

一. Supervised Classification 监督分类

在第四章中，我们使用了预训练好的模型对文本进行分类，训练过程是被冻结的。这些模型要么被训练来预测情感（特定于任务的模型），要么被训练来生成嵌入（嵌入模型），如图 11-1 所示。

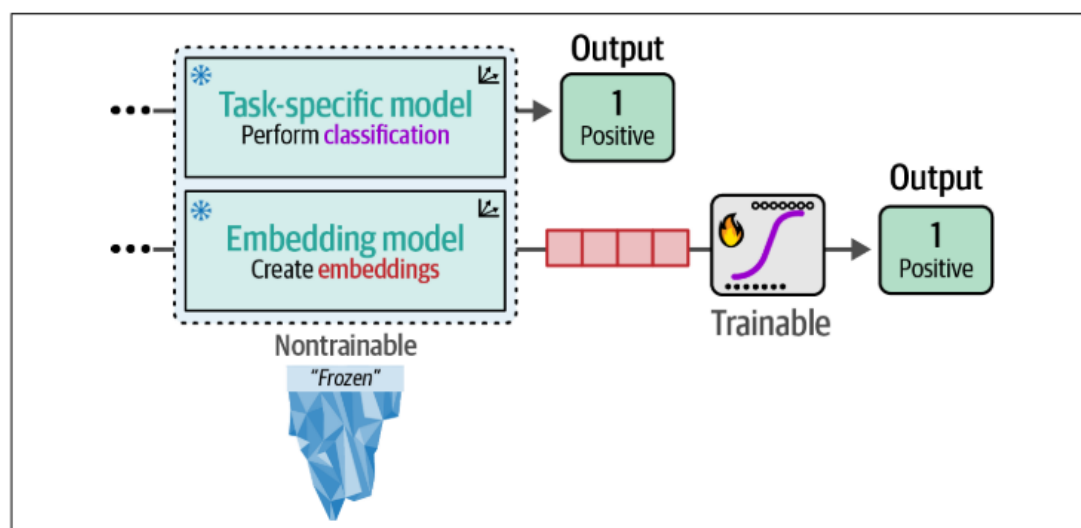


Figure 11-1. In [Chapter 4](#), we used pretrained models to perform classification without updating their weight. These models were kept “frozen.”

这两个模型都被冻结（不可训练），以展示利用预训练模型进行分类任务的潜力。在第二个模型中，在嵌入模型后可以使用单独的可训练分类头（分类器）来预测电影评论的情感。

在本节中，我们将采用类似的方法，但允许在训练期间更新模型和分类头。如图 11-2 所示，我们将不使用嵌入模型，而是微调预训练的 BERT 模型，以创建类似于在第 2 章中使用的特定任务模型。与嵌入模型方法相比，我们将把表示模型和分类头作为一个架构进行微调。

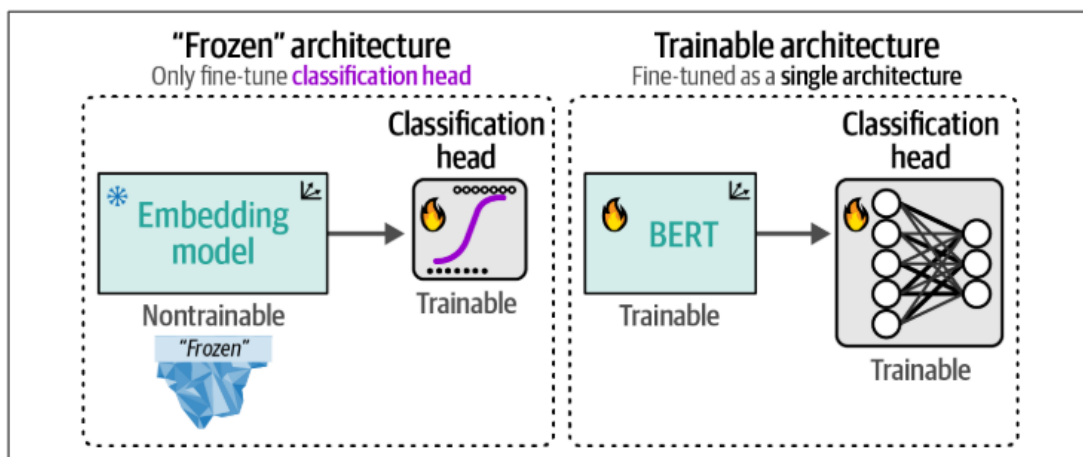


Figure 11-2. Compared to the “frozen” architecture, we instead train both the pretrained BERT model and the classification head. A backward pass will start at the classification head and go through BERT.

为了做到这一点，我们不是冻结模型，而是允许它是可训练的，并在训练过程中更新其参数。我们将使用预训练的 BERT 模型，并添加一个神经网络作为分类头，两者都将进行微调以进行分类。

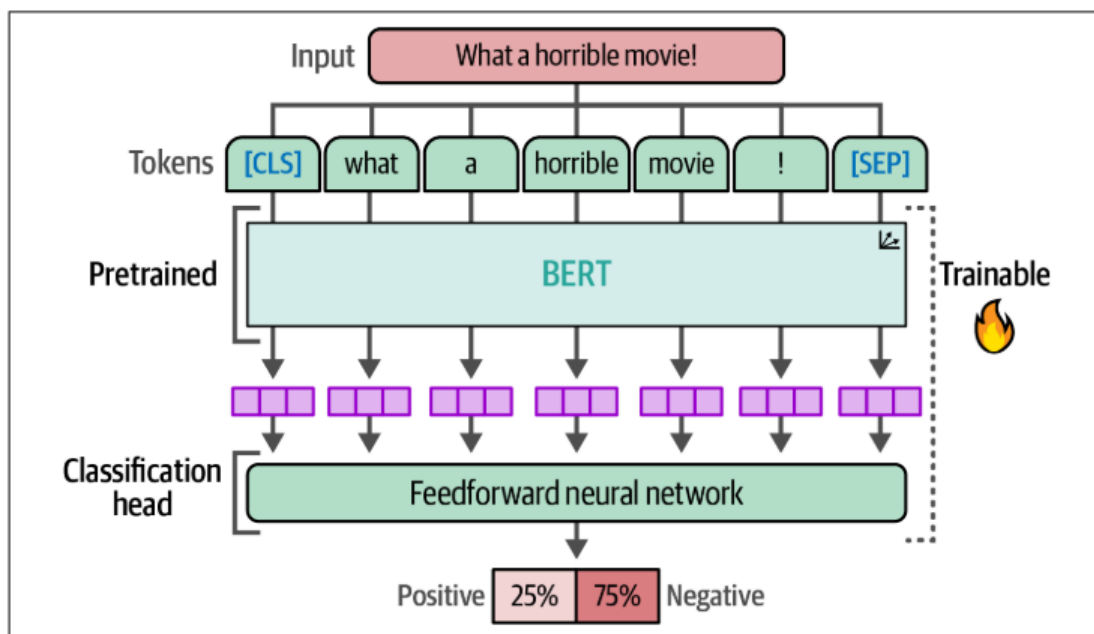


Figure 11-3. The architecture of a task-specific model. It contains a pretrained representation model (e.g., BERT) with an additional classification head for the specific task.

1. 微调预训练 BERT 模型

我们将使用第 4 章中使用过的相同数据集来微调我们的模型，即烂番茄数据集，其中包含来自烂番茄网站的 5,331 个正面和 5,331 个负面电影评论：

```
~ Data

from datasets import load_dataset

# Prepare data and splits
tomatoes = load_dataset("rotten_tomatoes")
train_data, test_data = tomatoes["train"], tomatoes["test"]

/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), see https://huggingface.co/docs/huggingface\_hub/quick-look#authentication.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
warnings.warn(
Downloading readme: 100% ██████████ 7.46k/7.46k [00:00<00:00, 35.9kB/s]
Downloading data: 100% ██████████ 699k/699k [00:00<00:00, 2.89MB/s]
Downloading data: 100% ██████████ 90.0k/90.0k [00:00<00:00, 342kB/s]
Downloading data: 100% ██████████ 92.2k/92.2k [00:00<00:00, 488kB/s]
Generating train split: 100% ██████████ 8530/8530 [00:00<00:00, 135123.32 examples/s]
Generating validation split: 100% ██████████ 1066/1066 [00:00<00:00, 43011.05 examples/s]
Generating test split: 100% ██████████ 1066/1066 [00:00<00:00, 68012.29 examples/s]
```

分类任务的第一步是选择我们想要使用的底层模型“bert-base-cased”。

```
~ Supervised Classification

~ HuggingFace Trainer

from transformers import AutoTokenizer, AutoModelForSequenceClassification

# Load Model and Tokenizer
model_id = "bert-base-cased"
model = AutoModelForSequenceClassification.from_pretrained(model_id, num_labels=2)
tokenizer = AutoTokenizer.from_pretrained(model_id)

config.json: 100% ██████████ 570/570 [00:00<00:00, 56.1kB/s]
model.safetensors: 100% ██████████ 436M/436M [00:05<00:00, 95.2MB/s]
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-cased and are newly initialized from the random normal distribution. You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
tokenizer_config.json: 100% ██████████ 49.0/49.0 [00:00<00:00, 4.56kB/s]
vocab.txt: 100% ██████████ 213k/213k [00:00<00:00, 13.7MB/s]
tokenizer.json: 100% ██████████ 436k/436k [00:00<00:00, 32.3MB/s]
```

然后用 tokenizer 对数据进行分词处理。

在自然语言处理（NLP）任务中，文本序列的长度通常是可变的。

DataCollatorWithPadding 是 Hugging Face transformers 库中一个非常重要的数据整理器（Data Collator）。它的核心作用是动态地对一个批次（batch）内的样本进行填充（padding），确保批次内所有样本具有相同的长度，从而可以被有效地输入到神经网络模型中进行批量训练或推理。神经网络（尤其是 Transformer 模型）通常要求一个批次内的所有输入张量具有相同的维度（即


相同的序列长度)。

```
from transformers import DataCollatorWithPadding

# Pad to the longest sequence in the batch
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)

def preprocess_function(examples):
    """Tokenize input data"""
    return tokenizer(examples["text"], truncation=True)

# Tokenize train/test data
tokenized_train = train_data.map(preprocess_function, batched=True)
tokenized_test = test_data.map(preprocess_function, batched=True)
```



The image shows two progress bars. The first bar is labeled 'Map: 100%' and shows a green bar at 100% completion, with the text '8530/8530 [00:00<00:00, 11019.69 examples/s]' to its right. The second bar is also labeled 'Map: 100%' and shows a green bar at 100% completion, with the text '1066/1066 [00:00<00:00, 2146.50 examples/s]' to its right.

