命名实体识别及Trainer微调预训练模型

什么是NER

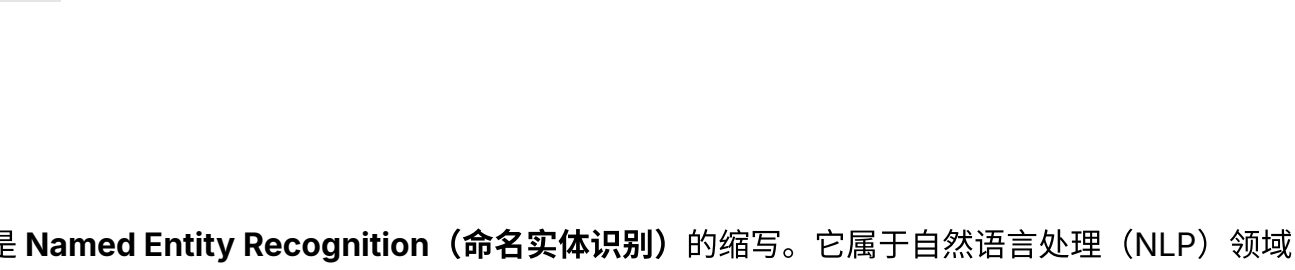
- 首先，NER 是 **Named Entity Recognition（命名实体识别）**的缩写。它属于自然语言处理（NLP）领域中的一个重要任务。

  语言中的实体(Entity)，通常都是指一些名词。例如：人名、地名、组织名、食物名等等。所以实体的类型也说可以多种多样，五花八门的。

  我们往往会根据具体工作业务的需要，而去识别不同的实体。例如：医生在日常工作中，会对**‘发烧’，‘红肿’，‘过敏’**等这类病理症状的名词实体关注较多；而学生在学习中，会对**‘成绩’，‘知识点’，‘测试’，‘科目’**等这类学习相关的实体关注度高。

  由此，命名实体识别的目的也就反映出，我们对于不同人群，在自然语言中分析并抽取与他们关联业务名词的过程。而这个抽取的过程，也是NLP领域中最常见和最基础的任务之一。

  另：基于**深度学习的命名实体识别**，也称为**基于深度学习的信息抽取技术**(Deep Learning for Information Extraction)。



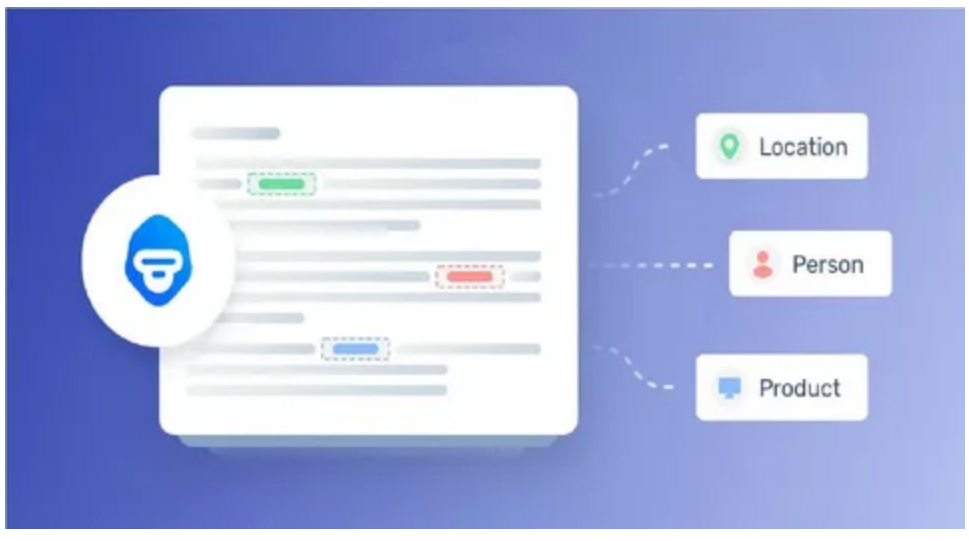
在上面这个例子中，“乔布斯” 是人物实体，“苹果公司” 是组织实体，“iPhone” 是产品实体。

简单来说，NER 就是给文本中的各种 “有意义的内容” 打上标签，告诉我们它们属于哪一类实体。

实体类别

由于实体类别本身也是个类别型名词，所以不同的行业领域，使用的实体类别名也各不相同。目前还没有一个统一的业内标准。但好在市场上已有的产品和数据集，已经在它们的说明文档中包含了实体类别的字典，所以我们也可以参照这些字典搬来照用，或者是进行自定义的类别设定。

