼리 레이블: ['new model']
쿼리 텍스트:
Implementing efficient self attention in T5

# 🌟 New model addition
My teammates and I (including @ice-americano) would like to use efficient self attention methods such as Linformer, Performer
=====
추출된 문서:
=====
텍스트:
Add Linformer model

# 🌟 New model addition
## Model description
### Linformer: Self-Attention with Linear Complexity
Paper published June 9th on Arxiv: https://arxiv.org/abs/2006.04768
La [...]
점수: 54.92
레이블: ['new model']
=====
텍스트:
Add FAVOR+ / Performer attention

# 🌟 FAVOR+ / Performer attention addition
...
## Model description
DeLight, that delivers similar or better performance than transformer-based models with sign [...]
점수: 60.12
레이블: ['new model']
=====
```

최근접 이웃 륙업을 수행한다. 임베딩 륙업으로 얻은 세 문서는 모두 레이블이 같다.

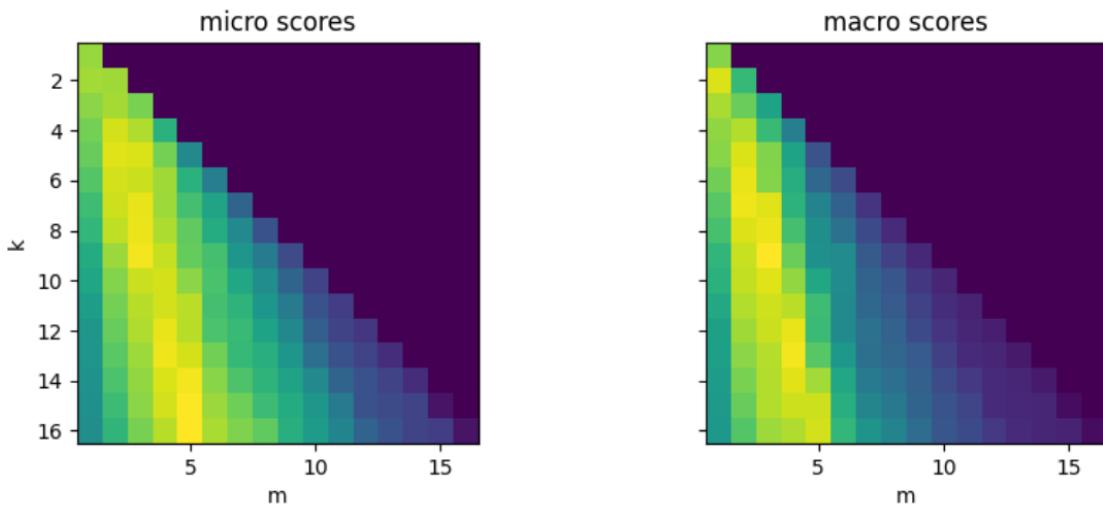
최적의 k 값은 얼마일까?

```

def get_sample_preds(sample, m):
    return (np.sum(sample["label_ids"], axis=0) >= m).astype(int)