定义模型性能的度量标准。这里用 f1 值进行评判。使用 `compute_metrics`，我们可以定义任何数量的我们感兴趣的指标，并且可以在训练期间打印或记录。

```
Define metrics.

[5] import numpy as np
import evaluate

def compute_metrics(eval_pred):
    """Calculate F1 score"""
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)

    load_f1 = evaluate.load("f1")
    f1 = load_f1.compute(predictions=predictions, references=labels)["f1"]
    return {"f1": f1}
```

`TrainingArguments` 类定义超参数，`trainer` 用于执行培训过程。

Train model.

```
from transformers import TrainingArguments, Trainer

# Training arguments for parameter tuning
training_args = TrainingArguments(
    "model",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=1,
    weight_decay=0.01,
    save_strategy="epoch",
    report_to="none"
)

# Trainer which executes the training process
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_train,
    eval_dataset=tokenized_test,
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)
```

训练并调用 evaluate 函数。

✓ 1 分钟

[7] `trainer.train()`

↕ [534/534 01:39, Epoch 1/1]

Step	Training Loss
500	0.417100

TrainOutput(global_step=534, training_loss=0.41323024949777437, metrics={'train_runtime': 101.1383, 'train_loss': 0.41323024949777437, 'epoch': 1.0})

Evaluate results.

✓ 3 秒

▶ `trainer.evaluate()`

↕ [67/67 00:03]

Downloading builder script: 6.79k/? [00:00<00:00, 567kB/s]

{'eval_loss': 0.36805158853530884,
'eval_f1': 0.8486536675951718,
'eval_runtime': 3.8766,
'eval_samples_per_second': 274.986,
'eval_steps_per_second': 17.283,
'epoch': 1.0}

2. 冻结层

为了进一步展示训练整个网络的重要性，下一个示例将演示如何使用 Hugging Face Transformers 来冻结网络的某些层。我们将冻结主 BERT 模型，只允许更新通过分类头。

首先，重新初始化模型，这样我们就可以从头开始：

Freeze Layers

```
[9] # Load Model and Tokenizer
model = AutoModelForSequenceClassification.from_pretrained(model_id, num_labels=2)
tokenizer = AutoTokenizer.from_pretrained(model_id)
```

BERT 模型包含了很多可以冻结的层。打印并检查这些层可以深入了解网络的结构以及我们想要冻结的内容：BERT 由嵌入层、编码层、池化层和分类器组成。

```
[10] # Print layer names
for name, param in model.named_parameters():
    print(name)

bert.embeddings.word_embeddings.weight
bert.embeddings.position_embeddings.weight
bert.embeddings.token_type_embeddings.weight
bert.embeddings.LayerNorm.weight
bert.embeddings.LayerNorm.bias
bert.encoder.layer.0.attention.self.query.weight
bert.encoder.layer.0.attention.self.query.bias
bert.encoder.layer.0.attention.self.key.weight
bert.encoder.layer.0.attention.self.key.bias
bert.encoder.layer.0.attention.self.value.weight
bert.encoder.layer.0.attention.self.value.bias
bert.encoder.layer.0.attention.output.dense.weight
bert.encoder.layer.0.attention.output.dense.bias
bert.encoder.layer.0.attention.output.LayerNorm.weight
bert.encoder.layer.0.attention.output.LayerNorm.bias
bert.encoder.layer.0.intermediate.dense.weight
bert.encoder.layer.0.intermediate.dense.bias
bert.encoder.layer.0.output.dense.weight
bert.encoder.layer.0.output.dense.bias
bert.encoder.layer.0.output.LayerNorm.weight
bert.encoder.layer.0.output.LayerNorm.bias
bert.encoder.layer.1.attention.self.query.weight
bert.encoder.layer.1.attention.self.query.bias
bert.encoder.layer.1.attention.self.key.weight
bert.encoder.layer.1.attention.self.key.bias
bert.encoder.layer.1.attention.self.value.weight
bert.encoder.layer.1.attention.self.value.bias
bert.encoder.layer.1.attention.output.dense.weight
bert.encoder.layer.1.attention.output.dense.bias
bert.encoder.layer.1.attention.output.LayerNorm.weight
bert.encoder.layer.1.attention.output.LayerNorm.bias
bert.encoder.layer.1.intermediate.dense.weight
bert.encoder.layer.1.intermediate.dense.bias
bert.encoder.layer.1.output.dense.weight
bert.encoder.layer.1.output.dense.bias
bert.encoder.layer.1.output.LayerNorm.weight
bert.encoder.layer.1.output.LayerNorm.bias
```

```
bert.encoder.layer.10.attention.self.query.weight
bert.encoder.layer.10.attention.self.query.bias
bert.encoder.layer.10.attention.self.key.weight
bert.encoder.layer.10.attention.self.key.bias
bert.encoder.layer.10.attention.self.value.weight
bert.encoder.layer.10.attention.self.value.bias
bert.encoder.layer.10.attention.output.dense.weight
bert.encoder.layer.10.attention.output.dense.bias
bert.encoder.layer.10.attention.output.LayerNorm.weight
bert.encoder.layer.10.attention.output.LayerNorm.bias
bert.encoder.layer.10.intermediate.dense.weight
bert.encoder.layer.10.intermediate.dense.bias
bert.encoder.layer.10.output.dense.weight
bert.encoder.layer.10.output.dense.bias
bert.encoder.layer.10.output.LayerNorm.weight
bert.encoder.layer.10.output.LayerNorm.bias
bert.encoder.layer.11.attention.self.query.weight
bert.encoder.layer.11.attention.self.query.bias
bert.encoder.layer.11.attention.self.key.weight
bert.encoder.layer.11.attention.self.key.bias
bert.encoder.layer.11.attention.self.value.weight
bert.encoder.layer.11.attention.self.value.bias
bert.encoder.layer.11.attention.output.dense.weight
bert.encoder.layer.11.attention.output.dense.bias
bert.encoder.layer.11.attention.output.LayerNorm.weight
bert.encoder.layer.11.attention.output.LayerNorm.bias
bert.encoder.layer.11.intermediate.dense.weight
bert.encoder.layer.11.intermediate.dense.bias
bert.encoder.layer.11.output.dense.weight
bert.encoder.layer.11.output.dense.bias
bert.encoder.layer.11.output.LayerNorm.weight
bert.encoder.layer.11.output.LayerNorm.bias
bert.pooler.dense.weight
bert.pooler.dense.bias
classifier.weight
classifier.bias
```

有 12 (0-11) 个编码器块 encoder blocks, 由 attention heads, dense networks and layer normalization 组成。

冻结除了 classifier 之外的所有内容:

```
[11] for name, param in model.named_parameters():

    # Trainable classification head
    if name.startswith("classifier"):
        param.requires_grad = True

    # Freeze everything else
    else:
        param.requires_grad = False
```


打印检查层级信息，我们已经冻结了除了 feedforward neural network（分类头）以外的所有部分。

```
[12] # We can check whether the model was correctly updated
      for name, param in model.named_parameters():
          print(f"Parameter: {name} ----- {param.requires_grad}")
```

➡ Parameter: bert.embeddings.word_embeddings.weight ----- False
Parameter: bert.embeddings.position_embeddings.weight ----- False
Parameter: bert.embeddings.token_type_embeddings.weight ----- False
Parameter: bert.embeddings.LayerNorm.weight ----- False
Parameter: bert.embeddings.LayerNorm.bias ----- False
Parameter: bert.encoder.layer.0.attention.self.query.weight ----- False
Parameter: bert.encoder.layer.0.attention.self.query.bias ----- False
Parameter: bert.encoder.layer.0.attention.self.key.weight ----- False
Parameter: bert.encoder.layer.0.attention.self.key.bias ----- False
Parameter: bert.encoder.layer.0.attention.self.value.weight ----- False
Parameter: bert.encoder.layer.0.attention.self.value.bias ----- False
Parameter: bert.encoder.layer.0.attention.output.dense.weight ----- False
Parameter: bert.encoder.layer.0.attention.output.dense.bias ----- False
Parameter: bert.encoder.layer.0.attention.output.LayerNorm.weight ----- False
Parameter: bert.encoder.layer.0.attention.output.LayerNorm.bias ----- False
Parameter: bert.encoder.layer.0.intermediate.dense.weight ----- False
Parameter: bert.encoder.layer.0.intermediate.dense.bias ----- False
Parameter: bert.encoder.layer.0.output.dense.weight ----- False
Parameter: bert.encoder.layer.0.output.dense.bias ----- False
Parameter: bert.encoder.layer.0.output.LayerNorm.weight ----- False
Parameter: bert.encoder.layer.0.output.LayerNorm.bias ----- False
Parameter: bert.encoder.layer.1.attention.self.query.weight ----- False
Parameter: bert.encoder.layer.1.attention.self.query.bias ----- False
Parameter: bert.encoder.layer.1.attention.self.key.weight ----- False
Parameter: bert.encoder.layer.1.attention.self.key.bias ----- False
Parameter: bert.encoder.layer.1.attention.self.value.weight ----- False
Parameter: bert.encoder.layer.1.attention.self.value.bias ----- False
Parameter: bert.encoder.layer.1.attention.output.dense.weight ----- False
Parameter: bert.encoder.layer.1.attention.output.dense.bias ----- False
Parameter: bert.encoder.layer.1.attention.output.LayerNorm.weight ----- False
Parameter: bert.encoder.layer.1.attention.output.LayerNorm.bias ----- False
Parameter: bert.encoder.layer.1.intermediate.dense.weight ----- False
Parameter: bert.encoder.layer.1.intermediate.dense.bias ----- False
Parameter: bert.encoder.layer.1.output.dense.weight ----- False
Parameter: bert.encoder.layer.1.output.dense.bias ----- False
Parameter: bert.encoder.layer.1.output.LayerNorm.weight ----- False
Parameter: bert.encoder.layer.1.output.LayerNorm.bias ----- False
Parameter: bert.encoder.layer.2.attention.self.query.weight ----- False


```
Parameter: bert.encoder.layer.9.output.LayerNorm.bias ----- False
Parameter: bert.encoder.layer.10.attention.self.query.weight ----- False
Parameter: bert.encoder.layer.10.attention.self.query.bias ----- False
Parameter: bert.encoder.layer.10.attention.self.key.weight ----- False
Parameter: bert.encoder.layer.10.attention.self.key.bias ----- False
Parameter: bert.encoder.layer.10.attention.self.value.weight ----- False
Parameter: bert.encoder.layer.10.attention.self.value.bias ----- False
Parameter: bert.encoder.layer.10.attention.output.dense.weight ----- False
Parameter: bert.encoder.layer.10.attention.output.dense.bias ----- False
Parameter: bert.encoder.layer.10.attention.output.LayerNorm.weight ----- False
Parameter: bert.encoder.layer.10.attention.output.LayerNorm.bias ----- False
Parameter: bert.encoder.layer.10.intermediate.dense.weight ----- False
Parameter: bert.encoder.layer.10.intermediate.dense.bias ----- False
Parameter: bert.encoder.layer.10.output.dense.weight ----- False
Parameter: bert.encoder.layer.10.output.dense.bias ----- False
Parameter: bert.encoder.layer.10.output.LayerNorm.weight ----- False
Parameter: bert.encoder.layer.10.output.LayerNorm.bias ----- False
Parameter: bert.encoder.layer.11.attention.self.query.weight ----- False
Parameter: bert.encoder.layer.11.attention.self.query.bias ----- False
Parameter: bert.encoder.layer.11.attention.self.key.weight ----- False
Parameter: bert.encoder.layer.11.attention.self.key.bias ----- False
Parameter: bert.encoder.layer.11.attention.self.value.weight ----- False
Parameter: bert.encoder.layer.11.attention.self.value.bias ----- False
Parameter: bert.encoder.layer.11.attention.output.dense.weight ----- False
Parameter: bert.encoder.layer.11.attention.output.dense.bias ----- False
Parameter: bert.encoder.layer.11.attention.output.LayerNorm.weight ----- False
Parameter: bert.encoder.layer.11.attention.output.LayerNorm.bias ----- False
Parameter: bert.encoder.layer.11.intermediate.dense.weight ----- False
Parameter: bert.encoder.layer.11.intermediate.dense.bias ----- False
Parameter: bert.encoder.layer.11.output.dense.weight ----- False
Parameter: bert.encoder.layer.11.output.dense.bias ----- False
Parameter: bert.encoder.layer.11.output.LayerNorm.weight ----- False
Parameter: bert.encoder.layer.11.output.LayerNorm.bias ----- False
Parameter: bert.pooler.dense.weight ----- False
Parameter: bert.pooler.dense.bias ----- False
Parameter: classifier.weight ----- True
Parameter: classifier.bias ----- True
```