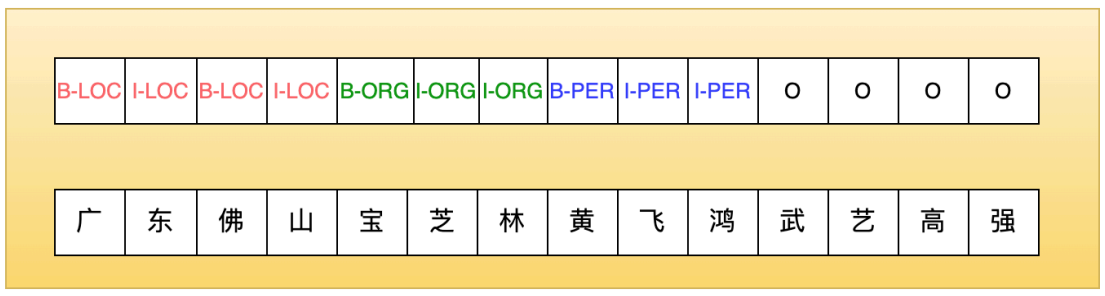
常用的五类实体类型包括：**人名(PER),标题(TTL),机构(ORG)，行政单位(GPE)，地理位置(LOC)**。



实体标注

- 标注方式可以分为：**序列标注、指针标注、全局指针标注**

序列标注



- 序列标注方法包括: ***BMES***、***BIO***、***BIOS***

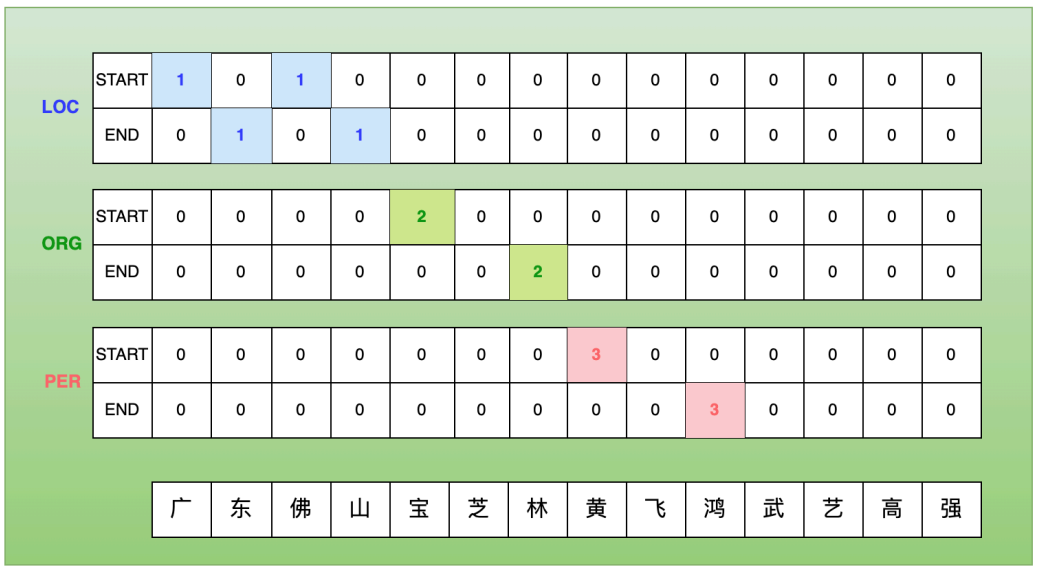
  BMEO 四位序列标注法 **B(egin)**表示一个词的词首位值，**M(iddel)**表示一个词的中间位置，**E(nd)**表示一个词的末尾位置，**O(outside)**表示不是实体。

  BIO 三位标注 (**B-begin，I-inside，O-outside**) IBO2格式的标注会以 **B**-X 代表实体X的开头， **I**-X代表实体的内容 **O** 代表不属于任何类型。

  BIOES **(B-begin，I-inside，O-outside，E-end，S-single) B** 表示开始，**I**表示内容， **O**表示非实体 ，**E**实体尾部，**S**表示该词本身就是一个实体。