def find_best_k(ds_train, valid_queries, valid_labels, max_k=17):
    max_k = min(len(ds_train), max_k)
    perf_micro = np.zeros((max_k, max_k))
    perf_macro = np.zeros((max_k, max_k))
    for k in range(1, max_k):
        for m in range(1, k + 1):
            _, samples = ds_train.get_nearest_examples_batch("embedding",
                                                               valid_queries, k=k)
            y_pred = np.array([get_sample_preds(s, m) for s in samples])
            clf_report = classification_report(valid_labels, y_pred,
                                                target_names=mlb.classes_, zero_division=0, output_dict=True)
            perf_micro[k, m] = clf_report["micro avg"]["f1-score"]
            perf_macro[k, m] = clf_report["macro avg"]["f1-score"]
    return perf_micro, perf_macro
```

해당 함수를 통하여 가장 좋은 성능을 내는 설정을 한다.

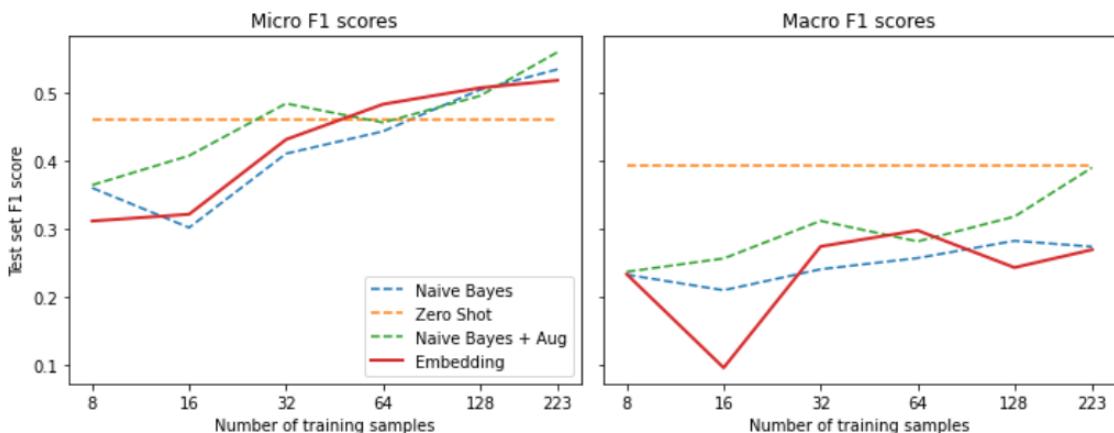


대략  $m/k = 1/3$  의 비율로 선택할 때 성능이 가장 좋다.

```
k, m = np.unravel_index(perf_micro.argmax(), perf_micro.shape)
print(f"최상의 k: {k}, 최상의 m: {m}")

->
최상의 k: 15, 최상의 m: 5
```

15개의 최근접 이웃을 검색하고 최소한 5회 등장한 레이블을 할당할 때 최상의 성능을 보이는 것을 확인할 수 있다.



마이크로 점수에서는 경쟁력이 있지만 매크로 점수에서는 나쁜 성능을 보인다.

### <기본 트랜스포머 미세 튜닝하기>

```
import torch
from transformers import (AutoTokenizer, AutoConfig,
                          AutoModelForSequenceClassification)

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

def tokenize(batch):
    return tokenizer(batch["text"], truncation=True, max_length=128)
ds_enc = ds.map(tokenize, batched=True)
ds_enc = ds_enc.remove_columns(['labels', 'text'])

ds_enc.set_format("torch")
ds_enc = ds_enc.map(lambda x: {"label_ids_f": x["label_ids"].to(torch.float)},
                    remove_columns=["label_ids"])
ds_enc = ds_enc.rename_column("label_ids_f", "label_ids")

->
```

```

DatasetDict({
    train: Dataset({
        features: ['input_ids', 'token_type_ids', 'attention_mask', 'label_ids'],
        num_rows: 223
    })
    valid: Dataset({
        features: ['input_ids', 'token_type_ids', 'attention_mask', 'label_ids'],
        num_rows: 106
    })
    test: Dataset({
        features: ['input_ids', 'token_type_ids', 'attention_mask', 'label_ids'],
        num_rows: 111
    })
    unsup: Dataset({
        features: ['input_ids', 'token_type_ids', 'attention_mask', 'label_ids'],
        num_rows: 9303
    })
})

```

훈련과 평가에 필요 없는 열인 labels, text를 삭제한다. 다중 레이블 손실 함수는 이산적인 레이블 대신 클래스 확률도 사용할 수 있기 때문에 실수 타입의 레이블을 기대한다. 그렇기에 타입을 바꾼다.