用 Trainer 进行训练并调用 evaluate 进行性能评价, F1 score 为 0.64, 和所有层都可以训练的模型相比 (F1 score = 0.85), F1 下降了 0.21。

```
[13] from transformers import TrainingArguments, Trainer
```

```
# Trainer which executes the training process
```

```
trainer = Trainer(  
    model=model,  
    args=training_args,  
    train_dataset=tokenized_train,  
    eval_dataset=tokenized_test,  
    tokenizer=tokenizer,  
    data_collator=data_collator,  
    compute_metrics=compute_metrics,  
)  
trainer.train()
```

```
→ /tmp/ipython-input-2563977997.py:4: FutureWarning: `tokenizer` is deprecated  
    trainer = Trainer(  
    [534/534 00:32, Epoch 1/1]
```

Step	Training Loss
------	---------------

500	0.694700
-----	----------

```
TrainOutput(global_step=534, training_loss=0.6948552721002129, metrics=  
'train_loss': 0.6948552721002129, 'epoch': 1.0))
```

```
[14] trainer.evaluate()
```

```
→ [67/67 00:03]
```

```
{'eval_loss': 0.6825135946273804,  
 'eval_f1': 0.64,  
 'eval_runtime': 3.7022,  
 'eval_samples_per_second': 287.936,  
 'eval_steps_per_second': 18.097,  
 'epoch': 1.0}
```

接下来我们将冻结 BERT 模型中的特定层，来观察模型的性能变化。首先冻结模型中的前十层 encoder。查看当前模型的层级信息，并输出每层的索引 Index：

Freeze blocks 0-9

```
# We can check whether the model was correctly updated
for index, (name, param) in enumerate(model.named_parameters()):
    print(f"Parameter: {index} {name} ----- {param.requires_grad}")
```

Parameter: 143bert.encoder.layer.8.intermediate.dense.weight ----- False
Parameter: 144bert.encoder.layer.8.intermediate.dense.bias ----- False
Parameter: 145bert.encoder.layer.8.output.dense.weight ----- False
Parameter: 146bert.encoder.layer.8.output.dense.bias ----- False
Parameter: 147bert.encoder.layer.8.output.LayerNorm.weight ----- False
Parameter: 148bert.encoder.layer.8.output.LayerNorm.bias ----- False
Parameter: 149bert.encoder.layer.9.attention.self.query.weight ----- False
Parameter: 150bert.encoder.layer.9.attention.self.query.bias ----- False
Parameter: 151bert.encoder.layer.9.attention.self.key.weight ----- False
Parameter: 152bert.encoder.layer.9.attention.self.key.bias ----- False
Parameter: 153bert.encoder.layer.9.attention.self.value.weight ----- False
Parameter: 154bert.encoder.layer.9.attention.self.value.bias ----- False
Parameter: 155bert.encoder.layer.9.attention.output.dense.weight ----- False
Parameter: 156bert.encoder.layer.9.attention.output.dense.bias ----- False
Parameter: 157bert.encoder.layer.9.attention.output.LayerNorm.weight ----- False
Parameter: 158bert.encoder.layer.9.attention.output.LayerNorm.bias ----- False
Parameter: 159bert.encoder.layer.9.intermediate.dense.weight ----- False
Parameter: 160bert.encoder.layer.9.intermediate.dense.bias ----- False
Parameter: 161bert.encoder.layer.9.output.dense.weight ----- False
Parameter: 162bert.encoder.layer.9.output.dense.bias ----- False
Parameter: 163bert.encoder.layer.9.output.LayerNorm.weight ----- False
Parameter: 164bert.encoder.layer.9.output.LayerNorm.bias ----- False
Parameter: 165bert.encoder.layer.10.attention.self.query.weight ----- False
Parameter: 166bert.encoder.layer.10.attention.self.query.bias ----- False
Parameter: 167bert.encoder.layer.10.attention.self.key.weight ----- False
Parameter: 168bert.encoder.layer.10.attention.self.key.bias ----- False
Parameter: 169bert.encoder.layer.10.attention.self.value.weight ----- False
Parameter: 170bert.encoder.layer.10.attention.self.value.bias ----- False
Parameter: 171bert.encoder.layer.10.attention.output.dense.weight ----- False
Parameter: 172bert.encoder.layer.10.attention.output.dense.bias ----- False
Parameter: 173bert.encoder.layer.10.attention.output.LayerNorm.weight ----- False
Parameter: 174bert.encoder.layer.10.attention.output.LayerNorm.bias ----- False
Parameter: 175bert.encoder.layer.10.intermediate.dense.weight ----- False
Parameter: 176bert.encoder.layer.10.intermediate.dense.bias ----- False
Parameter: 177bert.encoder.layer.10.output.dense.weight ----- False
Parameter: 178bert.encoder.layer.10.output.dense.bias ----- False
Parameter: 179bert.encoder.layer.10.output.LayerNorm.weight ----- False
Parameter: 191bert.encoder.layer.11.intermediate.dense.weight ----- False
Parameter: 192bert.encoder.layer.11.intermediate.dense.bias ----- False
Parameter: 193bert.encoder.layer.11.output.dense.weight ----- False
Parameter: 194bert.encoder.layer.11.output.dense.bias ----- False
Parameter: 195bert.encoder.layer.11.output.LayerNorm.weight ----- False
Parameter: 196bert.encoder.layer.11.output.LayerNorm.bias ----- False
Parameter: 197bert.pooler.dense.weight ----- False
Parameter: 198bert.pooler.dense.bias ----- False
Parameter: 199classifier.weight ----- True
Parameter: 200classifier.bias ----- True

初始化模型，冻结索引为 165 之前的所有层进行训练。只有 10 层 encoder 之后的层可训练。

```

# Load model
model_id = "bert-base-cased"
model = AutoModelForSequenceClassification.from_pretrained(model_id, num_labels=2)
tokenizer = AutoTokenizer.from_pretrained(model_id)

# Encoder block 10 starts at index 165 and
# we freeze everything before that block
for index, (name, param) in enumerate(model.named_parameters()):
    if index < 165:
        #冻结165层之前的层
        param.requires_grad = False

# Trainer which executes the training process
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_train,
    eval_dataset=tokenized_test,
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)
trainer.train()
trainer.evaluate()

```

```

[17] # Check whether the model was correctly updated again
for index, (name, param) in enumerate(model.named_parameters()):
    print(f"Parameter: {index} {name} ----- {param.requires_grad}")

```


Parameter: 143bert.encoder.layer.8.intermediate.dense.weight ----- False
Parameter: 144bert.encoder.layer.8.intermediate.dense.bias ----- False
Parameter: 145bert.encoder.layer.8.output.dense.weight ----- False
Parameter: 146bert.encoder.layer.8.output.dense.bias ----- False
Parameter: 147bert.encoder.layer.8.output.LayerNorm.weight ----- False
Parameter: 148bert.encoder.layer.8.output.LayerNorm.bias ----- False
Parameter: 149bert.encoder.layer.9.attention.self.query.weight ----- False
Parameter: 150bert.encoder.layer.9.attention.self.query.bias ----- False
Parameter: 151bert.encoder.layer.9.attention.self.key.weight ----- False
Parameter: 152bert.encoder.layer.9.attention.self.key.bias ----- False
Parameter: 153bert.encoder.layer.9.attention.self.value.weight ----- False
Parameter: 154bert.encoder.layer.9.attention.self.value.bias ----- False
Parameter: 155bert.encoder.layer.9.attention.output.dense.weight ----- False
Parameter: 156bert.encoder.layer.9.attention.output.dense.bias ----- False
Parameter: 157bert.encoder.layer.9.attention.output.LayerNorm.weight ----- False
Parameter: 158bert.encoder.layer.9.attention.output.LayerNorm.bias ----- False
Parameter: 159bert.encoder.layer.9.intermediate.dense.weight ----- False
Parameter: 160bert.encoder.layer.9.intermediate.dense.bias ----- False
Parameter: 161bert.encoder.layer.9.output.dense.weight ----- False
Parameter: 162bert.encoder.layer.9.output.dense.bias ----- False
Parameter: 163bert.encoder.layer.9.output.LayerNorm.weight ----- False
Parameter: 164bert.encoder.layer.9.output.LayerNorm.bias ----- False
Parameter: 165bert.encoder.layer.10.attention.self.query.weight ----- True
Parameter: 166bert.encoder.layer.10.attention.self.query.bias ----- True
Parameter: 167bert.encoder.layer.10.attention.self.key.weight ----- True
Parameter: 168bert.encoder.layer.10.attention.self.key.bias ----- True
Parameter: 169bert.encoder.layer.10.attention.self.value.weight ----- True
Parameter: 170bert.encoder.layer.10.attention.self.value.bias ----- True
Parameter: 171bert.encoder.layer.10.attention.output.dense.weight ----- True
Parameter: 172bert.encoder.layer.10.attention.output.dense.bias ----- True
Parameter: 173bert.encoder.layer.10.attention.output.LayerNorm.weight ----- True
Parameter: 174bert.encoder.layer.10.attention.output.LayerNorm.bias ----- True
Parameter: 175bert.encoder.layer.10.intermediate.dense.weight ----- True
Parameter: 176bert.encoder.layer.10.intermediate.dense.bias ----- True
Parameter: 177bert.encoder.layer.10.output.dense.weight ----- True
Parameter: 178bert.encoder.layer.10.output.dense.bias ----- True
Parameter: 179bert.encoder.layer.10.output.LayerNorm.weight ----- True

```
[18] trainer.train()

↔ [534/534 00:42, Epoch 1/1]

Step Training Loss
500 0.476100

TrainOutput(global_step=534, training_loss=0.4725381408291363, metrics={'train_loss': 0.4725381408291363, 'epoch': 1.0})

[19] trainer.evaluate()

↔ [67/67 00:03]

{'eval_loss': 0.4100644588470459,
 'eval_f1': 0.8072519083969466,
 'eval_runtime': 3.6901,
 'eval_samples_per_second': 288.881,
 'eval_steps_per_second': 18.157,
 'epoch': 1.0}
```