特点：应用广泛，每个标签对应一个类别。可通过多类别标注解决嵌套实体识别，但效果不好。

指针标注（范围标注）



特点：标注方式关注于类别和实体边界。分别使用start和end向量来记录边界位置。对嵌套实体标注支持不好。

全局指针标注

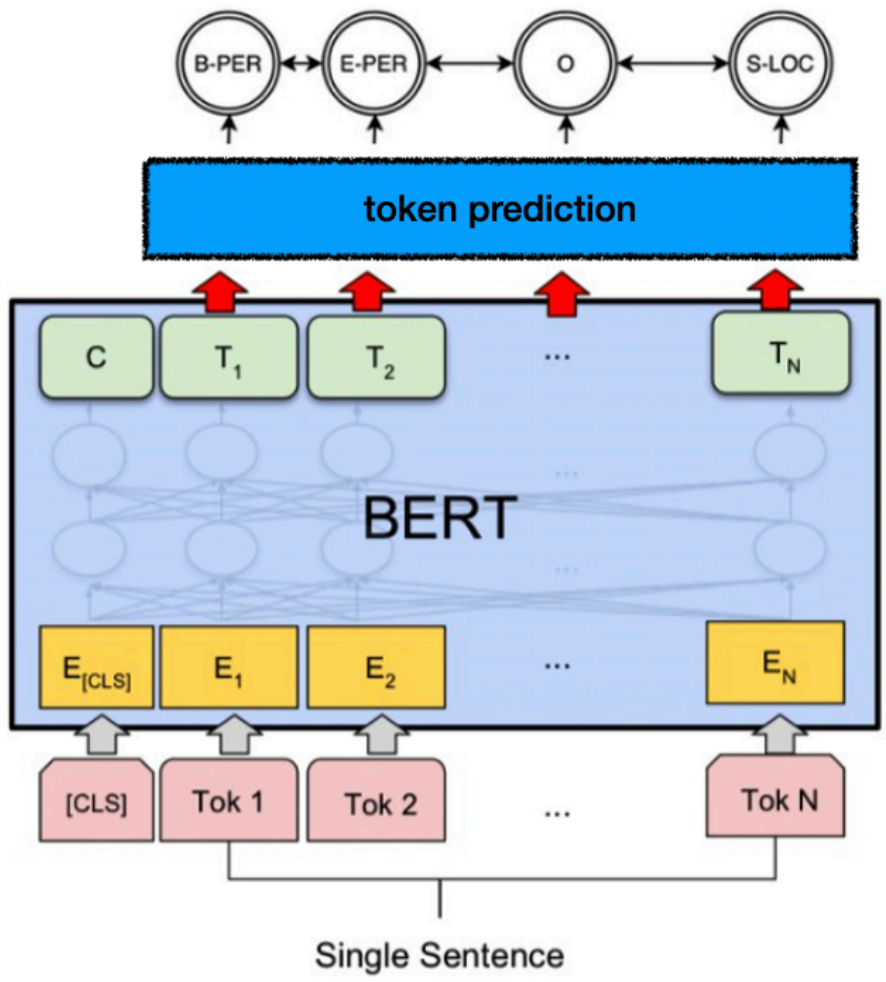
	广	东	佛	山	宝	芝	林	黄	飞	鸿	武	艺	高	强
广	0	1	0	0	0	0	0	0	0	0	0	0	0	0
东		0	0	0	0	0	0	0	0	0	0	0	0	0
佛			0	1	0	0	0	0	0	0	0	0	0	0
山				0	0	0	0	0	0	0	0	0	0	0
宝					0	0	2	0	0	0	0	0	0	0
芝						0	0	0	0	0	0	0	0	0
林							0	0	0	0	0	0	0	0
黄								0	0	3	0	0	0	0
飞									0	0	0	0	0	0
鸿										0	0	0	0	0
武											0	0	0	0
艺												0	0	0
高													0	0
强														0

特点：全局性标注，适合于嵌套和非嵌套实体的标注和训练。

命名实体识别(Named Entity Recognition)

通过大量标注了实体类别标签的语料样本，来训练一个模型。从而让模型可以在输入没有标签的文本语料时，可以快速准确的识别出其中不同类型的实体(名词)。

基于Bert的NER



- 网络输入层，通过Bert关联的Tokenizer将文本转换为相应的Bert输入
- Bert输出末端对接Linear推理层（token prediction）
- 通过交叉熵损失函数来对模型末端实体类别标签的预测顺序进行优化