```

from transformers import Trainer, TrainingArguments

training_args_fine_tune = TrainingArguments(
    output_dir='./results', num_train_epochs=20, learning_rate=3e-5,
    lr_scheduler_type='constant', per_device_train_batch_size=4,
    per_device_eval_batch_size=32, weight_decay=0.0,
    evaluation_strategy="epoch", save_strategy="epoch", logging_strategy="epoch",
    load_best_model_at_end=True, metric_for_best_model='micro f1',
    save_total_limit=1, log_level='error')

```

해당 Trainer를 정의한다. 훈련 데이터가 굉장히 작기 때문에 훈련 데이터에 금세 과대적합이 될 가능성이 높다.  
load\_best\_model\_at\_end = True로 설정하여 마이크로 F1-score를 기반으로 최선의 모델 선택.

```

from scipy.special import expit as sigmoid

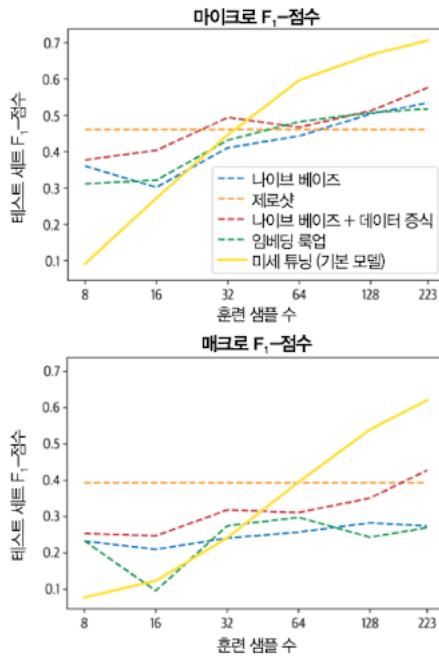
def compute_metrics(pred):
    y_true = pred.label_ids
    y_pred = sigmoid(pred.predictions)
    y_pred = (y_pred>0.5).astype(float)

    clf_dict = classification_report(y_true, y_pred, target_names=all_labels,
                                      zero_division=0, output_dict=True)
    return {"micro f1": clf_dict["micro avg"]["f1-score"],
            "macro f1": clf_dict["macro avg"]["f1-score"]}

```

최상의 모델 선택을 위해 F1-score를 가져온다. 로짓을 반환하기 때문에 시그모이드 함수로 정규화 한 후, 0.5 임곗값을 기준으로 모델의 예측 확률 출력을 이진 클래스 레이블로 변환한다. y\_pred 내부의 값들이 0.5보다 크면 True인 1을 float로 반환하고 반대라면 False인 0을 반환한다.

64개 샘플에서 경쟁력 있는 모델을 얻게 되었다.



### <프롬프트를 사용한 인-컨텍스트 학습과 퓨샷 학습>

- 프롬프트를 사용하고 모델의 토큰 예측을 파싱한다.
  - 훈련 데이터가 전혀 필요하지 않다.
  - 가지고 있는 레이블링된 데이터를 활용할 수 없다.

→ 인-컨텍스트 학습

#### ▼ 레이블링되지 않은 데이터 활용하기

- 분류기를 훈련할 최상의 시나리오는 고품질의 레이블링된 데이터를 대량으로 구하는 것이지만, 그렇다고 해서 레이블링되지 않은 데이터가 쓸모없다는 뜻은 아님.
- 예시로 BERT는 BookCorpus와 영어 위키피디아에서 사전 훈련한다. 하지만 코드가 담긴 텍스트와 깃허브 이슈는 확실히 작은 부분에 해당하는데 이를 위해 깃허브의 모든 이슈를 크롤링하여 학습하기엔 비용이 많이 듈다. 이렇게 밑바닥부터 재훈련하는 방법과 분류를 위해 그냥 모델을 사용하는 방법의 절충점
  - 도메인 적응(domain adaption) : 언어 모델을 밑바닥부터 재훈련하는 대신 주어진 도메인의 데이터에서 계속 훈련하는 방법, 마스킹된 단어를 예측하는 고전적인 언어 모델의 목표를 사용한다, 즉 레이블링된 데이터가 필요하지 않다.
- 도메인 적응은 레이블링된 데이터에 비해 레이블링되지 않은 데이터가 풍부한 경우가 많다는 사실을 활용

### <언어 모델 미세 튜닝하기>

- 토크나이저는 텍스트에 있는 일반적인 토큰 외에 분류와 다음 문장 예측에 사용할 [CLS]와 [SEP] 같은 특수 토큰도 시퀀스에 추가한다.