只开放最后两层 encoder 和分类头，F1 score 就可以达到 0.807。这表明，虽然我们通常希望训练尽可能多的层，但如果没有必要的计算能力，可以减少训练。

为了进一步说明冻结的层数和模型性能的关系，我们测试了迭代冻结编码器块并微调它们的效果。依次冻结前 12 层 encoder，训练模型并得到 F1 score。将 12 个 F1 score 放入 score[] 中用于后续作图。

▼ [BONUS] Freeze blocks

```
scores = []
for index in range(12): #依次冻结前12层，训练12个模型，比较12个模型的性能
    # Re-load model
    model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", num_labels=2)
    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")

    # Freeze encoder blocks 0-index
    for name, param in model.named_parameters():
        if "layer" in name:
            layer_nr = int(name.split("layer")[1].split(".")[1])
            if layer_nr <= index:
                param.requires_grad = False
        else:
            param.requires_grad = True

    # Train
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=tokenized_train,
        eval_dataset=tokenized_test,
        tokenizer=tokenizer,
        data_collator=data_collator,
        compute_metrics=compute_metrics,
    )
    trainer.train()

    # Evaluate
    score = trainer.evaluate()["eval_f1"]
    scores.append(score)
```

这些分数依次为冻结 1 层的 F1，冻结 2 层的 F1……冻结 12 层的 F1。

[21] scores

```
[0. 8504672897196262,
 0. 8541862652869238,
 0. 8528584817244611,
 0. 8466603951081844,
 0. 8388312912346843,
 0. 839134524929445,
 0. 8434864104967198,
 0. 8358490566037736,
 0. 8258801141769743,
 0. 8164435946462715,
 0. 7925270403146509,
 0. 7058823529411765]
```

最后进行汇总作图。

```

import matplotlib.pyplot as plt
import numpy as np

# Create Figure
plt.figure(figsize=(8,4))

# Prepare Data
x = [f"{index}" for index in range(12)]
x[0] = "None"
y = [
    0.8504672897196262,
    0.8541862652869238,
    0.8528584817244611,
    0.8466603951081844,
    0.8388312912346843,
    0.839134524929445,
    0.8434864104967198,
    0.8358490566037736,
    0.8258801141769743,
    0.8164435946462715,
    0.7925270403146509,
    0.7058823529411765
][::-1]

# Stylize Figure
plt.grid(color='#ECEFF1')
plt.axvline(x=5, color="#EC407A", linestyle="--")
plt.title("Effect of Frozen Encoder Blocks on Training Performance")
plt.ylabel("F1-score")
plt.xlabel("Trainable encoder blocks")

```

这段代码使用 matplotlib 库的 `annotate` 函数在图表上添加一个注释（带有箭头）

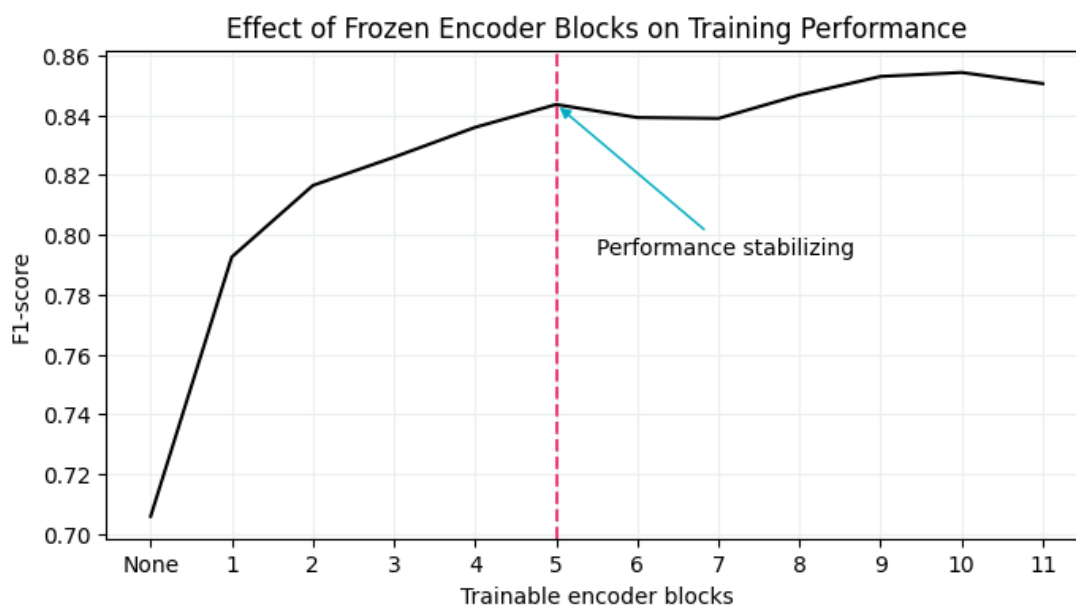
```

# Plot Data
plt.plot(x, y, color="black")

# Additional Annotation
plt.annotate(
    'Performance stabilizing', #性能稳定
    xy=(5, y[5]),
    xytext=(4.5, y[4]-.05),
    arrowprops=dict(
        arrowstyle="->",
        connectionstyle="arc3",
        color="#00ACC1")
)
plt.savefig("multiple_frozen_blocks.png", dpi=300, bbox_inches='tight')

```


仅训练前五个编码器块(红色垂直线)就足以几乎达到训练所有编码器块的性能。



二. Few-Shot Classification 少样本分类

Few-Shot Classification 是监督分类的一种技术，其中分类器仅基于少数标记的示例来学习目标标签。当我们有一个分类任务且手头上没有很多标记数据点时，可以使用这种方法。

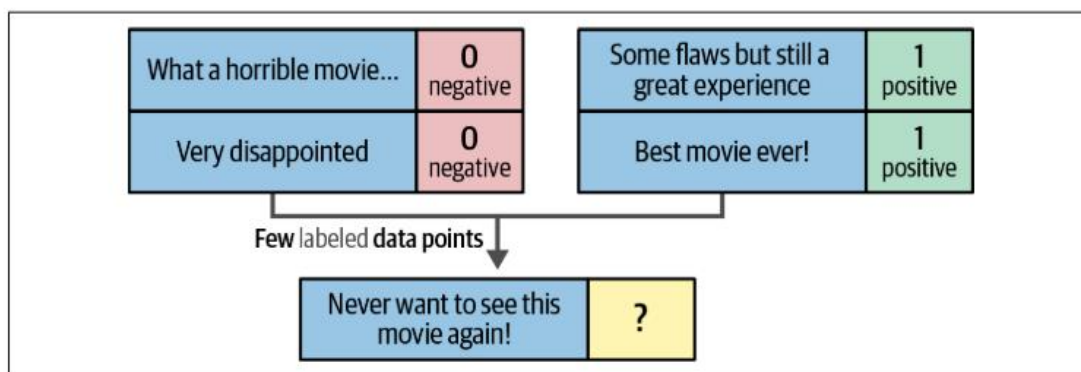


Figure 11-8. In few-shot classification, we only use a few labeled data points to learn from.

1. SetFit: 使用少量训练样本的高效微调

为了执行 Few-Shot Classification，我们使用了一个名为 SetFit 的高效框架。它构建在 sentence transformer 的架构之上，以生成在训练期间更新的高质量文本表示。这个框架只需要几个标记的例子，就可以与我们在上一个例子中探索的大型标记数据集上的 BERT 类模型进行竞争。

SetFit 的基础算法由以下三个步骤组成：①采样训练数据；基于对标记数

据的 in-class（类内）和 out-class（类外）选择，生成 positive(similar)和 negative(dissimilar) 的句子对。②微调嵌入；基于先前生成的训练数据微调预训练的嵌入模型。③训练分类器；在嵌入模型的基础上创建一个分类头，使用之前生成的训练数据进行训练。

句子可以有很多类，两个属于同一类的句子组成的句子对标记为 positive，两个属于不同类的句子组成的句子对标记为 negative。例如，我们有图 11-9 中的训练数据集，它将文本分为两类：关于编程语言的文本和关于宠物的文本。

Text	Class
I write my code in Python	Programming languages
I should practice SQL	Programming languages
My dog is a labrador	Pets
I have a Siamese cat	Pets

Figure 11-9. Data in two classes: text about programming languages and text about pets.