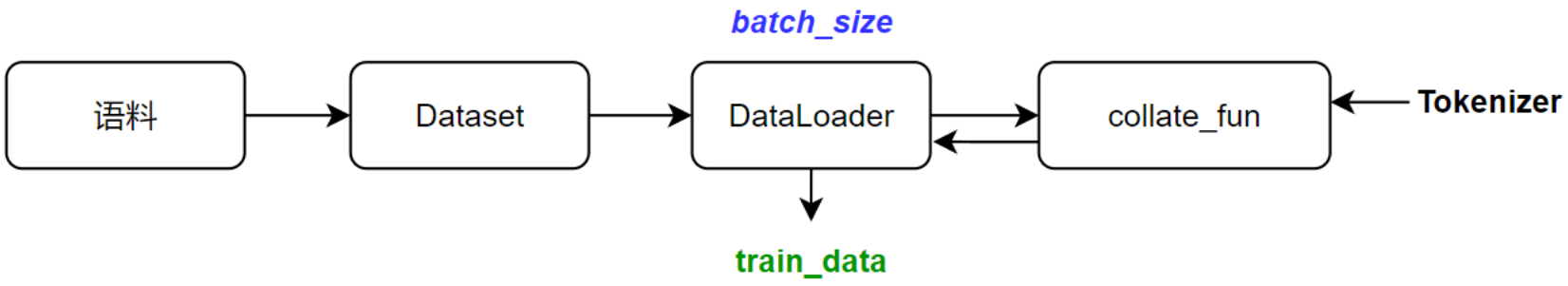
1. 语料处理

构建自定义的训练数据，需要先创建`Dataset`对象。PyTorch也提供了`torch.utils.data.TensorDataset`类，它可以把张量Tensor封装成Dataset。

由于我们处理的是中文文本，那就只能先通过Tokenizer转换后，再把值转换为张量。

```
ckpt = 'google/bert-base-chinese'
tokenizer = AutoTokenizer.from_pretrained(ckpt)

train_data = tokenizer(all_corpus, return_tensors='pt')
```

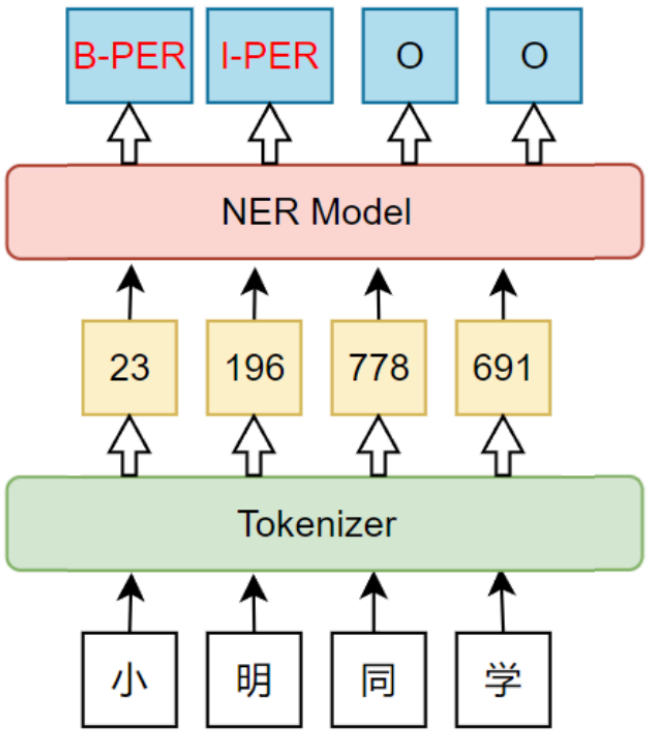
这种方式虽然可以把所有的语料都转换为bert模型输入需要的张量。但对于大数据集来说，占用的资源也会非常多。所以我们采用的策略是：



把语料封装到自定义Dataset，再通过DataLoader的参数collate_fun，在调用自定义的函数中实现批次语料的转换。这样资源的占用只限定在Dataset，而模型训练的语料是通过方法动态转换生成的。

```
def build_dataloader(corpus, tags_t2i, batch_size=4, shuffle=True):

    tokenizer = AutoTokenizer.from_pretrained(
        '../pretrained_models/bert-base-chinese')

    def batch_data_proc(batch_data):
        tokens, tags = [],[]
        for tks, tgs in batch_data:
            tokens.append(tks)
            tags.append(torch.tensor([tags_t2i[t] for t in tgs]))

        # 统一转换收集tokens集合
        data_input = tokenizer(tokens, padding=True, truncation=True,
                               return_tensors='pt',
                               is_split_into_words=True,
                               add_special_tokens=False)

        return data_input, pad_sequence(tags, batch_first=True, padding_value=-10)

    return DataLoader(corpus, batch_size=batch_size, shuffle=shuffle,
                      collate_fn=batch_data_proc)
```

由于Bert模型的输入包含`[CLS]`和`[SEP]`的token，为解决这个问题，我们在`tokenizer`方法调用时为参数`add_special_tokens=False`取值来取消这两个特殊标签的生成。这样label 便可以和bert的输出对齐。在实际预测时并不影响结果的正确性。