```

def tokenize(batch):
    return tokenizer(batch["text"], truncation=True,
                    max_length=128, return_special_tokens_mask=True)

ds_mlm = ds.map(tokenize, batched=True)
ds_mlm = ds_mlm.remove_columns(["labels", "text", "label_ids"])

->
DatasetDict({
    train: Dataset({
        features: ['input_ids', 'token_type_ids', 'attention_mask', 'special_tokens_mask'],
        num_rows: 223
    })
    valid: Dataset({
        features: ['input_ids', 'token_type_ids', 'attention_mask', 'special_tokens_mask'],
        num_rows: 106
    })
})

```

```

        })
    test: Dataset({
        features: ['input_ids', 'token_type_ids', 'attention_mask', 'special_tokens_mask'],
        num_rows: 111
    })
    unsup: Dataset({
        features: ['input_ids', 'token_type_ids', 'attention_mask', 'special_tokens_mask'],
        num_rows: 9303
    })
})

```

마스크드 언어 모델링을 시작하려면 입력 시퀀스에 있는 토큰을 마스킹하고 출력에서 타깃 토큰을 마련하는 메커니즘이 필요하다. → 랜덤한 토큰을 마스킹하고 시퀀스를 위해 레이블을 생성하는 함수 작성.

- 데이터 콜레이터 : 데이터셋과 모델 호출 사이를 연결하는 함수로, 배치를 쌓고 패딩하는 것 외에도 언어 모델 레이블 생성도 처리 한다. EX. 각 원소의 텐서를 하나의 텐서로 연결하는 경우에 동적으로 마스킹과 레이블 생성을 수행하기 위해 사용.

```

set_seed(3)
data_collator.return_tensors = "np"
inputs = tokenizer("Transformers are awesome!", return_tensors="np")
original_input_ids = inputs["input_ids"][0]
outputs = data_collator([{"input_ids": inputs["input_ids"][0]}])
masked_input_ids = outputs["input_ids"][0]

pd.DataFrame({
    "Original tokens": tokenizer.convert_ids_to_tokens(original_input_ids),
    "Masked tokens": tokenizer.convert_ids_to_tokens(masked_input_ids),
    "Original input_ids": original_input_ids,
    "Masked input_ids": masked_input_ids,
    "Labels": outputs["labels"][0]).T

```

<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
Original tokens	[CLS]	transformers	are	awesome	!
Masked tokens	[CLS]	transformers	are	awesome	[MASK]
Original input_ids	101	19081	2024	12476	999
Masked input_ids	101	19081	2024	12476	103
Labels	-100	-100	-100	-100	999

데이터 콜레이터는 다음과 같이 생성된다.