在步骤 1 中，SetFit 通过基于类内和类外选择生成句子对（训练数据），例如，当我们有 16 个关于编程的句子时，我们可以创建 $16 \times (16-1) / 2 = 120$ 个标记为 positive 的句子对。当我们有 2 个关于编程和 2 个关于宠物的句子时，可以生成 4 个标记为 negative 的句子对。

在步骤 2 中，我们使用生成的句子对来微调嵌入模型。这利用了一种称为对比学习的方法来微调预训练的 BERT 模型，正如第 10 章中所提到的。对比学习允许从成对的 similar(positive)和 dissimilar(negative) 句子中学习准确的句子嵌入。微调的目的是，使模型可以创建针对分类任务进行调整的嵌入。类的相关性及其相对意义通过微调嵌入模型被提取到嵌入中。

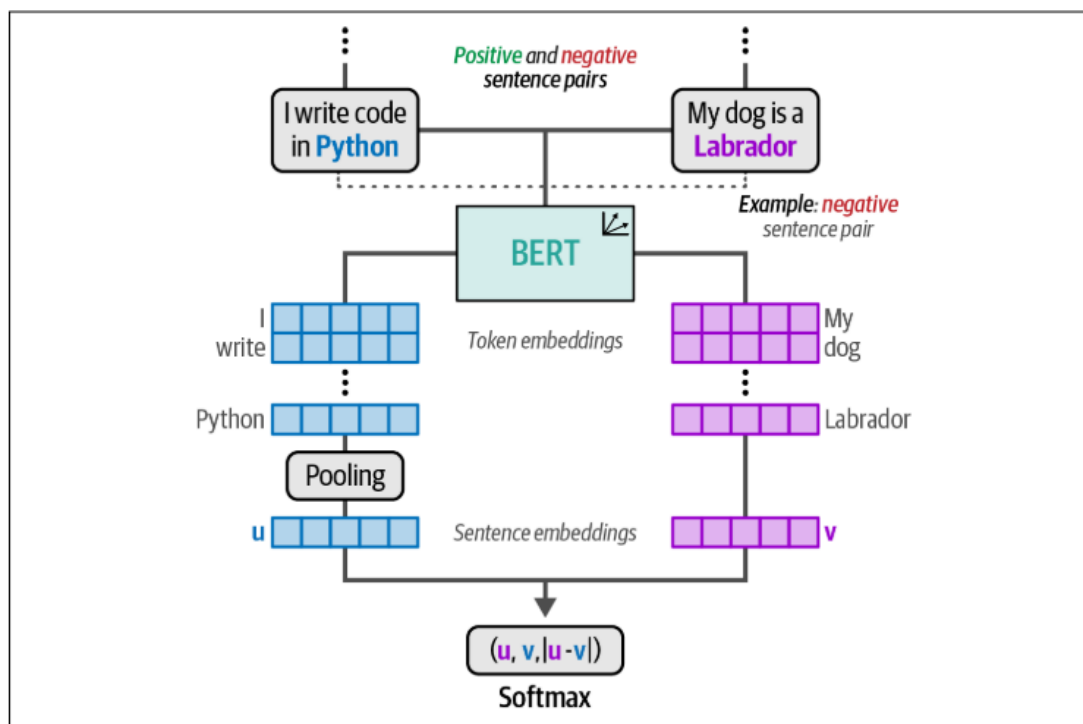


Figure 11-11. Step 2: Fine-tuning a SentenceTransformers model. Using contrastive learning, embeddings are learned from positive and negative sentence pairs.

在步骤 3 中，我们为所有句子生成嵌入，并将其用作分类器的输入。我们可以使用经过微调的 Sentence Transformers model 将我们的句子转换为可以用作特征的嵌入。分类器从我们微调的嵌入中学习，以准确地预测看不见的句子。最后一步如图 11-12 所示。

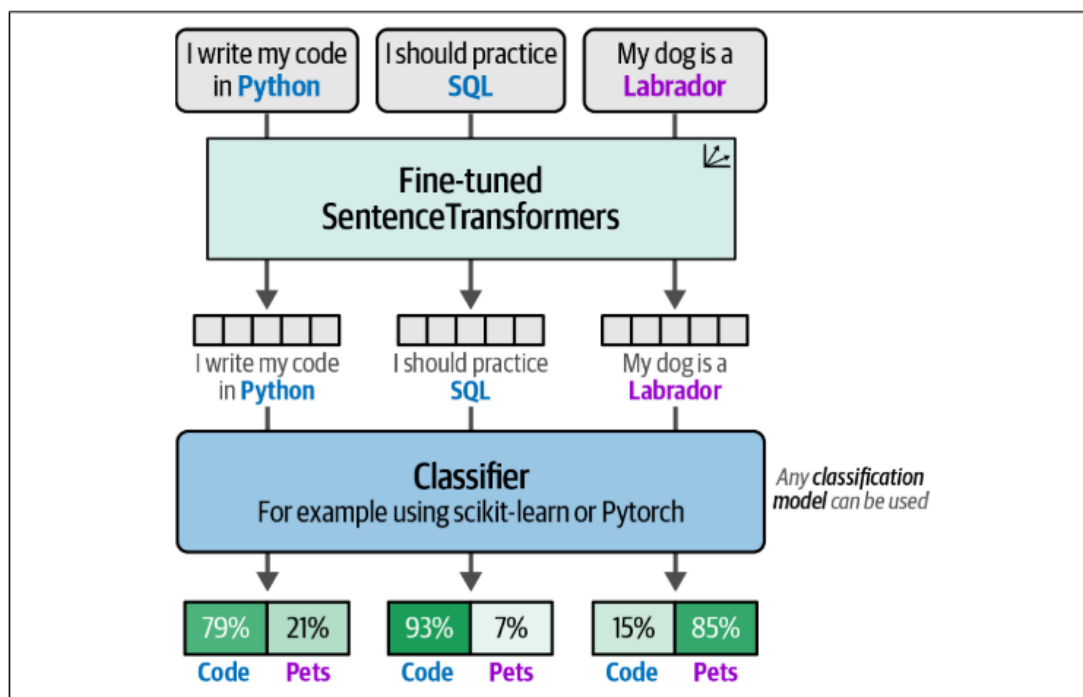


Figure 11-12. Step 3: Training a classifier. The classifier can be any scikit-learn model or a classification head.

Sentence Transformers 是一个专门用于将文本（句子、段落甚至短文档）转换成高维向量（称为“嵌入”或“句子嵌入”）的框架和模型集合。这些向量能够捕捉文本的语义含义，并且语义相似的文本在向量空间中会彼此靠近。与 **BERT** 的区别在于：BERT 直接输出的未经处理的句子表示（如[CLS] token 向量），一个向量代表一个 token；Sentence Transformers 的最终输出向量是经过专门池化和训练的，一个向量代表整个句子，目的和效果完全不同。

当这三个步骤整合在一起，如图 11-13 所示。首先，基于类内和类外选择生成句子对。其次，句子对用于微调预训练的 Sentence Transformer model。第三，句子通过微调过的嵌入模型生成嵌入，用于训练分类器以预测类别。

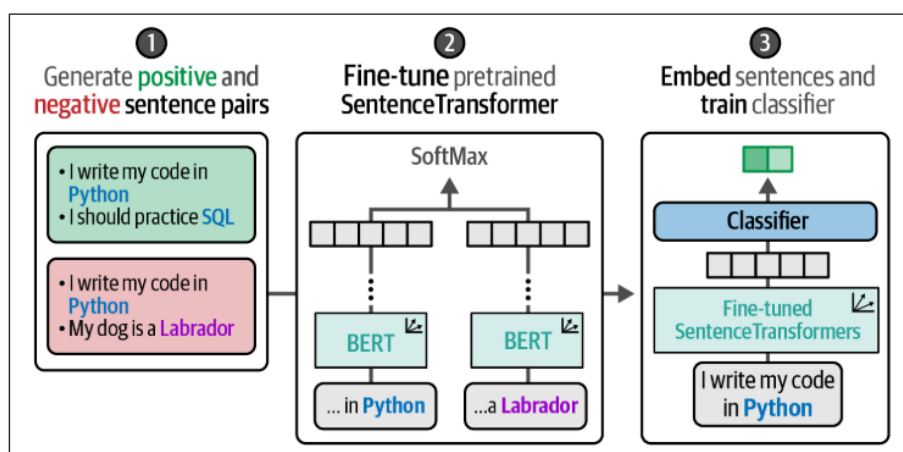


Figure 11-13. The three main steps of SetFit.

2. 针对 Few-Shot Classification 的微调

我们之前用一个包含大约 8,500 条电影评论的数据集来微调分类模型。为了展示 Few-Shot Classification 的潜力，我们仍然使用烂番茄数据集，但是每个类仅采样 16 个示例，并用由此得到的 32 个文档进行训练。

```

v Few-shot Classification

from setfit import sample_dataset

# We simulate a few-shot setting by sampling 16 examples per class
sampled_train_data = sample_dataset(tomatoes["train"], num_samples=16)

[ ] from setfit import SetFitModel

# Load a pre-trained SentenceTransformer model
model = SetFitModel.from_pretrained("sentence-transformers/all-mpnet-base-v2")

README.md: 11.6k/? [00:00<00:00, 1.04MB/s]
model_head.pkl not found on HuggingFace Hub, initialising classification head with random weights.
```

在对数据进行采样后，我们选择一个预训练的 Sentence Transformer model 进行微调。我们将使用“transformers/ all-mpnet-base-v2”作为基础模型，它是 MTEB 排行榜上表现最好的模型之一，该排行榜显示了嵌入模型在各种任务中的性能。

```

[ ] from setfit import TrainingArguments as SetFitTrainingArguments
    from setfit import Trainer as SetFitTrainer

# Define training arguments
args = SetFitTrainingArguments(
    num_epochs=3, # The number of epochs to use for contrastive learning
    num_iterations=20 # The number of text pairs to generate
)
args.eval_strategy = args.evaluation_strategy

# Create trainer
trainer = SetFitTrainer(
    model=model,
    args=args,
    train_dataset=sampled_train_data,
    eval_dataset=test_data,
    metric="f1"
)

Map: 100% 32/32 [00:00<00:00, 1723.99 examples/s]
```

与我们对 Hugging Face transformer 所做的类似，我们可以使用 trainer 来定义和使用相关参数。例如，我们将 num_epochs 设置为 3，以便在三个 epoch