3. 基于Bert的NER模型

使用transformers框架所提供的 `AutoModel` `ForTokenClassification` 可以快速构建基于Bert的NER模型。

```
from transformers import AutoModelForTokenClassification

tags_t2i = {'O':0, 'B-LOC':1, 'I-LOC':2, 'B-ORG':3, \
          'I-ORG':4, 'B-PER':5, 'I-PER':6 }
# 反向dict
tags_i2t = {v:k for k,v in tags_t2i.items()}

# ner模型
model = AutoModelForTokenClassification.from_pretrained(
    model_respo,
    num_labels=7).to(DEVICE)

# conifig标签映射指定
model.config.id2label = tags_t2i
model.config.label2id = tags_i2t
```

💡 ForTokenClassification末端包含了基于token的推理结果，同时也会返回损失函数结果。

4. 模型训练&评估

```
# optimizer
optimizer = optim.Adam(model.parameters(), lr=1e-5) # 优化器

# 最佳f1 score
max_f1 = 0

# 训练
for epoch in range(5):
    model.train()
    tpbar = tqdm(train_dl)
    for item, tags in tpbar:
        item = {k:v.to(DEVICE) for k,v in item.items()}
        tags = tags.to(DEVICE)
        optimizer.zero_grad()
        outputs = model(**item, labels=tags)
        loss = outputs.loss
        loss.backward()
```

```
        optimizer.step()

        tpbar.set_description(f'Epoch:{epoch+1}, Loss:{loss.item():.4f}')
```

- **NER任务序列标注的分类评估**

  在传统的分类识别任务中，模型评估的目标是以样本为单位。计算一个样本预测的正确与否，通过统计的结果，就可以使用混淆矩阵，精确率，召回率，F1 Score等计算公式来计算相关类别预测情况或者整体的预测情况。

  在中文命名实体识别任务中，一个实体是由一至多个汉字组成的。通常使用序列来标注实体范围。

  对于模型预测结果来讲，单个序列标注预测的准确性并不能代表该实体预测的正确性。

```
text:   特  斯  拉  上  海  超  级  工  厂
pred:     B  I  O  I  I  I  O  I  I
real:  B  I  I  I  I  I  I  I  I
```

  上面的例子就是个证明，`特斯拉上海超级工厂`是一个实体，但模型预测的序列标签中，有两个标签预测的类型和真实类型不一致。虽然标签预测的准确率达到了 7/9 ≈ 0.78%，但在利用标签界定实体时，还是错误的。这就是NER任务评估的难点

  那么进行实体预测的准确度评估，就是意味着需要检查预测实体标签序列的连续性和完整性。对于每个实体标签，都必须和真实标签长度和类型一致。这样才算是预测的实体结果正确。

  解决这个问题，我们可以采用编码的方式，但需要一些时间和代码维护。好在，我们现在可以使用**seqeval**来解决这个问题。

- **seqeval**

  seqeval 是一个用于序列标签评估的 Python 框架。 它可以评估分块任务的性能，例如命名实体识别、词性标注、语义角色标注等。

  安装

  `pip install seqeval`

  序列标注类型

  在seqeval 所支持的序列标注列表如下：

  - IOB1：标签`I`用于文本块中的字符，标签`O`用于文本块之外的字符。标签`B`用于在该文本块前面接着一个同类型的文本块情况下的第一个字符（分隔实体）。
  - IOB2：每个文本块都以标签`B`开始，除此之外，跟IOB1一样
  - IOE1：标签`I`用于独立文本块中，标签`E`仅用于同类型文本块连续的情况，假如有两个同类型的文本块，那么标签`E`会被打在第一个文本块的最后一个字符。
  - IOE2：每个文本块都以标签`E`结尾，无论该文本块有多少个字符，除此之外，跟IOE1一样。
  - IOBES(only in strict mode)
  - BILOU(only in strict mode)

IOB1、IOB2 、IOE1、IOE2几种标注方式对比:

```
text:   晓     明     晓     艳     一     起     出     游
IOB1:     I-PER  I-PER  B-PER  I-PER  O     O     O     O
IOB2:  B-PER  I-PER  B-PER  I-PER  O     O     O     O
IOE1:  I-PER  E-PER  I-PER  I-PER  O     O     O     O
IOE2:  I-PER  E-PER  I-PER  E-PER  O     O     O     O
```

seqeval官方演示

```
from seqeval.metrics import accuracy_score
from seqeval.metrics import classification_report
from seqeval.metrics import f1_score
y_true = [['O', 'O', 'O', 'B-MISC', 'I-MISC', 'I-MISC', 'O'], ['B-PER', 'I-PER', 'O']]
y_pred = [['O', 'O', 'B-MISC', 'I-MISC', 'I-MISC', 'I-MISC', 'O'], ['B-PER', 'I-PER', 'O']]
print(accuracy_score(y_true, y_pred))
print(f1_score(y_true, y_pred))
print(classification_report(y_true, y_pred))

"""
输出:
0.8
0.5
              precision    recall  f1-score   support

        MISC       0.00      0.00      0.00         1
         PER       1.00      1.00      1.00         1

   micro avg       0.50      0.50      0.50         2
   macro avg       0.50      0.50      0.50         2
weighted avg       0.50      0.50      0.50         2
"""
```