```

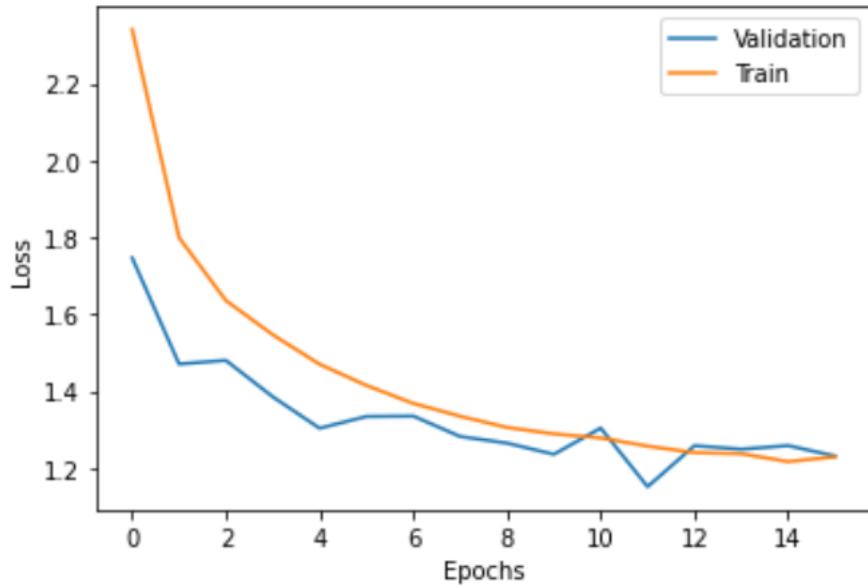
from transformers import AutoModelForMaskedLM

training_args = TrainingArguments(
    output_dir = f"{model_ckpt}-issues-128", per_device_train_batch_size=32,
    logging_strategy="epoch", evaluation_strategy="epoch", save_strategy="no",
    num_train_epochs=16, push_to_hub=True, log_level="error", report_to="none")

trainer = Trainer(
    model=AutoModelForMaskedLM.from_pretrained("bert-base-uncased"),
    tokenizer=tokenizer, args=training_args, data_collator=data_collator,
    train_dataset=ds_mlm["unsup"], eval_dataset=ds_mlm["train"])

trainer.train()

```



데이터 콜레이터와 특수 토큰을 생성하는 토크나이저를 사용하였을 때, 결과이다.

### <분류기 미세 튜닝하기>

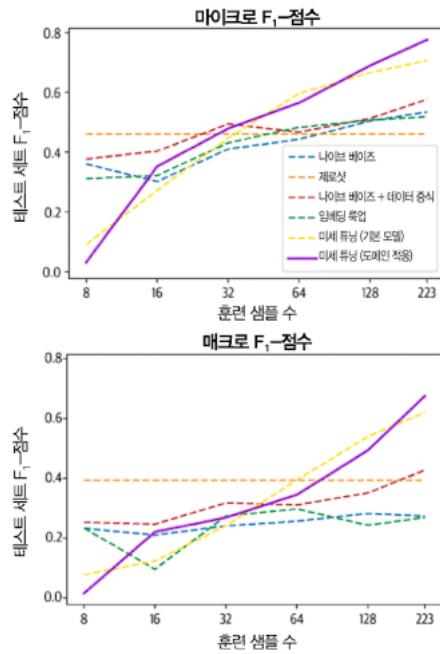
```

model_ckpt = f'{model_ckpt}-issues-128'
config = AutoConfig.from_pretrained(model_ckpt)
config.num_labels = len(all_labels)
config.problem_type = "multi_label_classification"