中执行对比学习。然后调用 train 启动训练循环。

```
# Training loop
trainer.train()

**** Running training ****
Num unique pairs = 1280
Batch size = 16
Num epochs = 3

/usr/local/lib/python3.11/dist-packages/notebook/utils.py:280: DeprecationWarning: distutils Version
return LooseVersion(v) >= LooseVersion(check)
wandb: Logging into wandb.ai. (Learn how to deploy a W&B server locally: https://wandb.me/wandb-serve)
wandb: You can find your API key in your browser here: https://wandb.ai/authorize?ref=models
wandb: Paste an API key from your profile and hit enter: . . . . .
wandb: WARNING If you're specifying your api key in code, ensure this code is not shared publicly.
wandb: WARNING Consider setting the WANDB_API_KEY environment variable, or running `wandb login` from
wandb: No netrc file found, creating one.
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
wandb: Currently logged in as: 1449185730 (1449185730-sun-yat-sen-university) to https://api.wandb.ai
/usr/local/lib/python3.11/dist-packages/wandb/analytics/sentry.py:263: DeprecationWarning: The `Scope
self.scope.user = {"email": email}
Tracking run with wandb version 0.21.1
Run data is saved locally in /content/wandb/run-20250819_111021-7hk8l3kp
Syncing run wild-deluge-4 to Weights & Biases (docs)
View project at https://wandb.ai/1449185730-sun-yat-sen-university/sentence-transformers
View run at https://wandb.ai/1449185730-sun-yat-sen-university/sentence-transformers/runs/7hk8l3kp
[240/240 01:27, Epoch 3/3]

Step Training Loss Validation Loss

/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:451: DeprecationWarning: sc
opt_res = optimize.minimize(
```

请注意，输出中提到为微调 SentenceTransformer 模型生成了 1,280 个句子对。我们只有 32 个句子，positive 类有 16 个句子，negative 类有 16 个句子，组成的句子对标签为 “similar” 有 $C(16, 2) * 2 = 240$ 。但 SetFit 不会直接使用原始的 240 个正样本对。为了增加数据的多样性，它对每一个训练样本都会进行多次数据增强。

SetFit 的默认**数据增强器**会为每个原始句子生成 num_duplicates 个增强后的句子，这个参数默认值为 4，这意味着 32 个原始样本，每个都变成了 4 个不同的版本。现在总共有 $32 * 4 = 128$ 个句子。Positive 类有 64 个句子，negative 类有 64 个句子。对于 positive 类，可以组成 2016 个正样本句子对，Negative 类也可以组成 2016 个正样本句子对。那么总正样本对数为 4032。

但是使用全部组合，这个数字太大了。因此，SetFit 采用了一种抽样策略。它默认会从所有可能的正样本对中为每个类别最多采样 640 个 pair。这样一来最终的正样本对总数为 1280。

SetFit 默认使用的是一种叫做 SentenceTransformersParaphrase

Augmenter 的增强器。它的核心思想是：对原始句子进行 paraphrase（释义），生成意思相同但表达方式不同的新句子。它主要通过以下技术来实现：

同义词替换：用同义词替换句子中的某些词。

句式转换：例如，主动句变被动句。

插入/删除：插入或删除一些不影响核心含义的副词、形容词等。

调用 `evaluate` 对模型的性能进行评估。

```
[ ] # Evaluate the model on our test data
    trainer.evaluate()

**** Running evaluation ****
Downloading builder script: 6.79k/? [00:00<00:00, 447kB/s]
{'f1': 0.8454011741682974}
```

随后可以训练逻辑回归分类头用于数据分类。

```
model.model_head

LogisticRegression
LogisticRegression()
```

三. 用掩码语言建模对预训练模型继续预训练

在上一节中，我们利用了一个预训练的模型，并对其进行了微调以执行分类。这个过程归纳为两个步骤：首先预训练一个模型（这已经为我们完成了），然后针对特定任务对其进行微调。我们在图 11-14 中说明了这个过程。

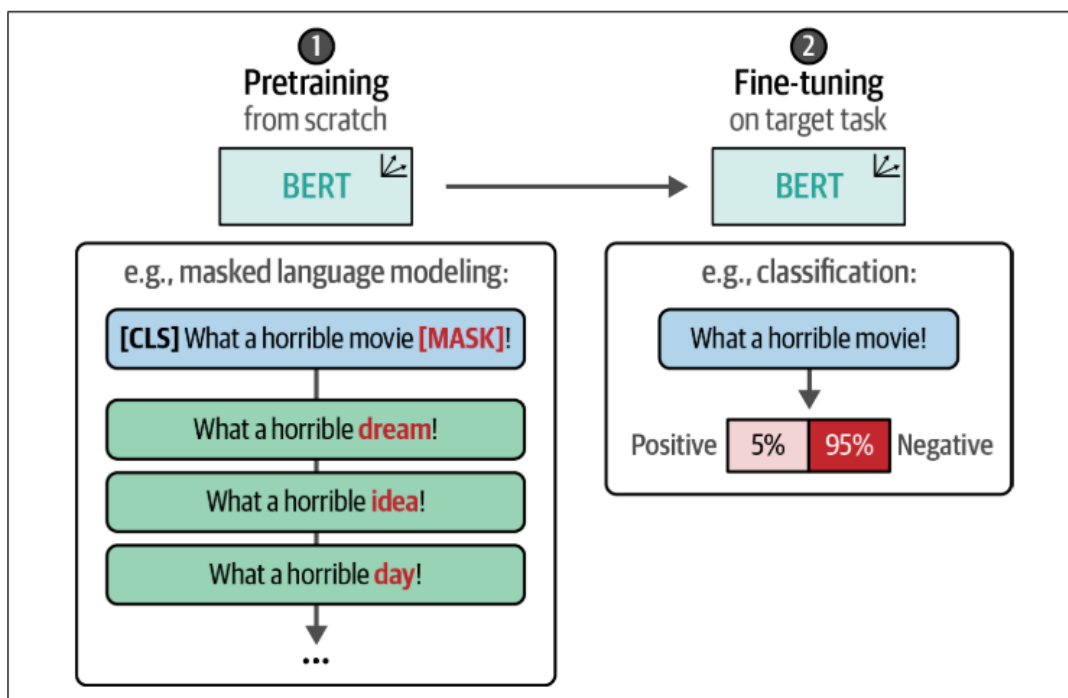


Figure 11-14. To fine-tune the model on a target task—for example, classification—we either start with pretraining a BERT model or use a pretrained one.

这种两步法通常在许多应用中使用。但当面对特定领域的的数据时，它存在局限性。预训练模型通常是在**非常通用的数据**上进行训练的，比如维基百科页面，可能无法针对特定领域的词汇进行优化。

我们可以在这两步之间插入另一步，继续对一个已经预训练好的 BERT 模型进行预训练。换句话说，我们可以简单地继续使用掩码语言建模（MLM）来训练 BERT 模型，但是使用的是特定领域的数据。这就好比从一个通用的 BERT 模型转变为一个专门用于医学领域的 BioBERT 模型，再转变为一个经过微调的、用于对药物进行分类的 BioBERT 模型。

下载模型和分词器：

```
▼ MLM

[ ] from transformers import AutoTokenizer, AutoModelForMaskedLM

# Load model for Masked Language Modeling (MLM)
model = AutoModelForMaskedLM.from_pretrained("bert-base-cased")
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

我们需要对原始句子进行分词化，并删除标签，因为这是无监督学习：

```
def preprocess_function(examples): #分词化
    return tokenizer(examples["text"], truncation=True)

# Tokenize data
tokenized_train = train_data.map(preprocess_function, batched=True)
tokenized_train = tokenized_train.remove_columns("label") #删除标签达到无监督的目的
tokenized_test = test_data.map(preprocess_function, batched=True)
tokenized_test = tokenized_test.remove_columns("label")

Map: 100% ██████████ 8530/8530 [00:01<00:00, 9177.28 examples/s]
Map: 100% ██████████ 1066/1066 [00:00<00:00, 10096.99 examples/s]
```

通常使用两种方法进行掩码: token masking 和 whole-word masking。使用 token masking, 我们随机遮掩一个句子中 15% 的 token, 因此可能会出现单词的一部分被遮掩的情况, 而 whole-word masking 是遮掩整个单词。

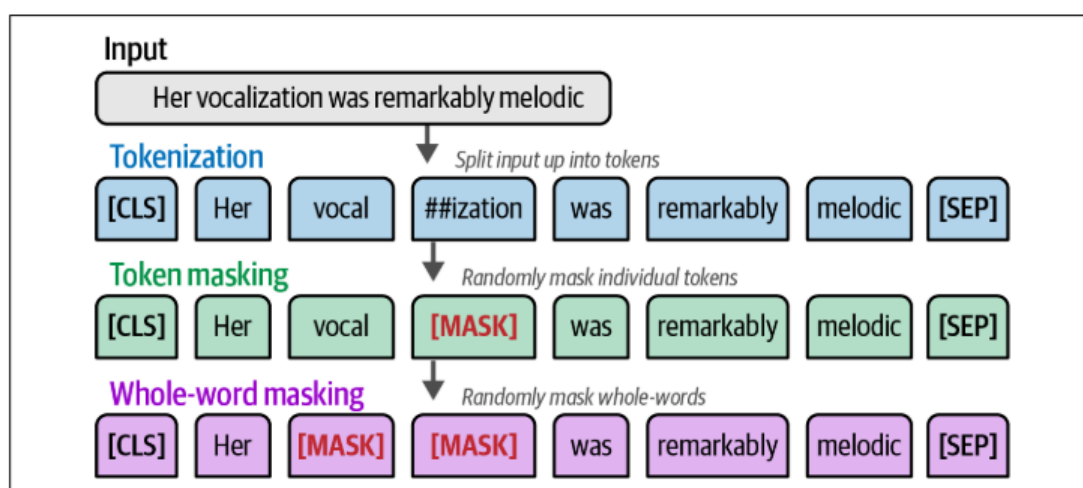


Figure 11-17. Different methods for randomly masking tokens.

一般来说, 预测整词往往比预测符号复杂, 这使得模型的性能更好, 因为它需要在训练过程中学习更准确和精确的表示。然而, 它往往需要更多的时间才能收敛。在本例中, 我们将使用令牌掩码, 使用 `DataCollatorForLanguageModel` 以加快收敛速度。若要使用全字掩码, 方法是将 `DataCollatorForLanguageModel` 替换为 `DataCollatorForWholeWord`。最后, 我们将 token 在给定句子中被屏蔽的概率设置为 15% (`mlm_probability`):

```
[ ] from transformers import DataCollatorForLanguageModeling

# Masking Tokens
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm=True,
    mlm_probability=0.15
)
```

创建 trainer 用于训练，并指定参数：

```
▶ # Training arguments for parameter tuning
training_args = TrainingArguments(
    "model",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=10,
    weight_decay=0.01,
    save_strategy="epoch",
    report_to="none"
)

# Initialize Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_train,
    eval_dataset=tokenized_test,
    tokenizer=tokenizer,
    data_collator=data_collator
)
```

保存预训练的 Tokenizer，在模型训练完毕后保存模型。

PS：在训练前保存 tokenizer 是一个防御性编程策略，它保证了实验的严谨性、结果的可复现性，

```
# Save pre-trained tokenizer
tokenizer.save_pretrained("mlm")

# Train model
trainer.train()

# Save updated model
model.save_pretrained("mlm")
```

[5340/5340 23:17, Epoch 10/10]

Step	Training Loss
500	2.604500
1000	2.371400
1500	2.306700
2000	2.191000
2500	2.150500
3000	2.089400
3500	2.053600
4000	1.993700
4500	1.987100
5000	1.955500

我们将继续预训练完毕后的模型保存为“mlm”，为了评估其性能，通常需要在各种任务上对模型进行微调。但是，我们可以运行一些掩码任务来观察它是否从继续预训练中有所学习。

我们将通过加载继续预训练之前的原始预训练模型来进行比较。使用句子 “What a horrible [MASK]!”，模型将预测哪个词会出现在 “[MASK]” 的位置。

```
[ ] from transformers import pipeline

# Load and create predictions
mask_filler = pipeline("fill-mask", model="bert-base-cased")
preds = mask_filler("What a horrible [MASK]!")