- 结合seqeval的NER评估

```
# 评估
model.eval()
with torch.no_grad():
    total_loss = 0
    total_count = 0
    total_pred, total_tags = [],[]
    tpar = tqdm(test_dl)
    for item, tags in tpar:
        item = {k:v.to(DEVICE) for k,v in item.items()}
        tags = tags.to(DEVICE)
        outputs = model(**item, labels=tags)
        total_loss += outputs.loss.item()
        total_count += 1

        # attention_mask筛选真实token结果

        # seqeval评估
        pred = outputs.logits.argmax(dim=-1)  #(batch_size, seq_len)

        tk_pred = pred.masked_select(item['attention_mask'].bool())
        tk_true = tags.masked_select(item['attention_mask'].bool())

        pred_tags = [tags_i2t[tk.item()] for tk in tk_pred]
        true_tags = [tags_i2t[tk.item()] for tk in tk_true]

        total_pred.append(pred_tags)
        total_tags.append(true_tags)

assert len(total_pred) == len(total_tags), '预测结果与真实结果长度不一致'
print('Accuracy:', accuracy_score(total_tags, total_pred))
f1 = f1_score(total_tags, total_pred)
print('F1 score:', f1)
```

```
    print(classification_report(total_tags, total_pred))

    print(f'Epoch:{epoch+1}, Test Loss:{total_loss/total_count:.4f}')
```

5.模型预测

```
nlp = pipeline("ner", model=model, tokenizer=tokenizer)
example = "我是王德发，目前住在北京。"

ner_results = nlp(example)
print(ner_results)
```

Trainer微调模型

Trainer是transformers框架提供的一个模型训练类，能够方便的对各类模型进行训练和微调。

Trainer类的构建参数多达 13 个，与其关联的**TrainingArguments** 的参数更是多达 116 个。所以学习掌握Trainer类的最好方式，是通过一个完整示例演示。

Trainer类模型微调流程

1. 准备Dataset
2. Dataset数据预处理
3. 创建TrainingArguments对象，封装模型训练参数
4. 创建Trainer对象，封装模型训练控制参数、Dataset、评估方法、优化器、学习率调度器等
5. 调用Trainer对象的train方法开始训练