for train_slice in train_slices:
    model = AutoModelForSequenceClassification.from_pretrained(model_ckpt,
                                                                config=config)
    trainer = Trainer(
        model=model,
        tokenizer=tokenizer,
        args=training_args_fine_tune,
        compute_metrics=compute_metrics,
        train_dataset=ds_enc["train"].select(train_slice),
        eval_dataset=ds_enc["valid"],
    )
    trainer.train()
    pred = trainer.predict(ds_enc['test'])
    metrics = compute_metrics(pred)
    # DA는 도메인 적용을 의미합니다
    macro_scores['Fine-tune (DA)'].append(metrics['macro f1'])
    micro_scores['Fine-tune (DA)'].append(metrics['micro f1'])

```

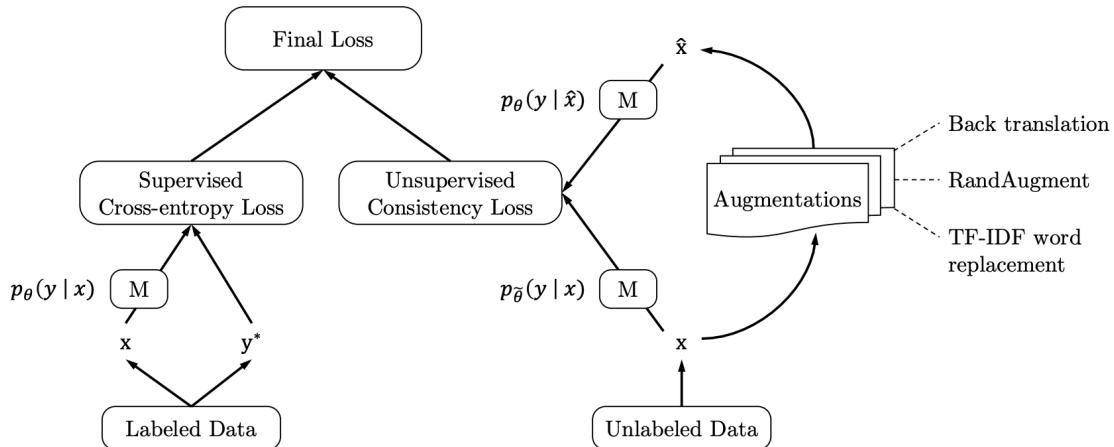
앞에서 만든 체크포인트를 로드하여 도메인 적응한 모델에 대해 점수를 측정해보자.



기본 BERT를 기반으로 미세 튜닝한 결과와 비교하면, 특히 데이터가 부족한 도메인에서 유리하다.

→ 도메인 적응이 레이블링되지 않은 데이터와 약간의 노력을 통해 모델 성능을 조금 향상시킬 수 있다는 사실을 보여준다.

### <고급 방법>



- 비지도 데이터 증식 (unsupervised data augmentation : UDA) : Labeled 데이터와 Unlabeled 데이터를 함께 학습에 활용하는 Semi-supervised Learning 방법론이다.
  - Supervised Loss는 일반적인 분류 학습에 활용하는 Cross-entropy Loss이므로 Labeled 데이터의 문장  $x$ 와 라벨  $y$ 가 있으면 쉽게 구성할 수 있다.
  - Consistency Loss는 두 개의 의미가 비슷한 문장이 필요하다. 따라서 Unlabeled 데이터의 문장  $x$ 뿐만 아니라  $x$ 와 비슷한 의미를 지니지만 문법적, 단어의 표현이 다른 문장  $x_{\sim}$ 를 생성해야 한다.
  - Data Augmentation을 통해 생성된 문장을 분류모델에 넣으면 특정 라벨에 속할 확률분포를 추출할 수 있다

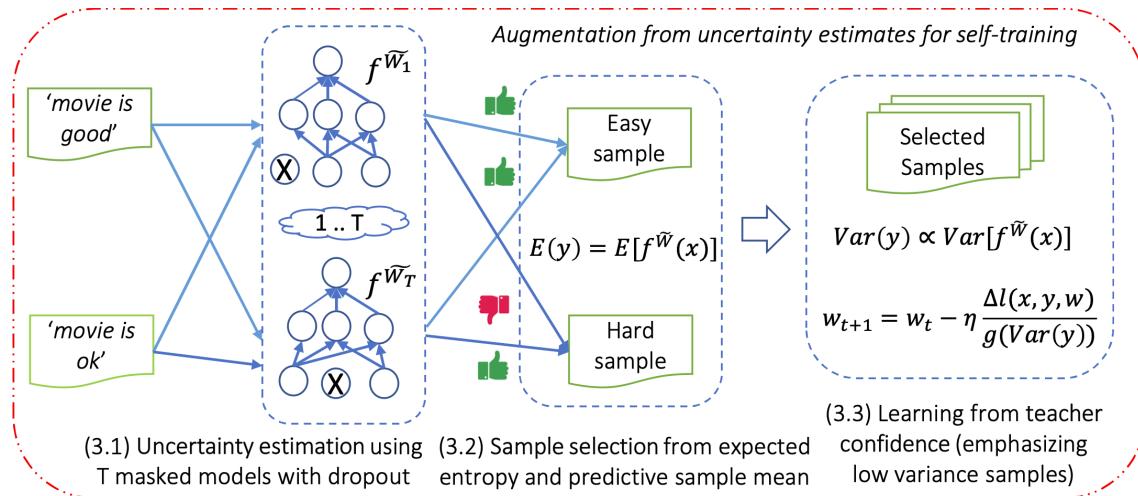
$$\text{Augmentation} = p_{\theta}(y|\hat{x})$$