# Print results
for pred in preds:
    print(f">>> {pred['sequence']}")
```

```
⇒ Some weights of the model checkpoint at bert-base-cased were not used
- This IS expected if you are initializing BertForMaskedLM from the checkpoint
- This IS NOT expected if you are initializing BertForMaskedLM from the
  weights directory
Device set to use cuda:0
>>> What a horrible idea!
>>> What a horrible dream!
>>> What a horrible thing!
>>> What a horrible day!
>>> What a horrible thought!
```

```
[ ] # Load and create predictions
mask_filler = pipeline("fill-mask", model="mlm")
preds = mask_filler("What a horrible [MASK]!")

# Print results
for pred in preds:
    print(f">>> {pred['sequence']}")
```

```
⇒ Device set to use cuda:0
>>> What a horrible movie!
>>> What a horrible film!
>>> What a horrible mess!
>>> What a horrible comedy!
>>> What a horrible story!
```

与预先训练的模型相比，该模型输出的结果更偏向于我们向其提供的数据，即电影领域的词出现更频繁。

下一步将根据我们在本章开始时所做的分类任务来微调此模型。只需按如下方式加载模型即可：

```
[ ] from transformers import AutoModelForSequenceClassification

# Fine-tune for classification
model = AutoModelForSequenceClassification.from_pretrained("mlm", num_labels=2)
tokenizer = AutoTokenizer.from_pretrained("mlm")
```

这是我们之前的代码，可以看到仅替换了模型的名称：

✓ Supervised Classification

✓ HuggingFace Trainer

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification

# Load Model and Tokenizer
model_id = "bert-base-cased"
model = AutoModelForSequenceClassification.from_pretrained(model_id, num_labels=2)
tokenizer = AutoTokenizer.from_pretrained(model_id)
```

接下来继续按步骤进行 tokenize data, define metrics, train model, evaluate results.

四. Named-Entity Recognition 命名实体识别

在这一节中，我们将深入研究专门针对 NER(命名实体识别)微调预训练的 BERT 模型的过程。NER 是自然语言处理（NLP）中的一项核心任务，属于信息提取的一个子领域。它的主要目标是：从非结构化的文本中识别出属于特定类别的命名实体，并将其分类到预定义的类别中。可以把它想象成一种文本高亮标记的高级形式：它不仅找到了文本中重要的“名字”，还告诉你这个名字属于哪种类型。主要用于“识别”和“分类”。常见的实体类别包括：人名、地点、组织、时间、货币、百分比和地缘政治实体（比如中国、美国，通常作为“地点”的子集）。

当存在敏感数据时，这对于去身份化和匿名化任务特别有帮助。

NER 与我们在本章开头探索的分类有相似之处，区别在于数据预处理和分类，我们关注的是对单个单词而不是整个句子。

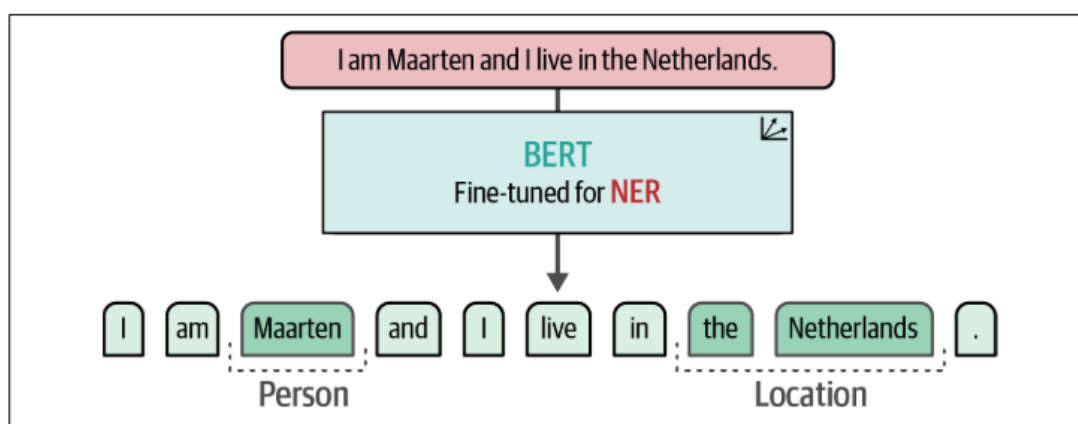


Figure 11-18. Fine-tuning a BERT model for NER allows for the detection of named entities, such as people or locations.

对预训练的 BERT 模型进行微调，其架构与我们之前在文档分类中观察到的模式类似。然而，分类方法发生了根本性的转变。模型不再依赖于 token 嵌入的聚合或池化操作，而是转为对序列中的每个独立词元进行预测。我们的 token 级分类任务并非对整个单词进行分类，而是对共同构成这些单词的词元进行分类。图 11-19 直观展示了这种 token 级别分类的实现方式。

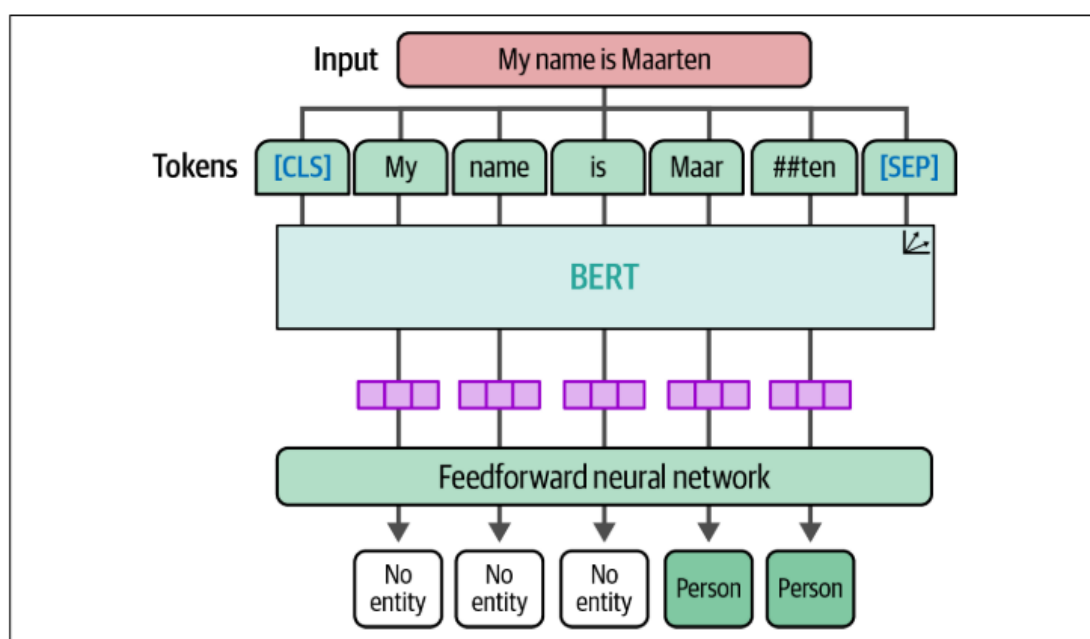


Figure 11-19. During the fine-tuning process of a BERT model, individual tokens are classified instead of words or entire documents.

Named Entity Recognition

Here are a number of interesting datasets you can also explore for NER:

- `tner/mit_movie_trivia`
- `tner/mit_restaurant`
- `wnut_17`
- `conll2003`

```
from transformers import AutoModelForTokenClassification, AutoTokenizer
from transformers import DataCollatorWithPadding
from transformers import TrainingArguments, Trainer
import numpy as np
```

```
[ ] !pip install datasets
```

下载数据集 `conll2003`。我们使用的是英文版的 CoNLL-2003 数据集，该数据集包含几种不同类型的命名实体(人名、组织、地区、杂项和无实体)，并具有大约

14,000 个训练样本。

```
[ ] # The CoNLL-2003 dataset for NER
dataset = load_dataset("conll2003", trust_remote_code=True)

Using the latest cached version of the module from /root/.cache/huggingface/modules/datasets_modules/datasets/conll2003
WARNING:datasets.load:Using the latest cached version of the module from /root/.cache/huggingface/modules/datasets_modules/datasets/conll2003
Downloading data: 100% ██████████ 983k/983k [00:00<00:00, 5.79MB/s]
Generating train split: 100% ██████████ 14041/14041 [00:04<00:00, 3219.88 examples/s]
Generating validation split: 100% ██████████ 3250/3250 [00:00<00:00, 3507.25 examples/s]
Generating test split: 100% ██████████ 3453/3453 [00:00<00:00, 3732.82 examples/s]
```

让我们通过一个例子来检查数据的结构：

```
example = dataset["train"][848]
example

{'id': '848',
 'tokens': ['Dean',
 'Palmer',
 'hit',
 'his',
 '30th',
 'homer',
 'for',
 'the',
 'Rangers',
 '.'],
 'pos_tags': [22, 22, 38, 29, 16, 21, 15, 12, 23, 7],
 'chunk_tags': [11, 12, 21, 11, 12, 12, 13, 11, 12, 0],
 'ner_tags': [1, 2, 0, 0, 0, 0, 0, 0, 3, 0]}
```

这个数据集为我们提供了句子中每个单词的标签。这些标签可以在“ner_tags” key 中找到，该键指的是以下可能的实体：PER 代表 person，ORG 代表 organization，LOC 代表 location，MISC 代表 miscellaneous entities，0 代表 no entity。每一类实体都分为 B 和 I，这个前缀用来区分是否是同一类短语的一部分。例如 Dean Palmer，这是一个人名，Dean 属于 B-PER，Palmer 属于 I-PER，表示他们属于一个人名的两个部分，不是独立的实体，是一个人而不是两个人。