Datasets

🤗 Datasets是一款功能强大的库，专为音频、计算机视觉和自然语言处理（NLP）任务设计。

它能让用户仅用一行代码就加载数据集，并凭借强大的数据处理方法，快速为深度学习模型训练准备数据。

Datasets 基于Apache Arrow格式，它在处理大型数据集时可实现零拷贝读取，突破内存限制，保障处理的高速与高效。

Datasets 与Hugging Face Hub深度集成，方便用户加载和分享数据集，推动机器学习社区的数据交流。

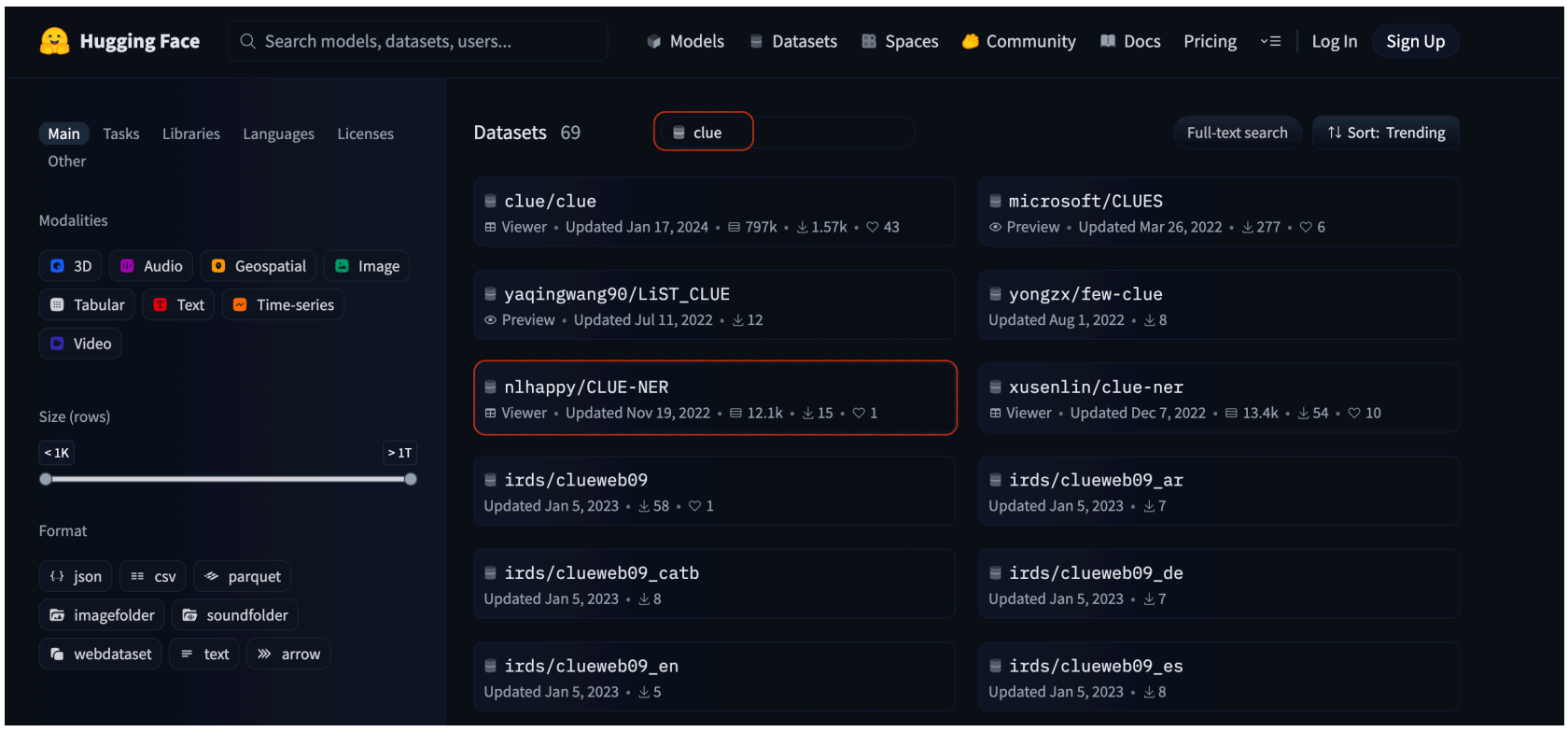
此外，Datasets 还提供了教程、操作指南、概念讲解和技术参考等丰富学习资源，助力用户全面掌握数据集的加载、访问和处理。

- **安装Datasets库**

  `pip install datasets`

- **检索加载Dataset**

  https://huggingface.co/datasets



选择CLUE的NER数据集

关于clue benchmark

https://cluebenchmarks.com/

```
from datasets import load_dataset

ds = load_dataset("nlhappy/CLUE-NER")
```

Datasets数据预处理

模型训练的需要，我们要指定 Label（对应标签名）与数值类型之间的映射（map）。

```
映射 entities 映射为 tag_index 索引值
O → 0
PER:B-PER → 1 I-PER → 2  (1*2)-1 = 1  (1*2)=2
LOC:B-LOC → 3 I-LOC → 4  (2*2)-1 = 3  (2*2)=4
ORG:B-ORG → 5 I-ORG → 6  (3*2)-1 = 5  (3*2)=6
```

```
# entity_index
entites = ['O'] + list({'movie', 'name', 'game', 'address', 'position', \
          'company', 'scene', 'book', 'organization', 'government'})
tags = ['O']
for entity in entites[1:]:
    tags.append('B-' + entity.upper())
    tags.append('I-' + entity.upper())

entity_index = {entity:i for i, entity in enumerate(entites)}
```

生成了entity_index的映射字典后，还要通过`Dataset`的`map`函数将实体标签映射为索引值。

```
def entities_tags_proc(item):
        # 实体大小
        data_size = len(item['entities'])
        ner_tags = []
        for i in range(data_size):
            text_len = len(item['text'][i])
            entites = item['entities'][i]
            tags = torch.tensor([entity_index['O']] * text_len)
            for entity in entites:
                start,end = entity['start_offset'], entity['end_offset']
```

```
            label = entity['label']
            tags[start] = entity_index[label] * 2 - 1
            tags[start+1:end] = entity_index[label] * 2
        ner_tags.append(tags)
    return {'ner_tags': ner_tags}

# map返回映射后包含ner_tags字段的Dataset新版本对象
ds1 = datasets.map(entities_tags_proc, batched=True)
```