$$\text{Original} = p_{\theta}(y|x)$$

- 책에서는 이를 왜곡된 샘플과 Unlabeled 샘플에 대해 모델의 예측이 일정해야 한다고 한다. 왜곡은 톤 교체 혹은 역 번역 같은 데이터 증식 전략으로 만든다.
- Back Translation(역 번역) : 대상 언어의 텍스트를 다시 소스 언어로 번역한 다음 다시 번역된 텍스트를 원본 텍스트와 비교하는 작업 (ex. 영어를 프랑스어로 번역한 다음 다른 기계 번역 시스템을 사용하여 프랑스어 문장을 다시 영어로 번역하는 작업)
- RandAugment(랜덤 증식) : 텍스트에 특정 비율을 무작위로 마스킹하거나 단어를 치환하거나 입력에 노이즈를 추가하는 방식
- TF-IDF word replacement(TF-IDF 단어 교체) : Term Frequency - Inverse Document Frequency)
  - TF : 특정 문서에서의 특정 단어의 등장 횟수, ex) 문서 [I am a great great boy]가 있을 때 각 단어의 'TF'는 [I, am, a, great, boy] == [1, 1, 1, 2, 1]
  - DF : 특정 단어가 등장한 문서의 수, ex) 문서 1 [I am a great great boy], 문서 2 [I am a boy]가 있을 때 'df'는 [I, am, a, great, boy] == [2, 2, 2, 1, 2]이 됩니다. 여기서 great == 1로 정의되는데 그 이유는 great이 문서 1에서 2번 사용되었지만, 사용된 문서수는 문서 1 하나이므로 1로 대응
  - IDF : DF에 반비례하는 수, 'the'나 'a'와 같이 너무 많이 사용되는 단어의 가중치를 줄이기 위해 필요한 값, log를 취한 이유는 숫자의 스케일을 줄여주기 위함입니다. 그 이유는 숫자간의 편차가 너무 크면 알아보기 힘들 뿐만 아니라 학습도 잘 되지 않습니다. 또한 분모에 '1'을 더해주는 이유는 분모가 '0'이 되는 것을 방지하기 위함입니다.

$$idf(d, t) = \log\left(\frac{n}{1 + df(t)}\right)$$

- TF-IDF 점수는 문서의 TF와 IDF를 곱하여 계산된다.
- TF-IDF word replacement는 문서의 단어를 해당 TF-IDF 점수로 바꾸는 것.
- ex) "The cat sat on the mat." 문서의 각 단어에 대한 TF-IDF 점수가 계산된 다음 해당 단어가 해당 점수로 대체됩니다. 결과 문서는 다음과 같습니다. "0.15 0.11 0.12 0.08 0.11"
- 텍스트 데이터의 차원을 줄이는데 도움이 되어 모델에서 더 쉽게 관리 가능하고 가장 중요한 단어를 강조 표시하여 텍스트 데이터의 해석 가능성을 향상시키는데 도움이 될 수 있음.
- 위와 같은 데이터 증식 파이프라인이 필요하고, 레이블링되지 않은 샘플과 증식 샘플에 대해 예측을 만들기 위한 정방향 패스를 여러 번 실행시켜야 하므로 훈련이 오래 걸림.