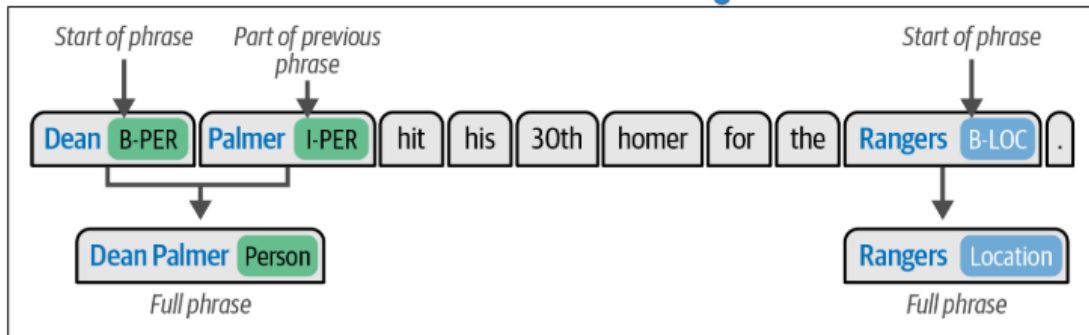


Figure 11-20. By indicating the start and end of the phrase with the same entity, we can recognize entities of entire phrases.

```

label2id = {
    'O': 0, 'B-PER': 1, 'I-PER': 2, 'B-ORG': 3, 'I-ORG': 4,
    'B-LOC': 5, 'I-LOC': 6, 'B-MISC': 7, 'I-MISC': 8
}
id2label = {index: label for label, index in label2id.items()}
label2id

```

```

{'O': 0,
 'B-PER': 1,
 'I-PER': 2,
 'B-ORG': 3,
 'I-ORG': 4,
 'B-LOC': 5,
 'I-LOC': 6,
 'B-MISC': 7,
 'I-MISC': 8}

```

数据集中的数据经过了预处理并被分割成单个单词，但是还不是 token，为此我们需要使用 tokenizer 进一步做分词处理。

```

from transformers import AutoModelForTokenClassification

# Load tokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")

# Load model
model = AutoModelForTokenClassification.from_pretrained(
    "bert-base-cased",
    num_labels=len(id2label),
    id2label=id2label,
    label2id=label2id
)

```

```
# Split individual tokens into sub-tokens
token_ids = tokenizer(example["tokens"], is_split_into_words=True)["input_ids"]
sub_tokens = tokenizer.convert_ids_to_tokens(token_ids)
sub_tokens
```

```
['[CLS]',
 'Dean',
 'Palmer',
 'hit',
 'his',
 '30th',
 'home',
 '##r',
 'for',
 'the',
 'Rangers',
 ',',
 ']',
 '[SEP]']
```

注意单词“homer”分成了“home”和“##r”，但是原来的标记不会改变，即 home 和##r 都是 B 开头的 label，我们需要创建一个函数对分词之后的 token 和 label 重新对齐，如果一个实体单词被分成了多个 token，那么第一个 token 的标签为 B 开头，其余的 token 都应该是 I 开头。

例如单词‘Maarten’，它有标签 B-PER 来表示这是一个人。如果我们将该单词传递给 tokenizer，它会将该单词拆分为标记‘Ma’、‘##arte’和‘##n’。我们不能对所有 token 使用 B-PER 实体，因为这将表明这三个 token 都是独立的人。每当实体被拆分成 token 时，第一个 token 应该有 B(表示开始)，后面的应该是 I(表示内部)。因此，‘Ma’会让 B-PER 表示短语的开始，而‘##arte’和‘##n’是 I-PER，表示它们属于某个短语。此对齐过程如图 11-21 所示。

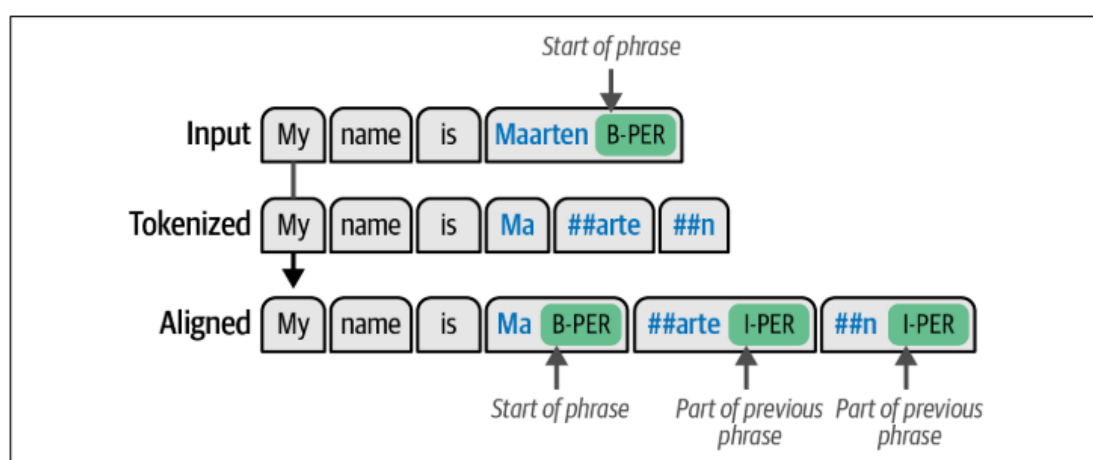


Figure 11-21. The alignment process of labeling tokenized input.

```
def align_labels(examples):
    token_ids = tokenizer(examples["tokens"], truncation=True, is_split_into_words=True)
    labels = examples["ner_tags"]

    updated_labels = []
    for index, label in enumerate(labels):

        # Map tokens to their respective word
        word_ids = token_ids.word_ids(batch_index=index)
        previous_word_idx = None
        label_ids = []
        for word_idx in word_ids:

            # The start of a new word
            if word_idx != previous_word_idx:

                previous_word_idx = word_idx
                updated_label = -100 if word_idx is None else label[word_idx]
                label_ids.append(updated_label)

            # Special token is -100
            elif word_idx is None:
                label_ids.append(-100)

            # If the label is B-XXX we change it to I-XXX
            else:
                updated_label = label[word_idx]
                if updated_label % 2 == 1:
                    updated_label += 1
                label_ids.append(updated_label)

        updated_labels.append(label_ids)

    token_ids["labels"] = updated_labels
    return token_ids

tokenized = dataset.map(align_labels, batched=True)
```

Map: 100% 14041/14041 [00:03<00:00, 3087.04 examples/s]
Map: 100% 3250/3250 [00:01<00:00, 2987.75 examples/s]
Map: 100% 3453/3453 [00:00<00:00, 4080.69 examples/s]

查看我们的示例，注意[CLS]和[SEP]被添加了其他标签(-100)：

```
# Difference between original and updated labels
print(f"Original: {example['ner_tags']}")
print(f"Updated: {tokenized['train'][848]['labels']}")
```

Original: [1, 2, 0, 0, 0, 0, 0, 0, 3, 0]
Updated: [-100, 1, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, -100]

既然我们已经 tokenizer 并对齐了标签，我们就可以开始考虑定义我们的评估指标了。这也与我们以前看到的不同。我们现在每个句子有多个 token，每个 token 都有自己的标签，而不是每个句子一个预测。我们将通过 Hugging Face 使

用 evaluate 包来创建一个 COMPUTE_METRICS 函数，该函数可以让我们在令牌级别评估性能：

```
import evaluate

# Load sequential evaluation
sequeval = evaluate.load("sequeval")

def compute_metrics(eval_pred):
    # Create predictions
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)

    true_predictions = []
    true_labels = []

    # Document-level iteration
    for prediction, label in zip(predictions, labels):

        # token-level iteration
        for token_prediction, token_label in zip(prediction, label):

            # We ignore special tokens
            if token_label != -100:
                true_predictions.append([id2label[token_prediction]])
                true_labels.append([id2label[token_label]])

    results = sequeval.compute(predictions=true_predictions, references=true_labels)
    return {"f1": results["overall_f1"]}
```

Downloading builder script: 6.34k/? [00:00<00:00, 335kB/s]


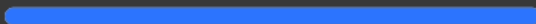
我们还需要一个可以在 token 级别工作的分类器 DataCollatorForTokenClassification：

```
[ ] from transformers import DataCollatorForTokenClassification

# Token-classification Data Collator
data_collator = DataCollatorForTokenClassification(tokenizer=tokenizer)
```

```
# Training arguments for parameter tuning
training_args = TrainingArguments(
    "model",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=1,
    weight_decay=0.01,
    save_strategy="epoch",
    report_to="none"
)


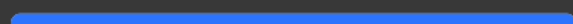
# Initialize Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized["train"],
    eval_dataset=tokenized["test"],
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)
trainer.train()
```

 /tmp/ipython-input-1046598781.py:14: FutureWarning: `tokenizer` is deprecated
trainer = Trainer(
 [878/878 02:27, Epoch 1/1]

Step	Training Loss
500	0.229300

TrainOutput(global_step=878, training_loss=0.1665827809813897, metrics={'train_loss': 0.1665827809813897, 'epoch': 1.0})

```
[ ] # Evaluate the model on our test data
trainer.evaluate()
```

  [216/216 00:07]

```
{'eval_loss': 0.14401467144489288,
 'eval_f1': 0.9010373576019856,
 'eval_runtime': 10.3085,
 'eval_samples_per_second': 334.967,
 'eval_steps_per_second': 20.954,
 'epoch': 1.0}
```

最后，让我们保存模型并在管道 pipeline 中使用它进行推理。这允许我们检查某些数据，以便我们可以手动检查推理过程中发生的情况，以及我们是否对输出满意：

```

▶ from transformers import pipeline

# Save our fine-tuned model
trainer.save_model("ner_model")

# Run inference on the fine-tuned model
token_classifier = pipeline(
    "token-classification",
    model="ner_model",
)
token_classifier("My name is Maarten.")

```

```

⇒ Device set to use cuda:0
[{'entity': 'B-PER',
  'score': np.float32(0.98402),
  'index': 4,
  'word': 'Ma',
  'start': 11,
  'end': 13},
 {'entity': 'I-PER',
  'score': np.float32(0.9593786),
  'index': 5,
  'word': '##arte',
  'start': 13,
  'end': 17},
 {'entity': 'I-PER',
  'score': np.float32(0.95604),
  'index': 6,
  'word': '##n',
  'start': 17,
  'end': 18}]

```