日期和英文单词，在中文预训练bert词典中，并不会按字符拆分。所以可能会发生字符与token不对齐的情况。

`“2000年2月” → “2000”, ”年”, “2”, ”月”`

可以通过tokenizer返回对象的`word_ids`函数来返回 ***输入字符 → token index*** 之间的位置映射关系。

从而纠正实体标签位置偏移的问题。

整体处理过程也通过`Dataset`的`map`函数来映射并返回处理后新的字段。

```
def corpus_proc(item):
    """语料中文本转换成为模型输入项"""
    input_data = tokenizer(item['text'],
                truncation=True,
                add_special_tokens=False,
                max_length=512）# 返回单条语句处理结果（python类型）
# 生成token_index索引和tags对齐!
# 遍历token index，匹配tag标签类别和token数量
total_adjusted_labels = []
for k in range(len(input_data['input_ids'])):
    # token和输入字符之间映射关系
    word_ids = input_data.word_ids(k)

    exits_label_ids = item['ner_tags'][k]  # 输入项每个字符的tag
    adjust_label_ids = []                  # 修正后每个token的tag

    prev_wid = -1
    i = -1
    for wid in word_ids:    # 假设输入 2000年2月 → word_ids[1,1,1,1,2,3]
        if wid != prev_wid:
            prev_wid = wid
            i += 1
        adjust_label_ids.append(exits_label_ids[i])
    total_adjusted_labels.append(adjust_label_ids)
input_data['labels'] = total_adjusted_labels
    return input_data

ds2 = ds1.map(corpus_proc, batched=True)
```

最后，通过Dataset的`set_format`函数来设置要映射的类型

```
ds2.set_format('torch', columns=['input_ids','attention_mask', \
            'token_type_ids', 'labels'])
```

函数会筛选columns参数指定的字段传给模型，同时第1个参数的 ’torch’ 表明Dataset中的每条记录都转换为pytorch张量类型。

TrainingArguments

在`Trainer`创建之前，需要先创建`TrainingArguments`对象。由于模型训练兼容性的设计，`TrainingArguments`的参数有116个之多。

https://huggingface.co/docs/transformers/v4.52.2/en/main_classes/trainer#transformers.TrainingArguments

这里并非所有的参数都需要创建和初始化。只需要根据经验选择初始化一些参数就可以完成模型训练或微调的要求。

```
# 模型训练参数
training_args = TrainingArguments(
    output_dir='example_trainer',  # 训练输出和保存目录
        learning_rate=4e-5,           # 模型微调学习率
    num_train_epochs=3,           # 训练epoches
    save_safetensors=False,       # 不以safetensors格式保存模型（使用pytorch格式）
    save_only_model=True,         # 只保存模型（不包含训练参数）
    per_device_train_batch_size=32, # 训练批次（支持多设备）
    eval_strategy='epoch',        # 评估策略
    eval_steps=0.1,               # 每10%批次 进行一次评估
)
```

DataCollator

Data collators是用于***将数据集元素列表转换成批次矩阵***的对象。`DataCollator`会实现一些预处理（如填充）功能，或进行进行随机数据增强。

具体到`DataCollatorForTokenClassification`子类，主要是为label标签矩阵进行填充，以便于让label矩阵和模型推理结果矩阵之间对齐，实现更高效的矩阵运算。

```
collator = DataCollatorForTokenClassification(tokenizer=tokenizer,
                                                                padding=True，
                                                                label_pad_token_id=-100)
```

首先，把`tokenizer`参数传入，以便后续协助矩阵完成填充。

参数`padding`为`True`，表示label矩阵需要填充。填充值由`label_pad_token_id`指定（默认值`-100`）

Trainer

`Trainer`类是 transformers 中用于模型训练的一个重要类，它与`TrainingArguments`类配合使用，提供了完整的训练API。

https://huggingface.co/docs/transformers/v4.52.2/en/main_classes/trainer

```
trainer = Trainer(
        model=model,                # 训练模型
        args=training_args,          # 训练参数
        train_dataset=ds2['train'],    # 训练数据集
        tokenizer=tokenizer,
        data_collator=data_collator,    # 矩阵填充处理
    )
```

构建完Trainer类对象后，就可以对预训练模型进行微调了。

```
trainer.train()
```

trainer会按训练的batch长度自动保存模型，以方便于后续的预测加载。

我们还可以在完成训练后通过调用predict方法来对训练后的模型进行测试。

```
predict = trainer.predict(ds2['test'])

print(predict)
```