- 불확실성 인지 자기 훈련(uncertainty-aware self-training : UST) : 레이블링된 티처 모델을 훈련하고 이 모델을 사용하여 레이블링되지 않은 데이터에서 의사 레이블(pseudo-label)을 만든다는 개념에 기초. 스튜던트 모델은 의사 레이블 데이터에서 훈련되고, 훈련 후에는 다음 반복에서 티처가 된다.
  - 의사 레이블 생성 방법 : 모델 예측의 불확실성을 측정하기 위해 동일한 입력을 드롭아웃이 적용된 모델에 여러 번 주입한다. 이런 예측의 분산이 특정 샘플에 대한 모델의 불확실성을 대변한다.
  - BALD(Bayesian Active Learning by Disagreement)라는 방법을 사용해 의사 레이블을 샘플링.

- 베이지안 모델을 사용하여 레이블이 지정되지 않은 데이터 세트에 대한 예측을 수행한 다음 모델의 불확실성 또는 불일치가 가장 높은 예를 선택하여 레이블을 지정하는 것. 이것은 불확실성의 측정으로 사용되는 예측 레이블과 실제 레이블 간의 상호 정보를 측정하여 수행.
- 티처는 이런 반복 구조를 통해 지속적으로 의사 레이블을 만드는 데 능숙해지므로 모델 성능이 향상
- ex) 뉴스 기사의 대규모 데이터 세트를 다른 범주로 분류하기 위해 텍스트 분류기를 훈련하려는 경우, 사전 훈련된 모델을 사용하여 기사의 레이블을 예측한 다음 모델이 사람에 의해 레이블링되기가 가장 불확실한 기사를 선택할 수 있습니다. 이러한 방식으로 가장 불확실한 예를 사용하여 모델의 성능을 반복적으로 개선할 수 있습니다.
- 
- 불확실성 인식 자체 훈련은 제한된 레이블링된 데이터를 사용할 수 있을 때 기계 학습 모델의 성능을 향상시키는 데 사용되는 기술입니다. 모델의 자체 예측을 추가 훈련 데이터로 사용하는 준지도 학습 기법인 자가 훈련과 레이블링할 가장 불확실한 예제를 선택하는 방법인 불확실성 샘플링을 결합합니다.
- 기본 아이디어는 사전 훈련된 모델을 사용하여 레이블이 지정되지 않은 데이터 세트에 대해 예측한 다음 모델이 인간 주석자에 의해 레이블이 지정될 것으로 가장 불확실한 예를 선택하는 것입니다. 그런 다음 레이블이 지정된 예제를 교육 세트에 추가하여 모델을 개선하는 데 사용합니다. 이 과정은 모형의 성능이 만족스러운 수준에 도달할 때까지 여러 번 반복됩니다.
- 불확실성 인식 자체 훈련은 텍스트 분류, 명명된 엔티티 인식 및 감정 분석과 같은 광범위한 자연어 처리(NLP) 작업에 유용할 수 있습니다. 레이블이 지정된 데이터가 부족하거나 얻기기에 비용이 많이 드는 상황에서 특히 유용할 수 있습니다.
- 예를 들어, 뉴스 기사의 대규모 데이터 세트를 다른 범주로 분류하기 위해 텍스트 분류기를 훈련하려는 경우, 사전 훈련된 모델을 사용하여 기사의 레이블을 예측한 다음 모델이 인간 주석자에 의해 레이블링되기가 가장 불확실한 기사를 선택할 수 있습니다. 이러한 방식으로 가장 불확실한 예를 사용하여 모델의 성능을 반복적으로 개선할 수 있습니다.